JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2017; **00**:1–37
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

MORE: A Multi-objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells

Ali Ouni[1,*,†], Marouane Kessentini[2], Mel Ó Cinnéide[3], Houari Sahraoui[4], Kalyanmoy Deb[5], and Katsuro Inoue[1]

[1] *Graduate School of Information Science and Technology, Osaka University, Japan*
[2] *Department of Computer and Information Science, University of Michigan, USA*
[3] *School of Computer Science, National University of Ireland, Dublin*
[4] *Department of Computer Science and Operations Research, University of Montreal, Canada*
[5] *Department of Electrical and Computing Engineering, Michigan State University, USA*

SUMMARY

Refactoring is widely recognized as a crucial technique applied when evolving object-oriented software systems. If applied well, refactoring can improve different aspects of software quality including readability, maintainability and extendibility. However, despite its importance and benefits, recent studies report that automated refactoring tools are underused much of the time by software developers. This paper introduces an automated approach for refactoring recommendation, called MORE, driven by three objectives: (1) to improve design quality (as defined by software quality metrics), (2) to fix code smells, and (3) to introduce design patterns. To this end, we adopt the recent non-dominated sorting genetic algorithm, NSGA-III, to find the best trade-off between these three objectives. We evaluated the efficacy of our approach using a benchmark of seven medium and large open-source systems, seven commonly-occurring code smells (god class, feature envy, data class, spaghetti code, shotgun surgery, lazy class, and long parameter list), and four common design pattern types (visitor, factory method, singleton and strategy). Our approach is empirically evaluated through a quantitative and qualitative study to compare it against three different state-of-the art approaches, two popular multi-objective search algorithms, as well as random search. The statistical analysis of the results confirms the efficacy of our approach in improving the quality of the studied systems while successfully fixing 84% of code smells and introducing an average of six design patterns. In addition, the qualitative evaluation shows that most of the suggested refactorings (an average of 69%) are considered by developers to be relevant and meaningful.
Copyright © 2017 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Refactoring; code smells; Design Patterns; Software Quality, Search-based Software Engineering

1. INTRODUCTION

Software systems are continuously subject to maintenance and evolution activities to add new features, to fix bugs or to adapt to new environmental changes [1]. Such activities are often performed in an undisciplined manner due to many reasons including time pressure, poorly planned changes or limited knowledge/experience of some developers about the system's design [2, 3]. Refactoring techniques are a fundamental support for improving software quality that has been

*Correspondence to: Ali Ouni, Graduate School of Information Science and Technology, Osaka University, 1-5, Yamadaoka, Suita, Osaka 565-0871, Japan.
†Email: ali@ist.osaka-u.ac.jp

Copyright © 2017 John Wiley & Sons, Ltd.

practiced for many years [4]. Refactoring has been defined by Fowler as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [2]. If applied well, refactoring brings many benefits to support software developers in terms of understanding, changing, maintaining and evolving the existing software implementation. To this end, various refactoring recommendation approaches have been proposed in the literature [5–8].

Despite its significant benefits, recent studies show that automated refactoring tools are underused much of the time [5, 9, 10]. One of the possible reasons is that most of existing refactoring tools [6, 11, 12] focus mainly only on improving some specific aspects of a system (e.g., coupling, cohesion, complexity, etc.). Indeed, improving some quality metrics in a software system does not necessary fix existing code smells [13]. Thus, quality metric values can be significantly improved but the original program may still contain a considerable number of code smells, which may lead to several maintenance and evolution difficulties. On the other hand, design patterns are known as "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to change [14]. Design patterns can be automatically introduced using refactoring [15, 16], however, most existing refactoring tools do not consider the use of design patterns to fix code smells and improve the quality of software systems. To make the situation worse, applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system for no benefit [17–20]. In addition, an excessive introduction of design patterns can make the system less maintainable and understandable, and even create performance problems [19]. For instance, introducing the Strategy design pattern to an inappropriate class will decrease the cohesion of the class and increase the overall coupling in the system.

A further challenge is that while it may appear that introducing design patterns, improving software quality metrics and removing code smells are all non-competing goals and hence can be simultaneously optimized, empirical evidence indicates otherwise. Studies by Soetens and Demeyer [21], Stroggylos and Spinellis [22] and Kannangara and Wijayanayake [23] all found little or no correlation between refactoring activity and improvement in software quality metrics. In a similar vein, Wilking et al. [24] found that refactored code was neither more maintainable nor easier to modify than unrefactored versions of the same code. Studies by Counsell et al. [25] and Chatzigeorgiou and Manakos [26] found no evidence to support the claim that refactoring is used to remove code smells. Even considering software quality metrics on their own, Ó Cinnéide et al. [27] found such conflict between the cohesion metrics they studied that it is likely that any observed correlation with a software quality metric will not extend to other software quality metrics.

To address the above-mentioned challenges, we developed a search-based refactoring recommendation approach [28] to fix code smells, introduce design patterns, and improve quality attributes of a system while preventing semantic incoherencies in the refactored program. This multi-objective search-based approach was embodied in a tool called MORE, based on the non-dominated sorting genetic algorithm (NSGA-II) [29]. The proposed approach aims to recommend refactoring operations to (1) improve software quality attributes (i.e., understandability, flexibility, maintainability, etc.), (2) introduce "good" design practices (i.e., design patterns) and (3) fix "bad" design practices (i.e., code smells). In addition, MORE is based on a set of constraints, for each refactoring operation, in order to ensure the semantic coherence of the refactored program, e.g., that a method is not moved to a class where it makes no sense.

This paper extends our previous work [28] that was published in the proceedings of the North American Search Based Software Engineering Symposium (NasBASE) in three ways.

1. Our initial approach was based on the popular multi-objective algorithm NSGA-II [29]. In this paper, we adopt NSGA-III, a recent many-objective search algorithm proposed by Deb et al. [30], to improve the performance of our approach. We conduct a experiment to compare the performance of NSGA-III against other popular multi-objective algorithms including NSGA-II and MOEA/D, as well as random search.
2. We extend MORE to support (1) three additional code smell types (shotgun surgery, lazy class, and long parameter list) and (2) one additional design pattern (Strategy). Moreover,

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

we provide detailed descriptions of the refactoring transformations for each of the considered design patterns, and the semantic constraints employed to preserve the design semantics when applying refactorings.
3. We present an empirical study based on a quantitative and qualitative evaluation on an extended benchmark composed of seven real-world Java software systems of various sizes. The quantitative evaluation investigates the efficacy (ability to achieve the intended effect under experimental conditions) of our approach in fixing seven common code smell types (*god class*, *feature envy*, *data class*, *spaghetti code*, *shotgun surgery*, *lazy class*, and *long parameter list*), introducing four types of design patterns (*factory method*, *visitor*, *singleton*, and *strategy*), and improving six quality attributes according to the popular software quality model QMOOD [31]. For the qualitative evaluation, we conducted a non-subjective evaluation with software developers to evaluate the meaningfulness and usefulness of our refactoring approach from a user's perspective.

The remainder of this paper is organized as follows. Section 2 describes necessary background and basic concepts related to our approach. Section 3 introduces our search-based approach, MORE. Section 4 describes in more detail the adaptation of NSGA-III to the refactoring recommendation problem. Section 5 describes the design of the empirical study we employ to evaluate our approach, while in Section 6 we present and discuss the experimental results. Section 7 describes the threats to validity and the limitations of the present study. Section 8 outlines the related work. Finally, in Section 9, we conclude and describe our future research directions.

2. BACKGROUND

2.1. Definitions

Refactoring is the process of changing the structure of software while preserving its external behavior. The term *refactoring* was introduced by Opdyke and Johnson [32], and popularized by Martin Fowler's book [2]. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions. This reorganization is used to improve different aspects of software quality such as maintainability, extensibility, reusability, etc. [33, 34].

Refactoring is widely recognized as an efficient technique to fix code smells and reduce the increasing complexity of software systems. Code smells, also called bad smells, design defects, design flaws or antipatterns, are symptoms of poor design and implementation practices that describe a bad solution to a recurring design problem that leads to negative effects on code quality [13]. Software developers may introduce code smells unintentionally during initial design or during software development due to bad design decisions, ignorance or time pressure. Table I describes the studied code smells in this paper: *god class*, *feature envy*, *data class*, *spaghetti code*, *shotgun surgery*, *long parameter list* and *lazy class*. Indeed, we selected these code smell types because (i) they are representative of problems with data, complexity, size, and the features provided by classes; and (ii) they are the most important and frequently-occurring ones in open-source and industrial projects based on recent studies [35–38].

On the other hand, design patterns are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to change [14]. We focus in this paper on four of the Gamma et al. design patterns namely *factory method*, *visitor*, *singleton* and *strategy* [14]. Table II provides definitions and characteristics of these design patterns. We choose these patterns because they are partially automatable [16], and also because they address problems related to classes and their associations and are commonly used to improve the structure of an object-oriented software design.

2.2. Search-Based Software Engineering

Our approach is largely inspired by contributions in the field of Search-Based Software Engineering (SBSE). The term SBSE was coined by Harman and Jones in 2001, and the goal of the field is to

Copyright © 2017 John Wiley & Sons, Ltd.

Prepared using smrauth.cls
DOI: 10.1002/smr

Table I. code smell types supported by MORE.

Name	Description
God class	It is found in design fragments where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily contain data. It is a large class that declares many fields and methods with low cohesion [2, 39, 40].
Feature Envy	It is found when a method heavily uses attributes from one or more external classes, directly or via accessor methods. [2].
Data Class	It contains only data and performs no processing on this data. It is typically composed of highly cohesive fields and accessors [13].
Spaghetti Code	It is code with a complex and tangled control structure. This code smell is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with few or no parameters, and utilizing global variables. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit, and indeed prevents, the use of object-oriented mechanisms such as polymorphism and inheritance [13].
Shotgun surgery	It occurs when a method has a large number of external operations calling it, and these operations are spread over a significant number of different classes. As a result, the impact of a change in this method will be large and widespread [2].
Long Parameter List	It is found when a method signature declares numerous parameters. Long parameter lists are prone to continuous change, difficult to use, and hard to understand and maintain. In object-oriented programming one should use objects instead of passing a large number of parameters. [2].
Lazy class	It refers to a class that is not doing enough to justify its existence. It represents a class having very small dimensions, few methods and with low complexity [2].

Table II. Design pattern types supported by MORE.

Name	Description
Visitor	The visitor pattern represents an operation to be performed on the elements of an object structure. In essence, the visitor pattern allows a new method to be added to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the dynamically-bound method [14].
Factory Method	Factory method is a creational pattern that uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. It defines an interface for creating an object, but defers to subclasses the decision as to exactly which class to instantiate [14].
Singleton	It restricts the instantiation of a class to one single object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists [14].
Strategy	It defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime. Strategy lets the algorithm vary independently from clients that use it. There are common situations when classes differ only in their behavior. In this case, best design practice suggests isolating the algorithms in separate classes in order to have the ability to select different algorithms at runtime [14].

move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search and evolutionary computation paradigms [41]. The idea is to exploit human creativity with machine tenacity and reliability, rather than requiring humans to perform the more tedious, error-prone and thereby costly aspects of the engineering process.

In 2007, Harman reviewed the state of the SBSE field [42] and found a strong adoption of a variety of metaheuristic search techniques (e.g., hill climbing, genetic algorithms, simulated annealing. etc.) by software practitioners to solve different software engineering problems. In later studies

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Harman et al. [43, 44] observed that multi-objective evolutionary optimization techniques (NSGA-II, MOGA, etc.) were becoming popular. In fact, this new tendency to adopt multi-objective search techniques to software engineering is justified by the new challenges that software practitioners face to solve more complex software engineering problems.

3. APPROACH

This section describes our approach, its components and the semantic constraints employed.

3.1. The General Architecture of MORE

The general structure of our approach is described in Figure 1. It takes as input the source code of the program to be refactored, and as output it recommends a sequence of refactorings that should provide a near-optimal trade-off between: (1) improving quality, (2) fixing code smells, and (3) introducing design patterns. The proposed approach comprises seven main components:

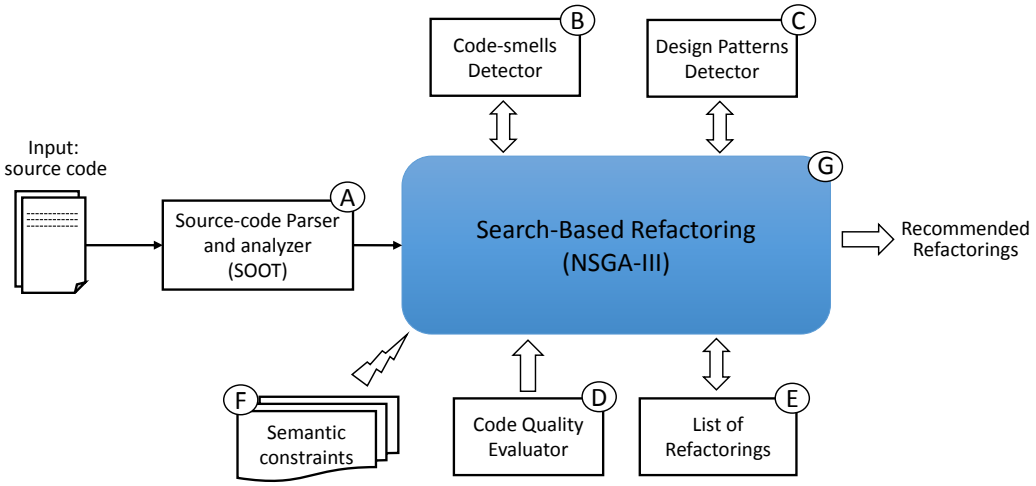


Figure 1. Overview of MORE architecture.

A) Source code parser and analyzer (Label A). This component is responsible for parsing and analyzing the source code of the program being refactored. Our approach is based on Soot [45], a Java optimization framework. The original source code is analyzed in order to extract from it the relevant code elements (i.e., classes, methods, attributes, etc.) and the existing relationships between them. The outputs are (1) the parsed code in a specific representation that is simple to manipulate during the search process, and (2) a call graph for the entire program that will be used for calculating semantic constraints and software metrics (e.g. coupling, cohesion, etc.).

B) Code smell detector (Label B). This component scans the entire software program in order to find existing code smell instances using a set of code smell detection rules [36]. Detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific code smell type (e.g., god class, feature envy, etc.) and is expressed as a logical combination of a set of quality metrics/threshold values. These detection rules are generated from real instances of code smells using genetic programming [36].

C) Design pattern detector (Label C). This component is responsible for detecting existing design pattern instances in the code being refactored. Extensive research has been devoted to develop techniques to automatically detect instances of design patterns both at code and design levels. In our approach, we use a detection mechanism that is inspired by the work of Heuzeroth et al. [46]. A design pattern P is defined by a tuple of program elements such as classes and methods conforming to the restrictions or rules of a certain design pattern. The detection strategy [46] is based on static

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

and dynamic specifications of the pattern. In MORE, we use only the static specifications, along with a post-processing step to eliminate redundancies. Static specifications are based on predicates to identify the types of code elements, e.g., classes, methods, calls, etc., and relates them to the roles in the pattern.

D) Code quality evaluator (Label D). This component consists of a set of software metrics that serves to evaluate the software design improvement achieved by refactoring. Indeed, the expected benefit from refactoring is to enhance the overall software design quality as well as fixing code smells [2]. In our approach we use the QMOOD (Quality Model for Object-Oriented Design) model [31] to estimate the effect of the suggested refactoring solutions on quality attributes.

E) List of refactorings (Label E). MORE currently supports the following refactoring operations: move method, move field, pull up field, pull up method, push down field, push down method, inline class, extract method, extract class, move class, extract superclass, extract subclass, and extract interface [2]. We selected these refactorings because they are the most frequently used refactorings and they are implemented in most contemporary IDEs such as Eclipse and Netbeans. In addition, we considered four specific blocks of refactorings to automatically introduce different types of design pattern instances: factory method refactoring, visitor pattern refactoring, singleton pattern refactoring and strategy pattern refactoring. We referred to some guidelines from the literature for introducing instances of design patterns [17, 34, 47]. MORE currently supports the following four design pattern types: visitor, factory Method, singleton and strategy.

- *Visitor Pattern Refactoring*. To introduce a visitor pattern, a sequence of refactoring operations should be applied in the right order. Algorithm 1 illustrates the necessary refactorings to be applied to introduce a visitor. The starting point is a class hierarchy H that has a superclass/interface SC and a set of subclasses CC. The first step is to create, for each functional method, a corresponding visitor class (lines 6-11). Then, functional code fragments should be moved from the class hierarchy H to the visitor classes. To this end, we apply the Extract Method refactoring to extract the functional code from the functional methods (Line 15). The original method will now simply delegate the new extracted one (at a later stage, these methods can be deleted and their call sites updated to use the appropriate visitor). The extracted method will be moved from the class hierarchy to the appropriate newly-created visitor class (Line 16). The new methods in visitor classes are named "visit*" using a Rename Method refactoring (Line 18). An abstract Visitor class is introduced as a superclass for all the created visitors using an Extract Superclass refactoring (Line 21). Now, an "accept" method is introduced in all the subclasses CC in H by extracting it from the initial methods, using an Extract Method refactoring (Line 24). All functional methods now call the accept method with an instance of the appropriate Visitor subclass. Therefore, their definition can be pulled up to the SC class by using a Pull Up Method refactoring.
- *Factory Method Refactoring*. As described in Algorithm 2, which uses the approach developed by Ó Cinnéide and Nixon [17], a factory method pattern can be introduced starting from a `Creator` class that creates instances of `Product` class(es). The first step is to apply an extract interface refactoring (Line 3) to abstract the public methods of the `Product` classes into an interface. All references to the `Product` classes in the `Creator` class are then updated to refer to this interface (Lines 4-7). Then, for each constructor in each of the `Product` classes, a similar method is added in the `Creator` class that returns an instance of the corresponding `Product` class (Lines 9-16). Finally all creations of `Product` objects in the `Creator` class are updated to use these new methods (Lines 17-20).
- *Singleton Pattern Refactoring*. Our formulation for the singleton pattern is derived from [48] and [3]. Algorithm 3 describes the basic steps to introduce the singleton pattern. A singleton class can be introduced starting from a candidate class `Singleton`. The first step (Line 3) is to apply the classic refactoring operation, defined in Fowlers catalogue [2], Replace Constructor with Factory Method. The aim of this step is to make the constructor private. Then access to this class will be performed via the newly-generated static method `getSingleton()`, which will be the global access point to the `Singleton` instance. The second step is to create a static field `singleton` of type Singleton with access level

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Algorithm 1 Pseudo-code of the Visitor Pattern Refactoring

```
1: Input: hierarchy H
2: Input: SC = getSuperClass(H)
3: Input: CC = getSubClasses(H)
4: Input: visitors = ∅
5: Process:
6: for each method m in SC do
7:     if m ∉ SC.constructors() then
8:         v = createEmptyClass(m.name)
9:         v = renameClass(c.name+"visitor")
10:        visitors = visitors ∪ {v}
11:    end if
12: end for
13: for each class c in CC do
14:    for each method m in c do
15:        visClass = V(m) //find visitor class that maps to the name of method
16:        extractMethod (c, m, m')
17:        moveMethod (c, m', visClass )
18:        renameMethod (visClass, m', "visit"+c.name )
19:    end for
20: end for
21: Visitor=extractSuperClass (Visitors,"Visitor"+SC.name )
22: for each class c in CC do
23:    for each method m in c do
24:        extractMethod (c, m, "accept" )
25:        pullUpMethod (m, c, SC )
26:    end for
27: end for
```

Algorithm 2 Pseudo-code of the Factory Method Refactoring

```
1: Input: Class Creator, Class [] Products
2: Process:
3: extractInterface(Products[], "abstract"+ Products.getName() )
4: for each Object o in Creator do
5:     if o.getType ∈ Products[] then
6:         o.renameType (o.getType()+"abstract"+ o.getType() )
7:     end if
8: end for
9: for each p ∈ Products[] do
10:    for each constructor c in p do
11:        m = addMethod (Creator, "create"+p.name() )
12:        m.setReturnType ("abstract"+p.name())
13:        m.setParamList (c.paramList)
14:        m.setBody ("return new P("+c.paramList+");"))
15:    end for
16: end for
17: for each Object o in Creator do
18:    if o.getType ∈ Products[] then
19:        Creator.replaceObjectCreations(o.getType(), "create"+ o.getType())
20:    end if
21: end for
```

private (Line 4) that will be initialized to `new Singleton()` in the body of the new method `getSingleton()` (Line 6). The selection statement ensures that the field singleton is instantiated only once, i.e., when it is null.

- *Strategy Pattern Refactoring*. Algorithm 4 describes the main steps to introduce a strategy pattern. The starting point of a strategy pattern is a method `m`. `m` can be turned into a strategy pattern by creating a new interface `i` (called with same name as `m` followed by "_Strategy") (Line 3) where the strategy method `m` is moved to (Line 4). Then for each '`if`' statement in `m`, a concrete class is created (Line 6) to implement the interface `i` (Line 8). An extract method refactoring is applied in order to move the correspondent code fragment from the

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Algorithm 3 Pseudo-code of the Singleton Pattern Refactoring

```
1: Input: Class Singleton
2: Process:
3: Replace_Constructor_with_Factory_Method(Singleton.constructor, "get"+ Singleton.name)
4: addField(singleton, Singleton, private, static)
5: if singleton == null then
6:     initialize(singleton, "new Singleton()")
7: end if
```

Algorithm 4 Pseudo-code of the Strategy Pattern Refactoring

```
1: Input: Method m
2: Process:
3: i = addNewInterface(m.name + "_Strategy ")
4: addMethod(i, m)
5: for each 'if' statement in m do
6:     c = addNewClass ( getName() + m.name + "_Strategy ");
7:     extractMethod (m, c )
8:     c.setInterface (i )
9:     f = addNewField (m.getClass(), c )
10:    f.setType (i)
11: end for
```

correspondent '`if`' statement (Line 7). A field of the concrete class type is added to the original class implementing `m`.

We selected these four design patterns because they are frequently used in practice, and it is widely believed that they embody good design practice [14]. Note that the four algorithms apply a typical implementation of the pattern, and leave some unfinished work to the developer to complete. Furthermore, if an atomic refactoring fails due to a non-satisfied precondition, the whole refactoring sequence that applies the design pattern will be rejected.

F) Semantic constraints checker (Label F). The aim of this component is to prevent arbitrary changes to code elements. Most refactorings are relatively simple to implement, and it is straightforward to show that they preserve behaviour assuming their pre-conditions are true [33]. However, until now there has been no consensual way to investigate whether a refactoring operation is semantically feasible and meaningful [49]. Preserving behavior does not mean that the coherence of the refactored program is also preserved. For instance, a refactoring solution might move a method `calculateSalary()` from the class `Employee` to the class `Car`. This refactoring could improve program structure by reducing the complexity and coupling of the class Employee while preserving program behavior. However, having a method `calculateSalary()` in the class `Car` does not make sense from the domain semantics standpoint. To avoid this kind of problem, we use a set of semantic coherence constraints that must be satisfied before applying a refactoring in order to prevent arbitrary changes to code elements. This will be described further in Section 3.2.

G) Search process (Label G). Our approach is based on multi-objective optimization search using the recent NSGA-III algorithm [30] to formulate the refactoring recommendation problem. We selected NSGA-III because it is a recent improvement of its previous version NSGA-II [29] which is widely-used in the field of multi-objective optimization, and demonstrates good performance compared to other existing metaheuristics in solving many software engineering problems [50]. Thus our approach can be classified as Search Based Software Engineering (SBSE) [41, 43, 44] for which it is established best practice to define a representation, fitness functions and computational search algorithm. Referring to Figure 1, the search process (NSGA-III) takes as input the source code that is then parsed into a more manipulable representation (Label A), a set of code smell detectors (Label B), a set of design patterns detectors (Label C), a code quality evaluator (Label D) that evaluates post-refactoring software quality, a set possible refactoring operations to be applied (Label E), and set of constraints (Label F) to ensure semantic coherence of the code after refactoring. As output, our approach suggests a list of refactoring operations that should be applied in the

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

appropriate order to find the best compromise between fixing code smells, introducing design patterns, and improving code quality.

3.2. Semantic Constraints

MORE defines and uses a set of semantic constraints to prevent arbitrary changes that may affect the semantic coherence of the refactored program. Indeed, applying a refactoring where it is not needed is highly undesirable as it may introduce semantic incoherence and unnecessary complexity to the original design. To this end, we have defined the following semantic constraints to steer the evolutionary process toward meaningful and useful refactoring solutions.

Vocabulary-based similarity constraint (VS). This kind of constraint is interesting to consider when moving methods, fields, or classes. For example, when a method has to be moved from one class to another, the refactoring would make sense if both actors (source class and target class) use similar vocabularies [49, 51]. The vocabulary can be used as an indicator of the semantic similarity between different actors (e.g., method, field, class, etc.) that are involved when performing a refactoring operation. We start from the assumption that the vocabulary of an actor is derived from the domain terminology and therefore can be used to determine which part of the domain semantics is encoded by an actor (e.g., class, method, package, interface, etc.). Thus, two actors are likely to be semantically similar if they use similar vocabularies.

The vocabulary can be extracted from the names of methods, fields, variables, parameters, types, etc. Tokenisation is performed using the Camel Case Splitter which is one of the most used techniques in Software Maintenance tools for the preprocessing of identifiers. A more pertinent vocabulary can also be extracted from comments, commit information, and documentation. We calculate the semantic similarity between actors using information retrieval-based techniques (e.g., cosine similarity). Equation 1 calculates the cosine similarity between two actors. Each actor is represented as a *n* dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two actors c_1 and c_2 is determined as follows:

$$Sim(c_1, c_2) = Cos(\overrightarrow{c_1}, \overrightarrow{c_2}) = \frac{\overrightarrow{c_1}.\overrightarrow{c_2}}{\parallel \overrightarrow{c_1} \parallel \times \parallel \overrightarrow{c_2} \parallel} = \frac{\sum_{i=1}^{n}(w_{i,1} \times w_{i,2})}{\sqrt{\sum_{i=1}^{n}(w_{i,1})^2 \times \sum_{i=1}^{n}(w_{i,2})^2}} \in [0,1] \quad (1)$$

where $\overrightarrow{c_1} = (w_{1,1}, ..., w_{n,1})$ is the term vector corresponding to actor c_1 and $\overrightarrow{c_2} = (w_{1,2}, ..., w_{n,2})$ is the term vector corresponding to c_2. The weights $w_{i,j}$ can be computed using information retrieval based techniques such as the Term Frequency - Inverse Term Frequency (TF-IDF) method.

Dependency-based similarity constraint (DS). This constraint aims at approximating semantics closeness between actors starting from their mutual dependencies. The intuition is that actors that are strongly connected (i.e., having dependency links) are semantically related. As a consequence, refactoring operations requiring semantic closeness between involved actors are likely to be successful when these actors are strongly connected. We consider two types of dependency link:

- *Shared Method Calls (SMC)* can be captured from call graphs derived from the whole program using CHA (Class Hierarchy Analysis) [52]. A call graph is a directed graph that represents the different calls (call in and call out) among all methods of the entire program. Nodes represent methods, and edges represent calls between these methods. CHA uses a basic call graph that considers class hierarchy information, e.g., for a call *c.m(...)* it assumes that any *m(...)* is reachable that is declared in a subclass of the declared class of *c*. For a pair of actors, shared calls are captured through this graph by identifying shared neighbours of nodes related to each actor. We consider both shared call-out and shared call-in. Equations 2 and 3 are used to measure respectively the shared call-out and the shared call-in between two actors c_1 and c_2 (two classes, for example).

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

$$SharedCallOut(c_1, c_2) = \frac{\mid callOut(c_1) \cap callOut(c_2) \mid}{\mid callOut(c_1) \cup callOut(c_2) \mid} \in [0, 1] \quad (2)$$

$$SharedCallIn(c_1, c_2) = \frac{\mid callIn(c_1) \cap callIn(c_2) \mid}{\mid callIn(c_1) \cup callIn(c_2) \mid} \in [0, 1] \quad (3)$$

where *callOut(c)* returns the set of methods called by the methods of the class *c*, and *callIn(c)* returns the set of external methods that call any method in the class c. Shared Method Calls (SMC) is then defined as the average of shared call-in and call-out.

- *Shared Field Access (SFA)* can be calculated by capturing all field references uncovered using static analysis to identify dependencies based on field accesses (read or modify). We assume that two software elements are semantically related if they read or modify the same fields. The rate of shared fields (read or modified) between two actors c_1 and c_2 is calculated according to Equation 4. In this equation, $fieldRW(c_i)$ returns the set of fields that may be read or modified by each method of the actor c_i. Thus, by applying suitable static program analysis to the whole method body, all field references that occur could be easily computed.

$$sharedFieldRW(c_1, c_2) = \frac{\mid fieldRW(c_1) \cap fieldRW(c_2) \mid}{\mid fieldRW(c_1) \cup fieldRW(c_2) \mid} \in [0, 1] \quad (4)$$

Implementation-based Similarity constraint (IS). Methods that have similar implementations in all subclasses of a superclass should usually be moved to the superclass using the Pull Up Method refactoring [2], assuming certain constraints are satisfied. The implementation similarity between methods is investigated at two levels: signature level and body level. To compare the signatures of methods, a semantic comparison algorithm is applied that takes into account the methods names, the parameter lists, and return types. Let $Sig(m_i)$ be the set of elements in the signature of method m_i. The signature similarity for two methods m_1 and m_2 is computed as follows:

$$sig\_sim(m_1, m_2) = \frac{\mid Sig(m_1) \cap Sig(m_2) \mid}{\mid Sig(m_1) \cup Sig(m_2) \mid} \in [0, 1] \quad (5)$$

To compare method bodies, MORE compares the statements in the body, the used local variables, the exceptions handled, the call-out, and the field references. Let $Body(m)$ (set of statements, local variables, exceptions, call-out, and field references) be the body of method m. The body similarity for two methods m_1 and m_2 is computed as follows:

$$body\_sim(m_1, m_2) = \frac{\mid Body(m_1) \cap Body(m_2) \mid}{\mid Body(m_1) \cup Body(m_2) \mid} \in [0, 1] \quad (6)$$

The IS score between two methods corresponds to the average score of their sig_sim and $body\_sim$ values. Although we simply used Jaccard similarity to compute the IS constraint in order to reduce the computational time of MORE, further improvement of the IS could be based on code clone detection techniques [53].

Feature Inheritance Usefulness constraint (FIU). This constraint is useful when applying refactorings such as Push Down Method or Push Down Field operations. In general, when a method or field is used by only few subclasses of a superclass, it is better to move it, i.e., push it down, from the superclass to the subclasses using it [2]. To do this for a method, we need to assess the usefulness of the method in the subclasses in which it appears. We use a call graph and consider polymorphic calls derived using XTA (Separate Type Analysis) [54]. XTA is more precise than CHA is it yields a more local view of what types are available. We use Soot [45] as a standalone tool to implement

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

and test all the program analysis techniques required in our approach. The inheritance usefulness of a method is given by Equation 7:

$$FIU(m,c) = 1 - \frac{\sum_{i=1}^{n} call(m,i)}{n} \quad \in [0,1] \tag{7}$$

where n is the number of subclasses of the superclass c, m is the method to be pushed down, and $call$ is a function that returns 1 if m is used (called) in the subclass i, and 0 otherwise.

For the refactoring operation Push Down Field, a suitable field reference analysis is used. The inheritance usefulness of a field is given by Equation 8:

$$FIU(f,c) = 1 - \frac{\sum_{i=1}^{n} use(f,c_i)}{n} \quad \in [0,1] \tag{8}$$

where n is the number of subclasses of the superclass c, f is the field to be pushed down, and use is a function that returns 1 if f is used (read or modified) in the subclass c_i, and 0 otherwise.

Furthermore, we introduced other semantic constraints related to the introduction of design patterns. Before introducing a design pattern to a particular design fragment, the basic intent of the pattern should exist in that design fragment already. This starting point is termed a precursor in the nomenclature of Ó Cinnéide and Nixon [17], and is not taken into account in much of the existing work in automated refactoring. MORE formulates the notion of precursor as a set of semantic constraints that should be satisfied before introducing a design pattern.

Factory Method constraint (FMC). The semantic constraint we use for the Factory Method pattern is that the Creator class must create a concrete instance of a Product class [17]. This situation could require the application of the Factory Method pattern, if the developer decides that the Creator class should be able to handle several different types of Product. MORE analyzes, using Soot [45], all the method bodies of a candidate Creator class to retrieve statements containing the operator "new" that occur within its functional methods' bodies. If the candidate Creator class does not create instances of the Product class, then there is no need to introduce a Factory Method pattern.

Visitor pattern constraint (VPC). The semantic constraints for the Visitor pattern involve the situation when a class do not support additional behavior. This relates in general to complex hierarchies that have a large number of inherited methods or with God classes that can be detected [17]. The goal is to increase the ability to add new operations to existing object structures without modifying those structures.

Singleton pattern constraint (SPC). The semantic constraints we use for the Singleton pattern is that the class under refactoring (the candidate Singleton): 1) has only one instance, 2) does not employ inheritance, and 3) provides a global point of access to it, i.e., a method called from other classes in the system [3]. These two constraints can be checked using static program analysis. Using Soot, all class instances, method calls and field accesses can be captured, thus classes that are instantiated once or accessed through static methods or fields are potential candidates for the Singleton pattern refactoring. Dynamically, we can simply check if at most one instance of the candidate singleton class is created during runtime. That would ensure that there are no false positives.

Strategy pattern constraint (StPC). The semantic constraint for the Strategy pattern is when a class defines many behaviors, and these appear as multiple conditional statements in its methods. Consider the situation where a method is structured as one large 'if' statement. This situation suggests that Strategy could be applied by extracting each branch of the if statement into its own class, where each class implements a common interface. The decision on which class to use can be moved to the original class, and made settable by the client.

Copyright © 2017 John Wiley & Sons, Ltd.

Prepared using smrauth.cls
DOI: 10.1002/smr

4. SEARCH-BASED REFACTORING USING NSGA-III

This section shows how the refactoring problem can be addressed using NSGA-III. We first present an overview of NSGA-III, then we provide the details of our approach.

4.1. NSGA-III Overview

NSGA-III is a recent many-objective search algorithm proposed by Deb et al. [30]. The basic framework remains similar to the original NSGA-II algorithm [29] with significant changes in its selection mechanism.

Algorithm 5 gives the pseudo-code of the NSGA-III procedure for a particular generation t. First, the parent population P_t (of size N) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected from the combined population to form the parent population for the next generation. The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. The original NSGA-II algorithm uses the crowding distance measure for selecting a well-distributed set of points, however, in NSGA-III the supplied reference points (Z_r) are used to select these remaining members (cf. Figure 2). To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

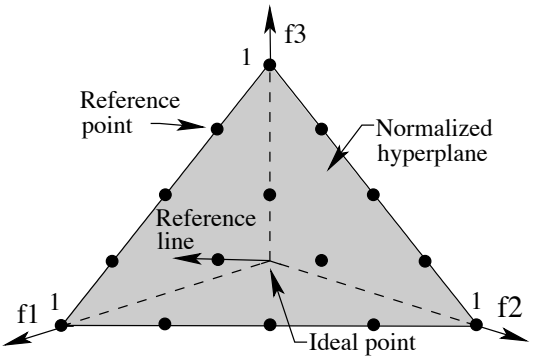


Figure 2. Normalized reference plane for a three-objective problem with $p = 4$.

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

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

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

Note that the set of reference points can either be predefined in a structured manner or supplied preferentially by the user. Our adopted version of NSGA-III in this paper is based on Das and Dennis's [55] systematic approach that automatically places points on a normalized hyper-plane – an (M-1)-dimensional unit simplex – which is equally inclined to all objective axes and has an intercept of one on each axis (cf. Figure 2). If p divisions are considered along each objective, the total number of reference points (H) in an M-objective problem is given by:

$$H = \binom{M+p-1}{p} \tag{9}$$

For our refactoring recommendation problem, which is a three-objective problem (M = 3), the reference points are created on a triangle with apex at (1, 0, 0), (0, 1, 0) and (0, 0, 1). If four divisions (p = 4) are chosen for each objective axis, $H = \binom{3+4-1}{4}$, or 15 reference points will be created. For clarity, these reference points are shown in Figure 2, where each axis f_1, f_2, and f_3 corresponds to an objective function, code smells correction ratio, quality metrics improvement, and the number of introduced patterns. Note that all fitness values are normalized in the range [0..1] so that they have an identical range with the reference points.

Algorithm 5 Generation t of NSGA-III procedure

Input: H structured reference points Z^s or supplied aspiration points Z^a, parent population P_t
Output: P_{t+1}
$S_t = \emptyset, i = 1$
Q_t = Recombination+Mutation(P_t)
$R_t = P_t \cup Q_t$
(F_1, F_2,...) = Non-dominated-sort(R_t)
repeat
    $S_t = S_t \cup F_i$ and $i = i + 1$
until $\mid S_t \mid \geq N$
Last front to be included: $F_l = F_i$
if $\mid S_t \mid = N$ **then**
    $P_{t+1} = S_t$, break
else
    $P_{t+1} = \bigcup_{j=1}^{l-1} F_j$
    Points to be chosen from $F_l : K = N - \mid P_{t+1} \mid$
    Normalize objectives and create reference set Z^r:
    `Normalize`($\mathbf{f}^n$, S_t, Z^r, Z^s, Z^a)
    Associate each member s of S_t with a reference point:
    [π(s),d(s)]= `Associate`(S_t, Z^r)
    %π(s): closest reference point,
    d: distance between s and π(s)
    Compute niche count of reference point $j \in Z^r$:
    $\rho_j = \sum_{s \in S_t/F_l}((\pi(s) = j)?1 : 0)$
    Choose K members one at a time from F_l to construct P_{t+1}:
    `Niching`($K, \rho_j, \pi, d, Z^r, F_l, P_{t+1}$)
end if

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

4.2. NSGA-III Adaptation for the Refactoring Recommendation Problem

4.2.1. Problem formulation The refactoring problem involves searching for a near-optimal refactoring solution among the set of candidate ones, which constitutes a huge search space. A refactoring solution is a sequence of refactoring operations where the goal is to apply the sequence to a software system S so as to to (1) minimize the number of code smells in S, (2) maximize the number of design patterns, and (3) improve the overall quality (using software metrics). This formulation is given as follows:

$$\begin{cases} \text{Minimize } F(x,S) = [f_1(x,S), f_2(x,S), f_3(x,S)] \\ \\ \text{Subject to } x = (x_1, x_2, ..., x_n) \in X \end{cases}$$

where X is the set of all legal refactoring sequences starting from S that satisfy the semantic constraints described in Section 3.2 , x_i is the i-th refactoring operation, and $f_k(x,S)$ is the k-th objective. Note that we formulate the refactoring problem as a minimization multi-objective problem (MOP) and observe that maximization can be easily turned to minimization based on the duality principle.

4.2.2. Solution approach This subsection describes how we adapted NSGA-III to the problem of refactoring recommendation in terms of solution representation, variation and evaluation.

Solution representation. As defined in the previous section, a solution consists of a sequence of n refactoring operations applied to different code elements in the source code. In order to represent a candidate solution (individual/chromosome), we use a vector-based representation. As depicted in Figure 3, each vector's dimension represents a refactoring operation where the order of applying these refactoring operations corresponds to their positions in the vector. For each of these refactoring operations, we specify pre-conditions in the style of Opdyke [33] to ensure the feasibility of their application. The initial population is generated by assigning randomly a sequence of refactorings to some code fragments. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles they play in performing the refactoring operation. An actor can be a package, class, field, method, parameter, statement, or variable.

1	move field (Person, Employee, salary)
2	extract class(Person, Adress, streetNo, city, zipCode, getAdress(), updateAdress())
3	move method (Person, Employee, getSalary())
4	push down field (Person, Student, studentId)
5	inline class (Car, Vehicle)
6	move method (Person, Employee, setSalary())
7	move field (Person, Employee, tax)
8	extract class (Student, Course, courseName, CourseCode, addCourse(), rejectCourse())

Figure 3. Example of solution representation in MORE.

Moreover, each refactoring operation should comply with its semantic constraints to be considered 'valid'. In Table III, we specify, for each refactoring operation, which semantic constraints are taken into account to ensure that the refactoring operation preserves design coherence.

Solution variation. In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table III. Semantic constraints considered for each refactoring operation.

Refactorings	VS	DS	IS	FIU	FMC	VPC	SPC	StPC
Move method	x	x						
Move field	x	x						
Pull up field	x	x		x				
Pull up method	x	x	x					
Push down field	x	x		x				
Push down method	x	x		x				
Inline class	x	x						
Extract class	x	x						
Move class	x	x						
Extract interface	x	x						
Visitor pattern refactoring						x		
Factory method refactoring					x			
Singleton pattern refactoring							x	
Strategy pattern refactoring								x

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 (cf. Figure 4). This operator must ensure to respect the length limits by eliminating randomly some refactoring operations. It is important to note that in multi-objective optimization, it is better to create children that are close to their both parents in order to have a more efficient search process [30]. For this reason, we control the cutting point k of the one-point crossover operator by restricting its position to be either belonging to the first third of the refactoring sequence or belonging to the last third, i.e., $k \in]0, 0.33]$, or $k \in [0.66, 1[$, respectively. For example, in Figure 4, $k = 0.3$ which corresponds to cutting 30% of Parent 1, and therefore the point-cut position is 2 in the vector of each parent solution. For mutation, we use the bit-string mutation operator. As depicted in Figure 5, the mutation operator picks probabilistically one or more refactoring operations from the associated sequence and replaces them by other ones from the initial list of possible refactorings.

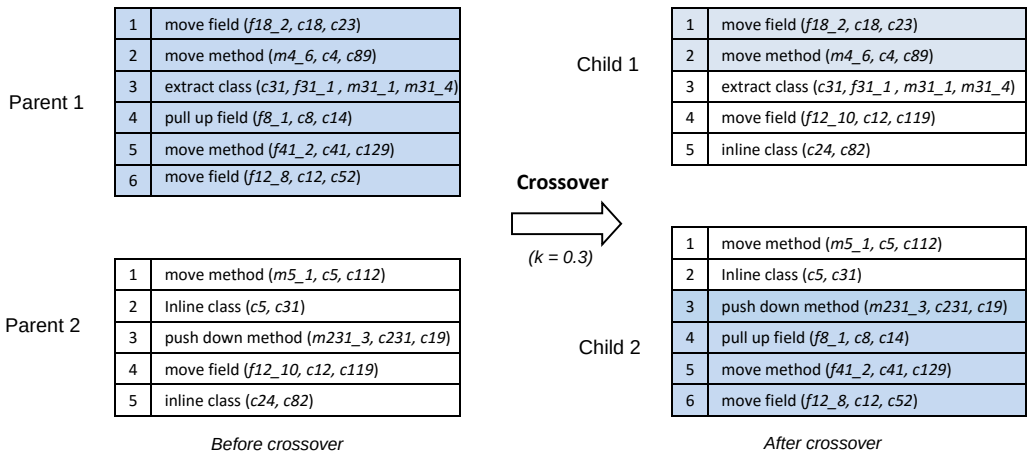


Figure 4. Example of crossover operator used.

Solution evaluation. To evaluate the fitness of each refactoring solution x to a system S, we used three objective functions according to each objective.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

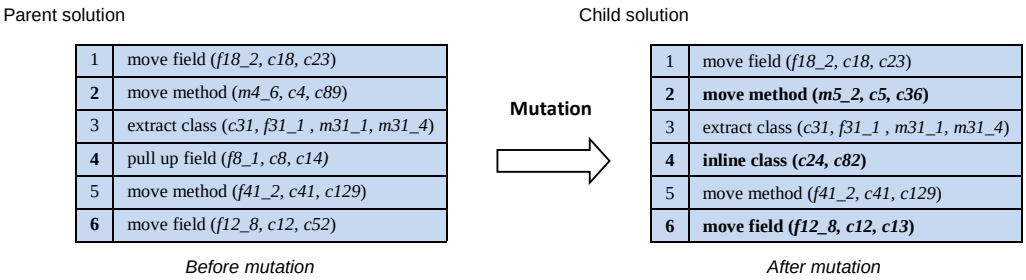


Figure 5. Example of mutation operator used.

1. *Code smells objective function*: It calculates the ratio of the number of corrected code smells to the initial number of code smells using the code smells detector component. The code smells correction ratio (CCR) is given by Equation 10:

$$CCR(x,S) = \frac{\text{number of corrected code smells}}{\text{initial number of code smells}} \tag{10}$$

2. *Design patterns objective function*: It calculates the number of produced design pattern instances (NP) using the design patterns detector component. NP is given by Equation 11:

$$NP(x,S) = DPA - DPB \tag{11}$$

where *DPA* and *DPB* are the number of design patterns, respectively, after and before refactoring. The *NP* values are then normalized in the range [0,1] using min–max normalization.

3. *Quality objective function*: It calculates the change in software quality using the QMOOD (Quality Model for Object-Oriented Design) model [31] to estimate the effect of the suggested refactoring solutions on quality attributes. We calculate the overall quality gain (QG) for the six QMOOD quality factors (reusability, flexibility, understandability, effectiveness, functionality, and extendibility) that are formulated using 11 low-level design metrics. Full details about these metrics are defined in Bansiya and Davis original work [31]. Let $Q = \{\mathrm{q}_1, \mathrm{q}_2, ...\mathrm{q}_6\}$ and $Q' = \{\mathrm{q}'_1, \mathrm{q}'_2, ...\mathrm{q}'_6\}$ be respectively the set of quality attribute values before and after applying the suggested refactorings, and $W = \{w_1, w_2, ...w_6\}$ the weights assigned to each of these quality factors. Then the total quality gain (QG) is estimated as follows:

$$QG(x,S) = \sum_{i=1}^{6} w_i \times (q'_i - q_i) \tag{12}$$

Creation of the initial population of solutions. To generate an initial population, we start by defining the maximum vector length (maximum number of operations per solution). The vector length is proportional to the number of refactorings that are considered and the size of the program to be refactored. A higher number of operations in a solution do not necessarily mean that the results will be better. Ideally, a small number of operations should be sufficient to provide a good trade-off between the fitness functions. This parameter can be specified by the user or derived randomly from the sizes of the program and the given refactoring list. During the creation, the solutions have random sizes inside the allowed range. To create the initial population, we normally generate a set of solutions ($PopSize$) randomly in the solution space.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table IV. Program statistics.

System	Release	#Classes	KLOC	#code smells	#Design patterns
Xerces-J	v2.7.0	991	240	97	36
JFreeChart	v1.0.9	521	170	84	23
GanttProject	v1.10.2	245	41	56	16
AntApache	v1.8.2	1191	255	112	38
JHotDraw	v 6.1	585	21	26	18
Rhino	v1.7R1	305	42	74	16
ArtOfIllusion	v2.8.1	459	87	63	13

5. THE DESIGN OF THE EMPIRICAL STUDY

In this section, we present our experimental study to evaluate the efficacy of our approach in fixing code smells, introducing design patterns and improving design quality.

5.1. Research Questions

Our study aims at addressing the following three research questions:

- **RQ1.** (*Sanity check*) How does the proposed approach perform compared to random search and other existing mataheuristic search methods?
- **RQ2.** (*Efficacy*) To what extent can the proposed approach improve the quality of software systems?
- **RQ3.** (*Comparison to state-of-the-art*) How does our approach perform compared to existing search-based refactoring approaches?

5.2. Software Systems Studied

We applied our approach to a set of seven well-known and well-commented industrial-size open source Java projects: Xerces-J[†], JFreeChart[‡], GanttProject[§], Apache Ant[¶], JHotDraw[‖], Rhino[**], and ArtofIllusion[††]. 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. Apache Ant 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. Finally, ArtOfIllusion is a 3D-modeler, renderer, and raytracer written in Java.

We selected these systems for our validation because they came from seven different organisations, involved different kinds of software engineering development and had different sizes, ranging from 245 to 1191 classes with a large number of both design pattern and code smell instances. Table IV provides some descriptive statistics about these seven programs.

Furthermore, as we previously note, in these corpora, we considered seven different types of code smell (*god class*, *feature envy*, *data class*, *spaghetti code*, *shotgun surgery*, *long parameter list* and *lazy class*) and four design patterns (*abstract method factory*, *visitor*, *singleton* and *strategy*).

[†]http://xerces.apache.org/xerces-j
[‡]http://www.jfree.org/jfreechart
[§]www.ganttproject.biz
[¶]http://ant.apache.org/
[‖]http://www.jhotdraw.org/
[**]http://www.mozilla.org/rhino
[††]www.artofillusion.org

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

5.3. Evaluation methodology

To answer our research questions, we conducted a set of experiments to apply MORE to our benchmark of seven medium and large-size open source software systems. Each experiment is repeated 31 times, and the obtained results are subsequently statistically analyzed with the aim of comparing our approach with existing multi-objective search algorithms as well as state-of-the-art refactoring recommendation approaches.

NSGA-III returns a set of near-optimal solutions instead of a single one; however, for our validation, we require that MORE proposes one single solution. To this end, and in order to fully automate our approach, MORE extracts and suggests only one optimal solution from the returned set of solutions. To this end, we used a technique based on Euclidean distance as described in [36]. Equation 13 is used to choose the solution that corresponds to the best trade-off between (*i*) CCR, (*ii*) NP and (*iii*) QG. The ideal solution should have the best CCR, NP and QG scores (ideally, their normalized score equals to 1). Hence, we select the nearest suggested solution to the ideal solution in terms of Euclidean distance. Let PF the set of solutions in the Pareto front, then the best solution to be returned by MORE is defined as follows:

$$BestSol = \min_{\forall s \in PF} \sqrt{(1 - CCR(s))^2 + (1 - NP(s))^2 + (1 - PF(s))^2)} \quad (13)$$

5.3.1. Research method for RQ1. To answer **RQ1**, we compared our NSGA-III formulation against random search (RS) [56] in terms of search space exploration. The goal is to make sure that there is a need for an intelligent method to explore our huge search space of possible refactoring solutions. In addition, to justify the adoption of NSGA-III, we compared our approach against two other popular search algorithms namely NSGA-II [29], and MOEA/D [57]. RQ1 serves the role of a *sanity check* and standard 'baseline' question asked in any attempt at an SBSE formulation [41].

Unlike mono-objective search algorithms, multi-objective evolutionary algorithms return as output a set of *non-dominated* (also called *Pareto optimal*) solutions obtained so far during the search process. A number of performance metrics for multi-objective optimization have been proposed and discussed in the literature, which aim to evaluate the performance of multi-objective evolutionary algorithms. Most of the existing metrics require the obtained set to be compared against a specified set of Pareto optimal reference solutions. In this study, the generational distance (GD) [58] and inverted generational distance (IGD) [59] are used as the performance metrics since thay have been shown to reflect both the diversity and convergence of the obtained non-dominated solutions.

- *Generational Distance (GD)*: computes the average distance between the set of solutions, *S*, from the algorithm measured and the reference set *RS*. The distance between *S* and *RS* in an *n* objective space is computed as the average n-dimensional Euclidean distance between each point in S and its nearest neighbouring point in *RS*. GD is a value representing how "far" S is from RS (an error measure).
- *Inverted Generational Distance (IGD)*: is used as a performance indicator since it has been shown to reflect both the diversity and convergence of the obtained non-dominated solutions [59]. The *IGD* corresponds to the average Euclidean distance separating each reference solution set (*RS*) from its closest non-dominated one *S*. Note that for each system we use the set of Pareto optimal solutions generated by all algorithms over all runs as reference solutions.

5.3.2. Research method for RQ2. To answer **RQ2**, we conducted a quantitative and qualitative study.

Quantitative evaluation. The quantitative study evaluates the efficacy of our approach for 1) fixing code smells, 2) introducing design patterns, 3) improving software quality.

- To evaluate the efficacy of our approach in fixing code smells, we calculated the code smells correction ratio (*CCR*) as given by Equation 10 on our benchmark.

Copyright © 2017 John Wiley & Sons, Ltd.

Prepared using smrauth.cls
DOI: 10.1002/smr

- To evaluate the efficacy of our approach in introducing design patterns, we calculated the number of new design pattern instances (*NP*) that are introduced as given by Equation 11.
- To evaluate the efficacy of our approach for improving software quality, we calculated the overall quality gain (*QG*) using the QMOOD (Quality Model for Object-Oriented Design) model [31] as given by Equation 12.

Qualitative evaluation. In addition to the quantitative evaluation which is widely used to evaluate existing refactoring approaches [6, 11, 12, 37], it is important to qualitatively evaluate the applicability and meaningfulness of the recommended refactorings from developer's perspective. That is, the recommended refactorings can be successfully applied, improve the code quality, but this can lead to arbitrary changes that affect the semantic coherence of the refactored program. Hence, the recommended refactoring operations should not only remove code smells and improve quality, but most importantly, should be meaningful from a developer's point of view.

To this end, we conducted a qualitative evaluation with potential users of our technique. Our evaluation is based on a survey to collect the feedback of developers about MORE's recommendations. Our study is conducted as follows:

Subjects: Our study involves seven volunteer participants to conduct our experiments; five participants are PhD students in software engineering at the University of Michigan, and two participants are working at General Motors as senior software developers. Participants were first asked to fill out a pre-study questionnaire containing six questions. The aim of the questionnaire is to collect background information of 1) their programming experience, 2) their familiarity with software refactoring, 3) their knowledge about code smells, 4) their knowledge about deign patterns, 5) their experience with quality assurance and software metrics; and 6) their experience with the studied open-source systems. The participants had a programming experience in Java ranging from 4 to 11 years. All participants were familiar with refactoring, code smells and design patterns. They have also an experience with some of the studied systems.

Process: All the participants who agreed to participate to the study received a questionnaire, a guide that advises on how to fill out the questionnaire, and the source code of the studied systems, in order to evaluate the relevance of the recommended refactorings to apply. The questionnaire is organized in an Excel file with hyperlinks to visualize the source code of the affected code elements easily. Participants were aware that they are going to evaluate the semantic coherence of refactoring operations, but do not know the particular experimental research questions (the approaches and algorithms being compared). We asked the participants to manually evaluate, for each system, 10 refactoring operations that are selected at random from the suggested list of refactoring solutions of each approach. Participants are asked to assign a correctness score for each refactoring according to its relevance and meaningfulness. Possible answers follow a five-point Likert scale [61] to express their level of agreement by a score in the range [0,5]: 1. *Not at all relevant*; 2. *Slightly relevant*; 3. *Moderately relevant*; 4. *Relevant*, and 5. *Extremely relevant*. Note that no 'neutral' option was offered, as we require that developers form and express an opinion regarding each evaluated refactoring.

Since the application of refactorings is a subjective process that depends on the developer's intention, it is normal that not all the participants have the same opinion. To this end, we consider a refactoring operation as meaningful if its assigned score is $\geqslant 3$. Then for each refactoring operation, we consider the majority of votes (at least 4 out of 7 of the participants) to determine if a recommended refactoring is relevant or not. We therefore define the metric refactoring meaningfulness (*RM*) that corresponds to the number of relevant refactoring operations over the total number of refactorings given to the participants to evaluate. RM is given by Equation 14.

$$RM = \frac{\text{\# meaningful refactorings}}{\text{\# evaluated refactorings}} \tag{14}$$

Note that the questionnaire is completed anonymously thus ensuring confidentiality. During the entire process, participants were encouraged to think aloud and to share their opinions, issues,

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

detailed explanations and ideas with the organizers of the study (one graduate student and one faculty from the University of Michigan) and not only answering the questions. In addition, a brief tutorial session was organized for every participant around refactoring to make sure that all of them have a minimum background to participate in the study. All the developers performed the experiments in a similar environment: similar configuration of the computers, tools (Eclipse, Excel, etc.) and facilitators of the study. We also added a short description of this instruction for the participants. The average time required to finish all the questions was 4h20min divided into two sessions.

5.3.3. Research method for RQ3. To answer **RQ3**, we compared MORE results against three state-of-the-art refactoring approaches: Seng et al. [11], Jensen et al. [18], and Kessentini et al. [60], in terms of CCR, NP, and QG. These approaches are designed each for a specific purpose. Seng et al.'s approach [11] aims to find a sequence of refactoring operations that improves specific quality metrics in the program being refactored. Jensen et al. [18] aim to find a combination of refactorings that introduces new design patterns to the initial software system, while Kessentini et al.'s approach [60] aims to find refactoring solutions that minimize as much as possible the number of code smells in the code being refactored. To make the comparison fair, we apply the refactorings suggested by each approach, and then calculate our evaluation metrics (CCR, NP, and QG).

5.4. Algorithms Parameter Tuning

An important aspect of metaheuristic search algorithms lies in parameter selection and tuning, something that is necessary to ensure not only fair comparison, but also potential replication. The initial population/solution of NSGA-III, NSGA-II, MOEA/D, and RS is completely random. The stopping criterion is when the maximum number of fitness evaluations, set to 350,000, is reached. After several trial runs of the simulation, the parameter values of the four algorithms are fixed to 100 solutions per population ($popSize$) and 3,500 iterations. For the variation operators, the crossover rate, p_c, is set to 0.9 and mutation, p_m, to a probability of 0.4. We used a high mutation rate to ensure the diversity of the population and to avoid premature convergence [62]. After several trial runs of the simulation, these parameter values are fixed. For instance, the $popSize$ parameter was tested with several values including 50, 100, 150, 200, 300, 500 and 1000; both p_c and p_m were tested with different values in the range $[0, 1]$ with a step equal to 0.1; the number of iterations was tested with values of 200,000, 250,000, 300,000, 350,000, 400,000 and 500,000. Indeed, there are no general rules to determine these parameters, and thus we set the combination of parameter values by trial-and-error, a method that is commonly used by the SBSE community [63, 64].

5.5. Inferential Statistical Tests Used

Due to the stochastic nature of the employed algorithms, they may produce different results when applied to the same problem instance over different runs. In order to cope with this stochastic nature, the use of statistical testing is essential to provide support and draw statistically sound conclusions derived from analyzing such data [63]. To this end, we used the Wilcoxon rank sum test in a pairwise fashion [65, 66] in order to detect significant performance differences between the algorithms under comparison. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. We set the confidence limit, α, at 0.05. In these settings, each experiment is repeated 31 times, for each algorithm and for each system. The obtained results are subsequently statistically analyzed with the aim to compare our NSGA-III approach with NSGA-II, MOEA/D and random search (RS). Furthermore, we used the Bonferroni [67] correction to reduce the chances of obtaining false-positive results when multiple pair wise tests are performed on a single set of data to compare NSGA-II, NSGA-III, and MOEA/D.

While the Wilcoxon rank sum test verifies whether the results are statistically different or not, it does not give any idea about the difference magnitude. To this end, we investigate the effect size using Cliff's delta statistic [68]. The effect size is considered: (1) negligible if $\mid d \mid < 0.147$, (2) small if $0.147 \leq \mid d \mid < 0.33$, (3) medium if $0.33 \leq \mid d \mid < 0.474$, or (4) high if $\mid d \mid \geq 0.474$.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table V. Median values of the quality indicators GD and IGD for the four compared algorithms NSGA-III, NSGA-II, MOEA/D and RS (best values are in bold).

System	GD				IGD			
	NSGA-III	NSGA-II	MOEA/D	RS	NSGA-III	NSGA-II	MOEA/D	RS
Xerces-J	**4.077E-03**	4.705E-03	7.682E-03	5.012E-03	**6.912E-02**	8.864E-02	7.542E-02	6.687E-01
JFreeChart	**5.6139E-02**	4.491E-01	4.657E-01	1.701E+00	**5.591E-04**	8.815E-04	7.687E-04	4.091E-02
GanttProject	7.636E-03	**6.385E-03**	2.645E-02	4.278E-01	3.827E-03	**2.912E-03**	4.336E-03	8.827E-02
AntApache	**4.777E-03**	6.118E-03	8.301E-03	1.528E-01	**3.592E-02**	6.392E-02	7.110E-02	4.705E-01
JHotDraw	**5.337E-03**	3.318E-02	1.947E-02	8.594E-02	6.231E-02	7.684E-02	**5.816E-02**	7.439E-01
Rhino	5.398E-02	**2.906E-02**	7.150E-02	8.0613E-01	**2.002E-03**	7.846E-03	7.123E-03	4.005E-01
ArtOfIllusion	**7.583E-03**	1.06677E-02	3.090E-02	5.529E-01	**4.603E-03**	8.440E-02	8.529E-02	6.756E-01

6. RESULTS

This section reports the results of our empirical study. We first start by answering our research questions. We then present further discussions on the obtained results.

6.1. Results for RQ1: Sanity check

Tables V and VI present respectively the results of the metric indicators GD and IGD, and the results of the statistical significance and effect size tests. We observe that NSGA-III clearly outperforms RS in all the seven studied systems with a Cliff's delta effect size of 'high' in both performance indicators *GD* and *IGD*. This is mainly due to the large search space to explore in order to find suitable combinations of refactoring operations. This requires a heuristic-based search rather than random search.

In more detail, Figure 6 and Table VI report our results for RQ1. We observe that over 31 runs, NSGA-III outperforms NSGA-II, in terms of *GD*, in 5 out of 7 systems with 'high' effect size. In the cases of Xerces-J and ArtOfIllusion the effect size was 'medium'. Similarly, NSGA-III significantly outperforms MOEA/D in the 7 systems with a 'high' effect size in 6 out of 7 cases; for JHotDraw, the effect size was 'medium'. In terms of *IGD*, NSGA-III provides better performance than both NSGA-II and MOEA/D in 5 out of 7 systems with 'high' effect size. NSGA-II provides better IGD results for GanttProject with high effect size, and MOEA/D achieved better results on JHotDraw with 'medium' effect size. Overall, we observed that NSGA-III tends to achieve better performance for large software systems when comparing to NSGA-II and MOEA/D.

Furthermore, we can also get a more informative sense of the distributions of results for the three competitive algorithms from the boxplots shown in Figure 6. From these boxplots, we can see that the variance in the results from NSGA-III is lower for both *GD* and *IGD* than the other two. The obtained results suggest that this is because there are simply fewer solutions that converge toward the last generation of NSGA-III. In addition, because *IGD* and *GD* metrics combine the information of convergence and diversity, the results indicate that NSGA-III has the best overall performance. This is very promising in multi-objective search problems, and it would be interesting to adopt it in solving other software engineering problems.

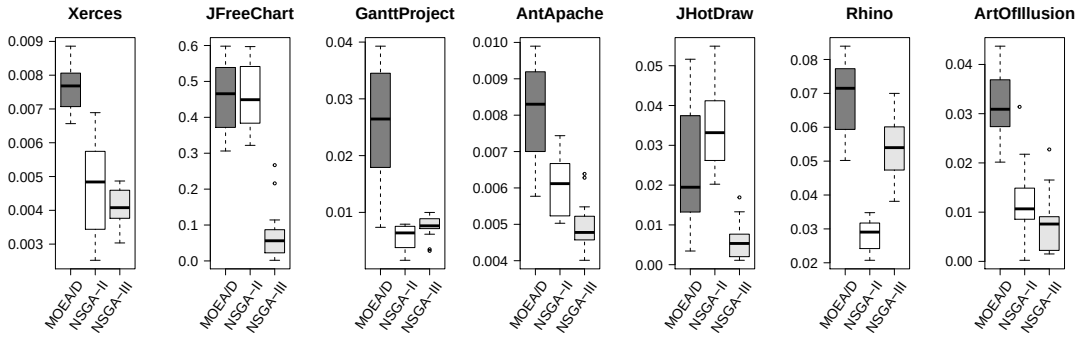
To conclude, the obtained results provide evidence that NSGA-III is the best search technique for the refactoring recommendation problem, particularly for the larger systems that we studied. Consequently, we can conclude that there is empirical evidence that our formulation passes the sanity check (RQ1).

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

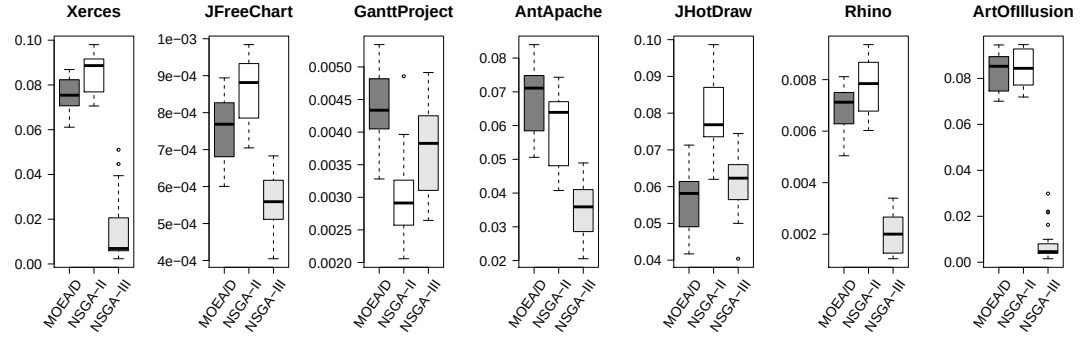
DOI: 10.1002/smr

Table VI. Statistical significance p-value (α=0.05) and effect size comparison results of NSGA-III against NSGA-II, MOEA/D and RS. A statistical difference is accepted at $p \leqslant 0.05$.

System		NSGA-III vs NSGA-II		NSGA-III vs MOEA/D		NSGA-III vs RS	
		GD	IGD	GD	IGD	GD	IGD
Xerces-J	p-value	0.06062	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	medium	high	high	high	high	high
JFreeChart	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	high	high	high	high	high	high
GanttProject	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	high	high	high	high	high	high
AntApache	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	high	high	high	high	high	high
JHotDraw	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	high	high	high	medium	high	high
Rhino	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	high	high	high	high	high	high
ArtOfIllusion	p-value	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
	effect size	mehuim	high	high	high	high	high



(a) Generational distance (GD)



(b) Inverted generational distance (IGD)

Figure 6. Boxplots for the quality measures GD and IGD results over 31 independent simulation runs of NSGA-III, NSGA-II, and MOEA/D.

6.2. Results for RQ2: Efficacy

The results for RQ2 are summarized in Table VII. After applying the refactoring operations proposed by MORE using NSGA-III, we found that, on average, 84% of the detected code smells were fixed (CCR) for all the seven studied systems. This high score is considered significant in improving the quality of the refactored systems by fixing the majority of existing code smells of

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

varying types (god class, feature envy, data class, spaghetti code, shotgun surgery, long parameter list and lazy class, as described in Table I). It is worth observing that we found the majority of non-fixed code smells are related to the god class type. Indeed, this type of code smell usually requires a large number of refactoring operations and is known to be very difficult to fix.

Moreover, MORE succeeded in introducing a reasonable number of instances of design patterns. Table VII shows the number of new design pattern instances, NP, introduced for each system. MORE successfully introduced an average of 6 design patterns (NP) per system, covering all types of supported design patterns: factory method, visitor, singleton and strategy. The recommended refactorings can support software developers who might be interested in automatically introducing new design patterns to make their software systems more understandable, flexible, and maintainable. The lowest NP value was recorded for JHotDraw (NP = 3). JHotDraw was developed by Erich Gamma and Thomas Eggenschwiler [14] as a design exercise to promote design patterns, so it is to be expected that many design patterns already exist in JHotDraw and there is little opportunity to add new design pattern instances.

It should be noted that introducing a design pattern does not necessarily imply that the design has been improved, and frequently the result of excessive pattern application is an over-engineered, hard-to-maintain system [69]. The fact that only a limited number of design patterns were introduced to each application suggests that our multi-objective approach, coupled with the semantic constraints of Section 3.2, did indeed prevent the blind application of many useless pattern instances. Whether or not the design patterns introduced by MORE make sense to a developer is assessed in the qualitative evaluation conducted with developers.

In terms of quality improvement (QG), as reported in Table VII, MORE succeeded in improving the quality of all the studied systems, with an average QG score of 0.4 in terms of QMOOD quality attributes. In more detail, Figure 7 shows the QG values for each QMOOD quality attribute after applying the recommended refactorings by MORE, for each studied system. We found that the systems quality increase across the six QMOOD quality factors. We observe that understandability is the quality factor that has the highest QG score; whereas the effectiveness quality factor has the lowest one. This finding can be explained in two possible ways: 1) the majority of non-fixed code smells are god class and spaghetti code which are known to increase the coupling (DCC) within classes (which heavily affect the quality index calculation of the effectiveness factor); 2) the vast majority of suggested refactoring types were move method, move field, and extract class (Figure 10) that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This can be justified by the fact that JHotDraw is known to be of high quality due to its exemplary design, its development model, and its widespread use. Indeed, it contains a few number of code smells comparing to the six other studied systems (cf. Table IV).

We can also get a more qualitative sense, we assess the relevance/meaningfulness of the suggested refactoring solutions from developers' perspective. To this end, we report in Figure 8 the results of the empirical study conducted with seven participants to evaluate the recommended refactorings on our studied systems. We observe from the figure that most of the recommended refactorings are evaluated as relevant with an average of 69% of the proposed refactoring operations being considered as semantically meaningful. The lowest RM value was 59% for Xerces. In fact, Xerces is known for its high change frequency and its complex design that have led to extensive refactoring activities in the coiurse of the past 15 years [71]. Moreover, we observed that, in general, for large-size programs (e.g., AntApache and JFreeChart, but excluding Xerces), the performance in terms of RM achieved by MORE is more notable than it is for the smaller programs.

Looking at this in more detail, Figure 9 shows the relevance (meaningfulness) of the recommended refactorings achieved by MORE for each refactoring type. Only less than 13% of recommended refactorings were marked as not at all relevant (score = 1) by the participants; and an average of 18% of recommended refactorings are assessed as slightly relevant. An average of 28.64% are marked as moderately relevant, 29.71% as relevant and 10.64% as extremely relevant. This confirms the importance of the recommended refactorings for developers, and shows that they

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

Table VII. CCR, # Patterns, and QG median values of 31 independent runs of MORE, Seng et al., Jensen et al., and Kessentini et al.

System	Approach	CCR (%)	# Patterns	QG
Xerces-J	MORE	90	12	0.49
	Seng et al.	23	0	0.54
	Jensen et al.	14	31	0.41
	Kessentini et al.	88	0	0.32
JFreeChart	MORE	88	6	0.54
	Seng et al.	21	0	0.59
	Jensen et al.	24	19	0.42
	Kessentini et al.	86	0	0.41
GanttProject	MORE	88	7	0.35
	Seng et al.	24	1	0.33
	Jensen et al.	33	14	0.35
	Kessentini et al.	84	0	0.21
AntApache	MORE	86	4	0.51
	Seng et al.	7	0	0.52
	Jensen et al.	12	28	0.51
	Kessentini et al.	87	0	0.39
JHotDraw	MORE	83	3	0.17
	Seng et al.	38	0	0.19
	Jensen et al.	25	9	0.14
	Kessentini et al.	88	0	0.1
Rhino	MORE	72	4	0.51
	Seng et al.	12	0	0.54
	Jensen et al.	18	11	0.32
	Kessentini et al.	78	0	0.36
ArtOfIllusion	MORE	83	6	0.26
	Seng et al.	10	0	0.29
	Jensen et al.	17	12	0.18
	Kessentini et al.	90	0	0.21
Average (all systems)	MORE	84	6	0.4
	Seng et al.	26	0.14	**0.43**
	Jensen et al.	20	**18**	0.31
	Kessentini et al.	**86**	0	0.29

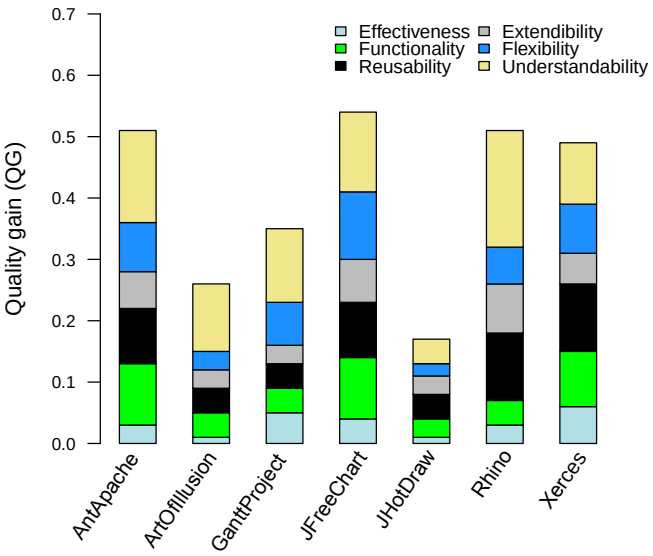


Figure 7. QMOOD quality factors gain obtained for MORE.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

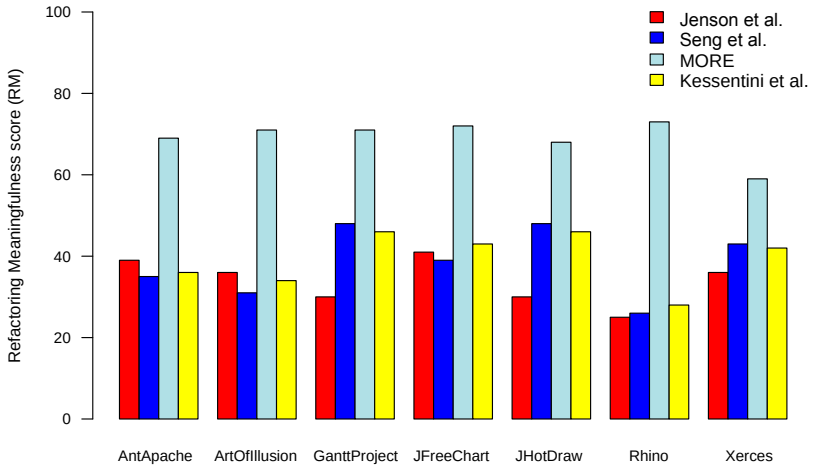


Figure 8. MORE comparison against Seng et al., Jensen et al., and Kessentini et al. in terms of refactoring meaningfulness (RM). A refactoring operation is considered meaningful if its assigned score is $\geqslant 3$.

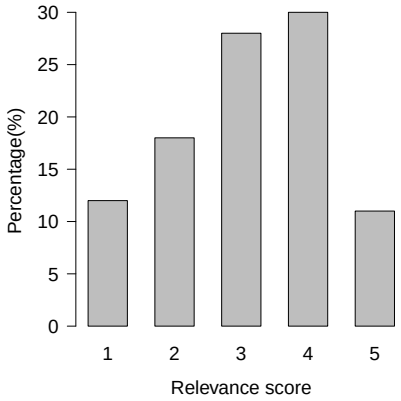


Figure 9. The average relevance score (RM) of the recommended refactorings by MORE. 1. *Not at all relevant*; 2. *Slightly relevant*; 3. *Moderately relevant*; 4. *Relevant*, and 5. *Extremely relevant*.

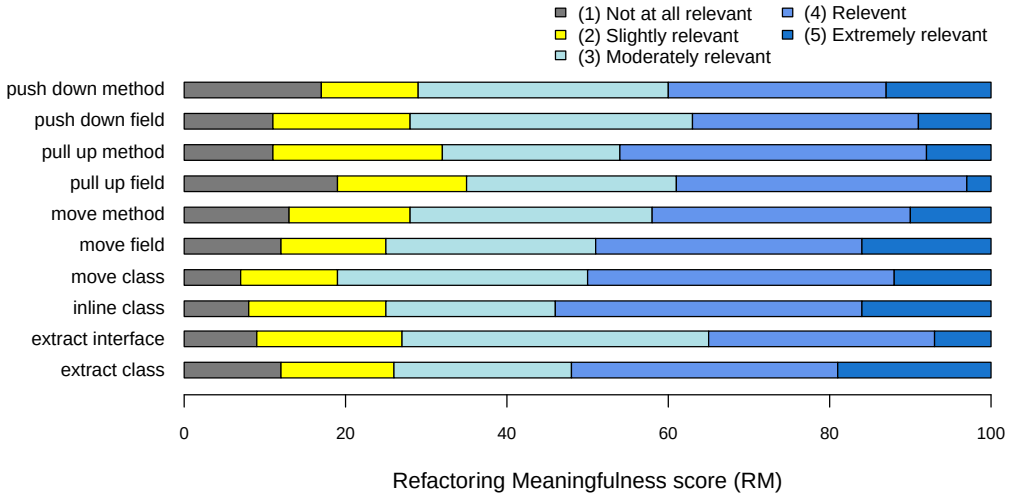
recognize that these refactorings can be useful in improving the quality of the studied software systems.

On the negative side, 31% of MOREs recommendations (13% definitely not at all relevant and 18% slightly relevant) were rejected by the participants. Indeed, we believe that it is difficult to automatically understand the semantics of source code through pre-defined heuristics and semantic constraints. The role of the developer remains fundamental to decide whether a refactoring could be applied or not.

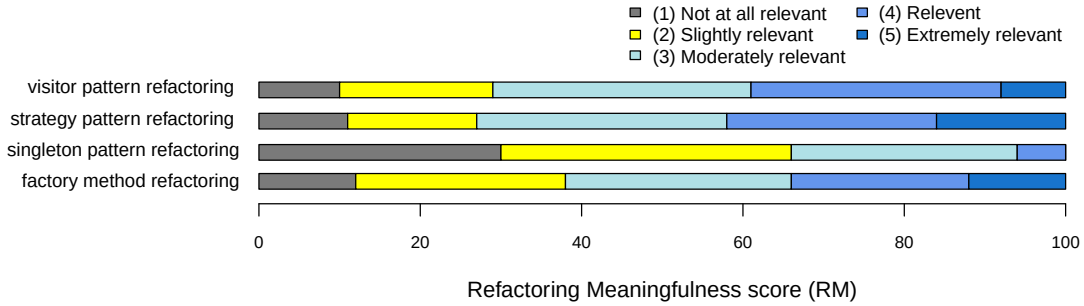
To better evaluate the relevance of the recommended refactorings, we investigated the types of refactorings that developers might consider more or less meaningful than others. Figure 10 shows that *move method*, *extract class* and *move class* are considered as the extremely relevant refactorings. In addition, the recommended *inline class* and *move field* are also considered relevant and meaningful. This can be explained by the fact that the developers are more focusing on quality issues that are related to class size, feature distribution, and coupling/cohesion problems. Additionally, the vocabulary-based similarity constraint (VS) was pertinent to prevent incoherent refactorings especially for *move method* and *move class*. On the other hand, *pull up field*, *extract interface* and *visitor pattern refactoring* recorded the lowest values for "extremely relevant". One possible explanation is that approving such refactorings requires in general that several coupled classes be investigated and studied and this is likely to be a more complex task for the developer.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr



(a) The RM results obtained for each refactoring type.



(b) The RM results obtained for each design pattern refactoring type.

Figure 10. The average refactoring meaningfulness score (RM) of different types of recommended refactorings by MORE.

It is also worth to notice that the worst results was obtained with *singleton pattern refactoring* where the RM score was relatively poor compared to the other refactorings. The reason might be due to intention to identify the singleton opportunity based only on static analysis. Dynamically, we can check if at most one instance of the candidate singleton class is created during runtime. That would ensure that there are no false positives. As future work, we plan to combine both static and dynamic analysis to improve the recommendation of singleton patterns refactorings.

6.3. Results for RQ3: Comparison to state-of-the-art

The results for RQ3 are presented in Table VII, that presents the median values of CCR, NP and QG over 31 independent simulation runs after applying the refactoring operations proposed by MORE, Seng et al., Jensen et al., and Kessentini et al.

As described in Table VII, after applying the refactoring operations proposed by MORE, we found that more than 84% of detected code smells were fixed (CCR) as an average across all the seven studied systems. For instance, for GanttProject, 75% (9 out of 12) of god classes, 86% (6 out of 7) of feature envy, 94% (15 out of 16) of spaghetti code, 93% (13 out of 14) of data classes are fixed. This score is comparable to the correction score achieved Kessentini et al. (an average of 86%). However, MORE achieved a significantly higher results than those of Seng et al., and Jensen et al., having respectively only 26% and 20% on average for all the studied systems.

In terms of introduced design patterns (NP), Jensen et al. achieves the highest score by introducing on average 18 design patterns for the seven systems. This score is higher than the one obtained by MORE (an average of 6 patterns per system). This can be explained by the fact that Jensen et al.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

apply design patterns without considering whether the design pattern is needed or not in that code fragment, i.e., the sole aim is to add as many design patterns as possible. From our perspective, this is unlikely to be useful and efficient in practice. For Seng et al. and Kessentini et al. we found that they are not able to produce design patterns (only 0.14 and 0 pattern per system respectively). This might be mainly due to the lists of generic refactorings they use which are not geared for the introduction of design patterns.

Furthermore, MORE achieves a QG score of 0.4 which is slightly less that the value of 0.43 achieved by Seng et al. This is mainly due to the fact that quality metrics improvement is the main component in the objective function of Seng et al. Furthermore, MORE provides comparable QG results to those of Jensen et al. (0.4). On the other hand, Kessentini et al. turns out to be the worst approach with a QG score of 0.29. Indeed, the approach of Kessentini et al. is driven only by code smell correction and not directed at improving quality metrics. This interesting result confirms that fixing code smells does not always mean that quality metrics will also be significantly improved. On the other hand, despite the significant improvement in terms of QG for Seng et al. (the highest score among the four compared approaches), it is not effective at fixing code smells (only 26% of code smells are fixed). Thus, these results provide further evidence that improving quality metrics does not necessarily mean that existing code smells are fixed. Indeed, the link between code smells and software metrics is not obvious [70].

In terms of refactoring meaningfulness (RM) from a developers perspective, Figure 8 reports the achieved results for the seven studied systems. We observe that MORE achieved 69% of RM which is significantly superior compared to Seng et al., Kessentini et al. and Jensen et al. having respectively only 39%, 38% and 34% as RM scores. This is mainly due the semantic constraints that are required to be satisfied before recommending refactorings, hence preventing arbitrary changes, unlike Seng et al., Kessentini et al. and Jensen et al. who do not consider the semantics of the program being refactored. Another reason can be that MORE provides a diversified sequence of refactorings to cover as much as possible the detected code smells and other quality issues.

This finding has actionable conclusions for software developers conducting software refactoring. Providing a trade-off between different conflicting quality objectives from different perspectives, i.e. code smells, design patterns and quality metrics, is required, which is one of the main purposes of MORE. Thus, developers who are interested mainly in fixing code smells can select solutions from the Pareto surface that provide high CCR values independently from the two other objectives. If the developer seeks to introduce design pattern instances to their code then they need to focus on the part of the Pareto surface that provides high NP values. Similarly, the developer who seeks to improve quality metrics of the code can the ignore the parts of the Pareto surface related to code smells and design patterns, and select a solution from the Pareto surface that maximizes the value of QG. If the developer seeks a trade-off between all the objectives, they should focus their attention on the middle part of the Pareto surface. Hence, as the three objectives are conflicting, maximizing the code smell correction score is indeed possible, but only at the cost of sacrificing some of the other objectives.

6.4. Discussions

To get more qualitative sense, we asked some of the participants to comment on their decisions in order to get a deeper view of the achieved results and to help us in future improvements of MORE. One of the refactoring operations suggested by MORE for Xerces-J is to move the method `fillXMLAttributes()` (cf. Figure 11) from the class `SchemaContentHandler` to the class `XMLAttributesImpl`. One of the participants comment on this as follows: "*I would strongly recommend apply this refactoring as the method `fillXMLAttributes()` manipulates objects of type `XMLAttributesImpl` and calls three of the methods of this type which are: `removeAllAttributes()`, `addAttributeNS()` and `setSpecified()`.*" Furthermore, the participant commented that the latter two methods are called by the original method within a `for` loop which may, in their opinion, significantly increase coupling. On the other hand, `fillXMLAttributes()` accesses only one service (`fillQName()`) from its current class. The participant explained that this method is in charge of removing all of the XML attributes along with

Copyright © 2017 John Wiley & Sons, Ltd.

Prepared using smrauth.cls
DOI: 10.1002/smr

```
private void fillXMLAttributes(Attributes atts) {
        fAttributes.removeAllAttributes();
        final int attrCount = atts.getLength();
        for (int i = 0; i < attrCount; ++i) {
            fillQName(fAttributeQName, atts.getURI(i), atts.getLocalName(i), atts.getQName(i));
            String type = atts.getType(i);
            fAttributes.addAttributeNS(fAttributeQName, (type != null) ? type :
XMLSymbols.fCDATASymbol, atts.getValue(i));
            fAttributes.setSpecified(i, true);
        }
    }
```

Figure 11. An example of one of MORE's refactoring suggestions for Xerces-J: move the method `fillXMLAttributes()` from the class `SchemaContentHandler` to the class `XMLAttributesImpl`.

all its associated entities. Then it adds a set of new attributes; each added attribute will be marked as specified in the XML instance document unless set otherwise using the `setSpecified()` method. It is clear that both source and target classes share several common identifiers (vocabulary) and therefore they are semantically related. Thus, it makes more sense to move the method `fillXMLAttributes()` to `XMLAttributesImpl`.

We asked another participant to justify their decision to accept (with a *relevant* vote) an extract class refactoring suggest by MORE for the class `GanttGraphicArea` in GanttProject[‡‡].MORE suggests to extract the following attributes: `margY`, `drag`, `arrayColor`, `myUIConfiguration` along with the following methods: `paintTasks()`, `paintATaskBilan()`, `paintATaskFather()`, `paintATaskchild()`. The participant confirmed that the class proposed to be extracted describes a separate entity which aims to draw different tasks in a specified graphic area. It is worth noting that MORE recommended another extract class refactoring for the same class `GanttGraphicArea`. This second extract class refactoring suggests extracting the following set of methods: `zoomMore()`, `zoomLess()`, `getZoom()`, `setZoom()` and the attributes: `zoomValue` and `oldDate` to a new class. The participant mentioned that although the first extract class would not be enough to fix the god class `GanttGraphicArea`, it is interesting to reapply other refactorings to the same smelly entity until the code smell is fixed. Furthermore, the participant commented: "*I would apply these two recommended refactorings as they would, in my opinion, significantly improve the understandability and flexibility of this badly implemented class* `GanttGraphicArea`". An interesting observation is that the participant indicated that even if some methods and attributes do not clearly describe a new concept, they can still be extracted into a new class only if they are structurally related, i.e., cohesive.

For the same class `GanttGraphicArea` depicted in Figure 12, MORE recommended applying a strategy pattern refactoring to the method `paintTasks()`. Indeed, strategy pattern is used when we have multiple algorithms for a specific task and the client should decide the actual implementation to be used according to the context. For this refactoring, the participant commented that *"`paintTasks()` would be a nice application of the strategy pattern that the original developers of GanttProject clearly missed"*. This method aims at deciding which paint method to execute for a `GanttTask` object (i.e., `task`) from 3 possible options: `paintATaskBilan()`, `paintATaskFather()`, or `paintATaskChild()`. Thus, this method could be easily turned into a strategy by extracting each branch of the '`if`' statement into its own class such that each class implements a common interface, and the decision on which one to use can be moved to the original class. Introducing several decision statements can obscure any calculation and make it likely to be misunderstood by others and harder to maintain, debug and extend, as shown in Figure 12.

[‡‡]`http://sourceforge.net/projects/ganttproject/files/OldFiles/ganttproject-1.10.2.zip`

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

```
 /** Paint all tasks  */
 public void paintTasks(Graphics g) {
     int sizex = getWidth();
     int sizey = getHeight();
     int headery = 45;
     float fgra = (float) sizex / (float) getGranit(true);

     g.setFont(myUIConfiguration.getChartMainFont());

     //Get all task
     //Probably optimised on next release
     listOfParam.clear();

     int y = 0;
     for (Iterator tasks = listOfTask.iterator(); tasks.hasNext();) {
         DefaultMutableTreeNode treeNode = (DefaultMutableTreeNode) tasks.next();
         GanttTask task = (GanttTask) treeNode.getUserObject();

//Is the task is visible, the task could be draw
         if (isVisible(task)) {
             int x1 = -10, x2 = sizex + 10;
             int e1; //ecart entre la date de debut de la tache et la date du debut du calendrier
             int fois;
             int type = 2;
             y++;

//difference between the start date of the task and the end
             e1 = date.diff(task.getStart());

//Calcul start and end pixel of each task
             float fx1, fx2;
             if (task.isMilestone()) {
                 fx1 = (float) e1 * fgra *
                         ((date.compareTo(task.getStart()) == 1) ? -1 : 1);
                 x1 = (int) fx1;
             } else {
                 fx1 = (float) e1 * fgra *
                         ((date.compareTo(task.getStart()) == 1) ? -1 : 1);
                 fx2 = fx1 + (float) task.getLength() * fgra;
                 x1 = (int) fx1;
                 x2 = (int) fx2;
             }

             int percent = 0;

//Meeting task
             if (task.isMilestone()) {
                 paintATaskBilan(g, x1, y, task);
                 x2 = x1 + (int) fgra;
                 type = 0;
             }
//A mother task
             else if (tree.getAllChildTask(treeNode).size() != 0) {

                 //Compute percent-complete
                 tree.computePercentComplete(treeNode);

                 paintATaskFather(g, x1, x2, y, task);
                 if (drawPercent) {
                     percent = paintAdvancement(g, x1, x2, y, task.getCompletionPercentage(), task.getShape(), task.getColor(), true);
                 }
                 type = 1;
             }
//A normal task
             else {
                 paintATaskChild(g, x1, x2, y, task);
                 if (drawPercent) {

                     percent = paintAdvancement(g, x1, x2, y, task.getCompletionPercentage(), task.getShape(), task.getColor(), false);
                 }
                 type = 2;
             }
//Add parameters on the array
             listOfParam.add(new GanttPaintParam(task.getName(), task.getTaskID(),
                     x1, x2, percent, y, type));
         }
     }
 }
```

Figure 12. An example of MORE's refactoring recommendation for GanttProject: the strategy pattern refactoring to be applied to the method `paintTasks()` in the class `GanttGraphicArea`.

As recommended by MORE and validated by the participants, the strategy refactoring applied to `paintTasks()` is a suitable design solution that deals well with this situation. This refactoring may lighten the original method by moving the conditional calculation logic to a small collection of independent calculation objects (strategies), each of which can handle one of the various ways of doing the calculation, making the design easier to understand and maintain.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

```
Node initFunction(FunctionNode fnNode, int functionIndex, Node statements, int functionType)
    {
        fnNode.itsFunctionType = functionType;
        fnNode.addChildToBack(statements);
        int functionCount = fnNode.getFunctionCount();
        if (functionCount != 0) {
            // Functions containing other functions require activation objects
            fnNode.itsNeedsActivation = true;
        }

        if (functionType == FunctionNode.FUNCTION_EXPRESSION) {
            String name = fnNode.getFunctionName();
            if (name != null && name.length() != 0) {
                // A function expression needs to have its name as a
                // variable (if it isn't already allocated as a variable).
                // See ECMA Ch. 13.  We add code to the beginning of the
                // function to initialize a local variable of the
                // function's name to the function value.
                Node setFn = new Node(Token.EXPR_VOID,
                                      new Node(Token.SETNAME,
                                          Node.newString(Token.BINDNAME, name),
                                          new Node(Token.THISFN)));
                statements.addChildrenToFront(setFn);
            }
        }

        // Add return to end if needed.
        Node lastStmt = statements.getLastChild();
        if (lastStmt == null || lastStmt.getType() != Token.RETURN) {
            statements.addChildToBack(new Node(Token.RETURN));
        }

        Node result = Node.newString(Token.FUNCTION, fnNode.getFunctionName());
        result.putIntProp(Node.FUNCTION_PROP, functionIndex);
        return result;
    }
```

Figure 13. An example of MORE's refactoring suggestion for Rhino: move the method `initFunction()` from the class `IRFactory` to the class `FunctionNode`.

Another example of MORE's suggestions for the Rhino project is depicted in Figure 13. This refactoring involves moving the method `initFunction()` from the class `IRFactory` to the class `FunctionNode`. Another participant commented on this refactoring: " *Looking at the `initFunction()` method which is implemented in `IRFactory` but it does not access any service in its original class; instead it uses services from the class `FunctionNode`. This method is calling the following methods from the class `FunctionNode`: `addChildToBack()`, `getFunctionCount()`, and `getFunctionName()`, and 2) access/modify two attributes `itsFunctionType` and `itsNeedsActivation`*". This might in turn cause a high coupling and there is no doubt that this method suffers from the feature envy code smell. For these reasons, the participant accepted the application of this refactoring.

To conclude, it was clear to our participants that MORE can provide useful refactoring recommendations that improve the design of the program under study. The refactoring of large systems can be time consuming and involves the improvement of several quality issues. We asked the participants to provide additional feedback. Firstly, MORE does not provide any ranking to prioritize the suggested refactorings. In fact, in practice, developers are unlikely to have enough time to understand, evaluate and apply all the suggested refactorings; rather they prefer to focus only on the most severe quality issues. Secondly, MORE does not provide support to fix or replace refactoring solutions that are not approved by the developer. Finally, most of participants mention that they prefer to include a procedure to automatically applies regression testing and generates test cases for the modified code fragments after refactoring. This is an interesting future research direction to explore in extending MORE.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

7. THREATS TO VALIDITY

Several factors can bias the validity of empirical studies. In this section, we discuss the different threats that can limit the validity of our study based on four types of threats, namely construct, conclusion, internal, and external validity.

Construct validity: It concerns the methodology employed to construct the experiment. The experiment conducted with the participants to evaluate the suggested refactorings represents a construct threat. As previously explained, we selected seven experienced participants in our study based on their programming experience with refactoring, code smells, design patterns, and software quality metrics. Furthermore, we diversified our participants from PhD students in software engineering to senior developers. To mitigate the diffusion threat, the participants were instructed not to share information about the experience prior to the completion of the study. We also randomized the ordering in which the refactorings were shown to the participants, to mitigate any sort of learning or fatigue effect. Another threat can be related to the number of evaluated refactoring operations. Applying a long sequence of refactoring operations can be a time-consuming task for participants who should understand the entire design of the systems being refactored, which range in size from 245 classes to 1,191 classes as reported in Table IV. In addition, the participants are asked to evaluate four solutions for each system; one solution per approach. Consequently, the task of evaluating the whole sequences of refactorings may potentially bias our experiments as fatigue threats to validity may arise, and can negatively affect their human judgement. To deal with this situation, we decided to focus on evaluating a random sample of the suggested solutions for each approach, which we believe a feasible and realistic scenario for the seven studied systems. Additionally, our results were compared to three other state-of-the-art approaches applied to the same subject systems and participants with the same settings.

Conclusion validity: Due to the stochastic nature of the implemented algorithms, we used the Wilcoxon rank sum test and effect size measures over 31 repeated runs of the algorithms with a 95% confidence level. The aim is to test whether significant differences exist between the measurements for different treatments. This test makes no assumption on the data distribution and is suitable for ordinal data. We are thus confident that the observed statistical relationships are significant.

Internal validity: It concerns the possible biases in the way in which the results were obtained. A possible threat to the internal validity concerns the technique used for detecting code smells, which may lead to a small number of false positives. While the employed detection rules are able to detect code smells with more than 90% of precision and recall scores as shown in [36], false positives/negatives may have an impact on the results of our experiments. To mitigate this threat, we manually inspect and validate each detected code smells. Moreover, our refactoring tool configuration is flexible and can support other state-of-the-art detection rules.

External validity: It concerns the possible biases related to the choice of experimental subjects/objects. Although we were able to select a set of subject systems that have a good degree of diversity in size, application domains and project teams, we cannot claim that our results can be generalized beyond these subject systems to other industrial contexts, other programming languages, and to other practitioners. Moreover, even if our approach succeeded in introducing four different design patterns (*factory method*, *visitor*, *singleton*, and *strategy*) and seven code smell types (*god class*, *feature envy*, *data class*, *spaghetti code*, *shotgun surgery*, *lazy class*, and *long parameter list*), we cannot generalize the results for other design pattern and code smell types.

In addition to these threats, and despite our encouraging results, our approach presents some limitations that should be addressed. First, some design pattern instances are difficult to implement and require an extra manual effort from the developer to tailor the implementation to fit the context. Second, some refactoring solutions require a significant number of code changes, which may take the code away from its initial design. To address this issue, we plan to consider new criteria to reduce the amount of code changes when recommending refactoring. Furthermore, in large-scale systems, the number of code smells to fix can be very large and not all of them can be fixed automatically. Thus, the prioritization of the list of detected code smells is required based on different criteria such as the severity and the risk.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

8. RELATED WORK

Search-based Software Engineering (SBSE) has been successfully employed to automate many software engineering tasks [43], including the problem of automated refactoring to improve various aspects of the software [6, 7, 11, 18, 36, 72]. These approaches cast refactoring as an optimization problem using a variety of optimisation techniques such as hill climbing, genetic algorithms, simulated annealing, etc. Most existing automated refactoring approaches can be classified into three main categories depending on the goal: (1) to improve quality factors; (2) to fix code smells; and (3) to introduce design patterns [14]. We explore each of these areas in the subsections below.

8.1. Automated Improvement of Design Quality

The majority of existing search-based refactoring approaches use software metrics as objective function(s) to find a good sequence of refactorings. This problem was initially studied by O'Keeffe and Ó Cinnéide [7, 73], who proposed an automated search-based refactoring approach for improving the quality of object-oriented programs based on standard software metrics from the QMOOD design quality model [31]. They conducted experiments on several medium-sized Java applications, where they found a significant and minimal improvements in *Understandability* and *Flexibility* respectively [7], but discovered that the *Reusability* function, as defined in the QMOOD suite, is unsuitable for a search-based refactoring process as it resulted in the addition of a large number of featureless classes.

Seng et al. [11] propose a single-objective optimization approach that uses a genetic algorithm to suggest a list of refactorings to improve software quality. The search process employs a single fitness function to maximize a weighted sum of several quality metrics (coupling, cohesion, complexity and stability). In contrast to fully-automated approaches, it is the designer's responsibility to take the decision that a suggested refactoring should be applied to the system or not.

Contrary to the aforementioned approaches that use a weighted-sum approach to combine metrics into a single objective fitness function, Harman and Tratt [6] use Pareto optimality to combine two metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), into a single fitness function. They demonstrated that this approach has several advantages over the weighted-sum approach in detecting opportunities to apply the Move Method refactoring.

Fatiregun et al. [74] showed how search-based transformations could be used to reduce code size and construct amorphous program slices. However, they use small atomic level transformations, rather than refactorings. In addition, their aim was to reduce program size rather than to improve its structure/quality.

Recently Simons et al. [75] conducted a survey with professionals to investigate the relationship between popular SBSE refactoring metrics and the subjective opinions of software engineers. The empirical study results suggest that (i) there is a little or no correlation between the two, and (ii) a simple static view of software is insufficient to assess software quality, and that software quality is dependent on factors that are not amenable to measurement via metrics. To address these issues, we introduced a set of semantic constraints to better drive the search process towards the optimal refactoring solutions. In addition, we evaluated our results from developers point of view as only metrics improvement would not be enough to guarantee that a suggested refactoring is good.

Although the approaches cited above are indeed powerful enough to improve quality as expressed by software quality metrics, this improvement does not mean that they are successful in removing actual instances of code smells.

8.2. Automated Correction of Code Smells

Search-based refactoring has been used to correct code smells by several authors. Kessentini et al. [60] proposed an approach using a mono-objective genetic algorithm to find a sequence of refactorings that attempts to minimize the number of code smells detected in the source code. Ouni et al. [49] proposed a multi-objective formulation of refactoring to find the best compromise between fixing code smells, and semantic coherence using two heuristics related to vocabulary similarity and structural coupling. The idea behind these two heuristics is to avoid violations of semantic

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

coherence when moving methods/fields between classes. Jdeodorant [37, 76, 77] is a well-known tool for detecting and repairing code smells, and currently supports four types of code smell. In a recent study [77] they propose a technique for detecting opportunities to apply the Extract Method refactoring and find in their study that in 42% of cases the applied refactoring resolved (or helped resolve) a code smell, usually Long Method or Duplicated Code.

Batova et al. [51] propose a technique based on relational topic models to identify Move Method refactoring opportunities and remove the Feature Envy smell from source code. A study with industrial developers indicated that their approach could provide meaningful recommendations for Move Method refactoring operations. In related work, these authors [78] also propose a method for automating the Extract Class refactoring by analyzing both structural and semantic relationships between the methods in a class to identify sets of strongly related methods, which are then used to define new classes with higher cohesion than the original class. They empirically evaluated this approach on Blobs in open source systems, and found that it outperformed the previous state-of-the-art and was found to be useful by industrial developers.

Furthermore, there are different research works related to our vocabulary-based similarity measure that have studied the semantic relatedness of source code elements, also called 'conceptual coupling'. Poshyvanyk et al. [79] first introduced the conceptual coupling measure, to capture new dimensions of coupling based on semantic information encoded in source code identifiers and comments. Later, Marcus et al. [80] introduced a new measure for the cohesion of classes in object-oriented software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers in order to construct models for predicting software faults. Bavota et al. [81] investigated several coupling measures including structural, dynamic, semantic, and logical coupling measures with respect to developer perception of coupling. An interesting result of this study indicates that the peculiarity of the semantic coupling measure allows it to better estimate the mental model of developers than other coupling measures, because interactions between classes are mostly encapsulated in the source code vocabulary. In common with [78], we show in MORE that the vocabulary in source code provides valuable information to guide the refactoring recommendation task.

8.3. Automated Introduction of Design Patterns

In one of the earliest works in this area, Eden et al. [82] noted that many design patterns could be described in terms of a common set of micro-patterns. They implemented transformations for these micro-patterns, thus creating a prototype 'patterns wizard' that could apply a number of design patterns to an Eiffel program. Another early work in the automated introduction of design patterns was that of Ó Cinnéide and Nixon [15, 17] who presented a methodology for the development of design pattern transformations in a behavior preserving fashion. They identified a number of 'pattern aware' composite refactorings called mini-transformations that, when composed, can create instances of design patterns. They defined a starting point for each pattern transformation, termed a precursor, which is where the basic intent of the pattern is present in the code, but not in its most flexible pattern form.

In more recent work, Jensen and Cheng [18] developed the first search-based refactoring approach that makes the introduction of design patterns a primary goal of the refactoring process. They used genetic programming, software metrics, and the set of mini-transformations identified by Ó Cinnéide and Nixon [17] to identify a sequence of mini-transformations that introduces the maximum number of design patterns to a software design.

Recently, Ajouli et al. [47] have described how to use refactoring tools (IntelliJ and Eclipse) to transform a Java program conforming to the Composite design pattern into a program conforming to the Visitor design pattern with the same external behavior, and vice versa. They consider several variations on each of the patterns, and validate their approach with a study of the JHotDraw system.

El Boussaidi and Mili [83] define a formal representation for the problem solved by a design pattern, the design pattern solution itself, and a transformation that converts an instance of the problem into an instance of the solution. They found that only 13 of the 23 Gamma et al. design patterns could be represented using this approach, and even where a pattern could be represented,

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

the codification of the solution was found to be too simplistic. In their conclusions they suggest that this could be due to textbook pattern examples being 'too perfect' or patterns being more widely applicable than their authors envisaged.

8.4. Related Refactoring Research

Researchers have examined various ways to improve automated refactoring. For instance, Murphy-Hill et al. [5, 84–86] proposed several techniques and empirical studies to support refactoring activities. In [85, 86], the authors proposed new tools to assist software engineers in applying refactoring manually such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, Ge and Murphy-Hill [87] proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, but check the correctness of their transformations automatically. However, the correction is based mainly on the structure of the code and does not consider its semantics. Mens et al. formalize refactoring by using graph transformations [88]. Bavota et al. [89] automatically identify method chains and refactor them to cohesive classes using the Extract Class refactoring. The aim of these approaches is to provide specific refactoring strategies; the aim of our approach is to provide a generic and automated refactoring framework to help developers to refactor their code.

Several works have taken a scripting approach to applying refactorings [90, 91], the most recent of these being the work of Kim et al. [16], which is relevant to our research as it also focuses on design patterns. They created a Java package to automate the creation of classical design patterns and encoded 18 out of the 23 Gang-of-Four design patterns [14] in their Java-based scripting language. In their experiments with a six real-world, non-trivial Java applications, they found that their approach could apply a complex pattern in a fraction of the time it would take a developer to perform the same process. This work does not attempt to find where a pattern should be applied however. By contrast, MORE applies a design pattern only when it is beneficial to do so, taking into account overall software quality, existing code smells in the code, and overall design coherence of the code base.

In summary, the area of identification of refactoring opportunities is a lively one [92] and significant contributions have been made in each of the areas on which we focus: design quality improvement, code smell removal, and introduction of design patterns. Our work is the first to our knowledge that combines all these research strands into a single coherent refactoring recommendation system.

9. CONCLUSION AND FUTURE WORK

This paper presented a search-based refactoring approach called MORE, that takes into consideration multiple perspectives to recommend refactoring solutions to: 1) improve software quality, 2) fix code smells, and 3) introduce design patterns. The selection of refactorings to propose is also guided by a set of semantic constraints to preserve the semantic coherence of the original program. MORE succeeded in finding near-optimal trade-offs between these multiple perspectives while providing more semantic and meaningful refactorings. To evaluate our approach, we conducted an empirical evaluation on seven medium and large-size open-source systems, and compared our results to three state-of-the-art approaches and two popular multi-objective algorithms as well as random search. Our empirical study shows the efficacy of our approach in improving the quality of the studied systems while successfully fixing 84% of code smells and introducing an average of six design patterns. In addition, the qualitative evaluation shows that most of the suggested refactorings (an average of 69%) are considered as relevant and meaningful by software developers.

As future work, we are planning to conduct an empirical study to investigate the correlation between fixing code smells, introducing design patterns and improving quality metrics to better understand the nature of relationship between them. We also plan to extend MORE to include other code smells and design pattern types and evaluate our approach in an industrial context. In addition,

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

we are planning to include a procedure to automatically apply regression testing techniques and generate test cases for the modified code fragments after refactoring.

REFERENCES

1. M. M. Lehman, L. A. Belady, Program evolution: processes of software change, Academic Press Professional, Inc., 1985.
2. M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
3. J. Kerievsky, Refactoring to patterns, Pearson Deutschland GmbH, 2005.
4. E. Murphy-Hill, C. Parnin, A. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18.
5. X. Ge, E. Murphy-Hill, Benefactor: a flexible refactoring tool for eclipse, in: ACM international conference companion on Object oriented programming systems languages and applications companion, 2011, pp. 19–20.
6. M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1106–1113.
7. M. OKeeffe, M. O. Cinnéide, Search-based refactoring for software maintenance, Journal of Systems and Software 81 (4) (2008) 502–516.
8. A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, Journal of Systems and Software 105 (0) (2015) 18 – 39.
9. S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, A comparative study of manual and automated refactorings, in: Object-Oriented Programming (ECOOP), Springer, 2013, pp. 552–576.
10. M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, p. 50.
11. O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in: 8th annual conference on Genetic and evolutionary computation, 2006, pp. 1909–1916.
12. G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, F. Palomba, Supporting extract class refactoring in eclipse: the aries project, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 1419–1422.
13. W. J. Brown, H. W. McCormick, T. J. Mowbray, R. C. Malveau, AntiPatterns: refactoring software, architectures, and projects in crisis, Wiley New York, 1998.
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.
15. M. O. Cinnéide, Automated application of design patterns: a refactoring approach, Ph.D. thesis, Trinity College Dublin. (2001).
16. J. Kim, D. Batory, D. Dig, Scripting parametric refactorings in java to retrofit design patterns, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 211–220.
17. M. Ó. Cinnéide, P. Nixon, A methodology for the automated introduction of design patterns, in: IEEE International Conference on Software Maintenance (ICSM), 1999, pp. 463–472.
18. A. C. Jensen, B. H. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, in: 12th annual conference on Genetic and evolutionary computation, 2010, pp. 1341–1348.
19. F. Khomh, Y. G. Gueheneuc, Do design patterns impact software quality positively?, in: 12th European Conference on Software Maintenance and Reengineering (CSMR), 2008, pp. 274–278.
20. F. Ververs, C. Pronk, On the interaction between metrics and patterns, in: Proceedings of the International Conference on Object Oriented Information Systems, IEEE Computer Society, Dublin, 1995.
21. Q. Soetens, S. Demeyer, Studying the effect of refactorings: A complexity metrics perspective, in: Seventh International Conference on the Quality of Information and Communications Technology (QUATIC'2010), 2010, pp. 313–318.
22. K. Stroggylos, D. Spinellis, Refactoring–does it improve software quality?, in: 5th International Workshop on Software Quality, WoSQ '07, 2007.
23. S. H. Kannangara, W. Wijayanayake, An Empirical Evaluation of Impact of Refactoring on Internal and External Measures of Code Quality, International Journal of Software Engineering & Applications 6 (1).
24. D. Wilking, U. Khan, S. Kowalewski, An Empirical Evaluation of Refactoring, e-Informatica Software Engineering Journal 1 (1).
25. S. Counsell, R. M. Hierons, H. Hamza, S. Black, M. Durrand, Exploring the Eradication of Code Smells: An Empirical and Theoretical Perspective, Advances in Software Engineering 2010 (2010) 1–12.
26. A. Chatzigeorgiou, A. Manakos, Investigating the Evolution of Bad Smells in Object-Oriented Code, in: International Conference on the Quality of Information and Communications Technology (QUATIC), 2010, pp. 106–115.
27. M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Hemati Moghadam, Experimental assessment of software metrics using automated refactoring, in: International Symposium on Empirical Software Engineering and Measurement, ESEM '12, 2012, pp. 49–58.
28. A. Ouni, M. Kessentini, H. Sahraoui, M. Ó Cinnéide, K. Deb, K. Inoue, A multi-objective refactoring approach to introduce design patterns and fix anti-patterns, in: 2015 North American Search Based Software Engineering Symposium (NasBASE), 2015.
29. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.
30. K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints, IEEE Transactions on Evolutionary Computation

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

18 (4) (2014) 577–601.
31. J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on Software Engineering 28 (1) (2002) 4–17.
32. W. F. Opdyke, Refactoring: An aid in designing application frameworks and evolving object-oriented systems, in: Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), 1990.
33. W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).
34. T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139.
35. M. Mantyla, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, in: IEEE International Conference on Software Maintenance (ICSM), 2003.
36. A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79.
37. M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings, in: 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1037–1039.
38. W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, Journal of systems and software 80 (7) (2007) 1120–1128.
39. R. Wirfs-Brock, A. McKean, Object design: roles, responsibilities, and collaborations, Addison-Wesley Professional, 2003.
40. A. J. Riel, Object-oriented design heuristics, Vol. 338, Addison-Wesley Reading, 1996.
41. M. Harman, B. F. Jones, Search-based software engineering, Information and Software Technology 43 (14) (2001) 833–839.
42. M. Harman, The current state and future of search based software engineering, in: Future of Software Engineering (FOSE), IEEE, 2007, pp. 342–357.
43. M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys 45 (1) (2012) 1–61.
44. M. Harman, S. Mansouri, Y. Zhang, Search based software engineering: A comprehensive analysis and review of trends techniques and applications, Kings College, London, UK, Tech. Rep. TR-09-03.
45. P. Lam, E. Bodden, O. Lhoták, L. Hendren, The soot framework for java program analysis: a retrospective, in: Cetus Users and Compiler Infastructure Workshop (CETUS), 2011.
46. D. Heuzeroth, T. Holl, G. Hogstrom, W. Lowe, Automatic design pattern detection, in: Program Comprehension, 2003. 11th IEEE International Workshop on, 2003, pp. 94–103.
47. A. Ajouli, J. Cohen, J.-C. Royer, Transformations between composite and visitor implementations in java, in: 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2013, pp. 25–32.
48. A. Ajouli, Vues et transformations de programmes pour la modularité des évolutions, Ph.D. thesis, Ecole des Mines de Nantes (2013).
49. A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 347–356.
50. M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys (CSUR) 45 (1) (2012) 11.
51. G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: Recommending move method refactorings via relational topic models, IEEE Transactions on Software Engineering 40 (7) (2014) 671–694.
52. J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: 9th European Conference on Object-Oriented Programming (ECOOP), 1995, pp. 77–101.
53. D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (7) (2013) 1165 – 1199.
54. D. F. Bacon, P. F. Sweeney, Fast static analysis of c++ virtual function calls, in: 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 1996, pp. 324–341.
55. I. Das, J. E. Dennis, Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems, SIAM Journal on Optimization 8 (3) (1998) 631–657.
56. D. C. Karnopp, Random search techniques for optimization problems, Automatica 1 (2) (1963) 111–121.
57. H. Li, Q. Zhang, Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii, IEEE Transactions on Evolutionary Computation 13 (2) (2009) 284–302.
58. D. A. Van Veldhuizen, G. B. Lamont, Multiobjective evolutionary algorithm research: A history and analysis (1998).
59. C. A. C. Coello, N. C. Cortés, Solving multiobjective optimization problems using an artificial immune system, Genetic Programming and Evolvable Machines 6 (2) (2005) 163–190.
60. M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design defects detection and correction by example, in: 19th International Conference on Program Comprehension (ICPC), 2011, pp. 81–90.
61. P. M. Chisnall, Questionnaire design, interviewing and attitude measurement, Journal of the Market Research Society 35 (4) (1993) 392–393.
62. K. Deb, T. Goel, Controlled elitist non-dominated sorting genetic algorithms for better convergence, in: Evolutionary Multi-Criterion Optimization, Springer, 2001, pp. 67–81.
63. A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 1–10.
64. A. E. Eiben, S. K. Smit, Parameter tuning for configuring and analyzing evolutionary algorithms, Swarm and Evolutionary Computation 1 (1) (2011) 19–31.
65. J. Demšar, Statistical comparisons of classifiers over multiple data sets, The Journal of Machine Learning Research 7 (2006) 1–30.
66. J. Cohen, Statistical power analysis for the behavioral sciences, Academic press, 1988.
67. S. Nakagawa, A farewell to bonferroni: the problems of low statistical power and publication bias, Behavioral Ecology 15 (6) (2004) 1044–1045.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr

68. N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychological Bulletin 114 (3) (1993) 494.
69. P. Wendorff, Assessment of design patterns during software reengineering: lessons learned from a large commercial project, in: 5th European Conference on Software Maintenance and Reengineering (CSMR), 2001, pp. 77–84.
70. W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, IEEE Transactions on Software Engineering 40 (9) (2014) 841–861.
71. S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: 3rd international symposium on empirical software engineering and measurement, 2009, pp. 390–400.
72. A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, Journal of Systems and Software.
73. M. O'Keeffe, M. Ó Cinnéide, A stochastic approach to automated design improvement, in: 2nd international conference on Principles and practice of programming in Java, Computer Science Press, Inc., 2003, pp. 59–62.
74. D. Fatiregun, M. Harman, R. Hierons, Evolving transformation sequences using genetic algorithms, in: 4th International Workshop on Source Code Analysis and Manipulation, SCAM '04, Los Alamitos, California, USA, 2004, pp. 65–74.
75. C. Simons, J. Singer, D. R. White, Search-based refactoring: Metrics are not enough, in: M. Barros, Y. Labiche (Eds.), Search-Based Software Engineering, Vol. 9275 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 47–61.
76. M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells, in: IEEE International Conference on Software Maintenance (ICSM), 2007, pp. 519–520.
77. N. Tsantalis, A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, Journal of Systems and Software 84 (10) (2011) 1757 – 1782.
78. G. Bavota, A. Lucia, A. Marcus, R. Oliveto, Automating extract class refactoring: an improved method and its evaluation, Empirical Software Engineering 19 (6) (2014) 1617–1664.
79. D. Poshyvanyk, A. Marcus, The conceptual coupling metrics for object-oriented systems, in: 22nd IEEE International Conference on Software Maintenance (ICSM), Washington, DC, USA, 2006, pp. 469–478.
80. A. Marcus, D. Poshyvanyk, R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE Transactions on Software Engineering 34 (2) (2008) 287–300.
81. G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, An empirical study on the developers' perception of software coupling, in: International Conference on Software Engineering (ICSE), 2013, pp. 692–701.
82. A. H. Eden, A. Yehudai, J. Gil, Precise specification and automatic application of design patterns, in: 12th International Conference on Automated Software Engineering (Formerly: KBSE), ASE '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 143–.
83. G. El Boussaidi, H. Mili, Understanding design patterns what is the problem?, Software: Practice and Experience 42 (12) (2012) 1495–1529.
84. E. Murphy-Hill, A. P. Black, Refactoring tools: Fitness for purpose, IEEE Software 25 (5) (2008) 38–44.
85. E. Murphy-Hill, A. P. Black, Breaking the barriers to successful refactoring: Observations and tools for extract method, in: 30th International Conference on Software Engineering (ICSE), ICSE '08, ACM, New York, NY, USA, 2008, pp. 421–430.
86. E. Murphy-Hill, A. Black, Programmer-friendly refactoring errors, IEEE Transactions on Software Engineering 38 (6) (2012) 1417–1431.
87. X. Ge, E. Murphy-Hill, Manual refactoring changes with automated refactoring validation, in: 36th International Conference on Software Engineering (ICSE), New York, NY, USA, 2014, pp. 1095–1105.
88. T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens, Formalizing refactorings with graph transformations: Research articles, Journal of Software Maintenance and Evolution 17 (4) (2005) 247–276.
89. G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, A two-step technique for extract class refactoring, in: IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, 2010, pp. 151–154.
90. M. Verbaere, R. Ettinger, O. de Moor, Jungl: a scripting language for refactoring, in: 28th International Conference on Software Engineering (ICSE), ACM Press, New York, NY, USA, 2006, p. 172181.
91. F. Steimann, C. Kollee, J. von Pilgrim, A refactoring constraint language and its application to eiffel, in: 25th European Conference on Object-oriented Programming (ECOOP), Springer-Verlag, Berlin, Heidelberg, 2011, pp. 255–280.
92. J. A. Dallal, Identifying refactoring opportunities in object-oriented code: A systematic literature review, Information and Software Technology 58 (2015) 231 – 249.

Copyright © 2017 John Wiley & Sons, Ltd.
Prepared using smrauth.cls

DOI: 10.1002/smr