A robust multi-objective approach to balance severity and importance of refactoring opportunities

Mohamed Wiem Mkaouer[1] · **Marouane Kessentini**[1] · **Mel Ó Cinnéide**[2] · **Shinpei Hayashi**[3] · **Kalyanmoy Deb**[4]

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. Both severity and importance of identified refactoring opportunities (e.g. code smells) are difficult to estimate. In fact, due to the dynamic nature of software development, these values cannot be accurately determined in practice, leading to refactoring sequences that lack robustness. In addition, some code fragments can contain severe quality issues but they are not playing an important role in the system. 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 three objectives to maximize: quality improvements, severity and importance of refactoring opportunities to be fixed. We evaluated

✉ Marouane Kessentini
kessentiniglp@yahoo.fr

Mohamed Wiem Mkaouer
mmkaouer@umich.edu

Mel Ó Cinnéide
mel.ocinneide@ucd.ie

Shinpei Hayashi
hayashi@se.cs.titech.ac.jp

Kalyanmoy Deb
kdeb@egr.msu.edu

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

[2] University College Dublin, Dublin, Ireland

[3] Tokyo Institute of Technology, Tokyo, Japan

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

our approach using 8 open source systems and one industrial project, and demonstrated that it is significantly better than state-of-the-art refactoring approaches in terms of robustness in all the 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, better prioritization of refactoring opportunities and to carry an acceptable robustness price.

Keywords Search-based software engineering · Refactoring under uncertainty · Software quality · Robust multi-objective optimization

1 Introduction

Software evolution is an essential component of software development process. It is mainly intended to keep the software system up to the user's requirements by regularly performing a set of development activities linked to adding new features, fixing reported bugs, migrating to different environments and platforms, and other function related tasks. Eventually. As a software system evolves, the code-related changes tend to degrade the system's structure as the evolution focuses mainly on the incorporation of required features and the correction of errors on the expense of the deterioration of the system's design. Since these code changes have become inevitable as they constitute a key-role in agile methodologies, strategies need to be adapted in order to preserve the software architecture's value over changes. Therefore, software maintenance gathers software change control and management strategies that aim to maintain the software quality during its evolution. On the other hand, the new functionalities, incorporated during the evolution of a software system, exhibit a growth of its size and complexity and so it becomes harder to maintain. As a consequence, 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 (Glass 2001). To facilitate maintenance tasks, one of the widely used techniques is *refactoring* which can be defined as the restructuration of the system's architecture with the intention of improving its internal design while preserving the overall external behavior of the software (Fowler et al. 1999).

Software refactoring, as an concept, has been introduced since the early nineties by Obdyke (Opdyke 1992) and became a key artifact of the agile development processes such as Extreme Programming (XP). Fowler (Fowler et al. 1999) has identified refactoring opportunities within code fragments and provides a refactoring operations catalog that can be applied to enhance the code's structure of code while preserving its semantics. There has been much work on different refactoring techniques and tools (Du Bois et al. 2004; Kessentini et al. 2011; O'Keeffe and Ó Cinnéide 2008; Ouni et al. 2012; Seng et al. 2006). 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, 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 (Du Bois et al. 2004; Kessentini et al. 2011; O'Keeffe and Ó Cinnéide 2008; Ouni et al. 2012; Seng et al. 2006).

Most of existing approaches propose refactoring solutions without a consideration of the severity and importance of detected refactoring opportunities to fix. In fact, both severity and importance are difficult to define and estimate. The estimation scores of these factors can change during the time due to the highly dynamic nature of software development. The importance and severity of code fragments can be different after new commits introduced by developers. Thus, it is important to consider the uncertainty related to these two factors when recommending refactoring solutions. In addition, the definition of severity and importance is very subjective and depends on the developers' perception.

In this paper, we take into account two dynamic aspects as follows:

- *Code Smell Severity*: Once a list of code smells is detected, the correction techniques treat these detected defects equally by suggesting which refactorings could be applied to the code in order to eliminate, or at least reducing them. Whereas, the effects of a defect in terms of potential introduction of faults may vary depending on the type of the code smell (Hall et al. 2014). Also, many studies has been investigating the impact of each defect type on the maintenance effort (Yamashita 2012). Thus, we consider a severity level assigned to a code smell type by a developer based on his prior knowledge and his preference on prioritizing the correction of a specific type of code smell among others. 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 (Harman et al. 2012) and classes are added/deleted/extracted.

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 a novel representation of the code refactoring problem, based on *robust* optimization (Beyer and Sendhoff 2007; Jin and Branke 2005) 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. (Deb et al. 2002) 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 three objectives to maximize: quality improvements (number of fixed code smells), severity and importance of refactoring opportunities to be fixed (e.g. 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. We extended our previous work (Mkaouer et al. 2014) by considering severity and importance of refactoring opportunities as separate objectives. In addition, we are considering additional types of code smells to fix and extended our validation to 8 open source systems and one industrial project.

- The paper reports on the results of an empirical study of our robust NSGA-II technique as applied different medium and large size systems. We compared our approach to random search, multi-objective particle swarm optimization (MOPSO) (Li 2003), search-based refactoring (Kessentini et al. 2011; O'Keeffe and Ó Cinnéide 2008) and a non-search-based refactoring tool (Fokaefs et al. 2011). 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.

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach, the nature of the refactoring challenge and related work. 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. In Section 5 we conclude and suggest future research directions.

2 Software Refactoring

2.1 Background and Challenges

A code-smell is defined as bad design choices that can have a negative impact on the code quality such as maintainability, changeability and comprehensibility which could introduce bugs (Yamashita 2012). Code-smells classify shortcomings in software that can decrease software maintainability. They are also defined as structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and trigger refactoring of code (Fowler et al. 1999). Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. In fact, most of code-smells can emerge during the evolution of a system and represent patterns or aspects of software design that may cause problems in the further development and maintenance of the system. As stated by Brown et al. (Brown et al. 1998a), code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. It is easier to interpret and evaluate the quality of systems by identifying code-smells than the use of traditional software quality metrics. In fact, most of the definitions of code-smells are based on situations that are daily faced by developers. Most of the code-smells identify locations in the code that violate object-oriented design heuristics, such as the situations described by Riel (Riel 1996) and Coad et al. (Zhang et al. 2011). The 22 Code Smells identified and defined informally by Fowler (Fowler et al. 1999) aim to indicate software refactoring opportunities and 'give you indications that there is trouble that can be solved by a refactoring'. Zhang et al. (Zhang et al. 2011) identified in their survey the code-smells that attracted more attention in current literature.

Van Emden and Moonen (Van Emden and Moonen 2002) developed one of the first automated code-smell detection tools for Java programs. Mantyla studied the manner of how developers detect and analyse code-smells (Mäntylä and Lassenius 2006). Previous

empirical studies have analysed the impact of code-smells on different software maintainability factors including defects and effort (D'Ambros et al. 2010). In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Anda et al. (Anda 2007) and Marinescu et al. (Marinescu 2010). Code-smells are associated with a generic list of possible refactorings to improve the quality of software systems. In addition, Yamashita et al. (Yamashita and Moonen 2012) show that the different types of code-smells can cover most of maintainability factors. Thus, the detection of code-smells can be considered as a good alternative of the traditional use of quality metrics to evaluate the quality of software products. Brown et al. (Brown et al. 1998b) define another category of code-smells that are documented in the literature, and named anti-patterns.

In our experiments, we focus on the seven following code-smell types:

- *Blob*: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data.
- *Feature Envy* (*FE*): It occurs when a method is more interested in the features of other classes than its own. In general, it is a method that invokes several times accessor methods of another class.
- *Data Class* (*DC*): It is a class with all data and no behavior. It is a class that passively store data
- *Spaghetti Code* (*SC*): It is a code with a complex and tangled control structure.
- *Functional Decomposition* (*FD*): It occurs when a class is designed with the intent of performing a single function. This is found in a code produced by non-experienced object-oriented developers.
- *Lazy Class* (*LC*): A class that is not doing enough to pay for itself.
- *Long Parameter List* (*LPL*): Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type.

We choose these code-smell types in our work because of their high frequency, detection difficulty (Palomba et al. 2013), cover different maintainability factors and also due to the availability of code-smell instances in the studied projects. However, the proposed approach in this paper is generic and can be applied to any type of code-smells.

To fix these code smells, the idea is to restructure variables, classes and methods to facilitate future adaptations and extensions and enhance comprehension. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, NetBeans, provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However, automatically suggesting/deciding where and which refactorings to apply is still a real challenge in Software Engineering. In order to identify which parts of the source code need to be refactored, most existing work relies on the notion of code smells (e.g., Fowler's textbook (Fowler et al. 1999)) as described previously.

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.

2.2 Related Work

In this section, we start by summarizing existing manual and semi-automated approaches to software refactoring, then we focus on some studies related to use of search based software engineering techniques (mono-objective and multi-objective optimization algorithms) to address software refactoring problems. The refactoring process includes the identification of code fragments to be refactored, i.e., bad smell detection (Zhang et al. 2011), and the determination of which refactorings should be applied to the fragments (Tom Mens and Tourwe 2004). There are a lot of work for supporting such activities.

The best-known work on manual refactoring is that of Martin Fowler's (Fowler et al. 1999). 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 (Opdyke 1992), 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 semi-automated approaches are based on quality metrics improvement to deal with refactoring, Tsantails et al. (Tsantalis and Chatzigeorgiou 2009) proposed a technique for detecting the opportunities of move method refactorings so that cohesion values become higher preventing coupling values from rising. Kerievsky (Kerievsky 2004) also proposed smells and refactorings for introducing design patterns. It is prefer- able to improve method assignment as early as possible, not in the step of source code level but design because of avoiding rework of development activities in later steps. Marinescu et al. (Radu 2004) presented a metric-based approach to identify smells with detection strategies, which capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds. Oliveto et al. (Oliveto et al. 2010) proposed a method to identify occurrences of antipatterns (Brown et al. 1998c) based on numerical analysis of metric values. Moha et al. (Moha et al. 2010) proposed DECOR method and DETEX technique (Moha et al. 2010) to specify and automatically generate identification algorithms for code/design smells based on metrics and structural characteristics. They showed the detection performance of 19 automatic generated algorithms by DETEX. These approaches are rule- based whereas our technique uses search-based

optimization. Zamani et al. (Zamani and Butler 2009) proposed a method to analyze a UML model using the in- formation on stereotypes and checks a smell whether enterprise architectural patterns (EAA patterns) are applied correctly to the model or not. In (Sunye et al. 2001), model refactoring on UML class and state diagrams is formalized as a sequence of transformation operations defined in OCL. Trifu et al. (Trifu and Reupke 2007) discussed relationship between a smell and the number of directly observable indicators. They defined specifications of the design flaw including context and indicators, and a diagnosis strategy using indicators and correction strategies written in a natural language. They also presented a tool to identify design flaws. Their indicators for design flaw identification are defined as a combination of design metrics and structural information. ClassCompass (Coelho and Murphy 2007), which is an automated software de- sign critique system, has a feature to suggest design correction based on rules written in a natural language. Sahraoui et al. (Sahraoui et al. 2000) 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. 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. (Kataoka et al. 2001) use of invariants to detect parts of program that require refactoring. Compared with these approaches and tools, our approach is search-based and then does not require users and developers to define detection rules.

To fully automate refactoring activities, new approaches have emerged that use search-based techniques (SBSE) (Marinescu 2010) 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. (Seng et al. 2006) 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 (O'Keeffe and Ó Cinnéide 2008) 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 (Bansiya and Davis 2002). 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 (Moghadam and Ó Cinnéide 2012).

Fatiregun et al. (Coad and Yourdon 1991) showed how search-based transformations can be used to reduce code size and construct amorphous program slices. In recent work, Kessentini et al. (Kessentini et al. 2011) 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. (Jensen and Cheng 2010) 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 (Bansiya and Davis 2002) to identify the most suitable set of refactorings to apply to a software design. Harman et al. (Harman and Tratt 2007) 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. (Ó Cinnéide et al. 2012) use multi-objective search-based refactoring to conduct an empirical investigation to assess structural cohesion metrics and to explore the relationships between them. Bavota et al. (Palomba et al. 2013) proposed an interactive optimization approach for software remodularization. However, the study of Bavota et al. was mainly limited to refactorings at the package level (remodularization) and did not consider the severity and importance of code smells. In addition, the approach of Bavota et al. requires more interactions with the user than our approach since it is based on interactive optimization.

According to a recent SBSE survey (Harman et al. 2012), robustness has been taken into account only in two software engineering problems: the next release problem (NRP) and the software management/planning problem. Paixao et al. (Esteves Paixao and De Souza 2013) propose a robust formulation of NRP where each requirement's importance is uncertain since the customers can change it at any time (Riel 1996). In work by Antoniol et al. (Antoniol et al. 2004), the authors propose a robust model to find the best schedule of developers' tasks where different objectives should be satisfied, robustness is considered as one of the objectives to satisfy (Gueorguiev et al. 2009). In this paper, for the first time, we have considered robustness as a separate objective in its own right.

3 Multi-objective Robust Software Refactoring

3.1 Robust Optimization

In dealing with optimization problems, including software engineering ones, most researchers assume that the 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 parameters with respect to nominal values. For instance, when handling the knapsack problem (KP), which is one of the most studied combinatorial problems (Beyer and Sendhoff 2007), 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 Beyer et al. (Beyer and Sendhoff 2007), 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 (Fowler et al. 1999; Du Bois et al. 2004) 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 (Kessentini et al. 2011; Ouni et al. 2012) 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 Fig. 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 (Jin and Branke 2005):

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

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.

- *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. (Beyer 2004) 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

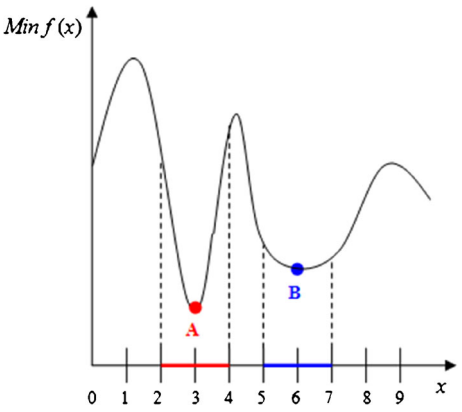


Fig. 1 Illustration of the robustness concept under uncertainty related to the decision variable x. Solution B is more robust than solution A

in order to yield more robust solutions (Das 2000). 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 (Jin and Sendhoff 2003). 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. (Beyer and Sendhoff 2007) for a comprehensive survey).

3.2 Multi-objective Robust Optimization for Software Refactoring

3.2.1 Problem Formulation

The refactoring problem, from search-based perspective, includes the exploration of a set of candidate solutions in order to determine the best one whose sequence of refactorings best satisfies the fitness function(s). A refactoring solution is a set of refactoring operations where the goal of applying the sequence to a software system S is typically to minimize the number of design defects 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 (Chatzigeorgiou and Manakos 2013).

We propose a robust formulation of the refactoring problem that separates class importance and smell severity into two different objectives. Consequently, we have three 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 the robustness of the refactoring solutions in relation to uncertainty in both (2) severity level of the code smells and (3) the importance of the classes that contain the code smells.

Analytically speaking, the formulation of the robust refactoring problem can be stated as follows:

$$\begin{aligned} &Maximize \\ &\begin{cases} f_1(x,S) = NCCS(x,S)/NDCS(S) \\ f_2(x,S) = \sum_{i=1}^{NCCS(x,S)} SmellSeverity(CCS_i,x,S) \\ f_3(x,S) = \sum_{i=1}^{NCCS(x,S)} ClassImportance(CCS_i,x,S) \end{cases} \\ &subject\ to\ \ x = (x_1,\ldots,x_n) \in X \end{aligned}$$

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, $NDCS(S)$ 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 $ClassImportance(CCS_i, x, S)$ 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. We used in our approach the widely used metrics by existing literature to estimate the importance and severity of code smells (Olbrich et al. 2010). In fact, few empirical studies were performed with software engineers to evaluate the severity and importance of several types of code smells (Olbrich et al. 2010; Fontana et al. 2015). We are taking these metrics and the severity scores as input for our approach.

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 three 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. In addition, severe quality issues may exist in code fragments that are not important for developers. Thus, the goal is to find a good compromise between these three conflicting objectives. This compromise is directly related to robustness cost, as discussed above. In fact, once the three objectives trade-off front 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 and smell severity and importance. In fact, robustness is inversely proportional to the severity and class importance, and this is how we make our algorithm robust. The approach is to use a multi-objective search algorithm to

explore the trade-off that can be obtained by ignoring the user's wishes to varying degrees. The results show this to be the case; quality can be most improved when importance and severity objective values are lower. For example, a refactoring applied to a class that is then deleted; the instantaneous quality improvement may be high, but in the project's timeline it is minimal. This makes a solid case for so-called "robust" optimization which seeks to identify good solutions that are resilient against changes in the developer's priorities.

In our robust formulation, we introduce "perturbations/variations" in the severity and importance scores of the code smells and classes at every iteration of our NSGA-II algorithm. As described later in the experiments section, the severity scores of 0.8, 0.6, 0.4, 0.3, 0.5, 0.3, and 0.2 of the different types of code represent just the initial values but these values slightly change at each iteration (a slight random increase or decrease of these scores using a variation rate parameter). These variations correspond to some artificially created changes in the environment (new code changes introduced, etc.) or priorities change or a different opinions of developers about the importance or severity of the classes.

3.2.2 The Solution Approach

Solution representation To represent a candidate solution (individual/chromosome), we use a vector-based representation which is widely adopted in the literature. According to this encoding, a solution is an array of refactoring operations where the order of their execution is accorded by their positions in the array. The standard approach of pre- and post-conditions, 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 Fig. 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 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 (Ouni et al. 2012; Seng et al. 2006).

Solution evaluation Each refactoring sequence in the population is executed on the system S. For each sequence, the solution is evaluated based on the three objective functions defined in the previous section. Since we are considering a three objectives 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

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	moved method
Inline class	class	source class, target class
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

one objective. The fitness of a particular solution in NSGA-II (Deb et al. 2002) 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

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

Fig. 2 A sample refactoring solution

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 $2N$. 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.

4 Design of the Empirical Study

This section describes our empirical study including the research questions to address, the systems examined, evaluation metrics considered in our experiments and the statistical tests results. In addition, we compared different refactoring algorithms quantitatively and qualitatively across several subject systems.

4.1 Research Questions and Systems Studied

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 other multi-objective algorithms? 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), (Li 2003) using the same adaptation (fitness functions). In addition, we compared our approach with our previous SSBSE2014 robust refactoring study (based on only 2 objectives) (Mkaouer et al. 2014).

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 (Kessentini et al. 2011; O'Keeffe and Ó Cinnéide 2008) 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 (Fokaefs et al. 2011) that are not based on optimization techniques.

RQ3: Developers' evaluation of the recommended refactorings. 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: a) exploring the trade-off of robust and qualitative solutions; and b) asking what developers think of the results.

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 eight large and medium sized open source Java projects: Xerces-J (Xerces J Online Available http://xerces.apache.org/xerces-j), JFreeChart (JFreeChart Online Available http://www.jfree.org/jfreechart/), GanttProject (GanttProject Online Available http://www.ganttproject.biz), ApacheAnt (ApacheAnt Online Available http://ant.apache.org), JHotDraw (JHotDraw Online Available http://www.jhotdraw.org), Rhino (Rhino Online Available https://developer.mozilla.org/en-US/docs/Rhino/), Log4J (Log4J Online Available http://logging.apache.org/), Nutch (Nutch Online Available http://nutch.sourceforge.net/) and JDI-Ford. 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. Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Nutch is an open source Java implementation of a search engine. Log4j is a Java-based logging utility. Table 2 provides some descriptive statistics about these eight 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 either detected and analyzed manually (Kessentini et al. 2011; O'Keeffe and Ó Cinnéide 2008; Ouni et al. 2012) or using an existing detection tool (Radu 2004). In these corpuses, the four following code smell types were identified manually (that are described in Section 2): Blob, Feature Envy (FE)

Table 2 Software studied in our experiments

Systems	Release	#Classes	#Smells	KLOC
Xerces-J	v2.7.0	991	82	240
JFreeChart	v1.0.9	521	73	170
GanttProject	v1.10.2	245	56	41
ApacheAnt	v1.8.2	1191	91	255
JHotDraw	v6.1	585	33	21
Rhino	v1.7R1	305	74	42
Log4J	v1.2.1	189	64	31
Nutch	v1.1	207	72	39
JDI-Ford	v5.8	638	88	247

, Data Class (DC), Spaghetti Code (SC), Functional Decomposition (FD), Lazy Class (LC) and Long Parameter List (LPL). 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.

We performed also a small industrial case study, based on one industrial project JDI-Ford. It is a Java-based software system that helps, our industrial partner, the Ford Motor Company, analyzes useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. This system is the main key software application used by the Ford Motor Company to improve their vehicle sales by selecting the right vehicle configuration to the expectations of customers. Several versions of JDI were proposed by software engineers at Ford during the past 10 years. Due to the importance of the application and the high number of updates performed during a period of 10 years, it is critical to make sure that all the JDI releases are within a good quality to reduce the time required by developers to introduce new features in the future. The software engineers from Ford manually evaluated the suggested refactorings based on their knowledge of the system since they are some of the original developers.

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 defined the following metrics:

– *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 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, 0.3, 0.5, 0.3, and 0.2 respectively for Blob, Spaghetti Code (SC), Functional Decomposition (FD), Lazy Class (LC), Feature Envy (FE), Data Class (DC) and Long Parameter List (LPL). We introduce "perturbations/variations" in the severity and importance scores of the code smells and classes at every iteration of our NSGA-II algorithm. Thus, these severity scores of the different types of code represent just the initial values but these values slightly change at each iteration (a slight random increase or decrease of these scores using a variation rate parameter between −0.2 and +0.2).

- *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.

- *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.

When we compared the different algorithms to NSGA-II, we used the original initial weights (before the perturbation).

4.4 Subjects

Our study involved 6 subjects from the University of Michigan and 4 software engineers from Ford Motor Company. Subjects include 1 master student in Software Engineering, 4 Ph.D. students in Software Engineering, 1 faculty member in Software Engineering, 3 junior software developers and 1 senior projects manager. All the subjects are familiar with Java development, software maintenance activities including refactoring. The experience of these subjects on Java programming ranged from 3 to 17 years. All the graduate students have an industrial experience of at least 2 years with large-scale object-oriented systems. The 6 subjects from the University of Michigan evaluated the refactoring results on the open source systems. The 4 software engineers from Ford evaluated the refactoring results only on the JDI-Ford system. They were selected, as part of a project funded by Ford, based on having similar development skills, their motivations to participate in the project and their availability. They are part of the original developers' team of the JDI system.

4.5 Parameter Tuning and Setting

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. 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 3), 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 (Li 2003). 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].

4.6 Approaches Compared

We compared our approach with different existing studies. For the mono-objective approaches, Kessentini et al. (Kessentini et al. 2011) used genetic algorithms to find the best sequence of refactoring that minimizes the number of code smells while O'Keeffe and Ó Cinnéide (O'Keeffe and Ó Cinnéide 2008) used different mono-objective algorithms to find the best sequence of refactorings that optimize a fitness function composed of a set of quality metrics. Kessentini et al. (Kessentini et al. 2011) and O'Keeffe et al. (O'Keeffe and Ó Cinnéide 2008), where uncertainties

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
Log4J	200	200	100
Nutch	100	200	150
JDI-Ford	500	600	460

are not taken into account. We also implemented a mono-objective Genetic Algorithm where one fitness function is defined as an average of the three objectives (quality, severity and importance). For the multi-objective algorithms, in Ouni et al.(Ouni et al. 2012), 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. They proposed a multi-objective approach to maximize the number of fixed defects and minimize the number of refactorings. We also compared our approach with our previous SSBSE2014 robust refactoring study (based on only 2 objectives) (Mkaouer et al. 2014) as described in the introduction. In addition, we compared our proposal to a popular design defects detection and correction tool JDeodorant (Tsantalis and Chatzigeorgiou 2009) that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them.

4.7 Results Analysis

This section describes and discusses the results obtained for the different research questions of Section 4.1. 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 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, our previous multi-objective work (Mkaouer et al. 2014) 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. The inference on the best result is done on the basis of ranking through multiple pair-wise tests.

The Wilcoxon signed-rank test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we

used the *Vargha and Delaney's A* statistics which is a non-parametric effect size measure (Vargha and Delaney 2000; Arcuri and Lionel 2011). In our context, given the different performance metrics (such as FCS, SCS and ICS), the A statistics measures the probability that running an algorithm B_1 (our robust multi-objective algorithm) yields better performance than running another algorithm B_2 (such as random search and MOPSO). If the two algorithms are equivalent, then $A = 0.5$. In our experiments, we have found the following results: a) On small and medium scale Software systems (JFreeChart and GanttProject) NSGA-II is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.9; b) On large scale Software systems (Xerces-J and JDI-Ford), NSGA-II is better than all the other algorithms with a an A effect size higher than 0.83.

4.7.1 Results for RQ1: Comparison Between NSGA-II and Random Search

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.

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).

4.7.2 Results for RQ2: Comparison with State of the Art Refactoring Approaches

In this section, we compare our NSGA-II adaptation to the current, state-of-the-art refactoring approaches. 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). Furthermore, we compare our proposal with our previous formulation (Mkaouer et al. 2014) based on two objectives using the *FCS*, *SCS*, *ICS*, *CR*, and *ICT*. To answer RQ2.2 we implemented a mono-objective Genetic Algorithm where one fitness function is defined as an average of the three objectives (quality, severity and importance). The multi-objective evaluation measures (*IHV*, *IGD* and *IC*) cannot be used in this comparison thus we considered the five metrics *FCS*, *SCS*, *ICS*, *CR*, and *ICT* defined in Section 4.3. To answer RQ2.3 we compared NSGA-II with two existing search-based refactoring approaches, Kessentini et al. (Kessentini et al. 2011) and O'Keeffe et al. (O'Keeffe and Ó Cinnéide 2008), where uncertainties are not taken into account. We considered the same metrics used to answer RQ2.2. To answer RQ2.4, we compared our proposal to a popular design defects detection and correction tool JDeodorant that does not use heuristic search techniques. We compared the results of this tool with NSGA-II using *FCS*, *SCS*, *ICS*, *CR*, and *ICT* since only one refactoring solution can be proposed and not a set of "non-dominated" solutions. To answer the above research questions, we selected the solution from the set of non-dominated ones providing the

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.
Log4J	NSGA-II	NSGA-II	NSGA-II
Nutch	No sign. diff.	No sign. diff.	NSGA-II
JDI-Ford	NSGA-II	NSGA-II	NSGA-II

maximum trade-off using the following strategy when comparing between the different algorithms (expect the mono-objective algorithm where we select the solution with the highest fitness function or the JDeodorant tool). In order to find the maximal trade-off solution of the multi-objective or many-objective algorithm, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan (Rachmawati and Srinivasan 2009) to evaluate the worthiness of each non-dominated solution in terms of compromise between the objectives.

Results for RQ2.1: Comparison with Multi-Objective 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: 20 out of 27 experiments (74 %). 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 6 out of 9 experiments, where only two cases were not statistically significant, namely GanttProject and Nutch. For IGD, the results were similar as for IC.

A more qualitative evaluation is presented in Fig. 3 illustrating the box plots obtained for the multi-objective metrics on the different projects.

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 four robust refactoring algorithms: our NSGA-II adaptation with the three objectives, MOPSO, our previous work based on NSGA-II with two objectives.

The results from 51 runs are depicted in Table 5. For FCS, the number of fixed code smells using NSGA-II is better than MOPSO in all systems except for GanttProject 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 at least 88 % of cases. Even

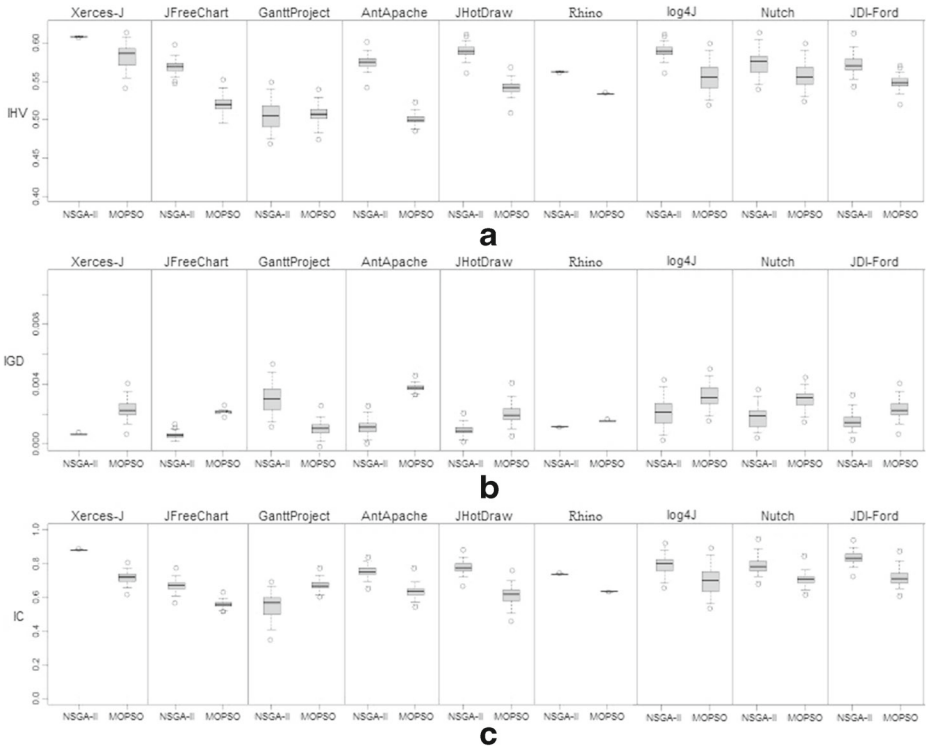


Fig. 3 Boxplots using the quality measures (**a**) IC, (**b**) IHV, and (**c**) IGD applied to NSGAII and MOPSO

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. The NSGA-II with the two objectives has a lower execution time than our adaptation with three objectives which normal. The different algorithms are using different change operators and number of objectives. In addition, our robust algorithm is using a perturbation function which not used by the non-robust algorithms. Thus it is normal that the execution time is different. However, it is not a very important factor since refactoring recommendation is not a real-time problem

In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 5a confirm that both multi-objective formulations are adequate and that NSGA-II outperforms MOPSO in most of the cases.

Results for RQ2.2, RQ2.3 and RQ2.4: Comparison with Mono-Objective and non Search-Based Approaches 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 as described previously. 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

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

(a)

Systems	NSGA-II (3 Obj)				MOPSO				NSGA-II (2 Obj)				Mono-EA			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	74/82	42.6	49.2	1 h56	69/82	34.6	46.8	1 h48	61/82	29.6	39.6	1 h32	54/82	21.8	32.8	1 h08
JFreeChart	61/73	39.8	37.8	2 h02	60/73	39.4	36.1	2 h21	51/73	24.8	31.2	1 h29	42/73	19.9	23.6	1 h06
GanttProject	42/56	31.4	29.6	1 h59	40/56	30.1	27.1	2 h09	32/56	21.1	22.4	1 h22	24/56	18.7	17.6	1 h01
ApacheAnt	83/91	44.6	46.4	2 h18	84/91	44.9	46.4	2 h22	69/91	32.6	38.6	1 h42	53/91	26.4	31.2	1 h11
JHotDraw	28/33	14.3	18.7	1 h49	25/33	12.2	16.2	1 h46	16/33	11.1	10.3	1 h34	12/33	06.3	07.1	1 h04
Rhino	62/74	42.8	32.3	1 h56	63/74	43.2	33.1	1 h59	58/74	33.1	28.4	1 h39	41/74	21.4	19.5	1 h12
Log4J	53/64	39.5	29.4	1 h38	50/64	35.9	27.3	1 h52	41/64	28.9	23.1	1 h14	32/64	19.4	18.9	1 h01
Nutch	66/72	41.1	36.8	1 h49	61/72	39.2	31.6	1 h58	49/72	30.2	21.8	1 h19	43/72	24.8	19.1	1 h02
JDI-Ford	74/88	43.8	42.1	1 h56	72/88	42.9	40.2	2 h12	64/88	37.3	28.4	1 h24	61/88	32.6	21.1	1 h04

(b)

Systems	Kessentini et al.'11				O'Keeffe et al.'08				JDedorant			
	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT	FCS	SCS	ICS	ICT
Xerces-J	52/82	20.2	32.8	1 h10	50/82	19.4	30.2	1 h02	46/82	19.1	29.1	N/A
JFreeChart	43/73	18.2	23.6	1 h04	38/73	17.9	21.6	1 h00	37/73	17.2	20.4	N/A
GanttProject	20/56	13.7	17.6	1 h06	21/56	16.9	15.8	0 h56	22/56	15.8	14.2	N/A
ApacheAnt	51/91	20.4	31.2	1 h14	48/91	22.3	27.1	0 h54	41/91	21.2	27.1	N/A
JHotDraw	13/33	05.3	07.1	1 h24	12/33	06.2	07.2	0 h51	10/33	05.9	06.3	N/A
Rhino	40/74	18.6	19.5	1 h16	38/74	17.2	17.4	1 h01	34/74	19.1	17.2	N/A
Log4J	32/64	18.4	17.6	1 h10	31/64	15.9	17.8	1 h04	29/64	16.5	15.1	N/A
Nutch	43/72	21.2	20.1	1 h02	40/72	18.4	19.6	0 h48	31/72	17.1	18.2	N/A
JDI-Ford	61/88	29.4	20.4	1 h01	56/88	27.1	18.2	0 h52	43/88	25.9	17.6	N/A

knee point in either direction is usually not interesting for the developer. 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 et al. to find the knee point (Rachmawati and Srinivasan 2009). 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.

Table 5 also shows the results of comparing our robust approach based on NSGA-II with two mono-objective refactoring approaches, a mono-objective genetic algorithm (Mono-EA) that has a single fitness function aggregating the two objectives, and a practical refactoring technique where developers used a refactoring plug-in in Eclipse to suggest solutions to fix code smells. It is apparent from Table 5 that our NSGA-II adaptation outperforms mono-objective approaches in terms of smell-fixing ability (FCS) all the cases. In addition, 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 the three objectives however the remaining approaches did not consider robustness but only quality. Thus, NSGA-II sacrifices a small amount of quality in order to improve importance and severity. Furthermore, the number of code smells fixed by NSGA-II is very close to the number fixed by the mono-objective and manual approaches, 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. Another interesting observation is that our refactoring solutions prioritized fixing code fragments containing severe code smells and also located in important classes. In fact, as described in Table 5 our approach fixed more important and severe code smells than all other existing approaches based on the SCS and ICS metrics. It is also well-known that a mono-objective algorithm requires lower execution time for convergence since only one objective is handled.

To answer RQ2.3 and RQ2.4, the results of Table 5b support the claim that our NSGA-II formulation provides a good trade-off between importance, severity and quality, and outperforms on average the state of the art of refactoring approaches, both search-based and manual, with a low robustness cost.

4.7.3 Results for RQ3: Manual Evaluation of the Results by Developers

To answer the last question RQ3 a manual evaluation were performed by subjects to estimate the correctness of the suggested refactoring. Figure 4 depicts the different Pareto surface obtained on the JDI-Ford system using NSGA-II to optimize the three objectives of quality, severity and importance. Due to space limitations, we show only this example of the Pareto-optimal front approximation. Similar fronts were obtained on the remaining systems. The 3-D projection of the Pareto front helps developers to select the best trade-off solution between the three objectives based on their own preferences. Based on the plot of Fig. 4, the developer could degrade quality in favor of importance and severity while controlling visually the

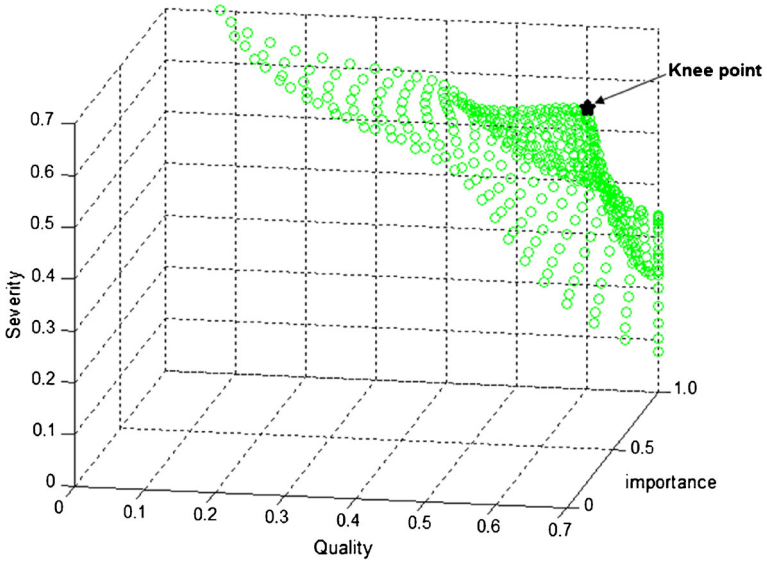


Fig. 4 3-D projection of the Pareto-Front solutions on the JDI-Ford system

robustness cost, which corresponds to the ratio of the quality loss to the achieved importance and severity gain. In this way, the preferred robust refactoring solution can be realized.

One striking feature is that starting from the highest quality solution the trade-off between the three objectives is in favor of quality, meaning that the quality degrades slowly with a fast increase in importance and severity up to the knee point, marked in each figure. Thereafter, there is a sharp drop in quality with only a small increase in importance. It is very interesting to note that this property of the Pareto-optimal front is apparent in all the problems considered in this study. 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 Fig. 4 shows how a better robust solution (importance and severity) can be obtained by sacrificing just a little in quality. Figure 4 shows the impact of different levels of perturbation on the Pareto-optimal front. However, it is difficult to generalize the observation that "the sacrifice in solution quality is quite small". The programmers may select the solution based on their preferences and the current context. In case that the programmers do not have enough time to fix all or most of the defects, they may select the refactoring solution fixing the most severe ones or those located in important classes. In other situations where there is enough time before the next release and several programmers are available, a solution that minimizes the sacrifice in quality is more adequate. The slight sacrifice on quality was only observed on few systems, thus it is hard to generalize the results.

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 Fig. 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.

Figure 5 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 70 % 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 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 (Ouni et al. 2012), and evaluating the impact of these changes on the robustness of suggested refactoring proposed by our NSGA-II algorithm and non-robust approaches. As depicted in Fig. 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 results achieved by the different techniques for different values of severity and class importance using the Wilcoxon test. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95 % confidence level ($\alpha = 5$ %). We also compared

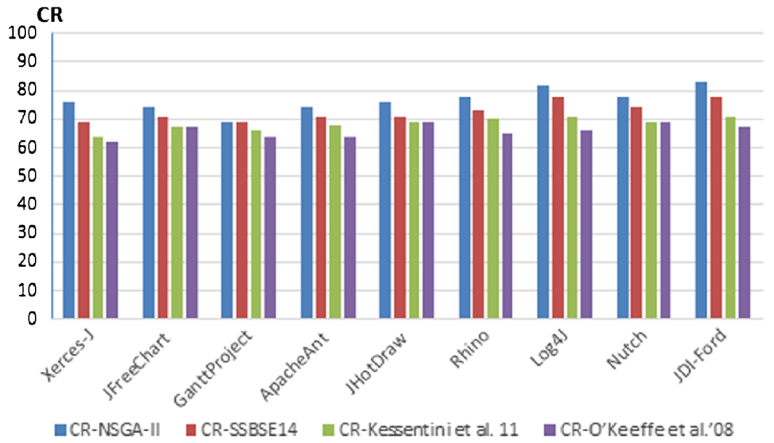


Fig. 5 The median correctness values (CR) of the recommended refactorings based on 51 runs. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95 % confidence level ($\alpha = 5$ %)

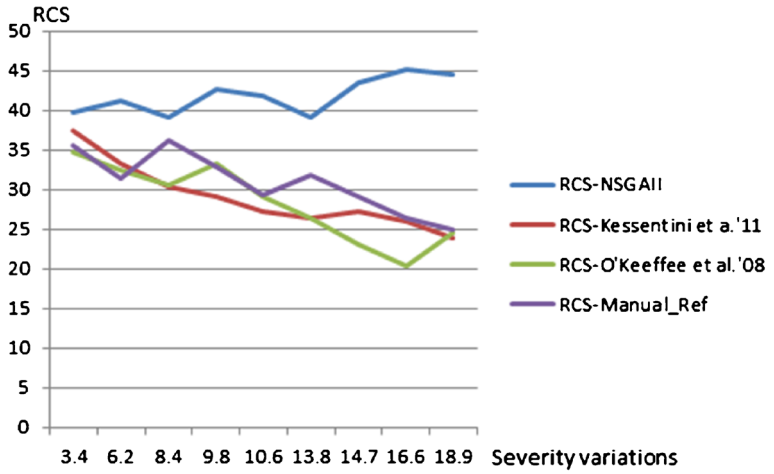


Fig. 6 The impact of code smells severity variations on the robustness of refactoring solutions for ApacheAnt. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95 % confidence level ($\alpha = 5$ %)

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.

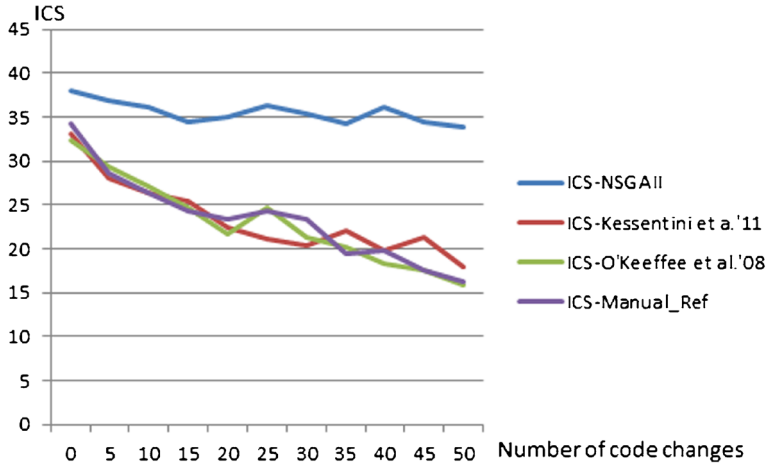


Fig. 7 The impact of class importance variation on the robustness of refactoring solutions for Apache Ant. The obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95 % confidence level ($\alpha = 5$ %)

4.8 Threats to Validity

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. Another threat is related to the fact that the original developers of the open source systems were not asked to evaluate the suggested refactorings. In addition, we should perform further experiments as part of our future work to compare recommended refactorings with expected ones (suggested by the original developers).

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-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 and just one industrial project. 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 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, 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 (Deb and Gupta 2011) 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.

References

Anda B (2007) Assessing software system maintainability using structural measures and expert assessments. In: International Conference on Software Maintenance (ICSM), pp 204–213

Antoniol G, Di Penta M, Harman M (2004) A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In METRICS '04, pp 172–183

Arcuri A, Lionel B (2011) "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In 33rd International Conference on Software Engineering (ICSE), 2011, pp 1–10. IEEE

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

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

Beyer H-G, Sendhoff B (2007) Robust optimization – A comprehensive survey. Comput Methods Appl Mech Eng 196(33–34):3190–3218

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

Brown W-J, Malveau R, McCormick H-W, Mowbray TJ (1998b) Antipatterns: refactoring software, architectures, and projects in crisis. Wiley, New York

Brown WJ, Malveau RC, McCormick HW, Mowbray TJ (1998) AntiPatterns: refactoring Software, Architectures, and Projects in Crisis. Wiley

Chatzigeorgiou A, Manakos A (2013) Investigating the evolution of code smells in object-oriented systems, innovations in systems and software engineering. NASA J

Coad P, Yourdon E (1991) Object-oriented design. Yourdon Press, Upper Saddle River

Coelho W, Murphy G (2007) ClassCompass: a software design men- toring system. Educ Resour Comput 7:1–18

D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects In: International Conference on Quality Software, pp 23–31

Das I (2000) Robustness optimization for constrained nonlinear programming problem. Eng Optim 32(5):585–618

Deb K, Gupta S (2011) Understanding knee points in bi-criteria problems and their implications as preferred solution principles. Eng Optim 43(11):1175–1204

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

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

Esteves Paixao M-H, De Souza J-T (2013) A scenario-based robust model for the next release problem. In Proceedings of the conference on Genetic and evolutionary computation (GECCO)

Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A (2011) JDeodorant: identification and application of extract class refactorings. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, pp 1037–1039

Fontana FA, Mäntylä MV, Zanoni M, Marino A (2015) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng:1–49

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring – Improving the design of existing code, 1st edn Addison-Wesley

Glass RL (2001) Frequently forgotten fundamental facts about software engineering. Softw Trends IEEE 18(3): 112–111. doi:10.1109/MS.2001.922739

Gueorguiev S, Harman M, 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). ACM, New York, pp 1673–1680

Hall T, Zhang M, Bowes D, Sun Y (2014) Some code smells have a significant but small effect on faults. ACM Trans Softw Eng Methodol 23(4), Article 33

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

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

Jensen A, 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, pp 1341–1348

Jin Y, Branke J (2005) Evolutionary optimization in uncertain environments – A survey. IEEE Trans Evol Comput 9(3):303–317

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

Kataoka Y, Ernst MD, Griswold WG, Notkin D (2001) Automated support for program refactoring using invariants. In International Conference on Software Maintenance (ICSM'01), pp 736–743

Kerievsky J (2004) Refactoring to patterns. Addison Wesley

Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A (2011) Design defects detection and correction by example. In: Proceedings of International Conference on Program Comprehension (ICPC), pp 81–90

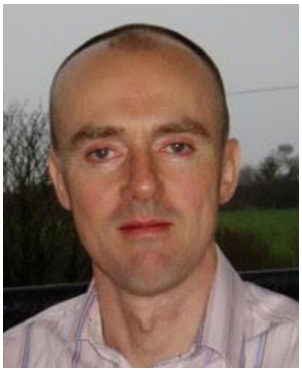
Li X (2003) A non-dominated sorting particle swarm optimizer for multiobjective optimization. In: The Genetic and Evolutionary Computation Conference (GECCO), pp 37–48

Mäntylä M-V, Lassenius C (2006) Subjective evaluation of software evolvability using code smells: an empirical study. Empir Softw Eng:395–431

Marinescu R (2010) inCode: continuous quality assessment and improvement. In Proceedings of the 14th Conference on Software Maintenance and Reengineering (CSMR)
Mkaouer M-W, Kessentini M, Bechikh S, Ó Cinnéide M (2014) A robust multi-objective approach for software refactoring under uncertainty. In: Symposium on Search Based Software Engineering (SSBSE), pp 168–183
Moghadam IH, Ó Cinnéide M (2012) Automated refactoring using design differencing. In Proceedings of European Conference on Software Maintenance and Reengineering (ECSM'12)
Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36
Ó Cinnéide M, Tratt L, Harman M, Counsell S, Moghadam IH (2012) Experimental assessment of software metrics using automated refactoring. In Proceedings of the ESEM'12, pp 49–58
O'Keeffe M, Ó Cinnéide M (2008) Search-based refactoring for software maintenance. J Syst Softw:502–516
Olbrich SM, Cruze DS, Sjøberg DI (2010, September) Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In Software Maintenance (ICSM), 2010 I.E. International Conference on (pp 1–10). IEEE
Oliveto R, Khomh F, Antoniol G, Gue- heneuc Y-G (2010) Numerical signatures of antipatterns: an approach based on B- Splines. In Proc. 14th European Conference on Software Maintenance and Reengineering, pp 248–251
Opdyke WF (1992) Refactoring object-oriented frameworks. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign. UMI Order No. GAX93-05645
Ouni A, Kessentini M, Sahraoui H, Boukadoum M (2012) Maintainability defects detection and correction: a multi-objective approach. J Autom Softw Eng:47–79
Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In Proceedings of the International Conference on Automated Software Engineering (ASE)
Rachmawati L, Srinivasan D (2009) Multiobjective evolutionary algorithm with controllable focus on the knees of the Pareto front. IEEE Trans Evol Comput 13(4):810–824
Radu M (2004) Detection strategies: metrics-based rules for detecting design flaws. In Proc. 20th International Conference on Software Mainte- nance, pp 350–359
Riel A (1996) Object-oriented design heuristics. Addison Wesley
Sahraoui H, Godin R, 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). IEEE Computer Society, pp 154–162
Seng O, Stammel J, 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), pp 1909–1916
Sunye G, Pollet D, T Yves Le, Jezequel J-M (2001) Refactoring UML models. In Proc. 4th International Conference on the Unified Modeling Language, volume LNCS 2185, pp 134–148
Tom Mens T, Tourwe T (2004) A survey of software refactoring. IEEE Trans Softw Eng (TSE) 30(2):126–139
Trifu A, Reupke U (2007) Towards automated restructuring of object oriented systems. In Proc. 12th Working Conference on Reverse Engineering, pp 39–48
Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng (TSE) 35(3):347–367
Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, Washington, pp 97–100
Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132
Yamashita A (2012) Assessing the capability of code smells to support software maintainability assessments: empirical inquiry and methodological approach. Doctoral Thesis, University of Oslo
Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects? In: 28th IEEE International Conference on Software Maintenance (ICSM), pp 306–315
Zamani B, Butler G (2009) Smell detection in UML designs which utilize pattern languages. Iran J Electr Comput Eng 8(1):47–52
Zhang M, Hall T, Baddoo N (2011) Code bad smells: a review of current knowledge. J Softw Maint 23(3):179–202

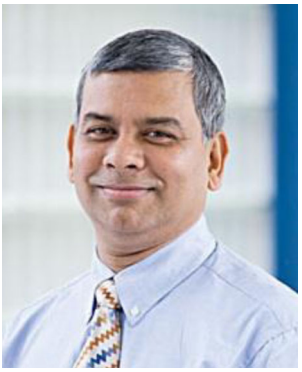
Mohamed Wiem Mkaouer is currently PhD candidate in Software Engineering at University of Michigan-Dearborn, USA, under the supervision of Dr. Marouane Kessentini. His research interests include software quality, systems refactoring, model-driven engineering and software testing. He is a member of the Search-based Software Engineering at Michigan research group, he is also a student member of the Association for Computing Machinery and the IEEE Computer Society. He has collaborations with industrial companies on the use computational search and evolutionary algorithms to address several software engineering problems such as software quality, software remodularization, software evolution, etc. He is expected to receive his PhD from University of Michigan in 2016. He published several papers in software engineering journals and conferences, including 1 best paper award. He has served as reviewer and program committee member in several major conferences (GECCO, EMSE, TEC, ASE, etc.) and an organization member of many conferences and workshops. He was also the web chair of the first North American Search Based Software Engineering Symposium (Nasbase2015) and he is now the web chair of of the 8th IEEE Search Based Software Engineering Symposium (SSBSE2016).

Marouane Kessentini is a tenure-track assistant professor at University of Michigan. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in Computer Science, University of Montreal (Canada), 2011. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 60 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program-committee/organization member in several conferences and journals.



Mel Ó Cinnéide holds BSc, MSc and PhD degrees in Computer Science from University College Cork, National University of Ireland and Trinity College Dublin respectively, and is a chartered engineer. He worked in the software industry for several years before moving to academia and is currently a lecturer in the School of Computer Science, University College Dublin. Dr. Ó Cinnéide has published over 45 papers in the areas of software quality and reuse, including two best paper awards. His primary research interest is currently the automated improvement of software design quality using Search-Based Software Engineering techniques.

Shinpei Hayashi is an assistant professor of Prof. Motoshi Saeki's research group, Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology. He recieved a B.Eng. degree from Hokkaido University in 2004. He also received M.Eng. and Dr. Eng. degrees from Tokyo Institute of Technology in 2006 and 2008, respectively. He published many papers in the area of software maintenance and evolution.



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