A Robust Multi-objective Approach for Software Refactoring under Uncertainty

Mohamed Wiem Mkaouer[1], Marouane Kessentini[1], Slim Bechikh[1], and Mel Ó Cinnéide[2]

[1] University of Michigan, MI, USA
firstname@umich.edu
[2] Lero, University College Dublin, Ireland
mel.ocinneide@ucd.ie

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.

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. In addition, it has been shown that software maintainers spend around 60% of their time in understanding the code. To facilitate maintenance tasks, one of the widely used techniques is refactoring which improves design structure while preserving the overall functionality of the software [12].

There has been much work on different techniques and tools for refactoring [12], [23], [21], [9]. 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 [12], 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

C. Le Goues and S. Yoo (Eds.): SSBSE 2014, LNCS 8636, pp. 168–183, 2014.

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 [23], [21], [9], [17], [20].

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 [15] and classes are added/deleted/split.

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 [3], [16] 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. [8] 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.

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 [15], 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) [18], search-based refactoring [17], [20] and a refactoring tool [24] not based on heuristic search. 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.

2 Multi-objective Robust Software Refactoring

2.1 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 [3], we can face such a problem. As stated by [3], 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 developer 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 [16].

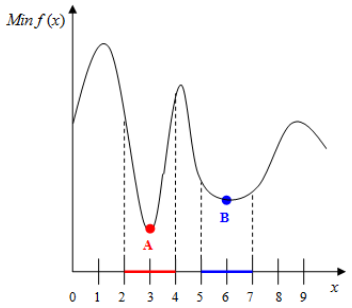

Fig. 1. Illustration of the robustness concept under uncertainty related to the decision variable x. Solution B is more robust than solution A.

2.2 Multi-objective Robust Optimization for Software Refactoring

2.2.1 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*.

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:

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

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 developer 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.

2.2.2 The Solution Approach

Solution Representation. 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 [12], 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.

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)

Fig. 2. A sample refactoring solution

Solution Variation. 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 [23][21].

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 [8] 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. 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. 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 developer will select his/her preferred refactoring solution.

3 Design of the Empirical Study

3.1 Research Questions and Systems Studied

RQ1: To validate the problem formulation of our approach, we compared our NSGA-II formulation with Random Search.

RQ2.1: How does NSGA-II perform compared to another multi-objective algorithm in terms of robustness cost, etc.?

RQ2.2: How do robust, multi-objective algorithms perform compared to mono-objective Evolutionary Algorithms?

RQ2.3: How does NSGA-II perform compare to existing search-based refactoring approaches?

RQ2.4: How does NSGA-II perform compared to existing refactoring approaches not based on the use of metaheuristic search?

RQ3: Insight. Can our robust multi-objective approach be useful for developers in real-world setting?

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, JFreeChart, GanttProject, ApacheAnt, JHotDraw, and Rhino. 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 [17], [20], [21].

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

3.2 Evaluation Metrics Used

We use the three following performance indicators [33] when comparing NSGA-II and MOPSO: *Hypervolume* (*IHV*), *Inverse Generational Distance* (*IGD*), *Contribution* (*IC*). 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 the previous section.

– *Correctness of the suggested Refactorings (CR)* is defined as the number of semantically correct refactorings divided by the total number of manually evaluated refactorings.

– *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.

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 ($\alpha = 5\%$).

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).

The MOPSO used in this paper is the Non-dominated Sorting PSO (NSPSO) proposed by Li [18]. 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 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

3.5 Results

3.5.1 Results for RQ1

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).

3.5.2 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 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 developer [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 except 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 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 [17], [20] and a practical refactoring technique where developers used a refactoring plug-in in Eclipse to suggest solutions to fix code smells. Kessentini et al. [17] used genetic algorithms to find the best sequence of refactoring that minimizes the number of code smells while O'Keeffe and Ó Cinnéide [20] 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. [21], 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. [17] 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.

3.6.3 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 developers 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 developer 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 preferred robust refactoring solution can be realized.

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 developer 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 developer 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, there still exists a knee point, which makes the decision making by a developer easier in such problems.

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.

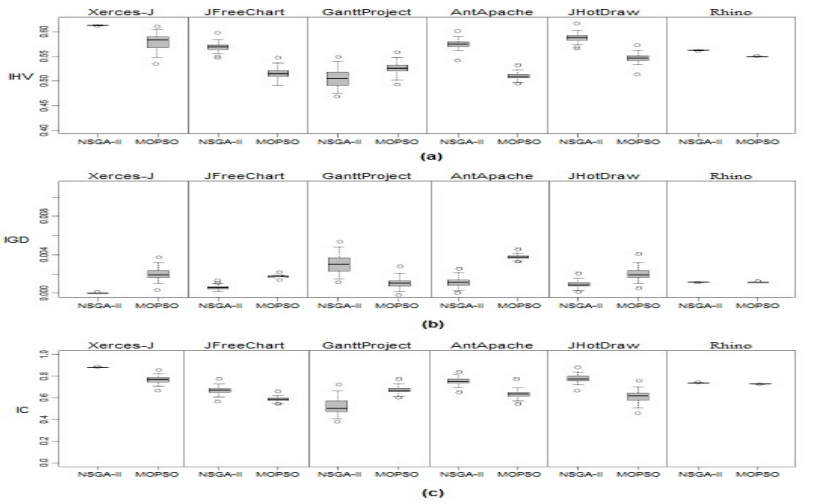

Fig. 3. Boxplots using the quality measures (a) IC, (b) IHV, and (c) IGD applied to NSGAII and MOPSO

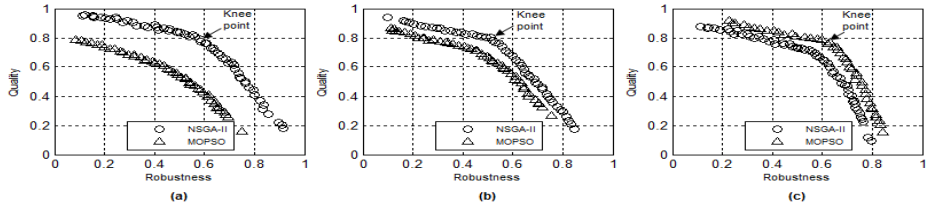

Fig. 4. Pareto fronts for NSGA-II obtained on three open source systems: (a) ApacheAnt (large), (b) JHotDraw (medium) and (c) GanttProject (small)

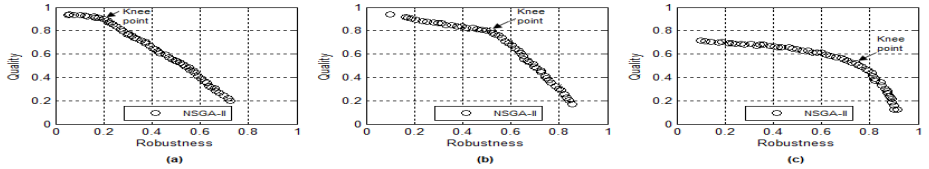

Fig. 5. Pareto fronts for NSGA-II obtained on JHotDraw with different perturbation levels variation (robustness): (a) low, (b) medium and (c) high

Figure 6 describes the manual qualitative evaluation of some suggested refactoring solutions. It is clear that results are almost similar between our proposal and existing approach in terms of the semantic coherence of suggested refactorings. We consider that a semantic precision more than 65% acceptable since most of the solutions should be executed manually by developers and our tool is a recommendation system. Thus, developers can evaluate if it is interesting or not to apply some refactorings based on their preferences and the semantic coherence.

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 developers can have divergent opinions about the severity of detected code smells. Figure 7 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 [21], and evaluating the impact of these changes on the robustness of suggested refactoring proposed by our NSGA-II algorithm and non-robust approaches [17], [20], [24]. As depicted in Figure 8, 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).

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

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

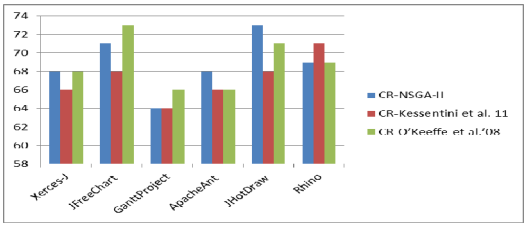

Fig. 6. The qualitative evaluation (CR) of some refactorings proposed by NSGA-II, [17] and [20]

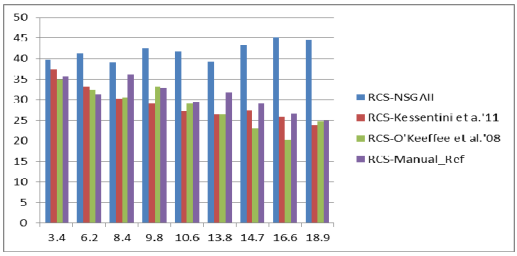

Fig. 7. The impact of code smells severity variations on the robustness of refactoring solutions for ApacheAnt proposed by NSGA-II, [17], [20] and [24]

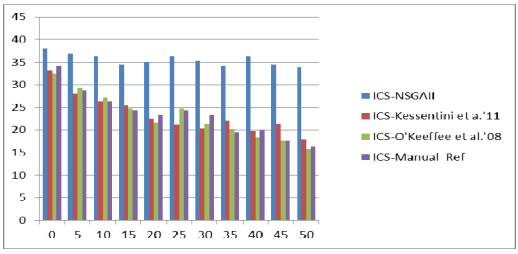

Fig. 8. The impact of class importance variation on the robustness of refactoring solutions for Apache Ant proposed by NSGA-II, [17], [20] and [24]

4 Related Work

The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [23] 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 [20] 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 [19]. In recent work, Kessentini et al. [17] 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. 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. [14] 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. [19] 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 [15], 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 [10]. 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], [13]. 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.

5 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 [17], [20], [24]. 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.

References

[1] Antoniol, G., Di Penta, M., Harman, M.: A Robust Search-Based Approach to Project Management in the Presence of Abandonment, Rework, Error and Uncertainty. In: METRICS 2004, pp. 172–183 (2004)

[2] Arcuri, A., Briand, L.C.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: ICSE 2011, pp. 1–10 (2011)

[3] Beyer, H.-G., Sendhoff, B.: Robust optimization – A comprehensive survey. Computer Methods in Applied Mechanics and Engineering 196(33-34), 3190–3218 (2007)

[4] Chatzigeorgiou, A., Manakos, A.: Investigating the evolution of code smells in object-oriented systems, Innovations in Systems and Software Engineering. NASA Journal (2013)
[5] Das, I.: Robustness optimization for constrained nonlinear programming problem. Engineering Optimization 32(5), 585–618 (2000)
[6] Deb, K., Gupta, H.: Introducing robustness in multi-objective optimization. Evolutionary Computation Journal 14(4), 463–494 (2006)
[7] Deb, K., Gupta, S.: Understanding knee points in bi-criteria problems and their implications as preferred solution principles. Engineering Optimization 43(11), 1175–1204 (2011)
[8] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
[9] Du Bois, B., Demeyer, S., Verelst, J.: Refactoring—Improving Coupling and Cohesion of Existing Code. In: WCRE 2004, pp. 144–151 (2004)
[10] Esteves Paixao, M.-H., De Souza, J.-T.: A scenario-based robust model for the next release problem. In: GECCO 2013 (2013)
[11] Ferrucci, F., Harman, M., Ren, J., Sarro, F.: Not going to take this anymore: Multi-objective overtime planning for software engineering projects. In: ICSE 2013, pp. 462–471. IEEE Press, Piscataway (2013)
[12] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley (1999)
[13] Gueorguiev, S., Harman, M., Antoniol, G.: Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In: GECCO 2009, pp. 1673–1680 (2009)
[14] Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: GECCO 2007, pp. 1106–1113 (2007)
[15] Harman, M., Mansouri, A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. (2012)
[16] Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments – A survey. IEEE Transactions on Evolutionary Computation 9(3), 303–317 (2005)
[17] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design Defects Detection and Correction by Example. In: Proceedings of ICPC 2011, pp. 81–90 (2011)
[18] Li, X.: A non-dominated sorting particle swarm optimizer for multiobjective optimization. In: GECCO 2003, pp. 37–48 (2003)
[19] Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., Moghadam, I. H.: Experimental Assessment of Software Metrics Using Automated Refactoring. In: Proceedings of the ESEM 2012, pp. 49–58 (2012)
[20] O'Keeffe, M., Ó Cinnéide, M.: Search-based Refactoring for Software Maintenance. Journal of Systems and Software, 502–516 (2008)
[21] Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M.: Maintainability Defects Detection and Correction: A Multi-Objective Approach. Journal of Automated Software Engineering, 47–79 (2012)
[22] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: Detecting Bad Smells in Source Code Using Change History Information. In: Proceedings of ASE 2013 (2013)
[23] Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of GECCO 2006, pp. 1909–1916 (2006)
[24] http://www.jdeodorant.com/