Empir Software Eng
DOI 10.1007/s10664-015-9414-4

On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach

Mohamed Wiem Mkaouer[1] · Marouane Kessentini[1] · Slim Bechikh[1] · Mel Ó Cinnéide[2] · Kalyanmoy Deb[3]

Abstract Search-based software engineering (SBSE) solutions are still not scalable enough to handle high-dimensional objectives space. The majority of existing work treats software engineering problems from a single or bi-objective point of view, where the main goal is to maximize or minimize one or two objectives. However, most software engineering problems are naturally complex in which many conflicting objectives need to be optimized. Software refactoring is one of these problems involving finding a compromise between several quality attributes to improve the quality of the system while preserving the behavior. To this end, we propose a novel representation of the refactoring problem as a many-objective one where every quality attribute to improve is considered as an independent objective to be optimized. In our approach based on the recent NSGA-III algorithm, the refactoring solutions are evaluated using a set of 8 distinct objectives. We evaluated this approach on one industrial project and seven open source systems. We compared our findings to: several other many-objective techniques (IBEA, MOEA/D, GrEA, and DBEA-Eps), an existing multi-objective approach

Communicated by: Marouane Kessentini and Guenther Ruhe

✉ Mohamed Wiem Mkaouer
mmkaouer@umich.edu

Marouane Kessentini
marouane@umich.edu

Slim Bechikh
sbechikh@umich.edu

Mel Ó Cinnéide
mel.ocinneide@ucd.ie

Kalyanmoy Deb
kdeb@egr.msu.edu

[1] University of Michigan, Dearborn, MI, USA

[2] University College Dublin, Dublin, Ireland

[3] Michigan State University, East Lansing, MI, USA

a mono-objective technique and an existing refactoring technique not based on heuristic search. Statistical analysis of our experiments over 31 runs shows the efficiency of our approach.

Keywords Search-based software engineering · Refactoring · Software quality · Many-objective optimization

1 Introduction

Search-based software engineering (SBSE) studies the application of meta-heuristic optimization techniques to software engineering problems (Harman and Jones 2001). Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, objective function, and solution change operators, there are a multitude of search algorithms that can be applied to solve that problem. Search-based techniques are widely applied to solve software engineering problems such as in testing, modularization, refactoring, planning, etc. (Harman 2013; Harman and Tratt 2007; Harman et al. 2012).

Based on a recent SBSE survey (Harman et al. 2012), the majority of existing work treats software engineering (SE) problems from a single-objective point of view, where the main goal is to maximize or minimize one objective, e.g., correctness, quality, etc. However, most SE problems are naturally complex in which many conflicting objectives need to be optimized such as model transformation, design quality improvement, test suite generation etc. The number of objectives to consider for most of software engineering problems is, in general, high (more than three objectives); such problems are termed *many-objective*. We claim that the reason that software engineering problems have not been formulated as many-objective problems is because of the challenges in constructing a many-objective solution. In this context, the use of traditional multi-objective techniques, e.g., NSGA-II (Deb et al. 2002), widely used in SBSE, is clearly not sufficient (Deb and Jain 2012).

There is a growing need for scalable search-based software engineering approaches that address software engineering problems where a large number of objectives are to be optimized. Recent work in optimization has proposed several solution approaches to tackle many-objective optimization problems (Jaimes et al. 2009; Zhang and Li 2007) using e.g., objective reduction, new preference ordering relations, decomposition, etc. However, these techniques have not yet been widely explored in SBSE (Harman et al. 2012). To the best of our knowledge and based on recent SBSE surveys (Harman et al. 2012), very few studies used many-objective techniques to address software engineering problems such as the work proposed by Sayyad et al. (2013a) that uses a many-objective approach, IBEA (Indicator-Based Evolutionary Algorithm) (Zitzler and Künzli 2004), to address the problem of software product line creation. However, the number of considered objectives is limited to five. In our recent work (Mkaouer et al. 2014), we proposed a many-objective approach for software refactoring using a set of 15 quality metrics as separate objectives. Software refactoring is one of those software engineering problems that require several quality objectives to be satisfied. Although, the approach has given promising results, some limitations have been raised, investigated and resulted in this extension that copes with the following concerns: the choice of using of 15 metrics as separate objectives, at the expense of higher computational complexity and higher number of solutions, made the benefit of selecting good refactoring solutions unclear due to conflict uncertainty between some objectives and the difficult interpretation of their metric values. Another limitation of our previous work is the exclusive optimization of the system's structure without taking into consideration the semantic coherence of its actors (classes, methods, attributes). Furthermore, one of the main objectives of Pareto-optimality is to allow the user to choose among equivalent solutions the one(s) that satisfies better his/her preferred objectives. However, in our context, it is harder to ask a developer to express his preference in terms of 15 internal quality attributes, so it is difficult for developers to select a solution from the high number of non-dominated refactoring solutions.

To address the above challenges, we propose to extend our previous work using a different many-objective formulation. We assume that it would be more convenient for developers to formulate their quality preferences in terms of external quality attributes such as reusability, flexibility and understandability instead of a large number of quality metrics (O'Keeffe and Ó Cinnéide 2008; Bansiya and Davis 2002). Thus, the goal of improving the software overall quality is still maintained while the number of objectives has been reduced. This is being done through the aggregation of metrics, previously optimized separately, into 8 objectives described as external quality attributes. This representation helps in analyzing the impact of applying refactoring operations on raising the conflicts between these objectives during the solution evolution. Thus, we propose in this paper to formulate the refactoring problem using the quality attributes of QMOOD as objectives (Bansiya and Davis 2002) along with the number of refactorings and design coherence preservation. To this end, we adapted the many-objective algorithm NSGA-III (Jain and Deb 2014). NSGA-III is a many-objective algorithm proposed by Deb and Jain (2013). The basic framework remains similar to the original NSGA-II algorithm (Deb et al. 2002), with significant changes in its selection mechanism. We implemented our approach and evaluated it on seven large open source systems and one industrial project provided by our industrial partner. We compared our findings to: several other many-objective techniques, an existing multi-objective approach (Ouni et al. 2012a), a mono-objective technique (Kessentini et al. 2011) and an existing refactoring technique not based on heuristic search (Tsantalis et al. 2008). Statistical analysis of our experiments over 31 runs shows the efficiency of our approach.

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

(1) The paper introduces a novel formulation of the refactoring problem through several objectives using NSGA-III.
(2) The paper reports the results of an empirical study with an implementation of our many-objective approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than several existing refactoring techniques based on a benchmark of seven open source systems and one industrial project. The paper also evaluates the relevance and usefulness of the suggested refactoring for software engineers in improving the quality of their systems.

The remainder of this paper is structured as follows. Section 2 provides background and an overview of many-objective optimization techniques and potential applications to software engineering problems. Section 3 describes our adaptation of NSGA-III to automate code refactoring and the results obtained from our experiment are presented then discussed in Sections 4 and 5. Threats to validity are described in Section 6. Section 7 ends this paper with concluding remarks and future work.

2 Background and related work

2.1 Software refactoring

Refactoring is defined as the process of improving code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions and enhance comprehension. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, NetBeans, provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in Software Engineering. In order to identify which parts of the source code need to be refactored, most existing work relies on the notion of code smells (e.g., Fowler's textbook (Fowler et al. 1999)), also called design defects or anti-patterns (Brown et al. 1998).

The impact of code smells on software systems has been the subject of several studies over the past decade since their first introduction by Fowler et al. (1999). They defined 22 code smells as structural code flaws that may decrease the overall software quality and serve as indicators to potential issues related to software evolution and maintenance. To cope with these smells, Fowler has introduced a set of 72 Refactoring operations to fix code smells and thus improving the system overall design.

The detection process can either be manual, semi-automated or fully automated. Mäntylä et al. (2003) provided an initial formulization of the code smells, in terms of metrics, based on analyzing Fowler's smells description, they studied the correlation between the smells and provided a classification according to their similarity. Mäntylä revealed that the manual detection of smells is dependent to the level of expertise of detection performers, which represents one of the main limitations of this approach. Marinescu (2004) presented an automated framework for smells identification using detection strategies which are defined by metric-based rules. Moha et al. (2009) presented a semi-automated technique called DECOR. This framework allows subjects to manually suggest their own defects through their description with domain specific language, then the framework automatically searches for smells and visually reports any finding. Most of the above mentioned work focuses mainly on smells specification in order to improve their detection, for the correction step, their proposals were limited to guidance on how to manually manage these smells by suggesting refactoring recommendations according to detected smell's type.

There is a growing interest in the area of identifying refactoring opportunities. Similar to the detection's state of art, refactoring techniques can be either manual, semi-automated or fully automated. Fowler et al. (1999) manually linked a set of suggested refactorings (Move Attribute, Extract Class, Move Method etc.) with each identified smell. Van Emden & Moonen (2002) focused on the detection of two code smells related to the Java language and suggested their specific correction. (Martin 2000) has used patterns to cope with the poor system design in presence of code smells. Mäntylä proposed refactoring solutions based on developers' opinions and driven by an automatic detection of structural anomalies at the source code level. Counsell et al. (2006) refined Fowler's suggested refactorings by prioritizing some refactorings in the execution order. Piveta el al. (2006) discussed when refactoring opportunities are eventually needed when detecting bad smells in aspect-oriented software. (Meananeatra 2012) presented another semi-automated heuristic for refactoring, it generates a graph or refactoring sequences that are being refined using three main objectives to minimize number of code smells, number of refactorings and number of refactored code elements.

JDeodorant (Tsantalis et al. 2008) is an automated refactoring tool implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. Du Bois et al. (2004) has investigated decreasing coupling and increasing cohesion metrics through the refactoring opportunities and used them to perform an optimal distribution of features over classes. They analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only some possible refactoring operations with few number of quality metrics. Murphy-Hill (2006; Ge and Murphy-Hill 2011, 2014) proposed several techniques and empirical studies to support refactoring activities. In (Murphy-Hill 2006) the authors proposed new tools to assist software engineers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, in (Ge and Murphy-Hill 2014) the authors have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, but check the correctness of the transformation automatically. BeneFactor (Ge and Murphy-Hill 2011) and WitchDoctor (Foster et al. 2012) can detect manual refactorings and then complete them automatically. Tahvildari et al. (2003) also proposed a framework of object-oriented metrics used to suggest to the software engineer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig (2011) proposed an interactive refactoring technique to improve the parallelism of software systems. Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-conditions). Kataoka et al. (2001) used invariants in the detection and extraction of source code fragments in need of refactoring.

Search-based refactoring represents fully automated refactoring driven by metaheuristic search and guided by software quality metrics and used subsequently to address the problem of automating design improvement. Seng et al. (2006) propose a search-based technique that uses a genetic algorithm over refactoring sequences. The employed metrics are mainly related to various class level properties such as coupling, cohesion, complexity and stability. The approach was limited only to the use of one refactoring operation type, namely 'move method'. In contrast to (O'Keeffe and Ó Cinnéide 2008) their fitness function is based on well-known measures of coupling between program components. Both these approaches use weighted-sum to combine metrics into a fitness function, which is of practical value but is a questionable operation on ordinal metric values. Kessentini et al. (2011) also propose a single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of code smells detected using a set of quality metrics.

Harman and Tratt were the first to introduce the concept of Pareto optimality to search-based refactoring (Harman and Tratt 2007). They use it to combine two metrics into a fitness function, namely CBO (coupling between objects) and SDMPC (standard deviation of methods per class), and demonstrate that it has several advantages over the weighted-sum approach, and its introduction, several multi-objective techniques have appeared (Ó Cinnéide M et al. 2012). The work by Ouni et al. (2012a) proposes a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. The proposed approach is based on two objective functions, quality (proportion of corrected code smells) and code modification effort, to recommend a sequence of refactorings that provide the best trade-off between quality and effort. In this study, we propose to address the refactoring problem using more than two objectives. To this end, we are adapting many objective algorithms that will be discussed in the next section.

2.2 Many-objective search-based software engineering

Recently many-objective optimization has attracted much attention in Evolutionary Multi-objective Optimization (EMO) which is one of the most active research areas in evolutionary computation (Bader and Zitzler 2011). By definition, a many-objective problem is multi-objective one but with a high number of objectives M, i.e., $M>3$. Analytically, it could be stated as follows (Deb 2001):

$$\begin{cases} Min\, f(x) = [f_1(x), f_2(x), \ldots, f_M(x)]^T, & M > 3 \\ g_j(x) \geq 0 & j = 1, \ldots, P; \\ h_k(x) = 0 & k = 1, \ldots, Q; \\ {x_i}^L \leq x_i \leq {x_i}^U & i = 1, \ldots, n. \end{cases} \tag{1}$$

where M is the number of objective functions and is *strictly greater* than 3, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the decision variable x_i (i.e., i^{th} component of x). A solution x satisfying the (P+Q) constraints is said to be feasible and the set of all feasible solutions defines the feasible search space denoted by Ω.

In this formulation, we consider a minimization multi-objective problem (MOP) since maximization can be easily turned to minimization based on the duality principle. Over the two past decades, several Multi-Objective Evolutionary Algorithms (MOEAs) have been proposed with the hope to work with any number of objectives M. Unfortunately, It has been demonstrated that most MOEAs are ineffective in handling such type of problems. For example, NSGA-II, which is one of the most used MOEAs, compares solutions based on their non-domination ranks. Solutions with best ranks are emphasized in order to converge to the Pareto front. When $M>3$, only the first rank may be assigned to every solution as almost all population individuals become non-dominated with each other. Without a variety of ranks, NSGA-II cannot keep the search pressure anymore in high dimensional objective spaces.

The difficulty faced when solving a many-objective problems could be summarized as follows. Firstly, most solutions become equivalent between each other according to the Pareto dominance relation which deteriorates *dramatically* the search process ability to converge towards the Pareto front and the MOEA behavior becomes very similar to the random search one. Secondly, a search method requires a very high number of solutions (some thousands and even more) to cover the Pareto front when the number of objectives increases. For instance, it has been shown that, in order to find a good approximation of the Pareto front for problems involving 4, 5 and 7 objective functions, the number of required non-dominated solutions is about 62 500, 1 953 125 and 1 708 984 375 respectively (Jaimes et al. 2009) which makes the decision making task very difficult. Thirdly, the objective space dimensionality increases significantly, which makes promising search directions very hard to find. Fourthly, the diversity measure estimation becomes very computationally costly since finding the neighbors of a particular solution in high dimensional spaces is very expensive. Fifthly, recombination operators become inefficient since population members are likely to be widely distant from each other which yields to children that are not similar to their parents; thereby making the recombination operation inefficient in producing promising offspring individuals. Finally, although it is not a matter that is directly related to optimization, the Pareto front visualization becomes more complicated, therefore making the interpretation of the MOEA's results more difficult for the user.

Recently, researchers have proposed several solution approaches to tackle many-objective optimization problems. Table 1 illustrates a summary of existing many-objective approaches. Firstly, we find the o*bjective reduction approach*, which involves finding the minimal subset of objective functions that are in conflict with each other. The main idea is to study the different conflicts between the objectives. The objective reduction approach attempts to eliminate objectives that are not essential to describe the Pareto-optimal front (Jaimes et al. 2009). Even when the essential objectives are four or more, the reduced representation of the problem has a favorable impact on the search efficiency, computational cost, and decision making. However, although this approach has solved benchmark problems involving up to 20 objectives, its applicability in real world setting is not straightforward and it remains to be investigated since most objectives are usually in conflict with each other in real problems (Ó Cinnéide M et al. 2012). Secondly, we have the *incorporation of decision maker's preferences*: When the number of objective functions increases, the Pareto optimal approximation would be composed by a huge number of non-dominated solutions. Consequently, the selection of the final alternative would be very difficult for the human decision maker (DM). In reality, the DM is not interested with the whole Pareto front rather than the portion of the front that best matches his/her preferences, called the Region of Interest (ROI). The main idea is to exploit the DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the ROI on problems involving more than 3 objectives (Ben Said et al. 2010). Preference-based MOEAs have demonstrated several promising results. Thirdly, we find *new preference ordering relations*. Since the Pareto dominance has the inability to differentiate between solutions with the increased number of objectives, researchers have proposed several new alternative relations. These relations try to circumvent the failure of the Pareto dominance by using additional information such as the ranks of the particular solution regarding the different objectives and the related population (di Pierro et al. 2007), but may not be agreeable to the decision makers. Fourthly, we have *decomposition*. This technique consists in decomposing the problem into several sub-problems and then solving these sub-problems simultaneously by exploiting the parallel search ability of evolutionary algorithms. The most reputable decomposition-based MOEA is MOEA/D (Zhang and Li 2007). Finally, we find the u*se of a predefined multiple targeted search*. Inspired by preference-based MOEAs and the decomposition approach, recently, Deb and Jain (Deb and Jain 2013), and Wang et al. (2013) have proposed a new idea that involves guiding the population during the optimization process based on multiple predefined targets (e.g., reference points, reference direction) in the objective space. This idea has demonstrated very promising results on MOPs involving up to 15 objectives.

Harman et al. (2012), stated that most of existing search-based software engineering (SBSE) related work treats software engineering (SE) problem from mono-objective point of view. As SE problems are naturally multi-objective, many techniques has been elaborated by handling more than one objective. However, as the number of objectives scales, non-dominance based methodologies' performance drastically degrades. It has been shown in (Garza-Fabre et al. 2011) that when the number of objectives goes beyond five, more than 90 % of the population becomes non-dominated to each other which indulges the stagnation of the population at an early stage of its evolution. In addition, visualization techniques, being widely used in SE, are no longer efficient in high dimensional problems. Subsequently, many-objective methodologies have been recently

Table 1 Summary of many-objective approaches

Approach	Basic idea	Example algorithms	No. of objectives	Real world many-objective application
Objective reduction	Find the minimal subset of conflicting objectives, then eliminate the objectives that are not essential to describe the Pareto optimal front.	1) PCA-NSGA-II (Deb and Saxena 2006)	10	1) Not found
		2) PCSEA (Singh et al. 2011)	20	2) Water resource problem
Incorporating decision maker's preferences	Exploit DM's preferences in order to differentiate between Pareto equivalent solutions so that we can direct the search towards the region of interest instead of the whole front.	1) r-NSGA-II (Ben Said et al. 2010)	10	1) Payment scheduling negotiation problem
		2) PBEA (Thiele et al. 2009)	10	2) Not found
		3) R-NSGA-II (Deb et al. 2006)	10	3) Welded beam design problem
New preference ordering relations	Propose alternative preference relations that are different from the Pareto dominance.	1) Preference Order Ranking–based algorithm (di Pierro et al. 2007)	8	1) Water distribution problem
		2) Ranking dominance-based algorithm (Kukkonen and Lampinen 2007)	10	2) Not found
		3) IBEA (Zitzler and Künzli 2004)	5	3) Software product line management
		4) HypE (Bader and Zitzler 2011)	20	4) Not found
Decomposition	Decompose the problem into several sub-problems and then solve these sub-problems simultaneously by exploiting the parallel search ability of EAs.	1) MOEA/D (Zhang and Li 2007)	5	1) Not found
Use of a predefined multiple targeted search	Guide the population during the optimization process based on multiple predefined targets (e.g., reference points, reference direction) in the objective space.	1) PICEA (Wang et al. 2013)	10	1) Not found
		2) NSGA-III (Deb and Jain 2013)	15	2) Crash-worthiness Design of Vehicles

investigated in the area of SBSE. The following table cites the previous work which its problem formulation has a number of objectives being equal or higher than five.

Although Search-based refactoring has shown promising results when addressing the problem of automating design improvement (O'Keeffe and Ó Cinnéide 2008), most of the current methodologies are still using a limited number of fitness functions, resulting on system repair from a single objective and it is difficult, in most cases, to extend the set of refactorings to handle other potential perspectives due to their conflicting nature. Thus, in this work, we consider the refactoring problem as many-objective optimization problem based on NSGA-III that will be described in the next section (Table 2).

3 Adapting NSGA-III for the software refactoring problem using quality attributes

3.1 NSGA-III

NSGA-III is a very recent many-objective algorithm proposed by Deb & Jain (2013). The basic framework remains similar to the original NSGA-II algorithm with significant changes in its selection mechanism. Figure 1 gives the pseudo-code of the NSGA-III procedure for a particular generation t. First, the parent population P_t (of size N) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected from the combined population to form the parent population for the next generation. The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members as described in Fig. 2. To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count ρ for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and a member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

It is worth noting that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as ρ_j for the j-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\mathrm{min}}=\{j\text{: argmin}_j\ (\rho_j)\}$ having minimum ρ_j. In case of multiple such reference points, one ($j^* \in J_{\mathrm{min}}$) is chosen at random. If $\rho_{j*}=0$ (meaning that there is no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exists

Table 2 Many-objectives approaches applied in software engineering (Mkaouer et al. 2015)

Author(s)	Year	Title	Area	Algorithm(s)	Number of objectives
Zhuang et al. (2007)	2007	Solving Multi-objective and Fuzzy Multi-attributive Integrated Technique for QoS Aware Web Service Selection	Design Engineering	MOGA	5
Wada et al. (2008)	2008	Multiobjective Optimization of SLA-Aware Service Composition	Design Engineering	E^3-MOGA	10
Bowman et al. (2010)	2010	Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms	Design Engineering	SPEA2	5
Kremmel et al. (2011)	2011	Software Project Portfolio Optimization with Advanced Multiobjective Evolutionary Algorithms	Management Engineering	mPOEMS,	5
Rodriguez et al. (2011)	2011	Multiobjective Simulation Optimisation in Software Project Management	Management Engineering	NSGA-II	5
Praditwong et al. (2010)	2011	Software Module Clustering as a Multi-Objective Search Problem	Design Engineering	Two-Archive GA	5
Barros (2012)	2012	An Analysis of the Effects of Composite Objectives in Multiobjective Software Module Clustering	Design Engineering	NSGA-II	5
Colanzi & Vergilio (2012)	2012	Applying Search Based Optimization to SPL Architectures: Lessons Learned	Design Engineering	NSGA-II	5
Sarro et al. (2012)	2012	Single and Multi Objective GP for Software Development Effort Estimation	Management engineering	MOGP	5
Sayyad et al. (2013a)	2013	On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines	Requirements Engineering	IBEA	Up to 5
Sayyad et al. (2013b)	2013	Scalable Product Line Configuration: A Straw to Break the Camel's Back	Requirements Engineering	IBEA	5

Table 2 (continued)

Author(s)	Year	Title	Area	Algorithm(s)	Number of objectives
Yao (2013)	2013	Some Recent Work on Multi-objective Approaches to Search-Based Software Engineering.	Design Engineering	Two-Archive Algorithm	5
Ramírez et al. (2014)	2014	On the Performance of Multiple Objective Evolutionary Algorithms for Software Architecture Discovery	Design Engineering	SPEA2, NSGA-II, ε-MOEA, MOEA/D, GrEA	6
Mkaouer et al. (2013)	2014	High Dimensional Search-based Software Engineering: Finding Tradeoffs among 15 Objectives for Automating Software Refactoring Using NSGA-III	Maintenance Engineering	NSGA-III	15
Kalboussi et al. (2013)	2014	Preference-Based Many-Objective Evolutionary Testing Generates Harder Test Cases for Autonomous Agents	Testing Engineering	P-MOET	7
Olaechea et al. (2014)	2014	Comparison of Exact and Approximate Multi-Objective Optimization for Software Product Lines	Requirements Engineering	GIA, IBEA	Up to 7
Mkaouer et al. (2015)	2015	Many-Objective Software Remodularization Using NSGA-III	Maintenance Engineering	NSGA-III	8

NSGA-III procedure at generation t

Input: H structured reference points Z^s, parent population P_t

Output: P_{t+1}

```
Begin
$S_t \leftarrow \emptyset$, $i \leftarrow 1$;
$Q_t \leftarrow$ Variation ($P_t$);
$R_t \leftarrow P_t \cup Q_t$;
($F_1$, $F_2$, ...) $\leftarrow$ Non-dominationed_Sort ($R_t$);
Repeat
  $S_t \leftarrow S_t \cup F_i$; $i \leftarrow i+1$;
Until | $S_t$ | $\geq N$;
$F_l \leftarrow F_i$; /*Last front to be included*/
If | $S_t$ | $= N$ then
  $P_{t+1} \leftarrow S_t$;
Else
  $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;
  /*Number of points to be chosen from $F_l$*/
  $K \leftarrow N - |P_{t+1}|$;
  /*Normalize objectives and create reference set $Z^r$*/
  Normalize ($F^M$; $S_t$; $Z^r$; $Z^s$);
  /*Associate each member $s$ of $S_t$ with a reference point*/
  /*$\pi(s)$: closest reference point*/
  /*$d(s)$: distance between $s$ and $\pi(s)$*/
  [$\pi(s)$, $d(s)$] $\leftarrow$ Associate ($S_t$, $Z^r$);
  /*Compute niche count of reference point $j \in Z^r$ */
  $\rho_j \leftarrow \sum_{s \in S_t / F_l} ((\pi(s) = j)\ ?\ 1 : 0)$;
  /*Choose $K$ members one at a time from $F_l$ to construct $P_{t+1}$*/
  Niching ($K$, $\rho_j$, $\pi(s)$, $d(s)$, $Z^r$, $F_l$, $P_{t+1}$);
End If
End
```

Fig. 1 Pseudo-algorithm of NSGA-III

one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count ρ_{j*} is then incremented by one. Second, the front F_l does not have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $\rho_{j*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count ρ_{j*} is incremented by one. After ρ_j counts are updated, the procedure is repeated for a total of K times to increase the population size of P_{t+1} to N.

3.2 Problem formulation

The refactoring problem involves searching for the best refactoring solution among the set of candidate ones, which constitutes a huge search space. A refactoring solution is a sequence of

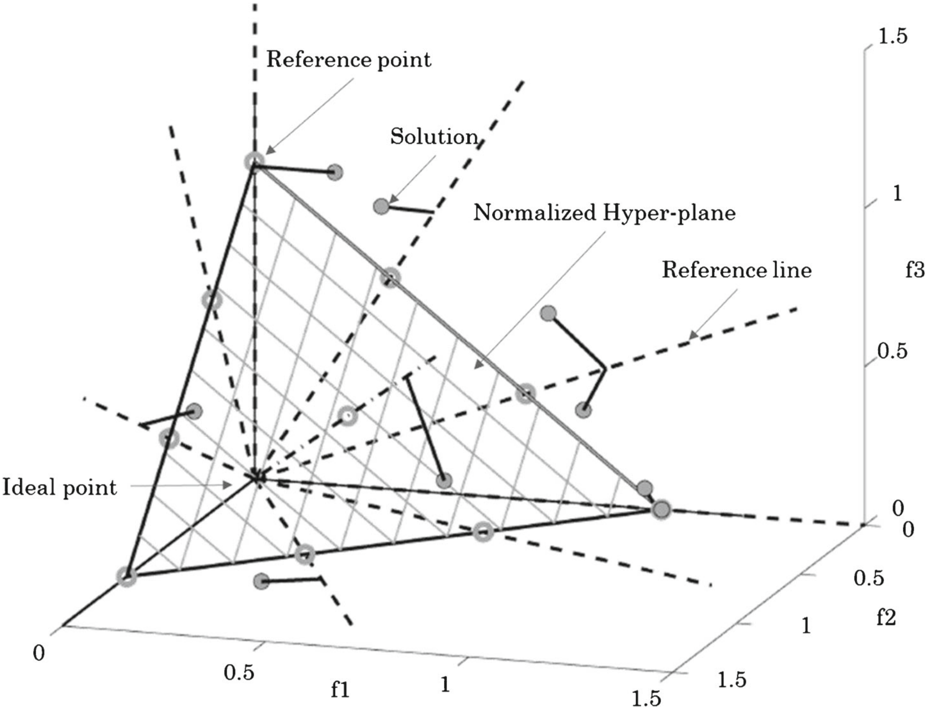


Fig. 2 Normalized reference plane for a three-objective case (Deb and Jain 2013)

refactoring operations where the goal of applying the sequence to a software system *S* is typically to fix maintenance issues in S. Usually in SBSE approaches, we use two or three metrics as objective functions for a particular multi-objective heuristic algorithm to find these design issues and correct them. In reality, we assume that increasing the number of metrics to optimize may increase the quality of the refactored code. However, the high number of suggested solutions in a 16-objective Pareto-Front quickly exceeds the developer's ability to manually choose between them. It has been known that developers most likely want a unique optimal solution that better satisfies his/her preferences that can be easily expressed in terms of quality objectives. Motivated by this observation, we propose in this research work to consider the six objectives of the QMOOD model where each represents a separate objective function along with two other objectives to reduce the number of refactorings to apply and maximize the design coherence after refactoring. In this way, we obtain a many-objective (8-objective) formulation of the refactoring problem that could not be solved using standard multi-objective approaches. This formulation is given as follows:

$$\begin{array}{l} Maximize \quad F(x,S) = [f_1(x,S), f_2(x,S), \dots, f_8(x,S)] \\ subject\ \ to \quad x = (x_1, \dots, x_n) \in X \end{array}$$

where X is the set of all legal refactoring sequences starting from S, xi is the ith refactoring operation, and f_k(x,S) is the kth objective.

The refactoring operations studied in this extension are described in Table 3.

The concern about using these operations is whether each one of them have a positive impact on the refactored code quality. In this context, previous work has studied the impact

Table 3 Refactoring types and their involved actors and roles

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

analysis of refactoring operations on internal and external quality metrics. Du Bois & Mens (2003) proposed the evaluation of a selected set of refactorings based on their impact on the internal CK quality metrics. They extended their work by including more studied operations while enhancing cohesion and coupling measures (Du Bois et al. 2004). They provided guidelines to distinguish between operations that optimize software quality and ruled out those which their application will increase coupling or decrease cohesion. Similarly, Alshayeb (Alshayeb 2009) has quantitatively assessed, using internal metrics, the effect of refactorings on different quality attributes to help developers in the estimation of refactoring effort in terms of the cost and time.

In the following, we will describe in details the different objectives considered in our formulation.

3.2.1 QMOOD model quality attributes as objectives

Many studies has been utilizing structural metrics as basis of defining quality indicators for a good system design (Chidamber and Kemerer 1994; Abreu 1995; Lorenz and Kidd 1994). As an illustrative example, Bansiya & Davis (2002) proposed a set of quality measures, using the ISO 9126 specification, called QMOOD. Each of these quality metrics is defined using a combination of high-level metrics detailed in Table 4.

Refactoring operations change the internal design of the system, and eventually the high level metrics values will change accordingly. To have a better understanding of the refactorings impact, external quality factors have been defined using the high level metrics. They provide

Table 4 QMOOD metrics description (Bansiya and Davis 2002)

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

meaningful measures to the system enhancement. These external quality factors are enumerated in the following Table 5.

The adaptation of the QMOOD model in the problem formularization has provided six quality attributes as separate objectives: *reusability*, *flexibility*, *understandability*, *functionality*, *extendibility* and *effectiveness*.

3.2.2 Number of code changes as an objective

It is known that multiple refactoring sequences may have a completely different set of operations which their execution will give two different resulting designs but they might have the same quality. So, the execution of a specific suggested refactoring sequence may require an effort that is comparable to the one of re-implementing part of the system from scratch. Taking this observation into account, it is trivial to minimize the number of suggested operations in the refactoring solution since the designer can have some preferences regarding the percentage of deviation with the initial program design. In addition, most developers prefer solutions that minimize the number of changes applied to their design (Kessentini et al. 2011). Thus, we formally defined the fitness function as the number of refactoring operations (size of the solution) to be minimized:$f_7(x)=Size(x)$ where x is the solution to evaluate.

The different code changes (refactoring types) used in our approach may have different impacts on the maintainability/quality objectives considered in our formulation. We show in

Table 5 Quality attributes and their computation equations (Bansiya and Davis 2002).

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

the following that the different quality objectives are conflicting since the different refactoring types considered in our approach may decrease some quality attributes and increase some others. In Shatnawi & Li (2011), the impact analysis of some of Fowler's catalog operations on some external quality factors (effectiveness, flexibility, extendibility and reusability) has shown that not all refactorings necessarily improve these quality factors. Rules of thumb have been established using heuristics to dictate which refactorings to use in order to enhance a given quality attribute. The following table has been extracted from (Shatnawi and Li 2011) to show the impact analysis of the refactorings applied on restructuring EclipseIU 2.1.3 and Struts (1.1 and 1.2.4).

Based on Table 6, the operations have various implications on the internal metrics that can reach the degree of conflict. The composition of these metrics values can be an indicator of

Table 6 Refactorings impact analysis on QMOOD internal metrics.

Refactoring Operation	DSC	NOH	ANA	DAM	DCC	CAMC	MOA	MFA	NOP	CIS	NOM
Extract class	+	0	0	0	+	+	+	0	0	0	0
Extract interface	+	0	+	0	0	0	0	+	+	0	0
Inline class	-	0	0	0	-	−	−	0	0	0	0
Move field	0	0	0	0	0	+	0	0	0	0	0
Move method	0	0	0	0	−	+	0	0	0	−	0
Push down field	0	0	0	0	0	+	0	0	0	0	0
Push down method	0	0	0	0	+	0	0	+	+	+	0
Pull up field	0	0	0	0	0	−	0	0	0	0	0
Pull up method	0	0	0	0	−	0	0	−	−	−	0

how the external quality attributes will be affected. Table 7 shows the resulting impact analysis on QMOOD quality attributes and the potential conflict between them.

Although the statistical significance of the reported results in Table 7 was not studied in this work and kept as part of our future investigations, it does not affect our problem formulation, in fact, if the degree of conflict between two objectives is not considerable for a random set of refactorings, this degrades the heuristics' performance by increasing the computational time of the heuristics but it does not affect the quality of the results.

In general, the related work has given the following observations: (1) Not all refactoring operations have a desired impact on internal and external quality attributes. (2) It is difficult to theoretically come up with an optimal set of corrections to increase a chosen quality attribute without decreasing another. (3) A goal-oriented process has been given to include or/and exclude refactorings based on developer's preferred quality attribute.

These limitations motivated our formulation that (1) is not limited to a specific types of refactorings and that (2) is not limited to optimizing a preferred quality attribute with disregard to the others.

3.2.3 Design coherence preservation as an objective

It is usually the designer's responsibility to manually inspect the feasibility of the suggested refactorings and evaluate their meaningfulness from the design coherence perspective. Sometimes, the new refactored design may be structurally improved but introduce several design incoherence. To preserve the semantics design, we present two main formulated different measures in which we describe the following sections. The fitness function is formulated as an average of the two following measures.

A. *Vocabulary-based similarity* (*VS*)

This kind of similarity should be eventually considered when moving methods, or attributes between classes. For instance, when moving a method or an attribute from one source class to another destination class, this operation would make sense if both source and target classes have similar vocabularies. In this case, it is assumed that the vocabulary of naming the code elements in classes is reflecting a specific domain terminology. That's why, two code elements could be semantically similar if they use similar vocabularies (Ouni et al. 2012a, b; Dean et al. 1995; Kim et al. 2010).

Table 7 Refactorings impact analysis on QMOOD quality attributes

Refactoring Operation	Reusability	Flexibility	Understandability	Functionality	Extendibility	Effectiveness
Extract class	+	+	−	+	−	+
Extract interface	+	+	−	+	+	+
Inline class	−	−	+	−	+	−
Move field	+	0	+	+	0	0
Move method	0	+	+	−	+	0
Push down field	+	0	+	+	0	0
Push down method	+	+	−	+	+	+
Pull up field	−	0	−	−	0	0
Pull up method	−	−	+	−	−	−

A token-based extraction (Corazza et al. 2012) of vocabulary is performed on the naming of classes, methods, attributes, parameters and comments. This Tokenization process is widely used in code clones detection techniques and it is known to be more robust to code changes compared to text-based approaches. The semantic similarity is calculated based on information retrieval-based techniques (e.g., cosine similarity). The following equation calculates the cosine similarity between two classes. Each actor is represented as an n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two classes, c1 and c2, is determined as follows:

$$Sim(c1, c2) = \cos\left(\overrightarrow{c1}, \overrightarrow{c2}\right) = \frac{\overrightarrow{c1}.\overrightarrow{c2}}{\left\|\overrightarrow{c1}\right\| * \left\|\overrightarrow{c2}\right\|} = \frac{\sum_{i=1}^{n} (wi, 1*wi, 2)}{\sqrt{\sum_{i=1}^{n} (wi, 1)^2}\sqrt{\sum_{i=1}^{n} (wi, 2)^2}} \in [0, 1] \quad (2)$$

where $\overrightarrow{c1} = \left(w_{1,1}, \dots, w_{n,1}\right)$ and $\overrightarrow{c2} = \left(w_{1,2}, \dots, w_{n,2}\right)$ are respectively two vectors corresponding to c1 and c2 The weights $w_{i,j}$ are automatically generated by information retrieval-based techniques such as the Term Frequency – Inverse Term Frequency (TF-IDF) method. We used a method similar to that described in (Corazza et al. 2012) to determine the vocabulary and represent the classes as term vectors.

B. *Dependency-based similarity* (*DS*)

Similarly to vocabulary similarity, the semantic closeness can be also extracted from mutual dependencies. In general, a high coupling (i.e., multiple call in and call out) between two classes is usually not recommended, and if it exists, developers are usually prompted to merge them to reduce the design complexity, this also hints that these two classes are semantically close. Intuitively, the application of refactoring on highly dependent classes is not only beneficial to the design quality, but also has a higher probability to eventually be meaningful. To follow up with dependency, we point out two types of dependency links:

1) *Shared Method Calls* (*SMC*) that can be easily detected through the application call graphs using the Class Hierarchy Analysis (CHA) (Dean et al. 1995). Methods are modeled as graphs while calls represent the edges between nodes. A call graph can either be a call in or call out. This technique is applied for each couple of classes, shared calls are being detected through the graph by identifying shared neighbors of nodes related to each actor. Shared call-in and shared call-out are distinguished and separately calculated for a given couple c1 and c2 (i.e., two classes) using the following equations.

$$\mathrm{sharedCallOut}(\mathrm{c1},\ \mathrm{c2}) = \frac{|\mathrm{callOut}(\mathrm{c1}) \cap \mathrm{callOut}(\mathrm{c2})|}{|\mathrm{callOut}(\mathrm{c1}) \cup \mathrm{callOut}(\mathrm{c2})|} \in [0, 1] \qquad (3)$$

$$\mathrm{sharedCallIn}(\mathrm{c1},\ \mathrm{c2}) = \frac{|\mathrm{callIn}(\mathrm{c1}) \cap \mathrm{callIn}(\mathrm{c2})|}{|\mathrm{callIn}(\mathrm{c1}) \cup \mathrm{callIn}(\mathrm{c2})|} \in [0, 1] \qquad (4)$$

2) *Shared field access* (*SFA*) is also known as data coupling and occurs when a class refers to another as a type, or shares a method that references another class as a parameter type.

Static analysis is adopted to view occurrences of possible invocation of calls of field accesses through methods or constructors. Two classes have a high shared field access rate if they read or modify the same fields belonging to one or both of them. This violation of the principle of modularity can be, for example, fixed by either merging these two classes. In this context, the rate of shared field access is used as an indicator to semantic closeness between two classes c1 and c2, and it is calculated according to the following equation.

$$\mathrm{sharedFieldsRW}(c1,\ c2)\ = \frac{|\mathrm{fieldRW}(c1) \cap \mathrm{fieldRW}(c2)|}{|\mathrm{fieldRW}(c1) \cup \mathrm{fieldRW}(c2)|} \in [0, 1] \tag{5}$$

where fieldRW(ci) refers to the number of fields that may be read or write by each method of the module ci.

To illustrate the dependency similarity measure, let us take the example of two classes A and B with no direct calls between them, if a third class C calls both of them, then the *callIn*(*A*) inter *callIn*(*B*) will be incremented, the intersection between *callIns* determines the number of classes that both call these two classes. It is divided by the overall number of *callIns* received by these two classes. Similarly for the *callsOut* which informs about the number of common entities called by two given classes. For the shared field access, the idea is similar, even if sharing attributes is a bad practice indeed, if it exists, then it creates a dependency between the class sharing the attribute and the other classes accessing it, we use this as an indicator of semantic closeness between them.

3.3 Solution approach

3.3.1 Solution representation

As defined in the previous section, a solution consists of a sequence of *n* refactoring operations applied to different code elements in the source code to fix. In order to represent a candidate solution (individual/chromosome), we use a vector-based representation. Each vector's dimension represents a refactoring operation where the order of applying these refactoring operations corresponds to their positions in the vector. For each of these refactoring operations, we specify pre- and post-conditions in the style of Fowler et al. (1999) to ensure the feasibility of their application. The initial population is generated by assigning randomly a sequence of refactorings to some code fragments. To apply a refactoring operation we need to specify which classes, i.e., code fragments, are involved/impacted by this refactoring and which roles they play in performing the refactoring operation. An actor can be a package, class, field or method. The list of refactoring operations along with their appropriate actors is listed in Table.

3.3.2 Solution variation

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure the respect of the length limits (size of the solution is limited to up-to 500 refactorings in our experiments) by eliminating randomly some refactoring operations. It is important to note that in many-objective optimization, it is better to create children that are close to their parents in order have a more efficient search process (Deb and Jain 2013). For this reason, we control the cutting point of the one-point crossover operator by restricting its position to be either belonging to the first tier of the refactoring sequence or belonging to the last tier. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings.

3.3.3 Solution evaluation

Each generated refactoring solution is executed on the system *S*. Once all required data is computed, the solution is evaluated based on the quality of the resulting design (the six quality attribute objectives to be maximized), along with the aggregation of semantics similarity functions (to be maximized) followed by the complexity of the refactoring operations (to be minimized). These values are the coordinates of the solution in the objectives space and so, it is assigned a non-domination rank as well as a particular reference point which is the closest to the solution.

3.3.4 Normalization of population members

Duo to the heterogeneous nature of objective functions (i.e., they have different ranges of values), we used the normalization procedure proposed by Deb & Jain (2013) to circumvent this problem. At each generation, the minimal and maximal values for each metric are recorded and then used by the normalization procedure to calculate respectively the Nadir and ideal points (Bechikh et al. 2010). Normalization allows the population members and the reference points to have the same range, which is a pre-requisite for diversity preservation.

3.3.5 Final solution selection

Once the final Pareto front has been generated, for each fitness function, the reference sets are calculated from the union of all Pareto approximations which are being normalized with respect to ideal and Nadir points. The choice of one individual among the large set of Pareto-optimal solutions is not trivial. The preference of the developer can be then used to determine which of the solution may better satisfy his/her needs. Since the objectives have been defined, in terms of quality attributes, along with minimizing the size and maximizing the semantic coherence, it is easier for the developer to specify a ranking that can be used as input to the hyperplane and consider only reference points that match (or closest to) the rank. This will reduce drastically the number of preferred reference points. Still, if few solutions satisfy such condition, the niche point selection will sort them and send the top of the queue as output. In case of absence of developer's input, we choose the nearest solution to the Knee point (Bechikh et al. 2011) (i.e., the vector composed of the best objective values among the population members in all iterations).

4 Design of the empirical study

The goal of the study is to evaluate the usefulness our many-objective refactoring tool in practice. We conducted experiments on popular open source systems and one industrial project using the Goal, Question, Metrics (GQM) assessment approach (Basili 1992). This Section is organized as follows. Section 4.1 poses three research questions that drive our experiments whose settings have been detailed in Section 4.2: Section 4.2.1 presents the selected software systems. Section 4.2.2 indicates the tuning of the various heuristics used. Section 4.2.3 describes the statistical tests details. Finally, Section 4.2.4 and 4.2.5 are dedicated to subjects and scenarios that constitute the non-subjective evaluation conducted with potential developers who can use our tool.

4.1 Research questions

In our study, we assess the performance of our refactoring approach by finding out whether it could generate meaningful sequences of operations that improve the quality of the systems while reducing the number of code changes and preserving the semantic coherence of the design. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful refactoring solutions that can help developers to improve the quality of their systems. To find an answer, we defined the following three research questions:

> **RQ1**: To what extent can the proposed approach improve the quality of the evaluated systems?
> **RQ2**: How does the proposed many-objective approach based on NSGA-III perform compared to other many/multi-objective algorithms or a mono-objective approach for software refactoring and to an existing approach that is not based on heuristic search?
> **RQ3**: How our many-objective refactoring approach can be useful for software engineers in real-world setting?

One of the challenges in SBSE is to find the most suitable search algorithm for a specific software engineering problem. The only proof is the experimental results, thus it is important to address this question when designing a new software engineering problem as an optimization problem. In addition, a comparison with a mono-objective algorithm may justify the need to use a many-objective approach to show that the different objectives are really conflicting and cannot be merged into one fitness function. It is maybe also not sufficient to show that the proposed many-objective formulation outperforms others search algorithms, thus it is important to compare with a non-search-based approach to confirm the practical value of the proposed search-based approach.

To answer **RQ1**, we validate the proposed refactoring technique on seven open-source systems and one industrial project to evaluate the quality improvements of systems after applying the suggested refactoring solution. We calculate the overall *Quality Gain* (QG) for the six quality attributes as follows: Let Q={q1, q2,… q6} and Q′={q′1, q′2,… q′6} be respectively the set of quality attribute values before and after applying the suggested refactorings, and {w1, w2,… w6} the weights assigned to each of these quality factors. Then the QG is calculated by:

$$QG = \sum_{i=1}^{6} wi^{*}(q'i - qi) \qquad (6)$$

In addition, we validate the proposed refactoring operations to fix code smells by calculating the *Defect Correction Ratio* (DCR) which is given by the following equation and corresponds to the ratio of the number of corrected design defects to the initial number of detected defects before applying the suggested refactoring solution. The code smells were collected using existing detection tools DÉCOR (Moha et al. 2009) and InCode (Marinescu et al. 2010).

$$DCR = \frac{\left|\text{Corrected Defects Instances}\right|}{|\text{All Defects Instances}|} \in [0, 1] \tag{7}$$

Since it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives, we use two different validation methods: manual validation and automatic validation of the efficiency of the proposed solutions. For the manual validation, we asked groups of potential users (software engineers) of our refactoring tool to evaluate, manually, whether the suggested operations are feasible, make sense semantically. We define the metric *Manual Precision* (MP) which corresponds to the number of semantically coherent operations over the total number of suggested operations. MP is given by the following equation

$$MP = \frac{\left|\text{Coherent Operations}\right|}{|\text{Suggested Operations}|} \in [0, 1] \tag{8}$$

For the automatic validation we compared the proposed refactorings with the expected ones using the different systems in terms of recall and precision. The expected refactorings are those applied by the software development team to the next software release. To collect these expected refactorings, we use Ref-Finder (Kim et al. 2010), an Eclipse plug-in designed to detect refactorings between two program versions. Ref-Finder allows us to detect the list of refactorings applied to the current version of a system (see Table 10):

$$\text{RE}\mathit{recall} = \frac{\left|\text{suggested operations}\right| \cap \left|\text{expected operations}\right|}{\left|\text{expected operations}\right|} \in [0, 1] \tag{9}$$

$$\text{PR}\mathit{precesion} = \frac{\left|\text{suggested operations}\right| \cap \left|\text{expected operations}\right|}{\left|\text{suggested operations}\right|} \in [0, 1] \tag{10}$$

To answer **RQ2**, we compared the performance of NSGA-III based approach with four many-objective techniques, Gr-EA (Yang et al. 2013), DBEA-Eps (Asafuddoula et al. 2013), IBEA (Zitzler and Künzli 2004) and MOEA/D (Zhang and Li 2007), an existing work based on a multi-objective NSGA-II algorithm (Ouni et al. 2012a) and also a mono-objective evolutionary algorithm (Kessentini et al. 2011). The approaches are briefly introduced as follows:

Grid-based Evolutionary Algorithm (Gr-EA) partitions the search space into grids (also called hypercubes). The number of divisions in Gr-EA is a parameter. Then, it recombines them based on the current objective values in the population. Just like in dominance-based algorithms, it also ranks the population by fronts but the crowding distance and the spread of solutions are calculated from grid-based metrics.

Decomposition Based Evolutionary Algorithm with Epsilon Sampling (DBEA-Eps) is another decomposition-based EA with a variation in the decomposition method which generates reference points via systematic sampling and deals with constraint by an adaptive epsilon scheme to manage balance between convergence and diversity.

Indicator-Based Evolutionary Algorithm (IBEA) is distinguished among other EAs by its continuous dominance criteria where each solution is assigned a weight that is calculated from quality indicators, usually given by the user.

Multiobjective Evolutionary Algorithm Based on Decomposition (MOEA/D) simultaneously performs an optimization of previously decomposed sub-problems, according to their neighborhood information. MOEA/D assigns a weight vector to every individual in the population, each one being focused on the resolution of the sub-problem represented by its weight vector. The solutions evaluation is done by the Tchebycheff approach and by computing their distance to a reference point.

The comparison between these many-objective algorithms is performed in terms of convergence of the Pareto Front and with respect to the diversity of the obtained solutions. In order to estimate the convergence and diversity, we used the Inverted Generational Distance (IGD), which is the sum of distances from each point of the true Pareto front to the nearest point of the non-dominated set found by the algorithm in all iterations, the lower the IGD value, the better the approximation is. Since both indicators measure the convergence and spread of the obtained set of solutions, we will only use the IGD as performance indicator and it will be statistically analyzed to assess the significance of results.

As part of our experiments, to demonstrate the importance of taking each quality attribute as separate objective instead of simply aggregating them into a single fitness function, a comparison with a mono-objective approach that aggregates several quality attributes in one objective is required. The comparison between a many-objective algorithms with a mono-objective one is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. In order to cope with this situation, for each many-objective algorithm the nearest solution to the Knee point is selected as a candidate solution to be compared with the single solution return by the mono-objective algorithm. We compared NSGA-III with an existing mono-objective refactoring approach (Kessentini et al. 2011) based on the use of QMOOD quality attributes aggregated in one fitness function.

We compared our proposal to the popular design defects detection and correction tool JDeodorant (Tsantalis et al. 2008) that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them.

For **RQ3**, we evaluated the benefits of our refactoring tool by several software engineers. To this end, they classify the suggested refactorings (*IR*) one by one as interesting or not. The difference with the *MP* metric is that the operations are not classified from a semantic coherence perspective but form a usefulness one.

$$IR = \frac{\left|\text{Useful Operations}\right|}{|\text{Suggested Operations}|} \in [0, 1]$$

To answer the above research questions, we selected the solution from the set of non-dominated ones providing the maximum trade-off using the following strategy when comparing between the different algorithms (expect the mono-objective algorithm where we select the solution with the highest fitness function). In order to find the maximal trade-off solution of the multi-objective or many-objective algorithm, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan (2009) to evaluate the worthiness of each non-dominated solution in terms of compromise between the objectives. This metric is expressed as follows:

$$\mu(x_i, S) = \underset{x_j \in S, x_i \nprec x_j, x_j \nprec x_i}{Min} T\left(x_i, x_j\right) \tag{12}$$

where

$$T\left(x_i, x_j\right) = \frac{\sum_{m=1}^{M} \max\left[0, \frac{f_m\left(x_j\right) - f_m\left(x_i\right)}{f_m^{\max} - f_m^{\min}}\right]}{\sum_{m=1}^{M} \max\left[0, \frac{f_m\left(x_i\right) - f_m\left(x_j\right)}{f_m^{\max} - f_m^{\min}}\right]}$$

We note that x_j denotes members of the set of non-dominated solutions S that are non-dominated with respect to x_i. The quantity $\mu(x_i,S)$ expresses the least amount of improvement per unit deterioration by substituting any alternative x_j from S with x_i. We note also that $f_m(x_i)$ corresponds to the m^{th} objective value of solution x_i and $f_m^{\max}/f_m^{\min}$ corresponds to the maximal/minimal value of the m^{th} objective in the population individuals. In the above equations, normalization is performed in order to prevent some objectives being predominant over others since objectives are usually incommensurable in real world applications. In the last equation, the numerator expresses the aggregated improvement gained by substituting x_j with x_i. However, the denominator evaluates the deterioration generated by the substitution (Table 8).

4.2 Experimental setting

4.2.1 Software systems

We used a set of well-known open-source java projects that have been investigated in our previous work (Mkaouer et al. 2013) and one project from our industrial partner Ford Motor Company. We applied our approach to the following open-source java projects: ArgoUML v0.26, Xerces v2.7, ArgoUML v0.3, Ant-Apache v1.5, Ant-Apache v1.7.0, Gantt v1.10.2 and

Table 8 Summary of the empirical study design

Research questions	Metrics and measurements
Quality improvement of refactored systems?	Total Quality Gain (QG) (Fig. 3) Defect Correction Ratio (DCR) (Fig. 4) Manual Precision (MP), Precision (PR) and Recall (RE) (Fig. 5)
NSGA-III performance compared to other mono/many/multi-objective algorithms and a non-search-based approach?	The Inverted Generational Distance (IGD) (Fig. 6) Computational Time (CT) (Fig. 7) Total Quality Gain (QG) (Fig. 8) Manual Precision (MP), Precision (PR) and Recall (RE) (Fig. 9) Average Number of Suggested Refactorings (Fig. 10)
Usefulness of the refactoring approach in real-world setting?	Useful Refactorings (IR) (Fig. 11)

Azureus v2.3.0.6. Xerces-J is a family of software packages for parsing XML. ArgoUML is a Java open source UML tool that provides cognitive support for object-oriented design. Apache Ant is a build tool and library specifically conceived for Java applications. GanttProject is a cross-platform tool for project scheduling. Azureus is a Java BitTorrent client for handling multiple torrents. We also considered in our experiments an industrial project, JDI, provided by our industrial partner the Ford Motor Company. It is Java-based software system that helps Ford Motor Company analyze useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is main key software application used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to the expectations of customers. JDI is a highly structured and several versions were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to ensure good refactoring solutions of JDI to reduce the time required by developers to introduce new features in the future and understand existing implementations.

We selected these systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years and because their defects are known and were subject of various previous studies (Moha et al. 2009; Ouni et al. 2012a; Palomba et al. 2013). Table 9 provides some descriptive statistics about these projects.

To collect operations applied in previous program versions, we used Ref-Finder. Table 10 shows the analyzed versions and the number of operations, identified by Ref-Finder, between each subsequent couple of analyzed versions, after the manual validation.

4.2.2 Parameter tuning

The algorithms have been configured according to the parameters detailed in Table 11. Different values have been used for the population size and the maximum number of evaluations, generating a variety of configurations related the projects sizes and the number of objectives. For the mono-objective EA, we adopted the same approach using best fitness value criterion since multi-objective metrics cannot be used for single-objective algorithms.

4.2.3 Statistical tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is

Table 9 Statistics of the studied systems

Systems	Release	classes	KLOC	Code smells
Xerces-J	v2.7.0	991	240	82
Azureus	v2.3.0.6	1449	264	108
ArgoUML	v0.26	1358	283	1358
ArgoUML	v0.3	1409	271	1409
Ant-Apache	v1.5.0	1024	266	103
Ant-Apache	v1.7.0	1839	294	124
GanttProject	v1.10.2	245	81	41
JDI-Ford	v5.8	638	247	88

Table 10 Analyzed versions and operations collection

Systems	Collected operation	
	Previous releases	Refactorings
Xerces-J	v1.4.2 - v2.6.1	52
GanttProject	v1.7 - v1.10.1	113
Azureus	v2.1.0.0- v2.3.0.0	146
ArgoUML	v0.11.4 - v0.17.2	182
Ant-Apache	v1.1.0- v1.4.0	177
JDI-Ford	v2.4 – v5.6	97

performed based on 31 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test (Arcuri and Fraser 2013) with a 95 % confidence level (α=5 %). The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α ($\leq$0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. For example, we compute the p-value obtained by comparing NSGA-II, IBEA, MOEA/D and mono-objective search results with NSGA-III ones. In this way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

4.2.4 Subjects

Our study involved 11 subjects from the University of Michigan and 5 software engineers from Ford Motor Company. Subjects include 6 master students in Software Engineering, 4

Table 11 Parameters configuration

Global parameters	
2003Population Size	190
Objectives	8
Max Evaluations	1400
Crossover Weight	0.8
Mutation Weight	0.2
Reference Points	156
NSGA-III/Gr-EA parameters	
Number of Divisions	4
IBEA parameters	
Archive Size	100
MOEA/D parameters	
Neighborhood Size	8
Max Replacements	2
H	99

PhD students in Software Engineering, 1 faculty member in Software Engineering, and 5 junior software developers. 3 of them are females and 8 are males. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 2 to 14 years. The evaluated solutions by the subjects are those that represent the maximum trade-off between the objectives using the trade-off worthiness metric proposed by Rachmawati as described in the previous section.

4.2.5 Scenarios

We designed our study to answer our research questions. The subjects were invited to fill a questionnaire that aims to evaluate the suggested refactorings. We divided the subjects into 8 groups according to 1) the number of studied systems (Table 9), 2) the number of refactoring solutions to evaluate, and 3) the number of techniques to be tested.

As shown in Table 12, for each system, several solutions have to be evaluated. In Table 12, we summarize how we divided subjects into 8 groups. In addition, as illustrated in Table 12, we are using a cross-validation to reduce the impact of subjects on the evaluation. Each subject evaluates different refactoring solutions for different systems.

Subjects were first asked to fill out a pre-study questionnaire containing seven questions. The questionnaire helped to collect background information such as their role within the

Table 12 Survey organization

Subject groups	Systems	Algorithms / Approaches
Group 1	Xerces-J v2.7.0	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	ArgoUML v0.26	MOEA/D / NSGA-II / GA
	Ant-Apach v1.5.0	DBEA-Eps / IBEA / NSGA-II / GA
Group 2	Azureus v2.3.0.6	MOEA/D / JDeodorant / Gr-EA
	ArgoUML v0.3	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	Ant-Apache v1.7.0	Gr-EA / DBEA-Eps / IBEA / JDeodorant
Group 3	GanttProject v1.10.2	Gr-EA / DBEA-Eps / IBEA / JDeodorant
	Xerces-J v2.7.0	MOEA/D / NSGA-II / GA
	ArgoUML v0.26	Gr-EA / DBEA-Eps / IBEA / JDeodorant
Group 4	Ant-Apach v1.5.0	MOEA/D / JDeodorant / Gr-EA
	Azureus v2.3.0.6	DBEA-Eps / IBEA / NSGA-II / GA
	ArgoUML v0.3	MOEA/D / NSGA-II / GA
Group 5	Ant-Apache v1.7.0	MOEA/D / NSGA-II / GA
	GanttProject v1.10.2	MOEA/D / NSGA-II / GA
	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D
Group 6	ArgoUML v0.3	DBEA-Eps / IBEA / NSGA-II / GA
	Ant-Apache v1.7.0	DBEA-Eps / IBEA / NSGA-II / GA
	JDI-Ford v5.8	NSGA-II / GA / JDeodorant
Group 7	GanttProject v1.10.2	DBEA-Eps / IBEA / NSGA-II /GA
	Xerces-J v2.7.0	DBEA-Eps / IBEA / NSGA-II / GA
	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D
Group 8	JDI-Ford v5.8	Gr-EA / DBEA-Eps / IBEA / MOEA/D / NSGA-II / GA / JDeodorant

company, their programming experience, their familiarity with software refactoring. In addition, all participants attended one lecture of 50 min about software refactoring and passed 10 tests to evaluate their performance to evaluate and suggest refactoring solutions. Then, the groups are formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. Group 8 is composed by only software engineers from Ford and evaluated only refactoring suggestions for JDI-Ford.

The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc and one PhD student). In addition, our experiments are not only limited to the manual validation but also the automatic validation can verify the effectiveness of our approach.

Subjects were aware that they are going to evaluate the design coherence and the usefulness of the suggested refactorings, but do not know the particular experiment research questions (algorithms used, different objectives used and their combinations). Consequently, each group of subjects who accepted to participate to the study, received an online questionnaire, a manuscript guide to help them to fill the questionnaire, and the source code of the studied systems, in order to evaluate the solutions. The questionnaire is organized in an excel file with hyperlinks to visualize easily the source code of the affected code elements. Subjects are invited to select for each operation one of the possibilities: "*Yes*", "*No*", or "*May be*" (if not sure) about the design coherence and usefulness. Since the application of refactroing solutions is a subjective process, it is normal that not all the programmers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not.

5 Results and discussions

5.1 Results for RQ1

Figure 3 summarizes the results of median values of the quality improvement metrics over 31 independent simulation runs after applying the proposed operations by the refactoring solution selected using the knee-point strategy (Bechikh et al. 2011). In our experiments, we used all the 8 objectives in our many-objective formulation. It is clear from Fig. 3 that all the six quality

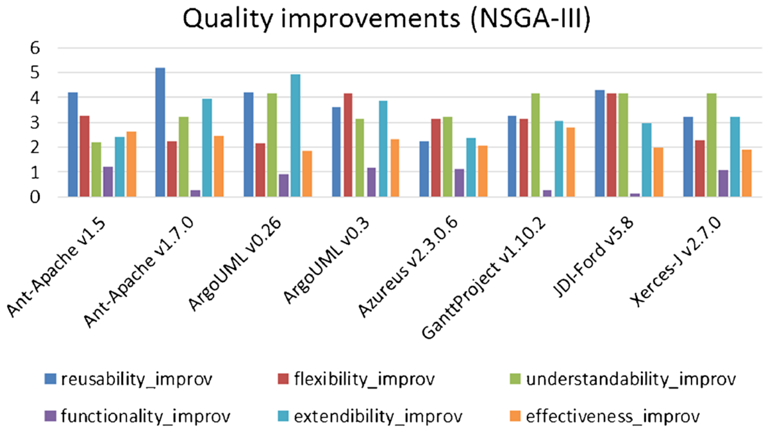


Fig. 3 Average quality improvements, over 31 runs, on the different systems using NSGA-III

objectives are improved using our NSGA-III algorithm compared to the program version before refactoring. The reusability, understandability and extendibility are the most improved metrics and this can be explained by the fact that refactoring is not expected to change a lot the behavior/functionality of a system and this explain that some objectives were not improved significantly such as the functionality improvements metric. The same observation regarding the behavior preservation is valid for the extendibility factor because it is, to some extent, a subjective quality factor and using a model of merely static measures to evaluate extendibility is may be not very good estimator. Overall, the NSGA-III algorithm was able to find a good trade-off between all the six quality objectives since most of them were significantly increased and no one of these metrics were decreased comparing the initial version of the system before refactoring. The variation in terms of quality improvements between the different systems is not high. ArgoUML and Ant-Apache were the main systems who are significantly improved and this can be explained by the lower quality of these projects comparing to the remaining systems before refactoring.

As described in Fig. 4, after applying the proposed refactoring operations by our approach (NSGA-III), we found that, on average, 82 % of the detected defects were fixed (DCR) for all the eight studied systems. This high score is considered significant in terms of improving the quality of the refactored systems by fixing the majority of defects of various types (blob, spaghetti code, functional decomposition (Moha et al. 2009), god class, data class and feature envy (Marinescu et al. 2010)).

A closer look to the fixed defects in Xerces-J v2.7.0, the one with the highest percentage of fixed defects (86 %), is detailed in Table 13.

On Table 13, it is noticeable that some code smells are harder to fix (such as God classes) compared to others (Feature Envy), further analysis needs to be done to better understand this observation. Although the bad smell detection literature suggests a wide variety of code smells to be corrected, we narrowed our selection to the five types that have given significant results compared to the others. This can be explained by the fact that the defects types that are not fixed require the considerations of more refactoring operations rather than those included in this work. In addition, some of these defects are difficult to detect just using structural metrics (Palomba et al. 2013).

We also need to assess the correctness/meaningfulness of the suggested refactorings from the developers' point of view. Figure 5 confirms that the majority of the suggested refactorings improve significantly the code quality while preserving design's semantic coherence. On average, for all of our studied systems, an average of around 91 % of proposed refactoring

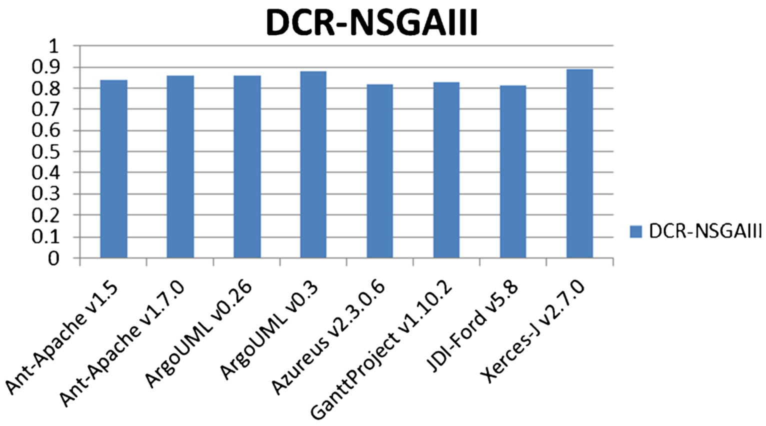


Fig. 4 Average percentage of fixed defects, over 31 runs, on the different systems using NSGA-III

Table 13 Fixed code smells distribution in Xerces-J v2.7.0

Code smell	Flawed classes	Number of fixed code smells
Blob	143 (32 % overlap)	30 (%89)
Data Class		19 (%83)
God Class		10 (%64)
Feature Envy		25 (%96)
Functional Decomposition		13 (%94)
Spaghetti Code		39 (%92)

operations are considered by potential users to be semantically feasible and do not generate semantic incoherence.

In addition to the manual evaluation, we automatically evaluate our approach without using the feedback of potential users to give more quantitative evaluation to answer RQ1. Thus, we compare the proposed refactorings with the expected ones. The expected refactorings are those applied by the software development team to the next software release as described in Table 10. We use Ref-Finder to identify refactoring operations that are applied between the program version under analysis and the next version. Figure 5 summarizes our results. We found that a considerable number of proposed refactorings (an average of 59 % for all studied systems in terms of recall) are already applied to the next version by software development team which is considered as a good recommendation score, especially that not all refactorings applied to next version are related to quality improvement, but also to add new functionalities, increase security, fix bugs, etc.

To conclude, we found that our approach produces good refactoring suggestions in terms of defect-correction ratio, semantic coherence from the point of view of 1) potential users of our refactoring tool and 2) expected refactorings applied to the next program version.

5.2 Results for RQ2

In this section, we focus first on the comparison between our NSGA-III adaption and other many-objective algorithms using the same adaptation. Table 14 shows the median IGD values over 31 independent runs for all algorithms under comparison. We have used pairwise

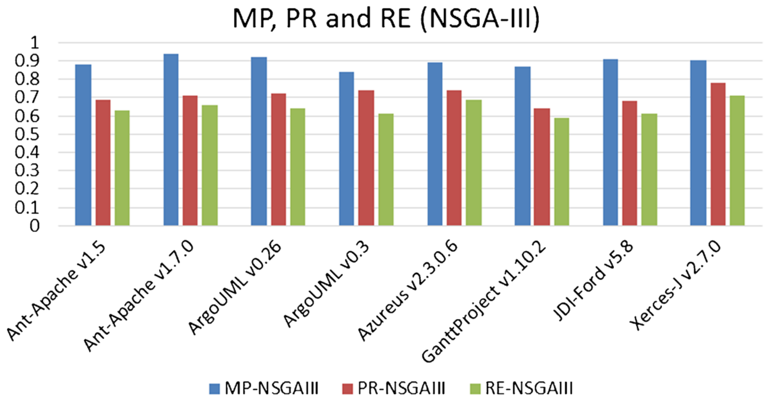


Fig. 5 Average manual and automatic design coherence measures (MP, PR and RE), over 31 runs, on the different systems using NSGA-III

Table 14 Median IGD values on 31 runs (best values are in bold and underlined, second best values are in bold)

System	NSGA-III	Gr-EA	DBEA-Eps	IBEA	MOEA/D	NSGA-II
ArgoUML v0.26	**4.113×10^{-3}**	4.229×10^{-3}	**4.206×10^{-3}**	4.329×10^{-3}	4.342×10^{-3}	~
Xerces v2.7	**7.998×10^{-3}**	8.308×10^{-3}	**8.181×10^{-3}**	8.399×10^{-3}	8.431×10^{-3}	~
ArgoUML v0.3	**5.499×10^{-3}**	5.677×10^{-3}	**5.603×10^{-3}**	5.712×10^{-3}	5.733×10^{-3}	~
Ant-Apache v1.5	**6.008×10^{-4}**	6.256×10^{-4}	**6.224×10^{-4}**	6.325×10^{-4}	6.333×10^{-4}	~
Ant-Apache v1.7.0	**6.202×10^{-3}**	6.412×10^{-3}	**6.377×10^{-3}**	6.489×10^{-3}	6.539×10^{-3}	~
Gantt v1.10.2	**7.806×10^{-3}**	8.002×10^{-3}	**7.968×10^{-3}**	8.088×10^{-3}	8.101×10^{-3}	~
Azureus v2.3.0.6	**6.933×10^{-4}**	7.112×10^{-4}	**7.075×10^{-4}**	7.191×10^{-4}	7.208×10^{-4}	~
JDI-Ford	**5.748×10^{-4}**	6.066×10^{-4}	**5.851×10^{-4}**	6.294×10^{-4}	6.646×10^{-4}	~

~ means a large value that is not interesting to show. The results were statistically significant on 31 independent runs using the Wilcoxon rank sum test with a 95 % confidence level (α=5 %)

comparisons, so we do not need to adjust p-values. After applying Cohen's d effect size we noticed that the effect size between the pair comparison of NSGA-III with each of the remaining algorithms is higher than 0.8 except for Gr-EA, which effect values were found to be medium.

All the results were statistically significant on the 31 independent simulations using the Wilcoxon rank sum test with a 95 % confidence level (α=5 %). NSGA-III strictly outperforms NSGA-II and gives slightly better results to those of the other many-objective algorithms. It is worth noting that for problems formulations with more 3 objectives, NSGA-II performance is dramatically degraded, which is simply denoted by the~symbol. The performance of NSGA-III could be explained by the interaction between: (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms.

Figure 6 describes value path plots of all algorithms for the 8-objective refactoring problem on Argo-UML. The horizontal axis shows the objective functions while the vertical axis marks its related values. The objectives values were normalized between 0 and 1 and set to be minimized. In terms of convergence, the algorithm whose solutions are closest to the ideal vector of height zeros has better convergence ratio. Thus, NSGA-III and DBEA-Eps outperform the remaining algorithms. Also, the spread of NSGA-III solutions vary in between [0, 0.9] presents a slightly better diversity than its follower DBEA-Eps whose solutions vary in between [0, 0.85]. On the other hand, the worst convergence is associated to NSGA-II as its solutions are so far from the ideal vector, and even it diversity is so reduced which may explain the stagnation of its evolutionary process. We conclude that although NSGA-II is the most famous multi-objective algorithm in SBSE, it is not adequate for problems involving over 3 objectives and NSGA-III is a very good candidate solution for tackling many-objective SBSE problems.

Figure 6 Value path plots of non-dominated solutions obtained by NSGA-III, DBEA-Eps, Gr-EA, IBEA, IBEA, and NSGA-II during the median run of the 8-objective refactoring problem on ArgoUML v0.26. The X-axis represents the different objectives while the Y-axis shows the variation of fitness values between [0..0.1]

When using optimization techniques, the most time consuming operation is the evaluation step. Thus, we studied the execution time of all many/multi-objective algorithms used in our experiments. Figure 7 shows the average running times of the different algorithms, over 31

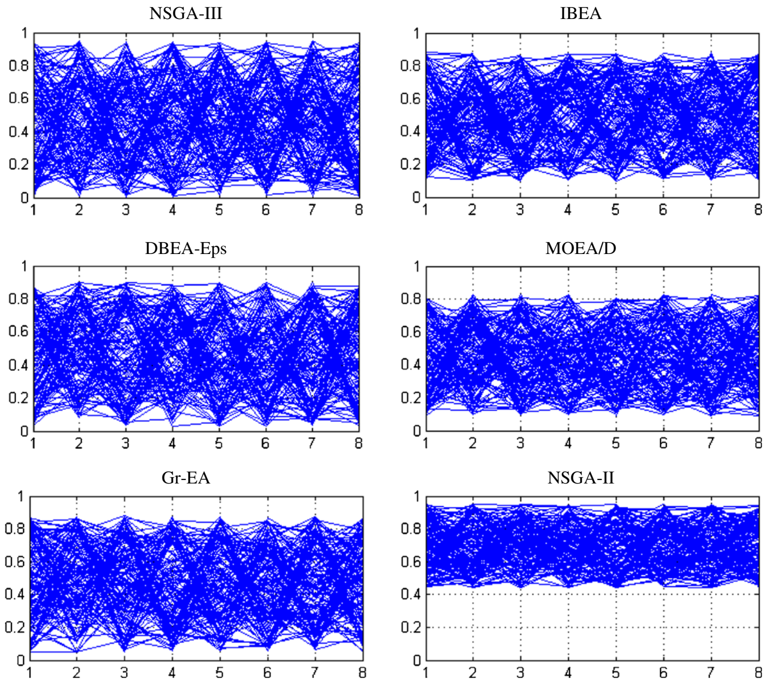


Fig. 6 Value path plots of non-dominated solutions obtained by NSGA-III, DBEA-Eps, Gr-EA, IBEA, IBEA, and NSGA-II during the median run of the 8-objective refactoring problem on ArgoUML v0.26. The X-axis represents the different objectives while the Y-axis shows the variation of fitness values between [0..0.1]

runs, on the ArgoUMLv0.26 system, the largest system in our experiments. It is clear from this figure that for an 8 objectives NSGA-III is faster than IBEA. This observation could be explained by the computational effort required to compute the contribution (IGD) of each solution. In comparison to MOEA/D, MOEA/D is slightly faster than NSGA-III since it does not make use of non-dominated sorting. Note that the experiments were conducted on a single machine (i7 – 2.70 GHz, 12.0 GB – DDR3, SSD - 520 MB/s) which may not be the optimal setting for some of these heuristics that can perform faster in an appropriate distributed or parallel environment.

We compared also the different search algorithms using metrics related to quality improvements, number of fixed defects, number of generated refactorings and a manual inspection of the results to check the correctness of the suggested operations. Figure 8 shows that our NSGA-III algorithm presents the best compromise between the different quality attributes

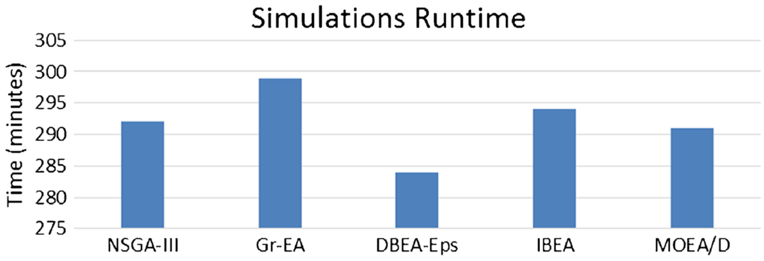


Fig. 7 Average Computational time values on 31 runs on refactoring ArgoUMLv0.26

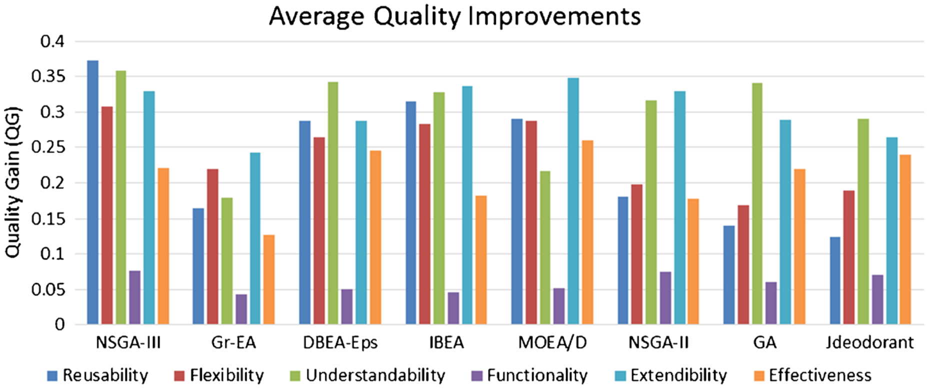


Fig. 8 Average quality improvements, over 31 runs, on the different systems

among all the other search algorithms. In addition, it is clear that the many-objective algorithms propose a better trade-off in terms of quality improvements than the mono and multi-objective techniques.

Since it is not sufficient to outperform existing search-based refactoring techniques, we compared our proposal to a popular design defects detection and correction tool JDeodorant. We first note that JDeodorant (like mono-objective approaches also) provides only one refactoring solution, while NSGA-III generates a set of non-dominated solutions. It can be seen that NSGA-III provides better results than JDeodorant, in average. The main reason can be related to the fact that JDeodrant provides a template of possible refactorings to apply for each detected defect but it is difficult to generalize such refactoring solutions since a defect can have several different refactoring strategies.

Figure 9 confirms that the majority of the suggested refactorings by NSGA-III improve significantly the code quality while preserving design's semantic coherence better than most of the other search algorithms. In addition, we automatically evaluated our approach

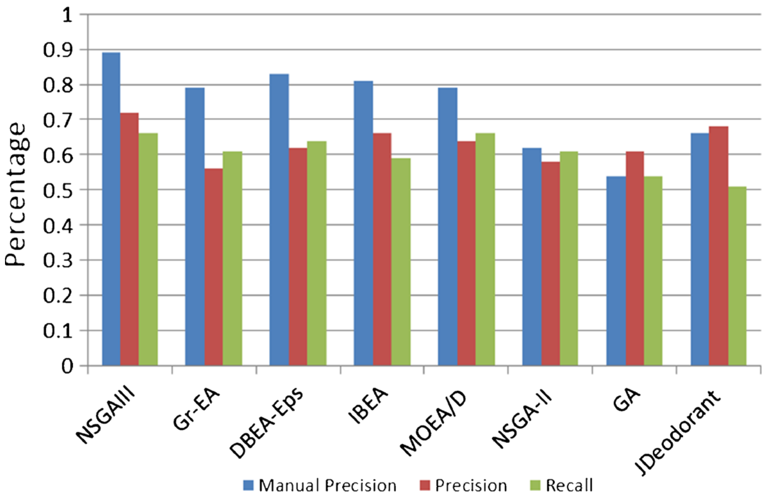


Fig. 9 Average manual and automatic design coherence measures (MP, PR and RE), over 31 runs, on the different systems

without using the feedback of potential users to give more quantitative evaluation. Thus, we compared the proposed refactorings with the expected ones. The expected refactorings are those applied by the software development team to the next software release as described in Table 10. Figure 9 confirms the outperformance of NSGA-III comparing to the remaining techniques.

We evaluated the number of operations (NO) suggested by the best refactoring solutions on the different systems over 31 runs. Figure 10 presents the code changes score for each algorithm, calculated by summing the size (number of operations) of each solution assigned to one project, divided by the number of projects. It is clear that our NSGA-III approach succeeded in suggesting solutions that do not require high code changes. However, IBEA generated less number of refactorings than our approach but this can be due to the fact that our technique improved better the quality comparing to IBEA's solutions. Thus, it may require higher number of refactorings to better improve the quality attributes. Another observation is that the number of refactorings proposed by JDeodorant is lower than the number of refactorings suggested by NSGA-III. However, the number of defects fixed by NSGA-III is higher than JDeodorant thus it is normal in this case that NSGA-III generates higher number of refactorings.

5.3 Results for RQ3

We asked the software engineers involved in our experiments to evaluate the usefulness of the suggested refactorings to apply one by one. In fact, sometimes these operations can improve the quality and preserve the semantics but developers will consider them as not useful due to many reasons such as some code fragments are not used/updated anymore or includes some features that are not important. Figure 11 shows that NSGA-III clearly outperforms existing work by suggesting useful refactoring operations for developers.

During the survey, the software engineers confirm that the main limitation related to the use of NSGA-III for software refactoring is the high number of equivalent solutions. However, found the idea of the use of the Knee point as described previously useful to select a good solution. We will investigate in our future work different other techniques to select the region of interest based on the preferences of developers.

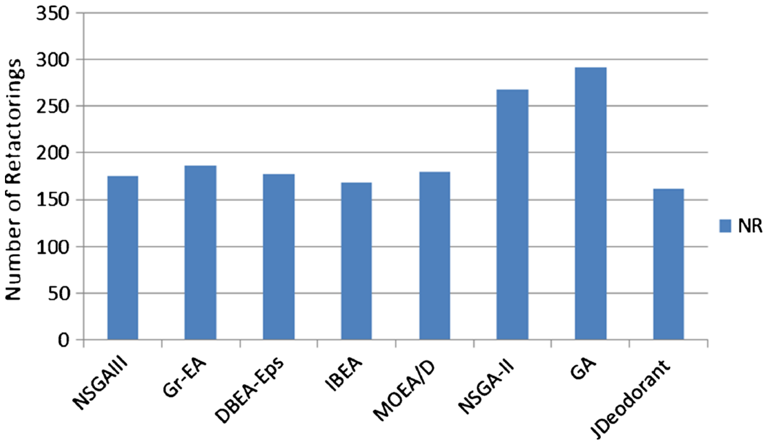


Fig. 10 Average suggested number of refactoring operations, over 31 runs, on the different systems

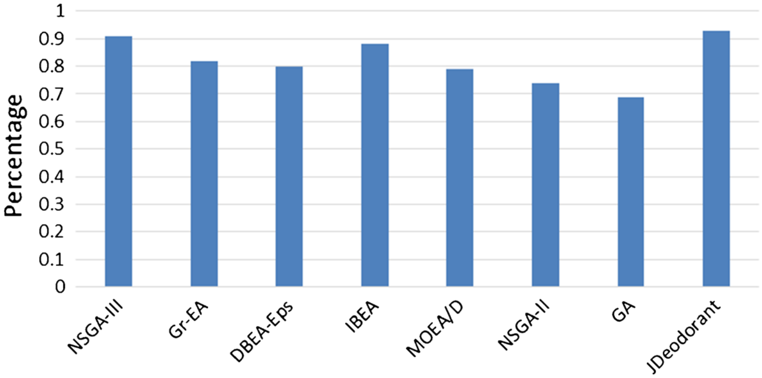


Fig. 11 Average of percentages of useful operations (IR), on the different systems using NSGA-III

6 Threats to validity

This section discusses potential threats to the validity of the study related to this work.

6.1 Threats to internal validity

The first threat is related to the variation of correctness and speed between the different groups when using our approach and other tools such as JDeodorant. In fact, our technique may not be the only reason for the superior performance because the subjects 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. Another threat concerns the collected code changes of the studied systems. In addition to the documented refactorings, we used Ref-Finder, which is known to be efficient. Indeed, Ref-Finder is able to detect refactoring operations with an average recall of 95 % and an average precision of 79 % (Kim et al. 2010). To ensure precision, we manually examined Ref-Finder generated refactorings by randomly selecting a set of detected changes and evaluating them by the participants in our experiments. We also identified three threats to internal validity: selection, learning and fatigue, and diffusion.

The selection threat refers to the participating subjects' profile and experience that could affect our study. Firstly, the subjects were all volunteers. We also minimized the selection threat by providing some guidelines and examples of previously evaluated refactorings. In addition, we also took care to randomize the selection refactorings to be evaluated for each refactoring solution.

Randomization also helps to prevent the learning and fatigue threats. We also mitigated the fatigue threat, by sending the questionnaires to the subjects by email and gave them enough time to complete the tasks.

Diffusion threat is limited because of the geographic distribution of our subjects in two different universities and a company, and they hardly know each other. For those who are in the same location, they were asked not to share information about the experience prior to the completion of the study.

Conclusion validity deals with the relation between the treatment and the outcome. Thus, we took special care to vary the subjects based on their professional status, university/company affiliations, gender, and years of experience. In addition, we clustered subjects into balanced groups. This has been said, we plan to test our tool with Java development companies, to draw better conclusions. Moreover, the automatic evaluation is also a way to limit the threats related to subjects as it helps to ensure that our approach is efficient and useful in practice.

6.2 Threats to extremal validity

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different widely-used systems and one industrial system belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to industrial Java applications, other programming languages, and to other practitioners. Future replications of this study are necessary to confirm our findings. Another limitation of our proposal is the selection of the best solution from the Pareto front. We used the technique of selecting the solution at the knee point. However, we plan in our future work to more integrate the preferences of developers to select the best solution from the set of non-dominated solutions. Another raised threat is due to the limited number of subjects and evaluated systems, which externally threatens the generalizability of our results. In addition, our study used a limited set of refactoring operations. Future replications of this study are necessary to confirm our findings.

7 Conclusions and future work

We propose a novel formulation of the refactoring problem as a many-objective problem, based on NSGA-III, using the quality attributes of QMOOD as objectives along with the number of refactorings and the preservation of design coherence. This paper represents one of the first real-world applications of NSGA-III. This initial empirical investigation has shown that a possible conflict among QMOOD objectives may occur depending on the kind of employed refactorings. In this context, a statistical analysis needs to be conducted to prove the validity of this insight. Furthermore, we did not yet prioritize any objective(s) although the definition of QMOOD and the capability of NSGA-III allows it. It would be interesting to compare multiple refactored systems, while each one of them is the result of high prioritization of one quality objective.

We implemented our approach and evaluated it on seven large open source systems and one industrial project provided by our industrial partner. We compared our findings to: several other many-objective techniques (IBEA, MOEA/D, GrEA, and DBEA-Eps), a mono-objective technique and an existing refactoring technique not based on heuristic search. Statistical analysis of our experiments over 31 runs shows the efficiency of NSGA-III as a powerful algorithm to tackle many objective formulations.

We also studied the impact of refactoring on fixing code smells. Although we were able to fix most of the detected defects, our defect selection types were limited due to limited types of used refactorings. Another limitation to take into account is the possibility of introducing code smells while performing the refactoring operations, for example, we noticed that repairing some instances of feature envy led to the introduction of the shotgun surgery defect. Since the latter type is not covered in this work, it did not affect our DCR, but it gives a strong indication for further investigation about how to perform refactoring interactively with the developer, combined with code smell detection, to avoid such drawbacks.

For the qualitative evaluation, we were able to show promising results, along with JDeodorant, which, as a tool, was highly appreciated by the participants because of its simplicity, ease of use and the possibility to preview changes and visualize entities. Unfortunately, it is limited to four types of smells.

In future work, multiple research directions are to be taken from some of the previously mentioned limitations and they are mainly linked to (1) the problem formulation and (2) NSGA-III tuning. Firstly, investigating the statistical significance of the impact of the refactorings on internal and external attributes can be an interesting research direction that can help in better understanding to what extent each refactoring type can affect the existing quality models. Moreover, code smells can be also described in terms of structural metrics, such statistical investigation will help in the validation of attributing a subset of refactorings to a known type of code smell. As for NSGA-III tuning, we will investigate the impact of different parameter settings on the quality of our results, in particular, the size of the reference set $|Z^r|$ can either be predefined and calculated based on the number of objectives and the number of desired divisions in the hyper-plane or preferentially introduced by the user. Augmenting the density of the hyper-plane i.e., increasing the number of used reference points will refine the niche count and thus will provide solutions with better diversity. Since, in our experiments, we only considered the predefined size of reference set, we plan in the future to investigate the impact of varying this parameter on the quality of the generated solutions. Another interesting research direction regards the prioritization of the code smells to be removed. For example, two candidate solutions extracted from the Pareto-front may have equivalent fitness function values, but their impact on reducing code smells may vary in terms of the number and types of fixed code smells. So this can be used as an additional developer preference to compare between given solutions. Moreover, the solution’s robustness can be also studied as to take into account the uncertainties related to analyzed classes importance and code smells severities while suggesting refactoring operations. Furthermore, we plan to work on adapting NSGA-III to additional software engineering problems and we will perform more comparative studies on larger open source systems. Nevertheless, this extensive study has shown a direction using NSGA-III to handle as many as 8 objectives in the context of solving software engineering problems and will remain one of the first studies in which such a large number of objectives have been considered.

Acknowledgments This work was supported, in part, by the Institute for Advanced Vehicle Systems-Michigan grant, the UM-Ford Alliance Program and the Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre.

References

Abreu F-B (1995) The MOOD Metrics Set, Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Workshop on Metrics, Vol. 95, p. 267

Alshayeb M (2009) Empirical investigation of refactoring effect on software quality. Inf Softw Technol 51(9): 1319–1326

Arcuri A, Fraser G (2013) Parameter tuning or default values? An empirical investigation in search-based software engineering. Empir Softw Eng 18(3):594–623

Asafuddoula M, Ray T, Sarker R (2013) A decomposition based evolutionary algorithm for many objective optimization with systematic sampling and adaptive epsilon control. Evol Multi-Criterion Optim 7811:413–427

Bader J, Zitzler E (2011) HypE: an algorithm for fast hypervolume-based many-objective optimization. Evol Comput 19(1):45–76

Bansiya J, Davis C-G (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28(1):4–17. doi:10.1109/32.979986

Barros M-O (2012) An analysis of the effects of composite objectives in multiobjective software module clustering. In Proceedings of the 14th annual conference on Genetic and evolutionary *computation* (GECCO '12), T Soule (Ed.). ACM, New York, NY, USA, 1205–1212. Doi: 10.1145/2330163.2330330

Basili V-R (1992) Software modeling and measurement: the Goal/Question/Metric paradigm. Technical Report. University of Maryland at College Park, College Park, MD, USA

Bechikh, S., Ben Said, L., & Ghedira, K. (2010). Estimating nadir point in multi-objective optimization using mobile reference points. In Evolutionary computation (CEC), 2010 I.E. congress on (pp. 1-9). IEEE

Bechikh S, Ben Said L, Ghédira K (2011) Searching for knee regions of the Pareto front using mobile reference points. Soft computing. Soft Comput Fusion Found, Methodologies Appl 15(9):1807–1823

Ben Said L, Bechikh S, Ghédira K (2010) The r-dominance: a new dominance relation for interactive evolutionary multicriteria decision making. Proc IEEE Trans Evol Comput 14(5):801–818

Bowman M, Briand L-C, Labiche Y (2010) Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. Softw Eng, IEEE Trans 36(6):817–837. doi:10.1109/TSE.2010.70

Brown, W. J., Malveau, R. C., Brown, W. H., and Mowbray, T-J (1998) Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1st Ed

Chidamber S-R, Kemerer C-F (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Colanzi T-E, Vergilio S-R (2012) Applying search based optimization to software product line architectures: lessons learned. In Proceedings of the 4th international conference on Search Based Software Engineering (SSBSE'12), Gordon Fraser and Jerffeson T. Souza de (Ed). Springer-Verlag, Berlin, Heidelberg, 259–266. Doi: 10.1007/978-3-642-33119-0_19

Corazza A, Di Martino S, Maggio V (2012) LINSEN: an efficient approach to split identifiers and expand abbreviations. In Proceedings of IEEE International Conference on Software Maintenance, pp.233–242

Counsell S, Hierons R-M, Najjar R, Loizou, G, Hassoun Y (2006) The effectiveness of refactoring, Based on a compatibility testing taxonomy and a dependency graph. Practice and research techniques, In Testing: Academic and Industrial Conference-Practice and Research Techniques, 2006. TAIC PART 2006. Proceedings, 181–192

Dean J, Grove D, Chambers G (1995) Optimization of object-oriented programs using static class hierarchy analysis, Proceedings of the 9th European Conference on Object-Oriented Programming, p.77–101

Deb K (2001) Multiobjective otpimization using evolutionary algorithms. Wiley, New York

Deb K, Jain H (2012) Handling many-objective problems using an improved NSGA-II procedure. In Proceedings of IEEE Congress on Evolutionary Computation. 1–8

Deb K, Jain H (2013) An evolutionary many-objective optimization algorithm using reference-point based Non-dominated sorting approach, part I: solving problems with Box constraints. Evol Comput, IEEE Trans 18(4): 577–601

Deb K, Saxena D-K (2006) Searching for pareto-optimal solutions through dimensionality reduction for certain large-dimensional multiobjective optimization problems. In Proceedings of IEEE Congress on Evolutionary Computation. 3353–3360

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. In proceedings of. IEEE Trans Evol Comput 6(2):182–197

Deb K, Sundar J, Uday N, Chaudhuri S (2006) Reference point based multi-objective optimization using evolutionary algorithms. Int J Comput Intell Res (IJCIR'06) 2(6):273–286

di Pierro F, Khu S-T, Savic D-A (2007) An investigation on preference order ranking scheme for multiobjective evolutionary optimization. Proc IEEE Trans Evol Comput 11(1):17–45

Dig D (2011) A refactoring approach to parallelism. IEEE Softw 28(1):17–22

Du Bois B, Mens T (2003) Describing the impact of refactoring on internal program quality. In International Workshop on Evolution of Large-scale Industrial Software Applications, 37–48

Du Bois B, Demeyer S, Verelst J (2004) Refactoring—Improving coupling and cohesion of existing code, Proceedings of the 11th Working Conference on Reverse Engineering. pp. 144–151

Foster S-R, Griswold W-G, Lerner S (2012, June) WitchDoctor: IDE support for real-time auto-completion of refactorings. In Proceedings of the 34th International Conference on Software Engineering, 222–232

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring – Improving the Design of Existing Code. 1st ed. Addison-Wesley

Garza-Fabre M, Toscano-Pulido G, Coello Coello C-A, Rodriguez-Tello E (2011) Effective ranking + speciation = Many-objective optimization, In Evolutionary Computation (CEC), 2011 I.E. Congress on pp. 2115–2122

Ge X, Murphy-Hill E (2011) BeneFactor: a flexible refactoring tool for eclipse. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and* *applications companion* (OOPSLA '11). ACM, New York, NY, USA, 19–20. Doi: 10.1145/2048147.2048157

Ge X, Murphy-Hill E (2014) Manual refactoring *changes with automated refactoring validation. In* Proceedings of the 36th International Conference on Software Engineering *(ICSE* 2014). ACM, New York, NY, USA, 1095–1105. Doi: 10.1145/2568225.2568280

Harman M (2013) Software Engineering: An Ideal Set of Challenges for Evolutionary Computation, In GECCO '13, 1759–1760

Harman M, Jones BF (2001) Search-based software engineering. Inf Softw Technol 43(14):833–839

Harman M, Tratt L (2007) Pareto optimal search based refactoring at the design level. In GECCO'07. 1106–1113

Harman M, Mansouri S-A, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Comput Surv (CSUR) 45(1):11

Jaimes A-L, Coello Coello, C-A, Barrientos J-E-U. (2009). Online Objective Reduction to Deal with Many-objective Problems. In the 5th international conference on Evolutionary Multicriterion Optimization. 423–437

Jain H, Deb K (2014). An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part II: handling constraints and extending to an adaptive approach. In Proceedings of IEEE Trans Evol Comput 18(4):602–622

Kalboussi S, Bechikh S, Kessentini M, Ben Said L (2013) Preference-based Many-objective Evolutionary Testing Generates Harder Test Cases for Autonomous Agents, in Proc. 5th International Symposium on Search-Based Software Engineering 2013 (SSBSE'13), 245–250

Kataoka Y, Notkin D, Ernst M-D, Griswold W-G. (2001) Automated support for program refactoring using invariants. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01) 736

Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A (2011) Design defects detection and correction by example. In ICPC'11. 81–90

Kim M, Gee M, Loh A, Rachatasumrit N (2010) Ref-Finder: a refactoring reconstruction tool based on logic query templates. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10). ACM, New York, NY, USA, 371–372. Doi: 10.1145/1882291.1882353

Kremmel T, Kubalík J, Biffl S (2011) Software project portfolio optimization with advanced multiobjective evolutionary algorithms. Appl Soft Comput 11(1):1416–1426

Kukkonen S, Lampinen J (2007) Ranking-dominance and many-objective optimization. In Proceedings of IEEE Congress on Evolutionary Computation (CEC), 3983–3990

Lorenz M, Kidd J (1994) Object-oriented software metrics: a practical guide. Prentice-Hall, Inc

Mäntylä L, Vanhanen J, Lassenius C (2003) A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In Proceedings of the International Conference on Software Maintenance (ICSM '03). IEEE Computer Society, Washington, DC, USA, 381–384

Marinescu R (2004) Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04). IEEE Computer Society, Washington, DC, USA, 350–359

Marinescu R, Ganea G, Verebi I (2010) InCode: continuous quality assessment and improvement. In Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, 274–275

Martin R-C (2000) Design principles and design patterns. Object Mentor 1:34

Meananeatra P (2012) Identifying refactoring sequences for improving software maintainability. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012). ACM, New York, NY, USA, 406–409. Doi: 10.1145/2351676.2351760

Mkaouer M-W, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M (2014). High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 objectives for Automated Software Refactoring using NSGA-III, In Proc. Genetic and Evolutionary Computation Conference (GECCO'14), 1263–1270

Mkaouer M-W, Kessentini M, Shaout A, Koligheu P, Bechikh S, Deb K, Ouni A (2015) Many-objective software remodularization using NSGA-III. ACM Trans Softw Eng Methodol (TOSEM) 24(3):17

Moha N, Guéhéneuc Y-G, Duchien L, Le Meur A.-F (2009) DECOR: A Method for the Specification and Detection of Code and Design Smells. In TSE, vol 12, 20–36

Murphy-Hill E (2006) Improving usability of refactoring tools. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA *'06). ACM*, New York, NY, USA, 746–747. Doi: 10.1145/1176617.1176705

Ó Cinnéide M, Tratt L, Harman M, Counsell S, Moghadam I-H (2012). Experimental Assessment of Software Metrics Using Automated Refactoring. In ESEM'12, 49–58

O'Keeffe M-K, Ó Cinnéide M (2008) Search-based refactoring for software maintenance. J Syst Softw 81(4):502–516

Olaechea R, Rayside D, Guo J, Czarnecki K (2014) Comparison of exact and approximate multi-objective optimization for software product lines. In Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14), Stefania Gnesi, Alessandro Fantechi, Patrick Heymans, Julia Rubin, Krzysztof Czarnecki, and Deepak Dhungana (Eds.), Vol. 1. ACM, New York, NY, USA, 92–101. Doi: 10.1145/2648511.2648521

Ouni A, Kessentini M, Sahraoui H, Boukadoum M (2012a) Maintainability defects detection and correction: a multi-objective approach. J Autom Softw Eng 20:47–79

Ouni A, Kessentini M, Sahraoui H A, Hamdi MS (2012) Search-based refactoring: Towards semantics preservation. ICSM 347–356

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting Bad Smells in Source Code Using Change History Information. IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 268–278

Piveta E-K, Pimenta M-S, Araújo J, Moreira A, Guerreiro P, Price R-T (2006) Detecting bad smells in aspectJ. J UCS 12(7):811–827

Praditwong K, Harman M, Yao X (2010) Software module clustering as a multi-objective search problem. Softw Eng IEEE Trans 37(2):264–282. doi:10.1109/TSE.2010.26

Rachmawati L, Srinivasan D (2009) Multiobjective evolutionary algorithm with controllable focus on the knees of the Pareto front. IEEE Trans Evol Comput 13(4):810–824

Ramírez A, Romero J-R, Ventura S (2014) On the performance of multiple objective evolutionary algorithms for software architecture discovery. In Proceedings of the 2014 Conference on Genetic and Evolutionary Computation (GECCO '14). ACM, New York, NY, USA, 1287–1294. doi: 10.1145/2576768.2598310

Rodriguez D, Ruiz M, Riquelme J-C, Harrison R (2011) Multiobjective simulation optimisation in software project management. In Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO '11), Natalio Krasnogor (Ed.). ACM, New York, NY, USA, 1883–1890

Sarro F, Ferrucci F, Gravino C (2012) Single and Multi Objective Genetic Programming for software development effort estimation. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, 1221-1226. Doi: 10.1145/2245276.2231968

Sayyad A, Menzies T, Ammar H (2013) On the value of user preferences in search-based software engineering: a case study in software product lines. In ICSE '13. 492–501

Sayyad A-S, Ingram J, Menzies T, Ammar H (2013) Scalable product line configuration: A straw to break the camel's back, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, vol., no., pp.465–474, 2013. doi: 10.1109/ASE.2013.6693104

Seng O, Stammel J, Burkhart D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems. In GECCO'06. 1909–1916

Shatnawi R, Li W (2011) An empirical assessment of refactoring impact on software quality using a hierarchical quality model. Int J Softw Eng Appl 5(4):127–149

Singh H-K, Isaacs A, Ray T (2011) A Pareto corner search evolutionary algorithm and dimensionality reduction in many-objective optimization problems. Proc IEEE Trans Evol Comput 99:1–18

Tahvildari L, Kontogiannis K, Mylopoulos J (2003) Quality-driven software re-engineering. J Syst Softw 66(3): 225–239. doi:10.1016/S0164-1212(02)00082-1

Thiele L, Miettinen K, Korhonen P-J, Luque J-M (2009) A preference-based evolutionary algorithm for multi-objective optimization. Evol Comput 17(3):411–436

Tsantalis N, Chaikalis T, Chatzigeorgiou A (2008) JDeodorant: Identification and Removal of Type-Checking Bad Smells. In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering *(CSMR* '08). IEEE Computer Society, Washington, DC, USA, 329–331. Doi: 10.1109/CSMR.2008.4493342

Van Emden E, Moonen L (2002) Java Quality Assurance by Detecting Code Smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE '02). IEEE Computer Society, Washington, DC, USA, 97

Wada H, Champrasert P, Suzuki J, Oba K (2008, July) Multiobjective optimization of sla-aware service composition. In Services-Part I, 2008. IEEE Congress on, 368–375 doi: 10.1109/SERVICES-1.2008.77

Wang R, Purshouse R-C, Fleming P-J (2013) Preference-inspired coevolutionary algorithms for many-objective optimization. Proc IEEE Trans Evol Comput 17(4):474–494

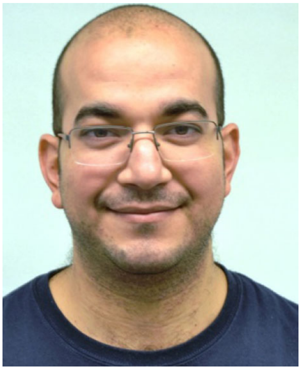
Yang S, Li M, Liu X, Zheng J (2013) A grid-based evolutionary algorithm for many-objective optimization. Evol Comput, IEEE Trans 17(5):721–736

Yao X (2013) Some Recent Work on Multi-objective Approaches to Search-Based Software Engineering In Proc. 5th Symp. on Search Based Software Engineering (SSBSE), vol. 8084, 4–15

Zhang Q, Li H (2007) MOEA/D: a multiobjective evolutionary algorithm based on decomposition. Proc IEEE Trans Evol Comput 11(6):712–731

Zhuang L, HeQing G, Dong L, Tao H, Juan Juan Z (2007) Solving Multi-Objective and Fuzzy Multi-Attributive Integrated Technique for QoS-Aware Web Service Selection, In Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on, 735–739 doi: 10.1109/WICOM.2007.190

Zitzler E, Künzli S (2004) Indicator-based selection in multiobjective search Parallel Problem Solving from Nature. In Parallel Problem Solving from Nature-PPSN VIII, 832–842, Springer: Berlin Heidelberg



Mohamed Wiem Mkaouer is currently a PhD candidate in Software Engineering at University of Michigan-Dearborn, USA, under the supervision of Dr. Marouane Kessentini. His research interests include software quality, software testing, model-driven engineering and search-based software engineering. He is a member of the Search-based Software Engineering at Michigan research group, he is also a student member of the IEEE and the IEEE Computer Society.

Marouane Kessentini is a tenure-track assistant professor at the Computer and Information Science department, University of Michigan, Dearborn campus. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 70 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program-committee/organization member in several conferences and journals.

Slim Bechikh received the BSc degree in computer science applied to management and the MSc degree in modeling from the University of Tunis, Tunisia, in 2006 and 2008, respectively. He also obtained the PhD degree in computer science applied to management from University of Tunis in January 2013. He worked, for four years, as an attached researcher within the Optimization Strategies and Intelligent Computing lab (SOIE), Tunisia. Now, he is a postdoctoral researcher at the SBSE@Michigan, University of Michigan. His research interests include multi-criteria decision making, evolutionary computation, multi-agent systems, portfolio optimization and search-based software engineering. Since 2008, he published several papers in well-ranked journals and conferences. Moreover, he obtained the best paper award of the ACM Symposium on Applied Computing 2010 in Switzerland among more than three hundreds participants. Since 2010, he serves as reviewer for several conferences such as ACM SAC and GECCO and various journals such as Soft Computing and IJITDM.

Mel Ó Cinnéide graduated in Computer Science from University College Cork, and then spent several years working in the software industry, initially with Philips in the Netherlands and later with Motorola in Cork. After this period in industry, he lectured in University College Cork while completing an MSc in the area of automated analysis and improvement of C++ inheritance hierarchies. He completed his PhD dissertation in the University of Dublin in 2001 on the topic of automated application of design patterns using refactorings, and is currently a member of academic staff in the School of Computer Science, University College Dublin. His present research interests centre around refactoring, and especially the use of search-based software engineering in automated refactoring. Related interests include design patterns, software metrics and code smell detection.

Professor Kalyanmoy Deb is Koenig Endowed Chair Professor at Electrical and Computer Engineering in Michigan State University, USA. Prof. Deb's research interests are in evolutionary optimization and their application in optimization, modeling, and machine learning. Prof. Deb has numerous awards and honours in his name, including the prestigeous Shanti Swarup Bhatnagar Prize in Engineering Sciences in 2005, 'Thomson Citation Laureate Award', an award given to an Indian Researcher for making most highly cited research contribution during 1996-2005 in a particular discipline according to ISI Web of Science, Friedrich Wilhelm Bessel Research Award and Humboldt Fellowship from Alexander von Humboldt Foundation, Germany. He is a fellow of Indian National Science Academy (INSA), Indian National Academy of Engineering (INAE), Indian Academy of Sciences (IASc), and International Society of Genetic and Evolutionary Computation (ISGEC). He has been awarded 'Distinguished Alumnus Award from his Alma mater IIT Kharagpur in 2011. Author of more than 275 research papers, two textbooks, 17 edited books, his 2001 book on Evolutionary Multiobjective Optimization Algorithms is the first ever compilation of multiobjective optimization algorithms. Because of his pioneering research in the field of evolutionary multi-objective optimization (EMO), he has been invited to present 35 Keynote lectures and more than 100 invited lectures and tutorials on the topic. His NSGA-II paper from IEEE Trans. on Evolutionary Computation (2000) is judged as the Fast-Breaking Paper in Engineering by ESI Web of Science and now this paper is awarded the 'Current Classic' and 'Most Highly Cited Paper' by Thomson Reuters. He is fellow of IEEE and three science academies in India. He has published 350+ research papers with Google Scholar citation of 55,000+ with h-index 77. He is in the editorial board on 20 major international journals. More information about his research can be found from http://www.egr.msu.edu/~kdeb.