Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach

Wiem Mkaouer, Marouane Kessentini, Slim Bechikh
University of Michigan, USA
firstname@umich.edu

Mel Ó Cinnéide
University College Dublin, Ireland.
mel.ocinneide@ucd.ie

Kalyanmoy Deb
Michigan State University, USA.
kdeb@egr.msu.edu

COIN Report Number 2013003

ABSTRACT

Refactoring large systems involves several sources of uncertainty related to the severity levels of code smells to be corrected and the importance of the classes in which the smells are located. Due to the dynamic nature of software development, these values cannot be accurately determined in practice, leading to refactoring sequences that lack robustness. To address this problem, we introduced a multi-objective robust model, based on NSGA-II, for the software refactoring problem that tries to find the best trade-off between quality and robustness. We evaluated our approach using six open source systems and demonstrated that it is significantly better than state-of-the-art refactoring approaches in terms of robustness in 100% of experiments based on a variety of real-world scenarios. Our suggested refactoring solutions were found to be comparable in terms of quality to those suggested by existing approaches and to carry an acceptable robustness price. Our results also revealed an interesting feature about the trade-off between quality and robustness that demonstrates the practical value of taking robustness into account in software refactoring tasks.

Categories and Subject Descriptors

D.2 [**Software Engineering**].

General Terms

Algorithms, Reliability.

Keywords

Search-based software engineering, software quality, code smells, robust optimization.

1. INTRODUCTION

Large-scale software systems exhibit high complexity and become difficult to maintain. It has been reported that the cost of maintenance and evolution activities comprises more than 80% of total software costs [8]. In addition, it has been shown that software maintainers spend around 60% of their time in understanding the code [48]. To facilitate maintenance tasks, one of the widely used techniques is refactoring which improves design structure while preserving the overall functionality of the software [23], [41].

There has been much work on different techniques and tools for refactoring [23], [41], [49], [42], [17], [31], [5]. The vast majority of these techniques identify key symptoms that characterize the code to refactor using a combination of quantitative, structural, and/or lexical information and then propose different possible refactoring solutions, for each identified segment of code. In order to find out which parts of the source code need to be refactored, most of the existing work relies on the notion of design defects or code smells. Originally coined by Fowler [23], the generic term code smell refers to structures in the code that suggest the possibility of refactoring. Once code smells have been identified, refactorings need to be proposed to resolve them. Several automated refactoring approaches are proposed in the literature and most of them are based on the use of software metrics to estimate quality improvements of the system after applying refactorings [49], [42], [17], [33], [40].

The existing literature on software refactoring invariably ignores an important consideration when suggesting refactoring solutions: the highly dynamic nature of software development. In this paper, we take into account two dynamic aspects as follows:

- *Code Smell Severity*: This is the severity level assigned to a code smell type by a developer. It usually varies from developer to developer, and indeed a developer's assessment of smell severity will change over time as well.
- *Code Smell Class Importance*: This is the importance of a class that contains a code smell, where importance refers to the number and size of the features that the class supports. A code smell with large class importance will have a greater detrimental impact on the software. Again, this property will vary over time as software requirements change [27] and classes are added/deleted/split.

Existing approaches to the refactoring problem assume a static environment to the problem, i.e., that all detected code smells are of the same severity and that the importance of the class in which is the code smell is situated is not liable to change.

We believe that the uncertainties related to class importance and code smell severity need to be taken into consideration when suggesting a refactoring solution. To this end, we introduce in this paper a novel representation of the code refactoring problem, based on *robust* optimization [10], [29] that generates robust refactoring solutions by taking into account the uncertainties related to code smell severity and the importance of the class that contains the code smell. Our robustness model is based on the well-known multi-objective evolutionary algorithm NSGA-II proposed by Deb et al. [16] and considers possible changes in class importance and code smell severity by generating different scenarios at each iteration of the algorithm. In each scenario, the detected code smell to be corrected is assigned a severity score and each class in the system is assigned an importance score. In our model, we assume that these scores change regularly due to reasons such as developers' evolving perspectives on the software or new features and requirements being implemented or any other code changes that could make some classes/code smells more or less important. Our multi-objective approach aims to find the best trade-off between maximizing the quality of the refactoring solution in terms of the number of code smells corrected and maximizing its robustness in terms of the severity of the code smells corrected and the importance of the classes that contains the code smells.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'14, Month 1–2, 2010, City, State, Country.

The primary contributions of this paper are as follows:

- The paper introduces a novel formulation of the refactoring problem as a multi-objective problem that takes into account the uncertainties related to code smell detection and the dynamic environment of software development. To the best of our knowledge, and based on recent search-based software engineering (SBSE) surveys [27], this is the first work to use robust optimization for software refactoring, and the first in SBSE to treat robustness as a helper objective during the search.
- The paper reports on the results of an empirical study of our robust NSGA-II technique as applied to six open source systems. We compared our approach to random search, multi-objective particle swarm optimization (MOPSO) [35], search-based refactoring [33], [40] and a practical refactoring technique [54]. The results provide evidence to support the claim that our proposal enables the generation of robust refactoring solutions without a high loss of quality using a variety of real-world scenarios [58], [55], [53], [52], [56], [57].

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach and the nature of the refactoring challenge. In Section 3, we describe robust optimization and explain how we formulate software refactoring as a robust optimization problem. Section 4 presents and discusses the results obtained by applying our approach to six large open-source projects. Related work is discussed in Section 5, while in Section 6 we conclude and suggest future research directions.

2. SOFTWARE REFACTORING: BACKGROUND AND CHALLENGES

Software refactoring is the process of improving code after it has been written by changing its internal structure without changing its external behavior [23]. The idea is to reorganize variables, classes and methods both to facilitate future adaptations and extensions, and to make the intention of the code clearer. To find out which parts of the source code need to be refactored, most existing work (e.g., [23], [41], [49], [42], [17], [31]) rely on the notion of code smells. In this paper, we do not focus on the first step related to the detection of refactoring opportunities. We consider that different code smells have already been detected, and need to be corrected.

Typically, code smells refer to design situations that adversely affect the development of software. As stated by Fenton and Pfleeger [21], code smells are unlikely to cause failures directly, but may do so indirectly. In general, they make a system more difficult to change, which may result in the introduction of bugs. Perhaps the most well known example of a code smell is the *blob*, which is where one large class monopolizes the behavior of a system, and other classes primarily encapsulate data. After detecting a blob, many refactoring operations can be used to reduce the amount of functionality in the blob class, such as move method and extract class.

Overall, there is no general consensus on how to decide if a particular code fragment constitutes a code smell. There is also a distinction between detecting the symptoms of a code smell and asserting that the code smell actually exists. For example, if one class in a system implements all the system behavior while the other classes are purely data classes, then this surely is an instance of the blob smell. Unfortunately, in real-life systems, matters are never so clear, and deciding if a class is a blob or simply a large class depends heavily on the interpretation of the developer making the decision. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, it is a common and acceptable practice to have a class that maintains a log of events in a program. However, as a large number of classes are coupled to it, it may be categorized as smelly. From this discussion we can conclude that it is difficult to estimate the severity of detected code smells since developers have divergent opinions on the topic.

Finally, detecting a large number of code smells in a system is not always helpful unless something is known about the priority of the detected code smells. In addition to the presence of false positives that may engender a rejection reaction in the developers, the process of assessing the detected smells, selecting the true positives, and finally correcting them is long, expensive, and not always profitable. Coupled with this, the priority ranking can change over time based on the estimation of smell severity and class importance scores.

To address these issues, we describe in the next section our robust refactoring model based on an evolutionary algorithm.

3. MULTI-OBJECTIVE ROBUST SOFTWARE REFACTORING

3.1 Robust Optimization

In dealing with optimization problems, including software engineering ones, most researchers assume that the input parameters of the problem are exactly known in advance. Unfortunately, this is an idealization often not the case in a real-world setting. Additionally, uncertainty can change the effective values of some input parameters with respect to nominal values. For instance, when handling the knapsack problem (KP), which is one of the most studied combinatorial problems [10], we can face such a problem. The KP problem requires one to find the optimal subset of items to put in a knapsack of capacity C in order to maximize the total profit while respecting the capacity C. The items are selected from an item set where each item has its own weight and its own profit. Usually, the KP's input parameters are not known with certainty in advance. Consequently, we should search for *robust* solutions that are *immune* to small perturbations in terms of input parameter values. In other words, we prefer solutions whose performance levels do not significantly degrade due to small perturbations in one or several input parameters such as item weights, item profits and knapsack capacity for a KP. As stated by [10], uncertainty is unavoidable in real problem settings; therefore it should be taken into account in every optimization approach in order to obtain robust solutions. Robustness of an optimal solution can usually be discussed from the following two perspectives: (1) the optimal solution is insensitive to small perturbations in terms of the decision variables and/or (2) the optimal solution is insensitive to small variations in terms of environmental parameters. Figure 1 illustrates the robustness concept with respect to a single decision variable named x. Based on the $f(x)$ landscape, we have two optima: A and B. We remark that solution A is very sensitive to local perturbation of the variable x. A very slight perturbation of x within the interval [2, 4] can make the optimum A unacceptable since its performance $f(A)$ would dramatically degrade. On the other hand, small perturbations of the optimum B, which has a relatively lower objective function value than A, within the interval [5,7] hardly affects the performance of solution B (i.e., $f(B)$) at all. We can say that although solution A has a better quality than solution B, solution B is more *robust* than solution A. In an uncertain context, the user would probably prefer solution B to solution A. This choice is justified by the performance of B in terms of robustness. It is clear from this discussion robustness has a price, called *robustness price or cost*, since it engenders a *loss in optimality*. This loss is due to preferring the robust solution B over the non-robust solution A. According to Figure 1, this loss is equal to $abs(f(B)-f(A))$. Several approaches have been proposed to handle robustness in the optimization field in general and more specifically in design engineering. These approaches can be classified as follows [29]:

- *Explicit averaging*: Assuming $f(x)$ to be the fitness function of solution x, the basic idea is to weighted average the fitness value [52] in the neighborhood $B_\delta(x)$ of solution x with a uncertainty distribution $p(\delta)$. The fitness function then becomes:

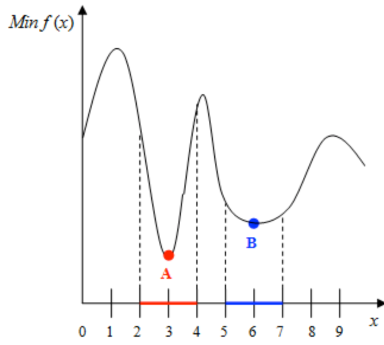

Figure 1. Illustration of the robustness concept under uncertainty related to the decision variable x. Among the two optima, solution B is more robust than solution A.

$$\int_{\delta \in B_\delta(x)} p(\delta) f(x+\delta) d\delta$$

However, because in the robustness term, case disturbances can be chosen deliberately, variance reduction techniques can be applied, allowing a more accurate estimation with fewer samples [11].

- *Implicit averaging*: The basic idea is to compute an expected fitness function based on the fitness values of some solutions residing within the neighborhood of the considered solution. Beyer et al. [9] noted that it is important to use a large population size in this case. In fact, given a fixed number of evaluations per generation, it was found that increasing the population size yields better results than multiple random sampling.
- *Use of constraints*: The difference between the weighted average fitness value and the actual fitness value at any point can be restricted to lie within a pre-defined threshold in order to yield more robust solutions [52]. Such an optimization process will prefer robust solutions to the problem and it then depends on the efficacy of the optimization algorithm to find the highest quality robust solution.
- *Multi-objective formulation*: The core idea in the multi-objective approach is to use an additional helper objective function that handles robustness related to uncertainty in the problem's parameters and/or decision variables. Thus we can solve a mono-objective problem by means of a multi-objective approach by adding robustness as a new objective to the problem at hand. The same idea could be used to handle robustness in a multi-objective problem; however in this case the problem's dimensionality would increase. Another reason to separate fitness from robustness is that using the expected fitness function as a basis for robustness is not sufficient in some cases [13], [30]. With expected fitness as the only objective, positive and negative deviations from the true fitness can cancel each other in the neighborhood of the considered solution. Thus, a solution with high fitness variance may be wrongly considered to be robust. For this reason, it may be advantageous to consider expected fitness and fitness variance as separate additional optimization criteria, which allows searching for solutions with different trade-offs between performance and robustness.

Robustness has a cost in terms of the loss in optimality, termed the *price of robustness*, or sometimes *robustness cost*. This is usually expressed as the ratio between the gain in robustness and the loss in optimality. Robustness handling methods have been successfully applied in several engineering disciplines such as scheduling, electronic engineering and chemistry (cf. [10] for a comprehensive survey).

3.2 Multi-Objective Robust Optimization for Software Refactoring

3.2.1 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 refactoring operations where the goal of applying the sequence to a software system S is typically to minimize the number of code smells in S. As outlined in the Introduction, in a real-world setting code smell severity and class importance are not certainties. A refactoring sequence that resolves the smells that one developer rates as severe may not be viewed as effective by another developer with a different outlook on smells. Similarly, a refactoring sequence that fixes the smells in a class that is subsequently deleted in the next commit is not of much value [12].

To address these issues, we propose a robust formulation of the refactoring problem that takes class importance and smell severity into account. Consequently, we have two objective functions to be maximized in our problem formulation: (1) the quality of the system to refactor, i.e., minimizing the number of code smells, and (2) the robustness of the refactoring solutions in relation to uncertainty in the severity level of the code smells and in the importance of the classes that contain the code smells. Analytically speaking, the formulation of the robust refactoring problem can be stated as follows:

$$\text{Maximize} \begin{cases} f_1(x,S) = NCCS(x,S)/NDCS(S) \\ f_2(x,S) = \sum_{i=1}^{NCCS} [SmellSeverity(CCS_i,x,S) + Importance(CCS_i,x,S)] \end{cases}$$
$$\text{subject to} \quad x = (x_1,\ldots,x_n) \in X$$

where X is the set of all legal refactoring sequences starting from S, x_i is the i-th refactoring in the sequence x, $NCCS(x,S)$ is the *Number of Corrected Code Smells* after applying the refactoring solution x on the system S, $NDCS$ is the *Number of Detected Code-Smells* prior to the application of solution x to the system S, CCS_i is the i-th Corrected Code Smell, $SmellSeverity(CCS_i, x, S)$ is the severity level of the i-th corrected code smell related to the execution of x on S, and $Importance(CCS_i, x, S)$ is the importance of the class containing the i-th code smell corrected by the execution of x on S.

The smell's severity level is a numeric quantity, varying between 0 and 1, assigned by the developer to each code smell type (e.g., blob, spaghetti code, functional decomposition, etc.). We define the class importance of a code smell as follows:

$$Importance(CCS_i,x,S) = \frac{(NC/MaxNC(S)) + (NR/MaxNR(S)) + (NM/MaxNM(S))}{3}$$

such that $NC/NR/NM$ correspond respectively to the *Number of Comments/Relationships/Methods* related to the CCS_i and $MaxNC/MaxNR/MaxNM$ correspond respectively to the *Maximum Number of Comments/Relationships/Methods* of any class in the system S. There are of course many ways in which class importance could be measured, and one of the advantages of the search-based approach is that this definition could be easily replaced with a different one. In summary, the basic idea behind this work is to maximize the resistance of the refactoring solutions to perturbations in the severity levels and class importance of the code smells while maximizing simultaneously the number of corrected code smells. These two objectives are in conflict with each other since the quality of the proposed refactoring solution usually decreases when the environmental change (smell severity and/or class importance) increases. Thus, the goal is to find a good compromise between (1) quality and (2) robustness. This compromise is directly related to *robustness cost*, as discussed above. In fact, once the bi-objective trade-off front (quality, robustness) is obtained, the user can navigate through this front in order to select his/her preferred refactoring solution. This is achieved through sacrificing some degree of solution quality while gaining in terms of robustness. In this way, the user can seek his/her preferred solution based on the robustness cost metric corresponding to the loss in terms of quality for achieving robustness.

3.2.2 The Solution Approach

The solution approach proposed in this paper lies within the SBSE field. We use the well-known multi-objective evolutionary

algorithm NSGA-II [16] to try to solve the robust refactoring problem. As noted by Harman et al. [27], a generic algorithm like NSGA-II cannot be used 'out of the box' – it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt NSGA-II to our problem, the required steps are to create: (1) solution representation, (2) solution variation and (3) solution evaluation. We examine each of these in the coming paragraphs.

Solution representation. As defined in the previous section, a solution comprises a sequence of *n* refactoring operations applied to certain elements in the source code under transformation. To represent a candidate solution (individual/chromosome), we use a vector-based representation. Each dimension of the vector represents a refactoring operation where the order of application of the refactoring operations corresponds to their positions in the vector. The standard approach of pre- and post-conditions [23], [41] is used to ensure that the refactoring operation can be applied while preserving program behaviour. For each refactoring operation, a set of controlling parameters (e.g., actors and roles as illustrated in Table 1) is randomly picked from the program to be refactored. Assigning randomly a sequence of refactorings to certain code fragments generates the initial population. An example of a solution is given in Figure 2 containing 3 refactorings. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles they play to perform the refactoring operation. An actor can be a package, class, field, method, parameter, statement or variable. Table 1 depicts, for each refactoring, its involved actors and its role.

Solution variation. In a search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards better 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 respect the refactoring sequence length limits by eliminating randomly some refactoring operations if necessary. 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 a list of possible refactorings. These two variation operators have already demonstrated good performance when tackling the refactoring problem [49][42].

Solution evaluation. Each refactoring sequence in the population is executed on the system *S*. For each sequence, the solution is evaluated based on the two objective functions (quality and robustness) defined in the previous section. Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) refactoring solutions. By definition, a solution *x* Pareto-dominates a solution *y* if and only if *x* is at least as good as *y* in all objectives and strictly better than *y* in at least one objective. The fitness of a particular solution in NSGA-II [16] corresponds to a couple (*Pareto Rank*, *Crowding distance*). In fact, NSGA-II classifies the population individuals (of parents and children) into different layers, called non-dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then discarded temporarily. This process is repeated until the entire population is classified with the domination metric. After that, a diversity measure, called *crowding distance*, is assigned front-wise to each individual. The crowding distance is the average side length of the cuboid formed by the nearest neighbors of the considered solution. Once each solution is assigned its Pareto rank, based on refactoring quality and robustness to change in terms of class importance and smell severity levels, in addition to its crowding distance, mating selection and environmental selection are performed. This is based on the crowded comparison operator that favors solutions having better Pareto ranks and, in case of equal ranks, it favors the solution having larger crowding distance. In this way, convergence towards the Pareto optimal bi-objective front (quality, robustness) and diversity along this front are emphasized simultaneously.

Table 1. Refactoring types and their involved actors and roles.

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

Inline_Class (Student, Person)
Pull_Up_Method (salary, Professor, Person)
Move_Method (grade, Registration, Student)

Figure 2. A sample refactoring solution.

The basic iteration of NSGA-II consists in generating an offspring population (of size N) from the parent one (of size *N* too) based on variation operators (crossover and mutation) where the parent individuals are selected based on the crowded comparison operator. After that, parents and children are merged into a single population *R* of size 2*N*. The parent population for the next generation is composed of the best non-dominated fronts. When including the last front, there are usually not enough available slots for all its members. Hence, based on the crowded comparison operator, solutions having largest crowding distances from the last front are included in the new parent population and the remaining ones are discarded. This process continues until the satisfaction of a stopping criterion. The output of NSGA-II is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the user will select his/her preferred refactoring solution.

4. DESIGN OF THE EMPIRICAL STUDY

This section describes our empirical study including the research questions to address, the open source systems examined, evaluation metrics considered in our experiments and the statistical tests results.

4.1 Research Questions

We defined six research questions that address the applicability, performance in comparison to existing refactoring approaches, and the usefulness of our robust multi-objective refactoring. The six research questions are as follows:

RQ1: Search Validation. To validate the problem formulation of our approach, we compared our NSGA-II formulation with Random Search. If Random Search outperforms an intelligent search method

thus we can conclude that our problem formulation is not adequate.

Since outperforming a random search is not sufficient, the next four questions are related to the comparison between our proposal and the state-of-the-art refactoring approaches.

RQ2.1: How does NSGA-II perform compared to another multi-objective algorithm in terms of robustness cost, etc.? It is important to justify the use of NSGA-II for the problem of refactoring under uncertainties. We compare NSGA-II with another widely used multi-objective algorithm, MOPSO (Multi-Objective Particle Swarm Optimization), [35] using the same adaptation (fitness functions).

RQ2.2: How do robust, multi-objective algorithms perform compared to mono-objective Evolutionary Algorithms? A multi-objective algorithm provides a trade-off between the two objectives where the developers can select their desired solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of both objectives and generates as output only one refactoring solution. This comparison is required to ensure that the refactoring solutions provided by NSGA-II and MOPSO provide a better trade-off between quality and robustness than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.

RQ2.3: How does NSGA-II perform compare to existing search-based refactoring approaches? Our proposal is the first work that treats the problem of refactoring under uncertainties. A comparison with existing search-based refactoring approaches [33], [40] is helpful to evaluate the cost of robustness of our proposed approach.

RQ2.4: How does NSGA-II perform compared to existing refactoring approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based refactoring approaches, developers will consider our approach useful, if it can outperform other existing refactoring tools [54] that are not based on optimization techniques.

RQ3: Insight. Can our robust multi-objective approach be useful for software engineers in real-world setting? In a real-world problem involving uncertainties, it is important to show that a robustness-unaware methodology drives the search to non-robust solutions that are sensitive to variation in the uncertainty parameters. However when robustness is taken into account, a more robust and insensitive solution is found. Some scenarios are required to illustrate the importance of robust refactoring solutions in a real-world setting.

4.2 Software Projects Studied

In our experiments, we used a set of well-known and well-commented open-source Java projects. We applied our approach to six large and medium sized open source Java projects: Xerces-J [58], JFreeChart [55], GanttProject [53], ApacheAnt [52], JHotDraw [56], and Rhino [57]. Xerces-J is a family of software packages for parsing XML. JFreeChart is a powerful and flexible Java library for generating charts. GanttProject is a cross-platform tool for project scheduling. ApacheAnt is a build tool and library specifically conceived for Java applications. JHotDraw is a GUI framework for drawing editors. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Table 2 provides some descriptive statistics about these six programs. We selected these systems for our validation because they range from medium to large-sized open source projects that have been actively developed over the past 10 years, and include a large number of code smells. In addition, these systems are well studied in the literature and their code smells have been detected and analyzed manually [33], [40], [5], [42]. In these corpuses, the four following code smell types were identified manually: Blob (one large class monopolizes the behavior of a system or part of it, and the other classes primarily encapsulate data); Data Class (a class that contains only data and performs no processing on these data); Spaghetti Code (code with a complex and tangled control structure) and Functional Decomposition (when a class performs a single function, rather than being an encapsulation of data and functionality). We chose these code smell types in our experiments because they are the most frequent ones detected and corrected in recent studies and existing corpuses [33], [40], [5], [42].

Table 2. Software studied in our experiments.

Systems	Release	#Classes	#Smells	KLOC
Xerces-J	v2.7.0	991	66	240
JFreeChart	v1.0.9	521	57	170
GanttProject	v1.10.2	245	41	41
ApacheAnt	v1.8.2	1191	82	255
JHotDraw	v6.1	585	21	21
Rhino	v1.7R1	305	61	42

4.3 Evaluation Metrics Used

When comparing two mono-objective algorithms, it is usual to compare their best solutions found so far during the optimization process. However, this is not applicable when comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. For this reason, we use the three following performance indicators [33] when comparing NSGA-II and MOPSO:

−*Hypervolume* (*IHV*) corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity. The reference point used in this study corresponds to the nadir point [6] of the Reference Front (*RF*), where the Reference Front is the set of all non-dominated solutions found so far by all algorithms under comparison.

−*Inverse Generational Distance* (*IGD*) is a convergence measure that corresponds to the average Euclidean distance between the Pareto front Approximation *PA* provided by the algorithm and the reference front *RF*. The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbour in *RF*. Lower values for this indicator mean better performance (convergence).

−*Contribution* (*IC*) corresponds to the ratio of the number of non-dominated solutions the algorithm provides to the cardinality of *RF*. Larger values for this metric mean better performance.

In addition to these three multi-objective evaluation measures, we used these other metrics mainly to compare between mono-objective and multi-objective approaches defined as follows:

−*Quality: number of Fixed Code-Smells* (*FCS*) is the number of code smells fixed after applying the best refactoring solution.

−*Severity of fixed Code-Smells* (*SCS*) is defined as the sum of the severity of fixed code smells:

$$SCS(S) = \sum_{i=1}^{k} SmellSeverity(d_i)$$

where k is the number of fixed code smells and *SmellSeverity* corresponds to the severity (value between 0 and 1) assigned by the developer to each code smell type (blob, spaghetti code, etc.). In our experiments, we use these severity scores 0.8, 0.6, 0.4 and 0.3 respectively for blob, spaghetti code, functional decomposition and data class.

−*Importance of fixed Code-Smells* (*ICS*) is defined using three metrics (number of comments, number of relationships and number of methods) as follows:

$$ICS(S) = \sum_{i=1}^{k} \text{importance}(d_i)$$

where *importance* is as defined in Section 3.2.1.

Table 3. Best population size configurations.

System	NSGA-II	MOPSO	Mono-EA
Xerces-J	1000	1000	1000
JFreeChart	500	200	500
GanttProject	100	100	100
ApacheAnt	1000	1000	1000
JHotDraw	200	200	200
Rhino	100	200	200

−*Computational time* (*ICT*) is a measure of efficiency employed here since robustness inclusion may cause the search to use more time in order to find a set of Pareto-optimal trade-offs between refactoring quality and solution robustness.

4.4 Inferential statistical test methods used

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 performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [2] with a 95% confidence level ($\alpha = 5\%$). The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value of random search, MOPSO and mono-objective search results with NSGA-II ones. In this way, we could decide whether the outperformance of NSGA-II over one of each of the others (or the opposite) is statistically significant or just a random result.

To answer the first research question RQ1 an algorithm was implemented where refactorings were randomly applied at each iteration. The obtained Pareto fronts were compared for statistically significant differences with NSGA-II using *IHV*, *IGD* and *IC*.

To answer RQ2.1 we implemented a widely used multi-objective algorithm, namely multi-objective particle swarm optimization (MOPSO) and we compared NSGA-II and MOPSO using the same quality indicators used in RQ1. In addition, we used boxplots to analyze the distribution of the results and discover the knee point (best trade-off between the objectives). To answer RQ2.2 we implemented a mono-objective Genetic Algorithm [24] where one fitness function is defined as an average of the two objectives (quality and robustness). The multi-objective evaluation measures (*IHV*, *IGD* and *IC*) cannot be used in this comparison thus we considered the three metrics *FCS*, *SCS* and *ICS* defined in Section 4.3. To answer RQ2.3 we compared NSGA-II with two existing search-based refactoring approaches, Kessentini et al. [33] and O'Keeffe and Ó Cinnéide [40], where uncertainties are not taken into account. We considered the same three metrics used to answer RQ2.2. To answer RQ2.4 we used a manual refactoring plug-in [54] used by a set of developers to fix the code smells in the systems considered in our experiments. We compared the results of this tool with NSGA-II using FCS, SCS and ICS since only one refactoring solution can be proposed by [54] and not a set of "non-dominated" solutions.

To answer the last question RQ3 we considered two real-world scenarios. The first scenario involves applying randomly a set of commits collected from the history of changes of the open source systems, and evaluating the impact of these changes on the robustness of the suggested refactorings proposed by our NSGA-II algorithm and non-robust approaches. In this first scenario we used the two metrics *FCS* and *ICS* since class importance changes when new commits are applied. In the second scenario we changed slightly and randomly the severity of code smells and evaluated the impact of these changes on the quality of refactoring suggested by NSGA-II and by non-robust refactoring approaches. In this second scenario we considered the two metrics *FCS* and *SCS*.

4.5 Parameter tuning and setting

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters [3]. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each multi-objective algorithm and for each system (cf. Table 2), we performed a set of experiments using several population sizes: 50, 100, 200, 500 and 1000. The stopping criterion was set to 250,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. Each algorithm was executed 51 times with each configuration and then comparison between the configurations was performed based on *IHV*, *IGD* and *IC* using the Wilcoxon test. Table 3 reports the best configuration obtained for each couple (algorithm, system). 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. The best configurations are also shown in Table 3.

The MOPSO used in this paper is the Non-dominated Sorting PSO (NSPSO) proposed by Li [35]. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 250,000 fitness evaluations. For MOPSO, the cognitive and social scaling parameters c_1 and c_2 were both set to 2.0 and the inertia weighting coefficient w decreased gradually from 1.0 to 0.4. Since refactoring sequences usually have different lengths, we authorized the length n of number of refactorings to belong to the interval [10, 250].

Table 4. The significantly best algorithm among random search, NSGA-II and MOPSO (No sign. diff. means that NSGA-II and MOPSO are significantly better than random, but not statistically different).

Project	*IC*	*IHV*	*IGD*
Xerces-J	NSGA-II	NSGA-II	NSGA-II
JFreeChart	NSGA-II	NSGA-II	NSGA-II
GanttProject	MOPSO	No sign. diff.	MOPSO
ApacheAnt	NSGA-II	NSGA-II	NSGA-II
JHotDraw	NSGA-II	NSGA-II	NSGA-II
Rhino	No sign. diff.	NSGA-II	No sign. diff.

4.6 Result analysis

This section describes and discusses the results obtained for the different research questions of Section 4.1.

4.6.1 Results for RQ1;

We do not dwell long in answering the first research question, **RQ1**, that involves comparing our approach based on NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach.

Table 4 confirms that NSGA-II and MOPSO are better than random search based on the three quality indicators IHV, IGD and IC on all six open source systems. The Wilcoxon rank sum test showed that in 51 runs both NSGA-II and MOPSO results were significantly better than random search. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

4.6.2 Results for RQ2;

In this section, we compare our NSGA-II adaptation to the current, state-of-the-art refactoring approaches. To answer the second research question, RQ2.1, we compared NSGA-II to another widely used multi-objective algorithm, MOPSO, using the same adapted fitness function. Table 4 shows the overview of the results of the significance tests comparison between NSGA-II and MOPSO. NSGA-II outperforms MOPSO in most of the cases: 13 out of 18

Table 5. FCS, SCS and ICS median values of 51 independent runs: (a) Robust Algorithms, and (b) Non-Robust algorithms.

Systems	NSGA-II				MOPSO				Mono-EA			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	52/66	31.7	29.3	1h38	48/66	28.4	26.7	1h44	41/66	24.9	24.1	1h21
JFreeChart	49/57	29.3	27.1	1h35	44/57	24.8	21.6	1h42	34/57	21.2	19.3	1h16
GanttProject	36/41	21.6	18.4	1h28	38/41	22.9	19.3	1h26	29/41	19.2	17.5	1h03
ApacheAnt	74/82	39.8	38.1	1h45	72/82	36.2	37.3	1h53	59/82	29.1	34.2	1h27
JHotDraw	17/21	11.3	10.3	1h33	15/21	9.8	8.2	1h47	13/21	8.3	8.2	1h14
Rhino	49/61	28.6	21.3	1h31	46/61	26.1	19.3	1h43	38/61	21.3	17.1	1h05

(a)

Systems	Kessentini et al.'11				O'Keeffe et al.'08				Manual			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	53/66	28.6	27.8	1h24	53/66	26.3	25.3	1h16	54/66	28.4	25.3	N/A
JFreeChart	49/57	25.8	22.3	1h13	48/57	23.6	21.9	1h04	50/57	23.9	21.2	N/A
GanttProject	37/41	19.2	17.1	1h08	37/41	20.2	17.8	1h06	37/41	19.3	16.9	N/A
ApacheAnt	76/82	32.4	33.4	1h25	75/82	33.5	34.1	1h23	71/82	31.2	32.4	N/A
JHotDraw	18/21	9.3	9.1	1h10	17/21	9.1	9.6	1h17	19/21	9.8	8.9	N/A
Rhino	52/61	24.9	16.4	1h01	51/61	23.2	17.6	1h04	51/61	24.2	16.2	N/A

(b)

experiments (73%). MOPSO outperforms the NSGA-II approach only in GanttProject, which is the smallest open source system considered in our experiments, having the lowest number of legal refactorings available, so it appears that MOPSO's search operators make a better task of working with a smaller search space. In particular, NSGA-II outperforms MOPSO in terms of IC values in 4 out 6 experiments with one 'no significant difference' result. Regarding IHV, NSGA-II outperformed MOPSO in 5 out of 6 experiments, where only one case was not statistically significant, namely GanttProject. For IGD, the results were the same as for IC.

A more qualitative evaluation is presented in Figure 3 illustrating the box plots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for NSGA-II have smaller variability than for MOPSO. This fact confirms the effectiveness of NSGA-II over MOPSO in finding a well-converged and well-diversified set of Pareto-optimal refactoring solutions.

Next, we use all four metrics FCS, SCS, ICS and ICT to compare three robust refactoring algorithms: our NSGA-II adaptation, MOPSO, and a mono-objective genetic algorithm (Mono-EA) that has a single fitness function aggregating the two objectives. We first note that the mono-EA provides only one refactoring solution, while NSGA-II and MOPSO generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-II and MOPSO using a *knee point* strategy [53]. The knee point corresponds to the solution with the maximal trade-off between quality and robustness, i.e., a small improvement in either objective induces a large degradation in the other. Hence moving from the knee point in either direction is usually not interesting for the user [50]. Thus, for NSGA-II and MOPSO, we select the knee point from the Pareto approximation having the median IHV value. We aim by this strategy to ensure fairness when making comparisons against the mono-objective EA. For the latter, we use the best solution corresponding to the median observation on 51 runs. We use the trade-off "worth" metric proposed by Rachmawati and Srinivasan [51] to find the knee point. This metric estimates the worthiness of each non-dominated refactoring solution in terms of trade-off between quality and robustness. After that, the knee point corresponds to the solution having the maximal trade-off "worthiness" value. The results from 51 runs are depicted in Table 5(a). It can be seen that both NSGA-II and MOPSO provide a better trade-off between quality and robustness than a mono-objective EA in all six systems. For FCS, the number of fixed code smells using NSGA-II is better than MOPSO in all systems expect for GanttProject (84% of cases) and also the FCS score for NSGA-II is better than mono-EA in 100% of cases. We have the same observation for the SCS and ICS scores where NSGA-II outperforms MOPSO and Mono-EA in at least 84% of cases. Even for GanttProject, the number of fixed code smells using NSGA-II is very close to those fixed by MOPSO. The execution time of NSGA-II is invariably lower than that of MOPSO with the same number of iterations, however the execution time required by Mono-EA is lower than both NSGA-II and MOPSO. It is well known that a mono-objective algorithm requires lower execution time for convergence since only one objective is handled. In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 5(a) confirm that both multi-objective formulations are adequate and outperform the

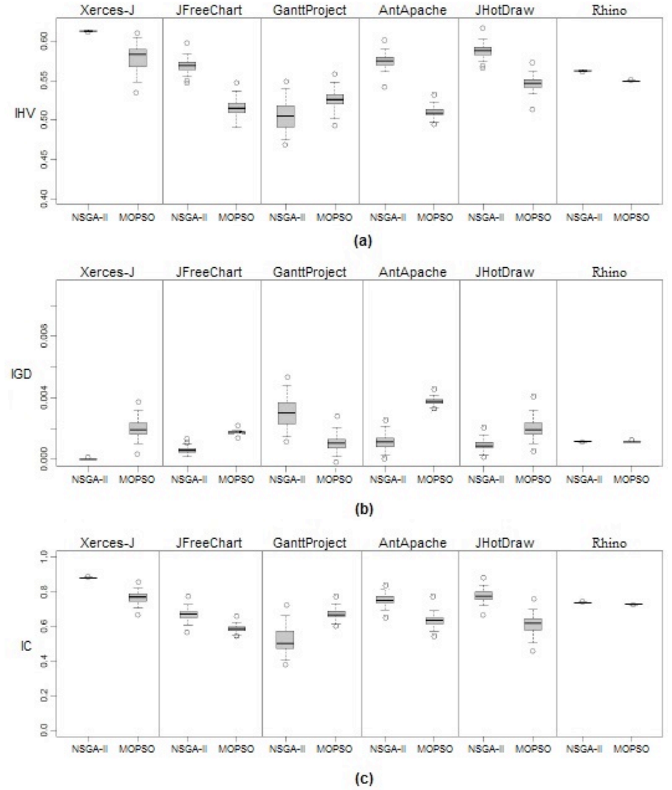

Figure 3. Boxplots using the quality measures (a) IC, (b) IHV, and (c) IGD applied to NSGAII and MOPSO.

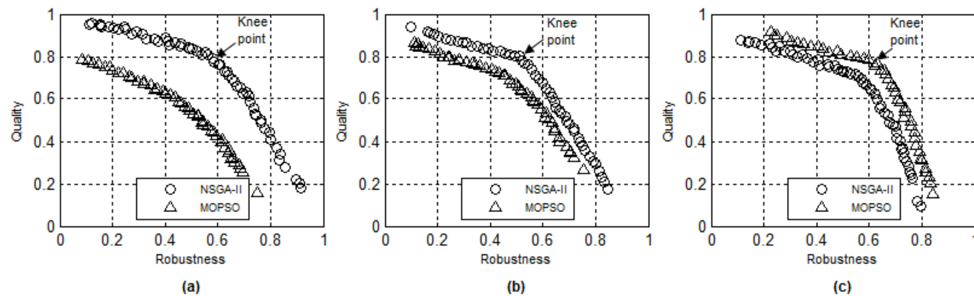

Figure 4. Pareto fronts for NSGA-II obtained on three open source systems: (a) ApacheAnt (large), (b) JHotDraw (medium) and (c) GanttProject (small).

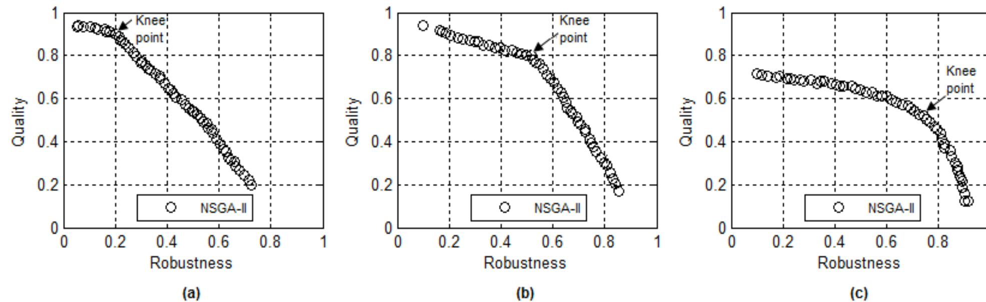

Figure 5. Pareto fronts for NSGA-II obtained on JHotDraw with different perturbation levels variation (robustness): (a) low, (b) medium and (c) high.

mono-objective algorithm based on an aggregation of two objectives (quality and robustness).

Table 5 also shows the results of comparing our robust approach based on NSGA-II with two mono-objective refactoring approaches [33], [40] and a practical refactoring technique where developers used a refactoring plug-in in Eclipse to suggest solutions to fix code smells. Kessentini et al. [33] used genetic algorithms to find the best sequence of refactoring that minimizes the number of code smells while O'Keeffe and Ó Cinnéide [40] used different mono-objective algorithms to find the best sequence of refactorings that optimize a fitness function composed of a set of quality metrics. In Ouni et al. [42], the authors ask a set of developers to fix manually the code smells in a number of open source systems including those that we are considering in our experiments. It is apparent from Table 5 that our NSGA-II adaptation outperforms mono-objective approaches in terms of smell-fixing ability (FCS) in only 11% of cases. However, our NSGA-II adaptation outperforms all the mono-objective and manual approaches in 100% of experiments in terms of the two robustness metrics, SCS and ICS. This is can be explained by the fact that NSGA-II aims to find a compromise between both quality and robustness however the remaining approaches did not consider robustness but only quality. Thus, NSGA-II sacrifices a small amount of quality in order to improve robustness. Furthermore, the number of code smells fixed by NSGA-II (277) is very close to the number fixed by the mono-objective and manual approaches (the best being Kessentini et al. [33] that fixed a total of 285 code smells), so the sacrifice in solution quality is quite small. When comparing NSGA-II with the remaining approaches we considered the best solution selected from the Pareto-optimal front using the knee point-based strategy described above. To answer RQ2.3 and RQ2.4, the results of Table 5(b) support the claim that our NSGA-II formulation provides a good trade-off between robustness and quality, and outperforms on average the state of the art of refactoring approaches, both search-based and manual, with a low robustness cost.

4.6.3 Results for RQ3;

Figure 4 depicts the different Pareto surfaces obtained on three open source systems (Apache Ant, JHotDraw and Gantt Project) using NSGA-II to optimize quality and robustness. Due to space limitations, we show only some examples of the Pareto-optimal front approximations obtained which differ significantly in terms of size. Similar fronts were obtained on the remaining systems. The 2-D projection of the Pareto front helps software engineers to select the best trade-off solution between the two objectives of quality and robustness based on their own preferences. Based on the plots of Figure 4, the engineer could degrade quality in favor of robustness while controlling visually the robustness cost, which corresponds to the ratio of the quality loss to the achieved robustness gain. In this way, the user can select the preferred robust refactoring solution to realize.

One striking feature about all the three plots is that starting from the highest quality solution the trade-off between quality and robustness is in favor of quality, meaning that the quality degrades slowly with a fast increase in robustness up to the knee point, marked in each figure. Thereafter, there is a sharp drop in quality with only a small increase in robustness. It is very interesting to note that this property of the Pareto-optimal front is apparent in all the problems considered in this study. It is likely that a software engineer would be drawn to this knee point as the probable best trade-off between quality and robustness. Without any robustness consideration in the search process, one would obtain the highest quality solution all the time (which is not robust at all), but Figure 4 shows how a better robust solution can be obtained by sacrificing just a little in quality.

Figure 5 shows the impact of different levels of perturbation on the Pareto-optimal front. Our approach takes as input as the maximum level of perturbation applied in the smell severity and class importance at each iteration during the optimization process. A high level of perturbation generates more robust refactoring solutions than those generated with lower variations, but the solution quality in this case will be higher. As described by Figure 4, the software engineers can choose the level of perturbation based on his/her preferences to prioritize quality or robustness. Although the Pareto-optimal front changes depending on the perturbation level, but there still exists a knee point, which makes the decision making by a software engineer easier in such problems.

To answer RQ3 more adequately, we considered two real-world scenarios to justify the importance of taking into consideration robustness when suggestion refactoring solutions. In the first scenario, we modified the degree of severity of the four types of code smells over time and we evaluated the impact of this variation on the robustness of our refactoring solution in terms of smell severity

(SCS). This scenario is motivated by the fact that there is no general consensus about the severity score of detected code smells thus software engineers can have divergent opinions about the severity of detected code smells. Figure 6 shows that our NSGA-II approach generates robust refactoring solutions on the Ant Apache system in comparison to existing state of the art refactoring approaches. In fact, the more the variation in severity increases over time the more the refactoring solutions provided by existing approaches become non-robust. Thus, our multi-objective approach enables the most severe code smells to be corrected even with slight modifications in the severity scores. The second scenario involved applying randomly a set of commits, collected from the history of changes of the open source systems [43], and evaluating the impact of these changes on the robustness of suggested refactoring proposed by our NSGA-II algorithm and non-robust approaches [33], [40], [54]. As depicted in Figure 7, the application of new commits modifies the importance of classes in the system containing code smells and the refactoring solutions proposed by mono-objective and manual approaches become ineffective. However, in all the scenarios it is clear that our refactoring solutions are still robust and fixing code smells in most of important classes in the system even with high number of new commits (more than 40 commits).

We also compared the refactoring solution at the knee-point (robust) for ApacheAnt with the best refactoring solution that maximizes only the quality (non-robust) to understand why the former solution is robust in both scenarios. We found that the knee-point solution rectified some code-smells that were not very risky and not located in important classes but these code-smells become more important after new commits. Thus, we can conclude that the simulation of changes in both importance and severity helps our NSGA-II to predict some future changes and adapt the best solutions according to that. Hence we conclude that RQ3 is affirmed and that the robust multi-objective approach has value for software engineers in a real-world setting.

4.7 Threats to validity

Following the methodology proposed by Wohlin et al. [50], there are four types of threat that can affect the validity of our experiments. We consider each of these in the following paragraphs.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used the Wilcoxon rank sum test with a 95% confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant.

Internal validity is concerned with the causal relationship between the treatment and the outcome. When we observe an increase in robustness, was it caused by our multi-objective refactoring approach, or could it have occurred for another reason? We dealt with internal threats to validity by performing 51 independent simulation runs for each problem instance. This makes it highly unlikely that the observed increased in robustness was caused by anything other than the applied multi-objective refactoring approach.

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as IHV, ICT etc. that are widely accepted as good proxies for quality. The notion of class importance we use in this paper is new and so constitutes a possible threat to construct validity. However, the formula we use for class importance is a routine size measure so, while many other definitions are possible, we consider the risk small that another formulation would yield very different results. We also assume that code smell severity is assigned on a per-type basis, so e.g., all blobs have the same severity. In reality, a developer would probably want to assign different blob instances different severities. While this is a weakness in our model, we do not anticipate that very different results would be obtained using per-

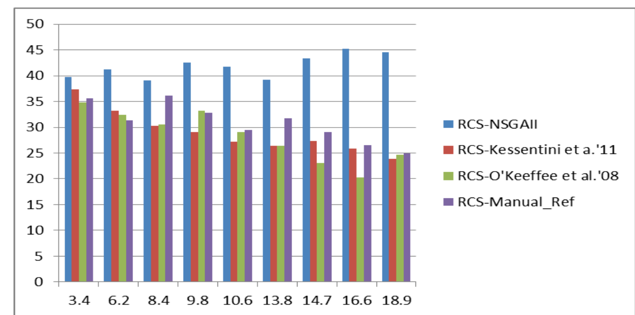

Figure 6. The impact of code smells severity variations on the robustness of refactoring solutions for ApacheAnt proposed by NSGA-II, [33], [40] and [54]

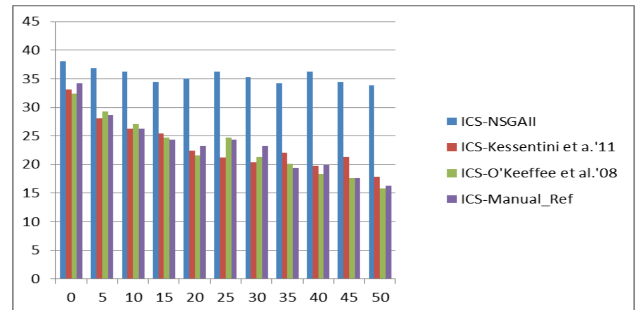

Figure 7. The impact of class importance variation on the robustness of refactoring solutions for Apache Ant proposed by NSGA-II, [33], [40] and [54].

instance model.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on six different widely used open-source systems belonging to different domains and with different sizes, as described in table 4. 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.

5. RELATED WORK

We start by summarizing existing manual and semi-automated approaches to software refactoring. The best-known work on manual refactoring is that of Martin Fowler's [23]. He provides a non-exhaustive list of code smells in source code, and, for each code smells, a particular list of possible refactorings are suggested to be applied by software maintainers manually. The first work on automated refactoring is that of Opdyke [41], who proposes the definition and the use of pre- and post-conditions with invariants to ensure that refactorings preserve the behavior of the software. Behavior preservation is thus based on the verification/satisfaction of a set of pre and post-conditions expressed in first-order logic. Opdyke's work has been the foundation for almost all subsequent automated refactoring approaches.

Most of the existing approaches are based on quality metrics improvement to deal with refactoring. Sahraoui et al. propose an approach to detect opportunities of code transformations (i.e., refactorings) based on the study of the correlation between certain quality metrics and refactoring changes [46]. To this end, different rules are defined as a combination of metrics/thresholds to be used as indicators for detecting code smells and refactoring opportunities. For each code smell a pre-defined and standard list of transformations are applied in order to improve the quality of the code. Other contributions in this field are based on rules that can be expressed as assertions (invariants, pre and post-condition), e.g., Kataoka et al. use of invariants to detect parts of program that require refactoring [32].

To fully automate refactoring activities, new approaches have emerged that use search-based techniques (SBSE) [27] to guide the search for better designs. These approaches cast refactoring as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. After formulating refactoring as an optimization problem, several different techniques can be applied for automating refactoring, e.g., genetic algorithms, simulated annealing, and Pareto optimality, etc. Hence, we classify those approaches into two main categories: mono-objective and multi-objective optimization approaches.

In the first category of mono-objective approaches, the majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [49] propose a single-objective optimization based-approach using genetic algorithm to suggest a list of refactorings to improve software quality. The search process uses a single fitness function to maximize a weighted sum of several quality metrics. Closely related work is that of O'Keeffe and Ó Cinnéide [40] where different local search-based techniques such as hill climbing and simulated annealing are used to implement automated refactoring guided by the QMOOD metrics suite [1]. In a more recent extension of their work, the refactoring process is guided not just by software metrics, but also by the design that the developer wishes the program to have [36].

Fatiregun et al. [20] showed how search-based transformations can be used to reduce code size and construct amorphous program slices. In recent work, Kessentini et al. [33] propose single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected on the source code. Jensen et al. [28] propose an approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. They use genetic programming and the QMOOD software metric suite [1] to identify the most suitable set of refactorings to apply to a software design. Harman et al. [26] propose a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The authors start from the assumption that good design quality results from good distribution of features (methods) among classes. Ó Cinnéide et al. [39] use multi-objective search-based refactoring to conduct an empirical investigation to assess structural cohesion metrics and to explore the relationships between them.

According to a recent SBSE survey [27], robustness has been taken into account only in two software engineering problems: the next release problem (NRP) and the software management/planning problem. Paixao and de Souza propose a robust formulation of NRP where each requirement's importance is uncertain since the customers can change it at any time [19]. In work by Antoniol et al., the authors propose a robust model to find the best schedule of developers' tasks where different objectives should be satisfied [1], [25]. Robustness is considered as one of the objectives to satisfy. In this paper, for the first time, we have considered robustness as a separate objective in its own right.

6. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel formulation of the refactoring problem that takes into account the uncertainties related to code smell correction in the dynamic environment of software development where code smell severity and class importance cannot be regarded as fixed. Code smell severity will vary from developer to developer and the importance of the class that contains the smell will vary as the code base itself evolves. We have reported the results of an empirical study of our robust technique compared to different existing approaches [33], [40], [54], and the results obtained have provided evidence to support the claim that our proposal enables the generation of robust refactoring solutions without a high loss of quality based on a benchmark of six large open source systems.

Our consideration of robustness as a separate objective has revealed an interesting feature of the refactoring problem in general. In our experiments, the trade-off between quality and robustness resulted in a knee solution in every case. From the highest quality solution to the knee point, the trade-off is in favor of quality, while after the knee point quality degrades more quickly than robustness. Based on this observation, we can recommend the knee solution to the software engineer as the most likely quality-robustness trade-off solution to consider.

Future work involves extending our approach to handle additional code smell types in order to test further the general applicability of our methodology. In this paper, we focused on the use of a structural metric to estimate class importance, but this can be extended to consider also the pattern of repository submits to achieve another perspective on class importance. In a similar vein, our notion of smell severity assumes each smell type has a certain severity, but a more realistic model is to allow each individual smell instance to be assigned its own severity. If further experiments confirm our observation that the knee point is indeed a trademark of the quality-robustness trade-off frontier for all software refactoring problems, then it would be interesting to apply straightway a knee-finding algorithm [53] to the bi-objective problem and determine if it yields any computational benefit. In an interactive software refactoring tool, the potential speed-up might be critical to success. Overall the use of robustness as a helper objective in the software refactoring task opens up a new direction of research and application with the possibility of finding new and interesting insights about the quality and severity trade-off in the refactoring problem.

7. REFERENCES

[1] Antoniol, G., Di Penta, M. and Harman, M. 2004. A Robust Search-Based Approach to Project Management in the Presence of Abandonment, Rework, Error and Uncertainty. In *Proceedings of the Software Metrics, 10th International Symposium* (METRICS '04). IEEE Computer Society, Washington, DC, USA, 172-183.

[2] Arcuri, A. and Briand, L. C. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 1-10. DOI=10.1145/1985793.1985795.

[3] Arcuri, A. and Fraser, G. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering*. vol. 18, no. 3.

[4] Bansiya, J. and Davis, C. 2002. A hierarchical model for object-oriented design quality assessment. In *Proceedings of IEEE Transactions on Software Engineering* (TSE'02). vol. 28, 4–17.

[5] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R. and Strollo, O. 2012. When does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation* (SCAM'12). 104-113.

[6] Bechikh, S., Ben Said, L. and Ghédira, K. 2010. Estimating Nadir Point in Multi-objective Optimization using Mobile Reference Points. In *Proceedings of the IEEE Congress on Evolutionary Computation* (CEC'10). 2129–2137.

[7] Bechikh, S., Ben Said, L. and Ghédira, K. 2011. Searching for Knee Regions of the Pareto Front using Mobile Reference Points. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*. vol. 15, no. 9, 1807–1823.

[8] Bennett, K. H. and Rajlich, V. T. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on*

The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 73-87. DOI=10.1145/336512.336534.

[9] Beyer, H-G. 2004. Actuator noise in recombinant evolution strategies on general quadratic fitness models. In *Genetic and Evolutionary Computation Conference* (GECCO'04). 654–665.

[10] Beyer, H-G. and Sendhoff, B. 2007. Robust optimization – A comprehensive survey. *Computer Methods in Applied Mechanics and Engineering*. vol. 196, no. 33–34, 3190–3218.

[11] Branke, J. 1998. Creating robust solutions by means of evolutionary algorithms. In *Parallel Problem Solving from Nature* (PPSN'98). LNCS. Springer-Verlag, 119–128.

[12] Chatzigeorgiou, A. and Manakos, A. 2013. Investigating the evolution of code smells in object-oriented systems, *Innovations in Systems and Software Engineering*. NASA Journal. DOI= 10.1007/s11334-013-0205-z.

[13] Das, I. 2000. Robustness optimization for constrained nonlinear programming problem. *Engineering Optimization*. vol. 32, no. 5, 585–618.

[14] Deb, K. and Gupta, H. 2006. Introducing robustness in multi-objective optimization. *Evolutionary Computation Journal,* vol. 14, no. 4. 463-494.

[15] Deb, K. and Gupta, S. 2011. Understanding knee points in bi-criteria problems and their implications as preferred solution principles. *Engineering Optimization*. vol. 43, no. 11, 1175-1204.

[16] Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, vol.6, no.2, 182-197. DOI= 10.1109/4235.996017.

[17] Du Bois, B., Demeyer, S. and Verelst, J. 2004. Refactoring—Improving Coupling and Cohesion of Existing Code. In *Proceedings of the Working Conference on Reverse Engineering* (WCRE'04). 144-151.

[18] Erlikh, L. 2000. Leveraging legacy system dollars for e-business. IT Professional. vol. 02. no. 3. 17–23.

[19] Esteves Paixao, M-H., De Souza, J-T. 2013. A scenario-based robust model for the next release problem. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'13).

[20] Fatiregun, D., Harman, M. and Hierons, R. 2004. Evolving transformation sequences using genetic algorithms. In *Proceedings of the 4th International Working Conference on Source Code Analysis and Manipulation* (SCAM'04). IEEE Computer Society Press. 65–74.

[21] Fenton, N. and Pfleeger, S. L. 1997. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. London, UK, International Thomson Computer Press.

[22] Ferrucci, F., Harman, M., Ren, J. and Sarro, F. 2013. Not going to take this anymore: multi-objective overtime planning for software engineering projects. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 462-471.

[23] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts D. 1999. *Refactoring – Improving the Design of Existing Code*, 1st ed. Addison-Wesley.

[24] Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co. Boston, MA, USA.

[25] Gueorguiev, S., Harman, M. and Antoniol, G. 2009. Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (GECCO '09). ACM, New York, NY, USA, 1673-1680. DOI=10.1145/1569901.1570125.

[26] Harman, M. and Tratt, L. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'07). 1106-1113.

[27] Harman, M., Mansouri, A. and Zhang, Y. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45, 1, Article 11 (December 2012), 61 pages. DOI=10.1145/2379776.2379787.

[28] Jensen, A. and Cheng, B. 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'10). ACM, New York, NY, USA, 1341-1348. DOI=10.1145/1830483.1830731.

[29] Jin, Y. and Branke, J. 2005. Evolutionary optimization in uncertain environments – A survey. *IEEE Transactions on Evolutionary Computation*. vol. 9, no. 3, 303–317.

[30] Jin, Y. and Sendhoff, B. 2003. Tradeoff between performance and robustness: An evolutionary multiobjective approach. In *international conference on Evolutionary Multi-Criterion Optimization* (EMO'03). 237–251.

[31] Kataoka, Y., Ernst, M. D., Griswold, W. G. and Notkin, D. 2001. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance* (ICSM'01). 736–743.

[32] Kataoka, Y., Ernst, M. D., Griswold, W. G. and Notkin, D. 2001. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance* (ICSM'01). 736–743.

[33] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M. and Ouni, A. 2011. Design Defects Detection and Correction by Example. In *Proceedings of the 19th International Conference on Program Comprehension* (ICPC'11). 81-90.

[34] Kim, M., Notkin, D., Grossman, D. and Wilson Jr, G. 2012. Identifying and Summarizing Systematic Code Changes via Rule Inference. In *Proceedings of the IEEE Transactions on Software Engineering* (TES'09). 45-62. DOI=10.1109/TSE.2012.16.

[35] Li, X. 2003. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation* (GECCO'03). 37-48.

[36] Moghadam, I. H. and Ó Cinnéide, M. 2012. Automated Refactoring using Design Differencing, In *Proceedings of European Conference on Software Maintenance and Reengineering* (ECSM'12).

[37] Moha, N., Guéhéneuc, Y-G., Duchien, L. and Meur, A-F. L. 2009. DECOR: A method for the specification and detection of code and design smells. In *Proceedings of IEEE Transaction in Software Engineering* (TES'09). 20–36.

[38] Monaci, M., Pferschy, U. and Serafini, P. 2011. On the robust knapsack problem. In *Proceeding of the 10th Cologne-Twente Workshop on graphs and combinatorial optimization* (Villa Mondragone, Frascati, Italy, June 14-16, 2011) CTW '11. 207-210.

[39] Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S. and Moghadam, I. H. 2012. Experimental Assessment of Software Metrics Using Automated Refactoring. In *Proceedings of the Empirical Software Engineering and Management* (ESEM'12). 49-58.

[40] O'Keeffe, M. and Ó Cinnéide, M. 2008. Search-based Refactoring for Software Maintenance. *Journal of Systems and Software*. vol. 81, no. 4, 502–516.

[41] Opdyke, W. F. 1992. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Doctoral Thesis. University of Illinois at Urbana-Champaign.

[42] Ouni, A., Kessentini, M., Sahraoui, H. and Boukadoum, M. 2012. Maintainability Defects Detection and Correction: A Multi-Objective Approach. *Journal of Automated Software Engineering*. Springer. vol. 20, no. 1, 47-79. DOI= 10.1007/s10515-011-0098-8

[43] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. and Poshyvanyk, D. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2013).

[44] Qayum, F. and Heckel, R. 2009. Local search-based refactoring as graph transformation. In *Proceedings of 1st International Symposium on Search Based Software Engineering*. 43–46.

[45] Rachmawati, L. and Srinivasan, D. 2009. Multiobjective Evolutionary Algorithm with Controllable Focus on the Knees of the Pareto Front. In *Proceedings of IEEE Transactions on Evolutionary Computation*. vol. 13, no. 4, 810-824.

[46] Sahraoui, H., Godin, R. and Miceli, T. 2000. Can Metrics Help to Bridge the Gap between the Improvement of OO Design Quality and its Automation? In *Proceedings of the International Conference on Software Maintenance* (ICSM'00). IEEE Computer Society. 154-162.

[47] Selfridge, P. 1990. Integrating Code Knowledge with a Software Information System. In *Proceedings of the 1990 Knowledge-Based Software Engineering Conference*. Hoebel, L. ed. Rome Laboratory Technical Report. 183-195.

[48] Selfridge, P. 1990. Integrating Code Knowledge with a Software Information System, In *Proceedings of the Knowledge-Based Software Assistant Conference*. 183-195.

[49] Seng, O., Stammel, J. and Burkhart, D. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO'06). 1909–1916.

[50] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B. and Wesslén, A. 2012. *Experimentation in Software Engineering*. Springer.

[51] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M. and da Fonseca, V. G. 2003. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transaction on Evolutionary Computation*. vol. 7, no. 2, 117-132.

[52] (ApacheAnt). [Online]. Available: http://ant.apache.org

[53] (GanttProject). [Online]. Available: www.ganttproject.biz

[54] (JDeodorant). [Online]. Available: http://jdeodorant.com

[55] (JFreeChart). [Online]. Available: http://www.jfree.org/jfreechart/

[56] (JHotDraw). [Online]. Available: http:// www.jhotdraw.org

[57] (Rhino). [Online]. Available: https://developer.mozilla.org/en-US/docs/Rhino

[58] (Xerces-J). [Online]. Available: http://xerces.apache.org/xerces-j