Model refactoring by Example: A Multi-Objective Search Based Software Engineering Approach

Adnane Ghannem1 *, Marouane Kessentini2, Mohammad Salah Hamdi1 and Ghizlane El Boussaidi3

1 Information Systems Department, Ahmed Bin Mohammed Military College., Qatar
2 Computer and Information Science Department, University of Michigan, USA
3 Software Engineering and IT Department, École de Technologie Supérieure., Canada

ABSTRACT

Declarative rules are frequently used in model refactoring in order to detect opportunities in the refactoring and to apply the appropriate ones. However, a high number of rules is required to obtain a complete specification of refactoring opportunities. In some situations, companies have accumulated examples of refactorings from past maintenance experiences. Based on these observations, we consider the model refactoring problem as multi objective problem by suggesting refactorings sequences that should maximize two objectives. The first objective is to maximize both structural and textual similarity between a given model (the model to be refactored) and a set of bad designed models in the base of examples (models that have undergone some refactorings). The second objective is to minimize the structural similarity between a given model and a set of well designed models in the base of examples (models that do not need any refactoring). To this end, we use the Non-dominated Sorting Genetic Algorithm (NSGA-II) to find a set of representative Pareto optimal solutions that present the best trade-off between structural and textual similarities of models. The validation results on eight real world models taken from open-source projects confirm the effectiveness of our approach with more than 80% of average correctness of our refactoring recommendations. In addition, our approach outperforms five other refactoring approaches.

Keywords: software maintenance, model evolution, refactoring by example, NSGA-II, Pareto front;

1. INTRODUCTION

Maintenance process includes the essential tasks that aim to existing software while preserving its integrity [1]. These modifications are incremental and aim to either update some functionality or correct some design flaws and fix some bugs. These software maintenance activities become more complex when the size of the system and the number of requirements increase during the time [2]. Therefore, it is important to provide automated and semi-automated software maintenance tools to improve the quality of software.

Software developers are invited to constantly restructure the software system in order to improve its structure and design. The restructuring process is called refactoring [3]. Fowler [2], defined the refactoring as a cleaning up code process that improves the software structure while preserving its external behavior. The process of refactoring involves several activities [3] including the activities of identifying refactoring opportunities in a given software and determining which refactorings to apply. Many researchers have been working on providing

* Corresponding author. Adnane Ghannem, ABMMC, Doha, Qatar.
E-mail address: adnane.ghannem@abmmc.edu.qa
support for refactoring (e.g., [4], [2], and [5]). Most of them have focused on refactoring at the source code level (e.g., code smells [6]) and few approaches attacked the refactoring at the model level (e.g., [7], [8] and [9]). However, model-driven engineering (MDE) approach, which is a promising approach to manage software systems’ complexity and specify domain concepts effectively [10] consider the models as primary artifacts in the software systems. MDE consists of refining and successively transforming abstract models into more concrete models including executable source code. In this context, refactoring is a specific type of model transformation that aims at improving the quality of a given model; for example improving the design of an existing design model by applying a design pattern which can be encoded as a model transformation [11].

The rise of MDE increased the interest and the needs for tools supporting refactoring at the model-level. However, many challenges need to be overcome when building such a tool. Some of these challenges were identified in [8] and they include issues related to assessing model quality, ensuring synchronization and coherence between models (including source code), preserving behavior, etc. Most of the existing refactoring tools offer a semi-automatic support [8] because part of the necessary knowledge for performing the refactoring remains implicit in designers’ heads. Undeniably, recognizing model refactoring opportunities remains a big challenge that is related to the model marking process within MDE context which requires design knowledge and expertise [11]. Furthermore, declarative rules represent a common technique in most of existing work on refactoring in order to detect and correct defects (i.e., refactoring opportunities) and defect types can be very important [12]. Another common issue to most of refactoring approaches is the problem of sequencing and composing refactoring rules that is, in general, related to the control of rules’ applications within a rule-based transformational approach.

To overcome some of these issues, many approaches to refactoring are using a search-based approach where the refactoring is considered as an optimization problem (e.g. [13] [14] [15] [16] [17] and [18]). Search-based refactoring approaches adapted some of the known heuristics methods such as Simulated annealing and Hill climbing as proposed in [13] and [14], and Genetic Algorithms as proposed in [15]. In previous work [16], we proposed a by example approach that recommends refactoring to correct models. The approach uses single-objective optimization to find the best refactoring sequences which maximize the structural similarity between the model under analysis and a set of model refactoring examples. The structural similarity is computed using a set of metrics. Other optimization goals were considered in search-based refactoring approaches (e.g., reducing the refactoring effort [17], improving the software structure [14]). Harman et al. [18] have proposed a multi-objective approach that uses two software metrics (CBO: coupling between objects, and SDMPC: standard deviation of methods per class) to define two optimization objectives. Most of these approaches relied on the structural information (i.e., a combination of software metrics) to formulate their fitness functions and do not consider the design consistency in the optimization process. However, to suggest meaningful refactorings and to reduce the number of possible refactorings, both quality and consistency of the model to be refactored should be considered.

This paper represents an extension of our previous work [19] having as goal to propose a new multi-objective optimization approach that aims at finding the best sequence of refactorings that optimize two objectives. The first objective is to maximize both the structural and the textual similarity between a given model (i.e., the model to be refactored) and a set of models in the base of examples (i.e., models that have undergone some refactorings). The second objective is to minimize the structural similarity between a given model (i.e., the model to be refactored) and a set of well-designed models in the base of examples (i.e., models that do not need any refactoring). We start in this work from the observation that required knowledge to propose suitable refactorings for a given new object-oriented model may be inferred from other similar existing models’ refactorings. We defined two levels of textual and structural similarities (measures) between these models and the given model. To combine these measures, we adapt the Non-dominated Sorting Genetic Algorithm (NSGA-II) [20] which aims at finding a set of representative Pareto optimal solutions in a single run. Our approach
takes as input an initial model to refactor, a base of examples of models and their subsequent refactoring operations, and a list of structural metrics and textual measures. It generates as output a solution as a sequence of refactorings that should be applied to the initial model. The process of generating this solution can be seen as the mechanism that finds a sequence of refactoring operations that represent the best trade-off between the two criteria of structural and textual similarities between the model to refactor and good/bad design examples.

The intuition behind this work is that we can use data collected from previous releases or projects within the same organization. The “good” and “bad” examples are defined based on the developers’ design and programming practices. In our experiments, we showed that open source systems could represent a very good starting point but to get optimal results it is always better that the developers/users provide a good and bad set of examples from their previous projects to get optimal results. These examples could be found, for example, in the review reports where some bad examples in terms of quality are described. Another option is to use a set of quality metrics to define what is considered as good or bad by the developers to extract a set of examples.

The approach could be considered as a data-mining technique since it is based on the exploration of design examples to identify relevant refactorings for a new context (project). We believe that one of the main key limits is related to the automated adaptation of refactorings applied in a similar context to a new one. In some situations, a user interaction is required and not all the refactorings that are recommended can be applied. However, the problem is a bit easier at the model level comparing to the code level. In our approach, we do not try to extract association rules like classical data mining techniques. However, the multi-objective approach try to adapt the refactoring solution as much as possible to an existing example while considering the applicability of the refactorings. Thus, we do not generate “templates” from the refactoring examples then apply them but our goal is to “optimize” as much as possible the similarity between the examples and the identified model fragment to be refactored.

The main contributions of the paper can be summarised as follows:

- We introduce a novel multi-objective refactoring approach based on the use of examples. We take into consideration the design consistency when comparing between the model to be refactored and existing model examples to suggest refactoring solutions.
- We report the result of an empirical study of NSGA-II technique applied to eight open source systems. We compare our approach to a mono objective approach (MONO), MOREX[16], random search, multi-objective particle swarm optimization (MOPSO)[19], a genetic programming approach GP[20], and our previous work that is not based on the use of good design examples [19]. The results provide evidence to support the claim that our proposal enables the generation of refactoring solutions with a high precision.

The goal of our experimentation is to find out whether our approach could propose meaningful sequences of refactorings to correct design defects within models (e.g., class diagrams). Indeed, our approach addresses two research questions:

- RQ1: To what extent can the proposed approach generate correct sequences of refactorings?
- RQ2: To what extent can the design consistency aspect improve the efficiency of our proposal to generate meaningful refactoring solutions?

To answer RQ1, we assessed the precision and recall of our approach that was applied on eight open source projects that we had several versions and refactorings they had undergone. To answer RQ2, we compared our results to those produced by our previous work called MOREX [16], MOREX [16], GP [20], MOPSO [19] and NSGA-II without the use of good design examples [19].

The rest of this paper is organized as follows. Section 2 presents the overall approach and the details of our adaptation of the multi-objective evolutionary algorithm NSGA-II to the model refactoring problem. Section 3
describes the supporting tools and experimental settings and presents results and discussion. Related works are discussed in section 4 and we conclude and outline some future directions to our work in section 5.

2. Model Refactoring using multi Objective optimization

2.1. Approach Overview

The proposed approach exploits examples of model refactorings and an evolutionary algorithm (NSGA-II [21]) to automatically suggest (to the user) sequences of refactorings that can be applied on a given model. The user is responsible for applying these refactorings. For example, for a rename refactoring, our approach does not decide for the new name but it suggests to apply the mentioned refactoring operation and we give hand to the user to choose the right name based on his experience. The general structure of our approach is introduced in Fig. 1. It takes as input a set of bad designed models (label A) (i.e., existing models and their related refactorings), a set of well designed models that do not need refactoring (label B), an initial model (label C) and takes as controlling parameters a set of software metrics (label D). The approach generates as output a sequence of refactorings that can be applied to the initial model. The goal of our approach is to recommend the refactorings that make the design similar to good ones in terms of quality thus we implicitly considered the fact that the refactorings have to improve the quality by meeting quality values similar the ones of the good design. The process of generating a sequence of refactorings (Fig. 1) can be seen as the mechanism that finds the best way to select and combine refactoring operations among the ones in the base of examples, in such a way to: 1) maximize the structural and the textual similarities between entities to be refactored in the initial model and entities of the models (bad designed models) and 2) minimize the structural similarity between entities to be refactored in the initial model and entities of the models (well designed systems). The structural similarity between two entities (e.g., classes) is computed using software metrics of these entities while their textual similarity is computed using textual measures based on WordNet [22].

![Fig. 1 Multi-objective model refactoring using examples](image)

In the proposed approach, a subset of the 72 refactorings defined in [2], that can be applied to UML class diagrams, was considered. In fact, some of the refactorings in [2] may be applied on design models (e.g. Rename Method, Move Method, Move Attribute, Extract Class etc.) while others cannot be (e.g. Inline Method, Extract Method, , Replace Temp With Query etc. ). Table 1 shows the list of twelve refactorings considered in in our approach chosen based on two factors: 1) they can be applied at the class
diagram level; and 2) they can be linked to a set of metrics (i.e., metrics which are impacted when applying these refactorings).

Table 1. Considered Refactorings in our approach

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract class</td>
<td>Consists of creating a new class and moving the relevant fields and methods from the old class into the new class.</td>
</tr>
<tr>
<td>Rename method</td>
<td>Consists of renaming method with a name that reveals its purpose in order to give more comprehensiveness to the model design.</td>
</tr>
<tr>
<td>Push down Method</td>
<td>Consists of moving method from a super-class to a specific subclass because it makes sense only there.</td>
</tr>
<tr>
<td>Push down Attribute</td>
<td>Consists of moving attribute from super class to a specific subclass because it makes sense only there.</td>
</tr>
<tr>
<td>Rename parameter</td>
<td>Consists of renaming parameter within the method parameter list.</td>
</tr>
<tr>
<td>Add parameter</td>
<td>Consists of adding a new parameter to the method parameter list.</td>
</tr>
<tr>
<td>Move Attribute</td>
<td>Consist of moving attribute from a source class to the class destination when it makes sense only there.</td>
</tr>
<tr>
<td>Move method</td>
<td>Consist of moving method from a class to another one when it is using or used more features of the destination class than the class on which it is defined.</td>
</tr>
<tr>
<td>Pull up method</td>
<td>Consists of moving method from some class(es) to the immediate super-class in order to eliminate duplicate methods among sibling classes, and hence reduce code duplication in general.</td>
</tr>
<tr>
<td>Pull up Attribute</td>
<td>Consists of moving attribute from some class(es) to the immediate super-class order to eliminate duplicate field declarations in sibling classes.</td>
</tr>
<tr>
<td>Extract interface</td>
<td>Consists of creating interface class when many classes use the same subset of class’s interface, or two classes have part of their interfaces in common.</td>
</tr>
<tr>
<td>Replace inheritance</td>
<td>Consists of changing the inheritance relation by a delegation when the subclasse uses only part of a super classes interface or does not want to inherit data.</td>
</tr>
</tbody>
</table>

Table 2 shows the list of sixteen metrics used in our approach and that could be applied to class diagrams (e.g., \( NA, NMeth, NDep \), etc.). These metrics include the eleven metrics defined in [23] to which we have added a set of simple metrics (e.g., \( NPvMeth, NPbMeth \)). All these metrics are related to the class entity which is the main entity in a class diagram. These metrics are used to compute the structural similarities between classes from the initial model and those in the base of examples. We believe that the textual similarity measure, described in the next section, consider the consistency between the elements.

Table 2. Considered metrics in our approach

<table>
<thead>
<tr>
<th>Ref</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>The number of attributes</td>
</tr>
<tr>
<td>NPvA</td>
<td>The number of private attributes</td>
</tr>
<tr>
<td>NPbA</td>
<td>The number of public attributes</td>
</tr>
<tr>
<td>NProtA</td>
<td>The number of protected attributes</td>
</tr>
<tr>
<td>NMeth</td>
<td>The number of methods</td>
</tr>
<tr>
<td>NPvMeth</td>
<td>The number of private methods</td>
</tr>
<tr>
<td>NPbMeth</td>
<td>The number of public methods</td>
</tr>
<tr>
<td>NProtMeth</td>
<td>The number of protected methods</td>
</tr>
<tr>
<td>NAss</td>
<td>The number of associations</td>
</tr>
</tbody>
</table>
To compute the textual similarity between two classes, we use the Rita toolkit [22]. To find the best compromise between the two objectives (structural and textual measures), we adapted the NSGA-II [21]. The next section describes the algorithm and its adaptation to the refactoring problem.

2.2. NSGA-II for model refactoring

NSGA overview

NSGA-II is an evolutionary algorithm based on the non-dominated sorting in order to solve multi-objective optimization problems [21]. It was designed to accept an exhaustive list of candidate solutions, which creates a large search space. The main idea of the NSGA-II consists of finding a representative set of Pareto optimal solutions, called non-dominated solutions. A solution called non-dominated when no other solution can improve some optimization objective without degrading another. Given a set of objectives $f_i$, $i \in 1, \ldots, n$, to maximize, a solution $x$ is said to Pareto dominate another solution $x'$ if and only if: $\forall i, f_i(x') \leq f_i(x)$ and $\exists j | f_j(x') < f_j(x)$

Three main steps characterize the NSGA-II algorithm:

- Create randomly the initial population $P_0$ of individuals encoded using a specific representation.
- Create a child population $C_0$ generated from the population of parents $P_0$ using genetic operators such as crossover and mutation.
- Merge both populations and select a subset of individuals, based on the dominance principle to create the next generation.

This process is repeated until reaching the last iteration according to stopping criteria.

NSGA-II adaptation

In this section, we describe how we adapted the NSGA-II to find the best trade-off between structural and textual similarity. As our aim is to maximize structural and textual similarity between a given model (the model to be refactored) and a set of bad designed models in the base of examples (models that have undergone some refactorings) and minimize the structural similarity between a given model and a set of well designed models in the base of examples (models that do not need any refactoring). We separate each criteria in an objective for NSGA-II. The algorithm takes as input a set of model refactorings’ examples (our base of examples), an initial model and set of metrics. We build an initial population as a set of individuals that stand for possible solutions representing sequences of refactorings that can be applied to the classes of the initial model (Lines 1-2). An individual is a set of blocks where each block contains a CIM (class chosen from the initial model), a CBE (class chosen the base of examples) that was matched to CIM, and a list of refactorings which is a subset of the refactorings that were applied to CBE (in its subsequent version) and that can be applied to CIM. The next
section explains and illustrates the Individuals’ representation.

Once, a population of refactoring solutions is generated, the main NSGA-II loop (Lines 4-21) consists of evolving a population of candidate solutions toward the best sequence of refactoring, i.e., an individual that maximises as much as possible both the textual and the structural similarities between the classes CIM and CBE that were matched within the individual’s blocks. In addition, CIM is compared to all the good examples to identify the closest class CGE using the structural similarity measure. An offspring population $C_t$ is generated from a parent population $P_t$ using genetic operators (selection, crossover and mutation) (Line 5) throughout each iteration $t$. After that, $C_t$ and $P_t$ will be merged together to create a global population $G_t$. Then, each individual $I$ in the population $G_t$ is evaluated using our two fitness functions.

After calculating these functions, a list of non-dominated fronts $F$ ($F_1$, $F_2$, ...) is returned by sorting all the solutions, where $F_1$ represents the set of non-dominated solutions, $F_2$ represents the set of solutions dominated only by solutions in $F_1$, etc (line 11). Then, we build the next population $P_{t+1}$ from the set of non-dominated fronts starting from front $F_i$ to $F_t$ (lines 14-17). In general, the number of solutions in all sets from front $F_i$ to $F_t$ is larger than the Max_size. To choose exactly Max_size solutions, we sort the solutions of the front $F_i$ using the crowded-comparison operator ($<_n$) defined in [21] (line 18). Then, we select the best solutions needed until we reach the Max-size (line 19). The crowded-comparison operator ($<_n$) is based on non-domination ranking and the crowding distance described in [21]. Finally, the loop terminates (line 21) announcing by that the achievement of termination criterion (i.e. maximum iteration number). The result of the algorithm is the set of best solutions, i.e., those in the Pareto front of the last iteration (line 22). In the next sub-sections, we give more details concerning the representation of solutions, genetic operators, and the fitness functions.

Algorithm 1. High level pseudo-code for NSGA-II adaptation to our problem

<table>
<thead>
<tr>
<th>Algorithm : NSGA-II Search-based Model refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td>Set of bad designed models with refactorings</td>
</tr>
<tr>
<td>Set of well designed models</td>
</tr>
<tr>
<td>Initial model</td>
</tr>
<tr>
<td>Set of software metrics</td>
</tr>
<tr>
<td><strong>Process:</strong></td>
</tr>
<tr>
<td>1. $I := \text{set (CIM, CBE, CPE, A set of applicable refactorings)}$</td>
</tr>
<tr>
<td>2. $P_0 := \text{set_of}(I)$</td>
</tr>
<tr>
<td>3. $t := 0$</td>
</tr>
<tr>
<td>4. Repeat</td>
</tr>
<tr>
<td>5. $C_t := \text{apply_Genetic_Operators}(P_t)$</td>
</tr>
<tr>
<td>6. $G_t := P_t \cup C_t$</td>
</tr>
<tr>
<td>7. For all $I \in G_t$</td>
</tr>
<tr>
<td>8. $\text{SimilarityBE}(I) := \text{calculate_Similarity}(\text{CIM, CBE})$</td>
</tr>
<tr>
<td>9. $\text{SimilarityGE}(I) := \text{calculate_Similarity}(\text{CIM, CGE})$</td>
</tr>
<tr>
<td>10. end For</td>
</tr>
<tr>
<td>11. $F := \text{fast_Non_Dominated_Sort}(G_t)$ // $F = (F_1, F_2, ...)$</td>
</tr>
<tr>
<td>12. $P_{t+1} := \emptyset$</td>
</tr>
<tr>
<td>13. $i := 1$</td>
</tr>
<tr>
<td>14. While $</td>
</tr>
<tr>
<td>15. $P_{t+1} := P_{t+1} \cup F_i$</td>
</tr>
<tr>
<td>16. $i := i+1$</td>
</tr>
</tbody>
</table>
Individual representation

To apply NSGA-II, we represent a candidate solution (i.e., an individual) as a set of blocks. A block is a triplet of (CIM, CBE, Applicable refactorings to CIM), where CIM is a class chosen from the initial model, CBE is a class chosen from the base of examples that was matched to CIM, and finally the list of refactorings which is a subset of the refactorings that were applied to CBE (in its subsequent versions and that can be applied to CIM) of where A is an artifact (software element) chosen from the software system (system being analyzed) and R is a requirement chosen from the set of requirements at hand. Fig. 2 shows an example of an individual (solution). The refactorings selection process adopted within a block considers some constraints in order to avoid conflicts and inconsistencies. For example, if the CIM class does not have any attribute or method and the CBE class requires a Move_attribute or Add_Parameter refactoring operation, then we discard these refactoring operations since we cannot apply them to the CIM class.

Fig. 2 Block representation

The bottom part of Fig. 3 shows an example of a candidate solution (i.e., an individual) composed of three blocks. Each block contains one refactoring operation. Therefore, the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply. The top part of Fig. 3 shows the fragments of an initial model before and after the sequence of refactoring proposed by the individual (at the bottom of the figure) were applied. Notice that the same refactoring operation could be included several times in the same individual.
The generation of the initial population require a maximum individual size as parameter. This parameter can be chosen randomly or specified the user. In our implementation, we adopted the first choice in order to obtain individuals with different sizes. After that, we assign randomly for each individual: (1) A set of CIM,(2) their matched CBE’s , and (3) the sub-set of refactorings that we can possibly apply on the CIM among the refactorings proposed from CBE.

**Selection and Genetic Operators**

**a. Selection**

There are many methods how to select the best individuals that will participate in the next generation (e.g., roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection, etc.). NSGA-II uses binary tournament selection [21] to derive a child population Qt (i.e., the set of individuals that will undergo the crossover and mutation operators) from a parent population Pt. The binary tournament selection involves running several "tournaments". Each tournament involves two randomly selected individuals from the population. This will give all individuals of the population the chance to be selected and preserves diversity.

**b. Crossover**

We use a simple, random, cut-point crossover. For each crossover, two individuals are selected by applying the tournament selection [21]. The crossover happens only with a certain probability. The crossover operator allows creating two offsprings P1’ and P2’ from the two selected parents P1 and P2. Fig. 4 illustrates a one-point crossover in which one point is selected from the parents P1 and P2. Everything before the cross point is swapped between the parents, producing two children P1’ and P2’.

---

**Fig. 3 Individual representation**

---
c. Mutation

The mutation operator consists of randomly changing one or more components of the list representing an individual (solution). Given a selected individual, the mutation operator first randomly selects one or more block of the sequence corresponding to the individual, then the CBE of each block will be replaced by another one chosen randomly from the base of examples. Fig. 5 illustrates the effect of a mutation operation. The CBE (Teacher and its refactorings list) of the second block is replaced by the CBE Student and its refactorings taking into consideration the constraints mentioned before.

Multi-criteria evaluation of individuals

In practice, we should formalized the evaluation of an individual as a mathematical function called “fitness function”. Two different fitness function were considered in this work. The first one calculate calculates the structural similarity between CIM and CDE. The second one calculate the textual similarity between CIM and CBRE. The idea behind is that a candidate solution that displays high structural and textual similarities between CIM and CBE should give the best sequence of refactorings.
a. Structural criterion

The structural criterion is evaluated using the fitness function denoted by Structural_Similarity by formula 1 and 2.

\[
\text{Structural Similarity (CMI, CBE)} = \frac{1}{m} \sum_{i=1}^{m} \text{Sim}(CMI_i, \text{CBE}_i)
\]  

(1)

\[
\text{Sim (CMI}_i, \text{CBE}_i) = \begin{cases} 
1 & \text{if CMI}_i = \text{CBE}_i \\
0 & \text{if (CMI}_i = 0 \text{ and CBE}_i \neq 0 \text{ or CMI}_i \neq 0 \text{ and CBE}_i = 0) \\
\frac{\text{CMI}_i}{\text{CBE}_i} & \text{if CMI}_i < \text{CBE}_i \\
\frac{\text{CBE}_i}{\text{CMI}_i} & \text{if CBE}_i < \text{CMI}_i
\end{cases}
\]  

(2)

Where \( m \) represents the number of metrics. \( \text{CMI}_i \), \( \text{CBE}_i \) represent the \( i \)th metric value of the CIM and \( i \)th metric value of the CBE respectively. Thus, we define the structural fitness function of a solution, normalized in the range \([0, 1]\), as:

\[
f_{\text{Structural}} = \frac{1}{n} \sum_{j=1}^{n} \text{Structural Similarity} (\text{CMI}_{Bj}, \text{CBE}_{Bj})
\]  

(3)

Where \( n \) represents the individual size (number of blocks), \( \text{CMI}_{Bj} \) and \( \text{CBE}_{Bj} \) represent the classes in the first two parts of the \( j \)th block of the individual. To illustrate how the structural fitness function is computed, we consider a system containing two classes as shown in Table 3 and a base of examples containing two classes shown in Table 4. In this example, we use six metrics and these metrics are given for each class in the model in Table 3 and each class of the base of examples in Table 4.

Here present two individuals \( I_1 \) and \( I_2 \) respectively composed by two blocks (Plane/Student and Product/Car) and one block (Product/Student). We consider the example of these two solutions (individuals). The fitness function calculated on these solutions has the value:

\[
f_{\text{Structural}I_1} = \frac{1}{2} \left[ \frac{1}{6} \left( \frac{2}{4} + 1 + 1 + 0 + \frac{1}{3} + 0 \right) + \left( \frac{2}{5} + \frac{1}{2} + \frac{1}{6} + 1 + 1 + 1 \right) \right] = 0.41
\]
In this case, the evaluation process will choose the individual having the maximum value of fitness function, then I₁ will be chosen as best individual.

b. Design consistency criterion

To formulate the textual similarity fitness function, we used Rita toolkit [22] which enables to compute the degree of likeness between two concepts based on their meaning. Specifically, we used this tool to calculate the textual similarity distance between two classes using their names. Thus, the textual similarity between two classes Class₁ and Class₂, denoted as Texture_similarity(Class₁, Class₂), corresponds to the textual similarity distance between Class₁’s name and Class₂’s name. The idea behind consist of breaking down the two class names, if the class name is a complex name, into different words. To this end, we used predefined functions in Rita toolkit that allow us the extraction of nouns, verbs, adverbs, etc. Then, we calculate the distance between different obtained combinations. Finally, the design consistency fitness function of a solution corresponds to the average of distances of all the blocks of the solution as shown by the formula 4.

\[
f_{\text{struc}} = \frac{1}{n} \sum_{j=1}^{n} \text{Texture_Similarity}(\text{CMI}_{Bj}, \text{CBE}_{Bj})
\]

Where \(n\) is the number of blocks in the solution and \(\text{CMI}_{Bj}\) and \(\text{CBE}_{Bj}\) are the classes composing the first two parts of the \(j\)th block of the solution. We considered the textual (semantic) similarity when the model is compared with the bad designed models since we adapted the refactorings applied to that bad designed model to our context. Thus, the semantic (textual) similarity is important along with the structural one. However, the disimilarity with the well-designed model is used as a helper objective just to identify refactoring opportunities but not to the correction of these identified defects.

The structural function is used by both fitness functions and the semantic function is just used by part the first fitness function that calculates the similarity between the model to refactor and bad examples (average of structural and textual similarities).

3. Experimentations with the approach

This section describes the evaluation steps of our approach. It starts by presenting our supporting tools. Then, we define our research questions. Finally, we describe our experimental settings and we present and discuss the results of the experimentations.

3.1. Supporting Tools

Preliminary steps are needed to validate our approach: (1) we implemented a parser to analyse Java source code and generate a predicate model as illustrated in Fig. 6; (2) we run the parser on 8 Java open source projects (Ant, JabRef, JGraphx, JHotDraw, GanttProject, J RDF, Xerces and Xom) in order to obtain their predicate models; (3) we completed the obtained models in the second step by manually entering the refactoring operations extracted with Ref-Finder [24], that these projects have undergone. Fig. 7 illustrates an example of a CBE. The Ref-Finder tool permits detection of 68 refactorings that include a set of atomic refactorings by using logic-based rules executed by a logic-programming engine. Ref-Finder helps finding refactorings that a system has undergone by comparing different versions of the system. Two reasons were behind the use of Ref-finder tool: (1) build the base of examples and (2) compute the precision and recall of our approach. We also used another
tool called Rita toolkit [22] to calculate the textual similarity. Behind

A plugin was developed to support our approach using Eclipse development environment. Fig. 8 shows a screenshot of the model refactoring plugin perspective. The plugin supports many heuristic-based algorithms for refactoring and hence enable to enter many controlling parameters depending on the chosen algorithm. For the NSGA-II refactoring algorithm, it takes as input a base of examples of models and their related refactorings, an initial model to refactor, and a set of metrics. The user also specifies the population size, the number of iterations and the solution size (we also can keep this value randomly). It generates as output a Pareto-front which contains optimal solutions of sequence of refactorings to be applied on the analyzed system.
3.2. Experimental Setup

Several tests are done in order to set the NSGA-II parameters. The stopping criterion was set to 1000 iterations, the Max_size of the population to 30, the crossover probability to 0.9 and the mutation probability to 0.4. These values were obtained by trial and error. In our case, we selected a high mutation rate because it allows the continuous diversification of the population, which discourages premature convergence to occur. A standard desktop computer (i7 CPU running at 3 GHz with 8GB of RAM) was used to run the NSGA-II algorithm. The run-time of our algorithm with a 1000 iterations (stopping criteria) was less than 4 min, which indicates the scalability of our approach from the performance standpoint. However, the run-time depends on the number of refactorings and the size of the models in the base of examples.

We analyzed eight open-source Java projects to answer the research questions reported above:

- Ant (v1.8.4): A Java library that is mainly used for building Java applications. Ant provides support to compile, assemble, test and run Java applications.
- GanttProject (v2.0.10): A Java project that supports project management and scheduling.
- JabRef (v2.7): A graphical application for managing bibliographical databases.
- JGraphx (v1.10.4.0): A Java Swing diagramming (graph visualisation) library.
- JHotDraw (v5.2): A framework for the creation of drawing editors.
- JRDF (v0.5.6.2): A Java library for parsing, storing and manipulating RDF (Resource Description Framework).
- Xerces (v2.7): A set of parsers compatible with Extensible Markup Language (XML).
- Xom (v1.2.8): A new XML object model.

The choice of these open source projects is based on the fact that they are medium-sized open-source projects and most of them were analyzed in related work (e.g., [24], [25], [26] and [17]). Most of these open source projects have been actively developed over the past 10 years. Table 5 provides some relevant information about these projects. Table 7 describes the number of good and bad examples and their sizes extracted from Ant 1.8.4, GanttProject 2.0.10, JHotDraw 5.2, Xerces 2.7. Most of the examples are selected based on the previous studies that analyzed these systems [50, 51, 52].

<table>
<thead>
<tr>
<th>Model</th>
<th>Classes</th>
<th>Methods</th>
<th>Attributes</th>
<th>Expected refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant 1.8.4</td>
<td>824</td>
<td>2090</td>
<td>1048</td>
<td>139</td>
</tr>
<tr>
<td>GanttProject 2.0.10</td>
<td>479</td>
<td>960</td>
<td>495</td>
<td>91</td>
</tr>
<tr>
<td>JabRef 2.7</td>
<td>594</td>
<td>253</td>
<td>237</td>
<td>32</td>
</tr>
<tr>
<td>JGraphx 1.10.4.0</td>
<td>191</td>
<td>1284</td>
<td>420</td>
<td>96</td>
</tr>
<tr>
<td>JHotDraw 5.2</td>
<td>160</td>
<td>519</td>
<td>141</td>
<td>71</td>
</tr>
<tr>
<td>JRDF v0.5.6.2</td>
<td>734</td>
<td>19</td>
<td>10</td>
<td>41</td>
</tr>
<tr>
<td>Xerces 2.7</td>
<td>625</td>
<td>2113</td>
<td>1408</td>
<td>182</td>
</tr>
<tr>
<td>Xom 1.2.8</td>
<td>252</td>
<td>186</td>
<td>31</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>#Bad Examples (min#classes, max#classes)</th>
<th>#Good Examples (min#classes, max#classes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant 1.8.4</td>
<td>72(8, 23)</td>
<td>42(7, 29)</td>
</tr>
<tr>
<td>GanttProject 2.0.10</td>
<td>49(6, 21)</td>
<td>28 (6, 22)</td>
</tr>
<tr>
<td>JHotDraw 5.2</td>
<td>34(9, 19)</td>
<td>19(8, 24)</td>
</tr>
<tr>
<td>Xerces 2.7</td>
<td>38 (6, 14)</td>
<td>34 (7, 19)</td>
</tr>
</tbody>
</table>

3.3. Measures of precision and recall

To answer our research questions, we used two measures: Precision and Recall. These measures stem originally from the area of information retrieval (IR). Precision is given by Equation (5). It is equal to the ratio of the “Number of correct refactorings detected” to the “Total number of refactorings detected”. A. Recall is given by Equation (6). It is equal to the ratio of the “Number of correct refactorings detected” to the “Number of correct refactorings”. Both values may range from 0% to 100%, whereas a higher value is better than a lower one. We consider that the threshold for the correctness of recommended refactorings is 85%.”

\[ R_{TPR} = \frac{\text{Number of correct refactorings detected}}{\text{Total number of refactorings detected}} \] (5)
\[ RT2R = \frac{\text{Number of correct refactorings detected}}{\text{Number of correct refactorings}} \] (6)

3.4. RTRP and RT2R were determined for each system based on the traceability matrix that is linking each artifact to the corresponding requirement (correct artifacts). This matrix is known for the systems (projects) used in the evaluation.

Results and discussion

Our results are presented based on two indicators: precision and recall. For our validation, we conducted multiple executions (40 executions) of our approach on the all considered projects. Fig. 9 illustrates the average of precision and recall values for each open source project over the 40 executions. Fig. 10 illustrates the box plots of the precision scores of our algorithm on the different projects based on 40 executions. Similarly, Fig. 11 illustrates the box plots of the precision scores of our algorithm on the different projects based on 40 executions.

![Fig. 9. Average of precision and recall over 40 execution of our approach on all projects](image)

We noticed very high values of the average of precision and recall which is over 90% for GanttProject, JHotDraw, JRDF and Xerces and less than 90% for the rest of projects (Ant, JabRef, JGraphx (88%) and Xom (86%)) under test. Indeed, the most common intervals [80%-100%] displayed by Fig. 10 and Fig. 11 prove that precision and recall scores are approximately the same for different executions in the eight projects under analysis. The most common range of 20% between the minimum (80%) and the maximum (100%) for each project is a sign of stability of the approach. For example, for JHotDraw and GanttProject, the average of the precision values is around 90% and is over than 94% for the average of the recall, while Xerces had approximately the same average of the precision and recall around 92%. These results allow us to positively answer our first research question RQ1 and conclude that the obtained results are very encouraging.
The advantage with the multi-objective approach is that NSGA-II, unlike GA, produce a set of solutions
called the Pareto-front. In our context, NSGA-II converges to Pareto-optimal solutions that are considered as good trade-off between structural and textual similarities. Fig. 12 display the Pareto-front for NSGA-II obtained on the considered open source projects. In these figures, each point is a solution with the structural similarity score represented in x-axis, the textual similarity score in the y-axis. The best solutions exist in the corner representing the Pareto-front that maximizes the values of the textual and the structural similarities.
To answer the second research question, RQ2, we compared our NSGA-II adaptation to the current, state-of-the-art refactoring approaches widely used multi-objective algorithm, MOPSO, using the same adapted fitness function, NSGA-II without the use of good design examples (NSGA-II BE) [19], to a mono objective approach (MONO) using genetic algorithm, to our previous work (MOREX: MOdel Refacttoring by EXamples) [16], to a genetic programming (GP) based approach [23] and to a random search approach. In MONO, we defined the fitness function as combination of both structural and textual fitness functions. The MOPSO used in this paper is the Non-dominated Sorting PSO (NSPSO) proposed by Li [19]. In a random search, the change operators (crossover and mutations) are not used, and populations are generated randomly and evaluated using the two fitness functions. To better evaluate the relevance of considering well designed model fragment examples, we compared our NSGA-II adaptation to our previous work based also on NSGA-II but without the use of good examples (called NSGA-II BE) [19]. In MOREX [16], we used a single-objective genetic algorithm to propose refactorings; i.e., we considered only one fitness function based on the structural similarity which is a combination of software metrics. In GP [20], the authors proposed an approach to generate detection rules based on quality metrics by using GP.

It is clear from Fig. 13 and Fig. 14 that our NSGA-II adaptation outperforms all the mono-objective approaches (MONO, MOREX and GP) in 100% of experiments. As shown by Fig. 13 and Fig. 14, NSGA-II had higher average of precision and recall than MONO, MOREX and GP, and it beats by far the random search approach. For example for GanttProject, the NSGA-II average precision and average recall values are 91% and 96%, respectively, while these values are 82% and 86% in MOREX, 78% and 82% in GP, and 41% and 43% in the random search algorithm where these values do not exceed 50% for all the eight projects. For example, we noticed an increase of 9% on average between NSGA-II and MOREX that could be considered as great improvement. This improvement can be explained by the fact that NSGA-II aims to find a compromise between two similarities (structural and textual). However MOREX did not consider textual similarity but only structural one. When comparing NSGA-II with the remaining approaches we considered the best solution selected from the Pareto-optimal front using the knee point-based strategy [27]. The results obtained in Fig. 13 and Fig. 14 support the claim that our NSGA-II formulation provides a good trade-off between structural and textual similarities, and outperforms on average the MOPSO approach, excepted for Ant, JRDF and XOM opens source systems where precision and recall are quite similar.
As the considered algorithms are meta-heuristics, they can produce different results on every run when applied to the same problem instance. For this reason, we used the p-values of the Wilcoxon rank-sum test [28] as a statistical test to compare the results of the five algorithms: NSGA-II, MONO, MOREX [16], NSGA-II BE [19], GP [20] and MOPSO [19]. We independently performed 40 executions using the six algorithms in MONO, MOREX [16], GP [20], NSGA-II BE [19] and MOPSO [19], for the eight open-source projects that we used in
In our experiment, a p-value that is less than or equal to $\alpha (=0.05)$ means that the distributions of the results of the five algorithms are different in a statistically significant way. In fact, we computed the p-values of MONO, MOREX, GP and MOPSO results compared with NSGA-II. In this way, we could decide whether the outperformance of our approach over the MONO, MOREX [16], [29]GP, NSGA-II BE [19] and MOPSO approach is statistically significant. Table 8 and Table 9 display the precision and recall median values of NSGA-II, MONO, MOREX, GP, NSGA-II BE [19] and MOPSO for the 8 open source projects respectively. For example, for Ant project, the p-value for the precision median results of MONO compared with NSGA-II is 0.0103 while the p-value of the recall median results of MONO compared with NSGA-II is 0.0113. The p-value for the precision median results of MOREX compared with NSGA-II is 0.0224 while the p-value of the recall median results of MOREX compared with NSGA-II is 0.0213. The p-value for the precision median GP compared with NSGA-II is 0.0112 while the p-value of the recall median results of GP compared with NSGA-II is 0.0142. In addition, the p-value for the precision median results of MOPSO compared with NSGA-II is 0.0098 while the p-value of the recall median results of MOPSO compared with NSGA-II is 0.0087. Accordingly, we infer that the precision and recall median values of our algorithm are statistically different from the MONO, MOREX, GP, NSGA-II BE [19] and the MOPSO ones on each of the systems based on the fact that these p-values are less than $\alpha (= 0.05)$. We consequently conclude that our approach is more effective than these four approaches and specifically it is more effective than an approach based only on the structural similarity without taking into account the context of the entities of analyzed models (i.e., MOREX). This observation allows us to positively answer our second research question RQ2.

Table 8. Precision median values of NSGA-II, MONO, MOREX [16], GP [20], NSGA-II BE [19] and MOPSO [19] over 40 independent simulation runs

<table>
<thead>
<tr>
<th>Model</th>
<th>NSGA-II</th>
<th>NSGA-II BE [19]</th>
<th>MONO</th>
<th>MOREX</th>
<th>GP</th>
<th>MOPSO</th>
<th>NSGA-II vs MONO</th>
<th>NSGA-II vs [19]</th>
<th>NSGA-II vs MOREX</th>
<th>NSGA-II vs GP</th>
<th>NSGA-II vs MOPSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>95</td>
<td>92</td>
<td>91</td>
<td>93</td>
<td>94</td>
<td>79</td>
<td>0.0103</td>
<td>0.0147</td>
<td>0.0224</td>
<td>0.0112</td>
<td>0.0098</td>
</tr>
<tr>
<td>Gantt</td>
<td>90</td>
<td>84</td>
<td>83</td>
<td>84</td>
<td>80</td>
<td>75</td>
<td>0.0015</td>
<td>0.0014</td>
<td>0.0158</td>
<td>0.0101</td>
<td>0.0102</td>
</tr>
<tr>
<td>JabRef</td>
<td>84</td>
<td>80</td>
<td>79</td>
<td>78</td>
<td>77</td>
<td>73</td>
<td>0.0101</td>
<td>0.0111</td>
<td>0.0124</td>
<td>0.0077</td>
<td>0.0045</td>
</tr>
<tr>
<td>JGraphx</td>
<td>86</td>
<td>80</td>
<td>75</td>
<td>77</td>
<td>73</td>
<td>70</td>
<td>0.0017</td>
<td>0.0047</td>
<td>0.0212</td>
<td>0.0111</td>
<td>0.0014</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>90</td>
<td>84</td>
<td>82</td>
<td>85</td>
<td>84</td>
<td>80</td>
<td>0.0078</td>
<td>0.0019</td>
<td>0.0222</td>
<td>0.0056</td>
<td>0.0075</td>
</tr>
<tr>
<td>JRDF</td>
<td>90</td>
<td>87</td>
<td>86</td>
<td>88</td>
<td>84</td>
<td>82</td>
<td>0.0028</td>
<td>0.0027</td>
<td>0.0189</td>
<td>0.0045</td>
<td>0.0084</td>
</tr>
<tr>
<td>Xerces</td>
<td>92</td>
<td>88</td>
<td>87</td>
<td>89</td>
<td>86</td>
<td>83</td>
<td>0.0102</td>
<td>0.0043</td>
<td>0.0211</td>
<td>0.0059</td>
<td>0.0047</td>
</tr>
<tr>
<td>Xom</td>
<td>87</td>
<td>87</td>
<td>84</td>
<td>87</td>
<td>86</td>
<td>70</td>
<td>0.0094</td>
<td>0.0052</td>
<td>0.0051</td>
<td>0.0091</td>
<td>0.0031</td>
</tr>
</tbody>
</table>
The proposed approach is different than a classical data mining technique or a manual inspection of the history to identify similar refactoring patterns. First, the proposed approach does not find similarities between refactoring but the current model design is compared to examples of bad and good design models then the applied refactorings to that similar bad design model are adapted to the current context. It will be difficult for a developer to identify similarities between model fragments based on the history of changes, especially when the number of changes are high. Second, our multi-objective approach does not require to use examples from previous releases of the project to evaluate. The experiments show that examples from open source systems could be a very good starting point for developers in practice/industry to use our technique. Finally, our approach generates a set of Pareto front solutions and not a single solution which is the case of machine learning and data mining algorithms. Thus, the developer can select a solution based on his preferences and not limited to one set of recommended refactorings.

Since we viewed the matching problem as a combinatorial problem addressed with heuristic search, it is important to contrast the correctness results with the execution time. We executed our algorithm on a standard desktop computer (Pentium CPU running at 3GHz with 8GB of RAM). The average execution time is shown in Figure 15. The execution time of the mono-objective algorithms is slightly lower than our NSGA-II approach and NSGA-II BE [19] but the MOPSO one was higher than ours (due to the used changes operators by MOPSO). In any case, our approach is meant to apply to situations where the execution time is not the primary concern. Figure 16 shows the distribution of the different types of refactoring recommended by our approach. It is clear that move method, extract class and push down method are the most frequent refactorings between the different solutions.

<table>
<thead>
<tr>
<th>Model</th>
<th>Recall (%)</th>
<th>P-value (&lt;= 0.05)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NSGA-II</td>
<td>NSGA-II BE [19]</td>
</tr>
<tr>
<td>Ant</td>
<td>81</td>
<td>79</td>
</tr>
<tr>
<td>Gantt</td>
<td>94</td>
<td>88</td>
</tr>
<tr>
<td>JabRef</td>
<td>91</td>
<td>90</td>
</tr>
<tr>
<td>JGraphx</td>
<td>87</td>
<td>83</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>94</td>
<td>87</td>
</tr>
<tr>
<td>JRDF</td>
<td>89</td>
<td>84</td>
</tr>
<tr>
<td>Xerces</td>
<td>91</td>
<td>84</td>
</tr>
<tr>
<td>Xom</td>
<td>82</td>
<td>79</td>
</tr>
</tbody>
</table>
Fig. 15. Comparison between NSGA-II, MONO, MOREX [16], GP [20], NSGA-II BE [19], MOPSO [19] and Random search in terms of average execution time on the different systems.

Fig. 16. The average distribution of refactoring types in the best solutions recommended by our approach on the different models.

4. Threats to validity

The most important point that we consider as threats to the generalization of our approach is the use of the Ref_finder tool to build the base of examples and at the same time, we compare the results obtained by our algorithm to those given by Ref_finder.
In general, we do not need a large number of examples to obtain good results which confirm the reliability of the proposed approach. We showed, in our experiment, that some open source projects used out of the box could produce good refactoring results for the systems under analysis. However, we agree that, sometimes, within specific contexts it is difficult to define and find refactorings’ opportunities. Applications from different domains may have dissimilar design practices and developers may have different opinions on how to define a good and bad design example. In fact, the quality of our results heavily depends on the examples introduced as inputs. In an industrial setting, we could expect a company to start with some few open source projects, and gradually migrate its set of refactoring examples to include context-specific data. In fact, the definition of “good” and “bad” examples are left to the developers based on their preferences and best/bad design practices. A possible use of previous releases of evolved models could be a good strategy to adapt in order to define a set of bad and good design fragments. This might be essential if we consider that different languages and software infrastructures have different best/worst practices. Another threat is related to the used textual measures. Sometimes two classes have similar names but they are semantically equivalent. Also, two classes might have different names, but be semantically similar.

5. Related Work

In this section, we classified a relevant existing works that tackled the automation of refactoring activities using search-based techniques’ into two main groups: single-objective and multi-objective optimization approaches.

In the first group, most of existing works defined their fitness function based on software metrics in order to find the best sequence of refactorings (e.g. [14] [30] [31] [20] [57] [58] [59] [60] [61] [62] [63] [64] [65] and [32]). A comparative study of four heuristic techniques applied to the refactoring problem was presented by O’Keeffe et al. [30] using a fitness function that combine 11 metrics to evaluate the quality improvements. They experiment the four techniques on five open-source systems. They find that hill-climbing outperforms the other three algorithms. Another single–objective optimization was proposed by Seng et al. [14] using on the genetic algorithm (GA) in order to suggest a list of refactorings. The single fitness function used in this proposal aims to maximize a weighted sum of a set of metrics to improve the class structure of a software system. These metrics are mainly related to the coupling, cohesion, complexity and stability. The authors have defined some refactorings’ preconditionsable to preserve the behaviour but not the semantics domain of the software program. However, authors have validated their proposal only on the move method refactoring. The refactoring schedule problem was considered by Qayum et al. [31] as a graph transformation problem expressings as a search for an optimal path. To do that, they used Ant colony optimization in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. However, the domain semantics of the software system and its runtime behavior was not considered. Last but not least, Kessentini et al. [20] have proposed a single–objective optimization approach using GA to find the best sequence of refactoring operations at code level aiming to improve the quality by minimizing the design defects’ number detected in the source code.

In the second group, Pareto-Front concept have been used by Harman et al. [18] in order to improve search based refactoring approaches. The authors have combined two software metrics: (1) CBO (coupling between objects) and (2) SDMPC (standard deviation of methods per class). Each metric have been assigned to one objective (fitness function). The authors found that the proposed multi-objective algorithm is able to find a good sequence of move method refactorings. The obtained sequence represents the best trade-off between CBO and SDMPC to improve code quality. Another multi-objective optimization approach have been proposed by Ouni et al. [17] using NSGA-II providing the best trade-off between two objectives: (1) quality based on the metric that calculates the number of corrected defects detected in the initial system and (2) effortbased on the hat calculates the code modifications score metric. In another work, Ouni et al. [33] tried to find the best trade-off to find the best sequence of refactorings by maximizing the quality improvements and minimizing


semantic errors. Ó Cinnéide et al. [34] have investigated, via an empirical study, about the assessment of some structural software metrics and their relationships between them, based on a variety of search techniques (Pareto-optimal search, semi-random search). In conclusion, vast majority of existing work on search-based software engineering approaches focused only on the program structure improvements based on a set of software metrics in both single and multi-objectives approaches. Thaina et al. [53] proposed a recent systematic literature review on search based refactoring.

Interactive techniques have been generally introduced in the literature of Search-Based Software Engineering and especially in the area of software modularization. Hall et al. [54] treated software modularization as a constraint satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer. The approach, called SUMO (Supervised Re-modularization), consists of iteratively feeding domain knowledge into the remodularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO.

Bavota et al. [55] presented the adoption of single objective interactive genetic algorithms in software remodularization process. The main idea is to incorporate the user in the evaluation of the generated remodularizations. Interactive Genetic Algorithms (IGAs) extend the classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the solution’s fitness function. The basic idea of the Interactive GA (IGA) is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Initially, the IGA evolves similarly to the non-interactive GA. After a user-defined set of iterations, the individual with the highest fitness value is selected from the population set (in the case of single-objective GA) or from the first front (in the case of multi-objective GA) and presented to the user. After analyzing the current modularization, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Overall, the above existing studies of interactive remodularization are limited to few types of refactoring such as moving classes between packages and splitting packages. Furthermore, the interaction mechanism is based on the manual evaluation of proposed remodularization solutions which could be a time-consuming process. A recent study [56] extended our previous work [57] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one.

In other side, there are some contributions that focused on the automation of refactoring activities at the model-level based on rules. In general, the used rules have been expressed either as assertions (i.e., invariants, pre-and post-condition) [35, 36], or as graph transformations targeting refactoring operations in general (e.g., [37, 38]) or refactorings related to design patterns’ applications (e.g., [7]). For example, Ragnhild et al. have [35] proposed to detect some parts of the model that need refactoring using declarative rules. However, this proposal require an important number of rules. In addition, refactoring rules must be complete, consistent, non-redundant and correct to specify clearly and completely of the refactorings. Refactoring rules have been used also by ElBoussaidi and Mili [7] in order to specify design patterns’ applications. To do that, the authors represent the design problems (solved by these patterns) based on models. Then, the obtained models will be transformed using refactoring rules according to the solutions proposed by the patterns. However, models are not able to represent all design problems at hand. For example, the problem space for the observer pattern is quite large and the problem cannot be captured in a single, or a handful of problem models [7]. In conclusion, a common issue for in most of the proposed approaches is how to sequence and compose the refactorings rules.
Some other studies have tackled the refactoring opportunities detection at model level using code smells defined as bad design choices which could negatively impact some code qualities such as maintainability, changeability and comprehensibility [39]. Indeed, during the evolution of the system, code-smells can emerge and then represent patterns of software design. These patterns can be a source of problems in the further development and maintenance of the system. Fowler et al [2] identified and defined 22 Code Smells aiming to indicate software refactoring opportunities and ‘give you indications that there is trouble that can be solved by a refactoring’. Zhang et al. [40] investigated about code-smells that need more attention than any other. Van Emden and Moonen [41] and Mantyla [42] proposed two approaches that aim to detect and analyze code smells for java programs. Previous empirical studies have analysed the impact of code-smells on different software maintainability factors including defects [43] and effort [44]. In fact, software metrics (quality indicators) are sometimes difficult to interpret and suggest some actions (refactoring) as noted by Anda et al. [45] and Marinescu et al. [46].

6. Conclusion

We presented a by example search-based approach that exploits both structural and design consistency information to improve the automation of suggesting refactoring. It takes as input a model to be refactored, a base of examples of models and their subsequent refactorings, and a list of metrics and textual measures calculated on both the initial model and the models in the base of examples. The output is a solution to the refactoring problem. A solution is a sequence of refactoring operations that should be applied to the initial model and that displays the best compromise between the two criteria: structural and textual similarities. In contrast to existing work on refactorings, the design consistency is a major concern in our paper.

Our experimentation shows that our technique outperforms state-of-the-art techniques where single-objective and multi-objective is used. We evaluated our approach on real-world models and the obtained results indicate that the proposed refactorings are comparable to those expected. We also tested the stability of our approach by performing multiple executions on the projects at hand and checking the it’s the stability of precision and recall values over the multiple executions. These results allowed us to conclude that the proposed approach is more efficient and promising than approaches that do not consider the design consistency in their optimization objectives.

In the future, we plan to extend our base of examples to include more refactoring operations. We also plan to analyse the domain-specific impact on the obtained results.

Acknowledgment

This work was made possible by NPRP grant # [7-662-2-247] from Qatar Research Fund (a member of Qatar Foundation). The findings achieved herein are solely the responsibility of the authors.

References

4. Opdyke, F.W., Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. 1992, University of Illinois at Urbana-Champaign.


Marouane Kessentini, Stéphane Vaucher, Houari A. Sahraoui: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. ASE 2010: 113-122


