Model Refactoring Using Interactive Genetic Algorithm

Adnane Ghannem[1], Ghizlane El Boussaidi[1], and Marouane Kessentini[2]

[1] École de Technologie Supérieure, Montréal, Canada
adnane.ghannem.1@ens.etsmtl.ca, ghizlane.Elboussaidi@etsmtl.ca
[2] Missouri University of Science and Technology, Rolla, MO, USA
marouanek@mst.edu

Abstract. Refactoring aims at improving the quality of design while preserving its semantic. Providing an automatic support for refactoring is a challenging problem. This problem can be considered as an optimization problem where the goal is to find appropriate refactoring suggestions using a set of refactoring examples. However, some of the refactorings proposed using this approach do not necessarily make sense depending on the context and the semantic of the system under analysis. This paper proposes an approach that tackles this problem by adapting the Interactive Genetic Algorithm (IGA) which enables to interact with users and integrate their feedbacks into a classic GA. The proposed algorithm uses a fitness function that combines the structural similarity between the analyzed design model and models from a base of examples, and the designers' ratings of the refactorings proposed during execution of the classic GA. Experimentation with the approach yielded interesting and promising results.

Keywords: Software maintenance, Interactive Genetic Algorithm, Model refactoring, Refactoring by example.

1 Introduction

Software maintenance is considered the most expensive activity in the software system lifecycle [1]. Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws. However, as the time goes by, the system's conceptual integrity erodes and its quality degrades; this deterioration is known in the literature as the software decay problem [2]. A common and widely used technique to cope with this problem is to continuously restructure the software system to improve its structure and design. The process of restructuring object oriented systems is commonly called refactoring [3]. According to Fowler [2], refactoring is the disciplined process of cleaning up code to improve the software structure while preserving its external behavior. Many researchers have been working on providing support for refactoring operations (e.g., [4], [2], and [5]). Existing tools provide different environments to manually or automatically apply refactoring operations to correct, for example, code smells. Indeed, existing work has, for the most part, focused on refactorings at the source code level. Actually, the rise of the model-driven engineering (MDE) approach increased the interest and

G. Ruhe and Y. Zhang (Eds.): SSBSE 2013, LNCS 8084, pp. 96–110, 2013.

the needs for tools supporting refactoring at the model-level. In MDE, abstract models are successively refined into more concrete models, and a model refactoring tool will be of great value within this context.

The search-based refactoring approaches proved their effectiveness to propose refactorings to improve the model's design quality. They adapted some of the known heuristics methods (e.g. Simulated annealing, Hill_climbing) as proposed in [6-8] and Genetic Algorithms as in [9]. These approaches relied, for the most part, on a combination of quality metrics to formulate their optimization goal (i.e., the fitness function). A major problem founded in these approaches is that the quality metrics consider only the structural properties of the system under study; the semantic properties of the system are not considered. In this context, Mens and Tourwé [3] argue that most of the refactoring tools cannot offer a full-automatic support because part of the necessary knowledge– especially those related to the semantics– for performing the refactoring remains implicit in designers' heads. Indeed, recognizing opportunities of model refactoring remains a challenging issue that is related to the model marking process within the context of MDE which is a notoriously difficult problem that requires design knowledge and expertise [10].

To take into account the semantics of the software system, we propose a model refactoring approach based on an Interactive Genetic Algorithm (IGAs) [11]. Two types of knowledge are considered in this approach. The first one comes from the examples of refactorings. For this purpose, we hypothesize that the knowledge required to propose appropriate refactorings for a given object-oriented model may be inferred from other existing models' refactorings when there is some structural similarities between these models and the given model. From this perspective, the refactoring is seen as an optimization problem that is solved using a Genetic Algorithm (GA). The second type of knowledge comes from the designer's knowledge. For this purpose, the designer is involved in the optimization process by continuously interacting with the GA algorithm; this enables to adjust the results of the GA progressively exploiting the designer's feedback. Hence the proposed approach (MOREX+I: MOdel REfactoring by eXample plus Interaction) relies on a set of refactoring examples and designer's feedbacks to propose sequences of refactorings. MOREX+I takes as input an initial model, a base of examples of refactored models and a list of metrics calculated on both the initial model and the models in the base of examples, and it generates as output a solution to the refactoring problem. In this paper, we focus on UML class diagrams. In this case, a solution is defined as a sequence of refactorings that maximize as much as possible the similarity between the initial and revised class diagrams (i.e., the class diagrams in the base of examples) while considering designer's feedbacks.

The primary contributions of the paper are 3-fold: 1) We introduce a model refactoring approach based on the use of examples. The approach combines implicitly the detection and the correction of design defects at the model-level by proposing a sequence of refactorings that must be applied on a given model. 2) We use the IGA to allow the integration of feedbacks provided by designers upon solutions produced during the GA evolution. 3) We report the results of an evaluation of our approach.

The paper is organized as follows. Section 2 is dedicated to the background where we introduce some basic concepts and the related work. The overall approach is described in Section 3. Section 4 reports on the experimental settings and results, while Section 5 concludes the paper and outlines some future directions to our work.

2 Background

2.1 Class Diagrams Refactorings and Quality Metrics

Model refactoring is a controlled technique for improving the design (e.g., class diagrams) of an existing model. It involves applying a series of small refactoring operations to improve the design quality of the model while preserving its behavior. Many refactorings were proposed and codified in the literature (see e.g., [2]). In our approach, we consider a subset of the 72 refactorings defined in [2]; i.e., only those refactorings that can be applied to UML class diagrams. Indeed, some of the refactorings in [2] may be applied on design models (e.g. *Move_Method, Rename_method, Move_Attribute, Extract_Class* etc.) while others cannot be (e.g. *Extract_Method, Inline_Method, Replace_Temp_With_Query* etc.). In our approach we considered a list of twelve refactorings (e.g. *Extract_class, Push_down_method, Pull_up_method,* etc.) based on [2]. The choice of these refactorings was mainly based on two factors: 1) they apply at the class diagram-level; and 2) they can be link to a set of model metrics (i.e., metrics which are impacted when applying these refactorings).

Metrics provide useful information that help assessing the level of conformance of a software system to a desired quality [12]. Metrics can also help detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero et al. [13]. In the context of our approach, we used a list of sixteen metrics (e.g. *Number of attributes: NA, Number of methods: NMeth, Number of dependencies: NDep*, etc.) including the eleven metrics defined in [13] to which we have added a set of simple metrics (e.g., *number of private methods in a class, number of public methods in a class*). All these metrics are related to the class entity which is the main entity in a class diagram.

2.2 Interactive Genetic Algorithm (IGA)

Heuristic search are serving to promote discovery or learning [14]. There is a variety of methods which support the heuristic search as hill_climbing [15], genetic algorithms (GA) [16], etc. GA is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [17]. The basic idea behind GA is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem. Each individual (i.e., a solution) of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. Exploration of the search space is achieved by selecting individuals (in the current population) that have the best fitness values and evolving them by using genetic operators, such as crossover and mutation. The crossover operator insures generation of new children, or offspring, based on

parent individuals while the mutation operator is applied to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search. Once selection, mutation and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations.

Interactive GA (IGAs) [18] combines a genetic algorithm with the interaction with the user so that he can assign a fitness to each individual. This way IGA integrates the user's knowledge during the regular evolution process of GA. For this reason, IGA can be used to solve problems that cannot be easily solved by GA [19]. A variety of application domains of IGA include development of fashion design systems [19], music composition systems [20], software re-modularization [21] and some other IGAs' applications in other fields [11]. One of the key elements in IGAs is the management of the number of interactions with the user and the way an individual is evaluated by the user.

2.3 Related Work

Model refactoring is still at a relatively young stage of development compared to the work that has been done on source-code refactoring. Most of existing approaches for automating refactoring activities at the model-level are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-conditions) [22, 23], or graph transformations targeting refactoring operations in general [24, 25] or design patterns' applications in particular (e.g., [26]). In [22] invariants are used to detect some parts of the model that require refactoring and the refactorings are expressed using declarative rules. However, a complete specification of refactorings requires an important number of rules and the refactoring rules must be complete, consistent, non-redundant and correct. In [26] refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models. Finally an issue that is common to most of these approaches is the problem of sequencing and composing refactoring rules. This is related to the control of rules' applications within rule-based transformational approaches in general.

Our approach is inspired by contributions in search-based software engineering (SBSE) (e.g. [6, 7, 9, 27, 28]). Techniques based on SBSE are a good alternative to tackle many of the above mentioned issues [9]. For example, a heuristic-based approach is presented in [6, 7, 27] in which various software metrics are used as indicators for the need of a certain refactoring. In [27], a genetic algorithm is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object oriented metrics. Harman and Tratt [6] propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in [27] and [6] were limited to the Move Method refactoring

operation. In [7], the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In [28], the authors proposed an automated refactoring approach that uses genetic programming (GP) to support the composition of refactorings that introduce design patterns. The fitness function used to evaluate the applied refactorings relies on the same set of metrics as in [12] and a bonus value given for the presence of design patterns in the refactored design. Our approach can be seen as linked to this approach as we aim at proposing a combination of refactorings that must be applied to a design model. Our approach was inspired by the work in [21] where the authors apply an Interactive Genetic Algorithm to the re-modularization problem which can be seen as a specific subtype of the refactoring problem. Our work is also related to the approach in [29] where the authors apply an SBSE approach to model transformations. However this approach focuses on general model transformations while our focus is on refactorings which are commonly codified transformations that aim at correcting design defaults.

To conclude, most of the approaches that tackled the refactoring as an optimization problem by the use of some heuristics suppose, to some extent, that a refactoring operation is appropriate when it optimizes the fitness function (FF). Most of these approaches defined their FF as a combination of quality metrics to approximate the quality of a model. However, refactoring operations are design transformations which are context-sensitive. To be appropriately used, they require some knowledge of the system to be refactored. Indeed, the fact that the values of some metrics were improved after some refactorings does not necessarily mean or ensure that these refactorings make sense. This observation is at the origin of the work described in this paper as described in the next section.

3 Heuristic Search Using Interactive Genetic Algorithm

3.1 Interactive Genetic Algorithm Adaptation

The approach proposed in this paper exploits examples of model refactorings, a heuristic search technique and the designer's feedback to automatically suggest sequences of refactorings that can be applied on a given model (i.e., a UML class diagram). A high-level view of our adaptation of IGA to the model refactoring problem is given in Fig. 1. The algorithm takes as input a set of quality metrics, a set of model refactoring examples, a percentage value corresponding to the percentage of a population of solutions that the designer is willing to evaluate, the maximum number of iterations for the algorithm and the number of interactions with the designer. First, the algorithm runs classic GA (line 2) for a number of iterations (i.e., the maximum number of iterations divided by the number of interactions). Then a percentage of solutions from the current population is selected (line 3). In lines 4 to 7, we get designers' feedbacks for each refactoring in each selected solution and we update their fitness function. We generate a new population (p+1) of individuals (line 8) by iteratively

selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (solutions). We include both the parent and child variants in the new population. Then we apply the mutation operator, with a probability score, for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the maximum iteration number is reached, and returns the best set of refactorings' sequences (i.e., best solutions from all iterations).

```
Input: Set of quality metrics
Input: Set of model refactoring examples
Input: Percentage (P%)
Input: MaxNbrIterations
Input: NbrOfInteractions
Output: A sequence of refactorings
1: for i = 1 . . . NbrOfInteractions do
2:  Evolve GA for NbrIterations
3:  Select P% of best solutions from the current population.
4:  for-each selected solution do
5:    Ask the designer whether each refactoring within
      the selected solution makes sense.
6:    Update the FF of the selected solution to integrate
      the feedback.
7:  end for-each
8:  Create a new GA population using the updated solutions
9: end for
10: Continue (non-interactive) GA evolution until it converges
    or it reaches maxNbrIterations
```

Fig. 1. High-level pseudo-code for IGA adaptation to our problem

In the following subsections we present the details of the regular GA adaptation to the problem of generating refactoring sequences and how we collect the designers' feedbacks and integrate it in the fitness function computation.

3.2 Representing an Individual and Generating the Initial Population

An individual (i.e., a candidate solution) is a set of blocks. The upper part of Fig. 2 shows an individual with three blocks. The first part of the block contains the class (e.g. Order) chosen from the initial model (model under analysis) called CIM, the second part contains the class (e.g Person) from the base of examples that was matched to CIM called CBE, and finally the third part contains a list of refactorings (e.g. *Pull_Up_Method(calc_taxes(), LineOrder, Orde)*) which is a subset of the refactorings that were applied to CBE (in its subsequent versions) and that can be applied to CIM. In our approach, classes from the model (CIMs) and the base of examples (CBEs) are represented using predicates that describe their attributes, methods and relationships. In addition, the representation of a CBE class includes a list of refactorings that were applied to this class in a subsequent version of the system's model to which CBE belongs. The subset of a CBE subsequent refactorings that are applicable to a CIM class constitutes the third part of the block having CIM as its first part and

CBE as its second part. Hence, the selection of the refactorings to be considered in a block is subjected to some constraints to avoid conflicts and incoherence errors. For example, if we have a *Move_attribute* refactoring operation in the CBE class and the CIM class doesn't contain any attribute, then this refactoring operation is discarded as we cannot apply it to the CIM class.

Hence the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply. The bottom part of Fig. 2 shows the fragments of an initial model before and after the refactorings proposed by the individual (at the top of the figure) were applied.

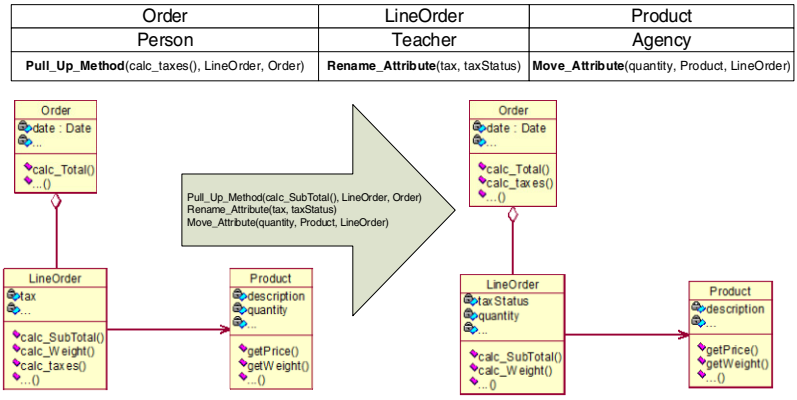

Fig. 2. Individual representation

To generate an initial population, we start by defining the maximum individual size. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual we randomly assign: 1) a set of classes from the initial model that is under analysis and their matched classes from the base of examples, and 2) a set of refactorings that we can possibly apply on the initial model class among the refactorings proposed from the base of examples class.

3.3 Genetic Operators

Selection: To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) [17], in which the probability of selection of an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population p+1. These (population_size/2) selected individuals will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover: For each crossover, two individuals are selected by applying the *SUS* selection [17]. Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring *p'1* and *p'2*

from the two selected parents *p1* and *p2* as follows: A random position, *k*, is selected. The first *k* refactorings of *p1* become the first *k* elements of *p'2*. Similarly, the first *k* refactorings of *p2* become the first *k* refactorings of *p'1*. The rest of refactorings (from position k+1 until the end of the sequence) in each parent *p1* and *p2* are kept. For instance, Fig. 3 illustrates the crossover operator applied to two individuals (parents) *p1* and *p2* where the position *k* takes the value *2*.

Mutation: The mutation operator consists of randomly changing one or more elements in the solution. Hence, given a selected individual, the mutation operator first randomly selects some refactorings among the refactoring sequence proposed by the individual. Then the selected refactorings are replaced by other refactorings. Fig. 4 illustrates the effect of a mutation on an individual.

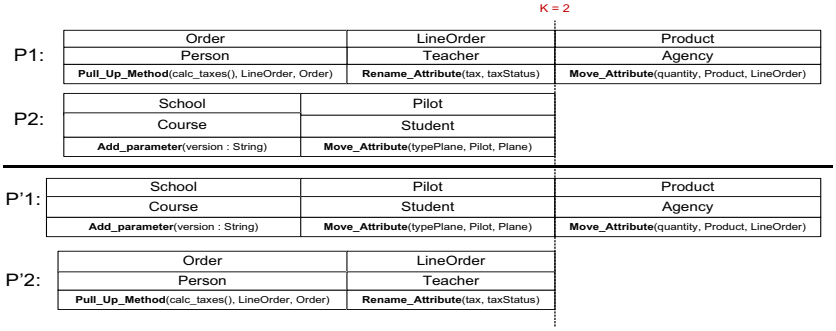

Fig. 3. Crossover operator

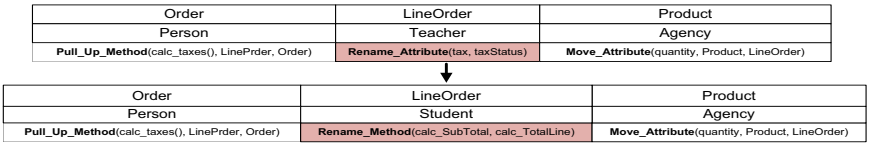

Fig. 4. Mutation operator

3.4 Evaluating an Individual within the Classic GA

The quality of an individual is proportional to the quality of the refactoring operations composing it. In fact, the execution of these refactorings modifies various model fragments; the quality of a solution is determined with respect to the expected refactored model. However, our goal is to find a way to infer correct refactorings using the knowledge that has been accumulated through refactorings of other models of past projects and feedbacks given by designers. Specifically, we want to exploit the similarities between the actual model and other models to infer the sequence of refactorings that we must apply. Our intuition is that a candidate solution that displays a high similarity between the classes of the model and those chosen from the examples base should give the best sequence of refactorings. Hence, the fitness function aims to

maximize the similarity between the classes of the model in comparison to the revised ones in the base of examples. In this context, we introduce first a similarity measure between two classes denoted by *Similarity* and defined by formula 1 and 2.

$$Similarity\ (CMI, CBE) = \frac{1}{m}\sum_{i=1}^{m} Sim(CMI_i, CBE_i) \quad (1)$$

$$Sim\ (CMI_i, CBE_i) = \begin{cases} 1 & if\ CMI_i = CBE_i \\ 0 & if\ CMI_i = 0\ or\ CBE_i = 0 \\ \frac{CMI_i}{CBE_i} & if\ CMI_i < CBE_i \\ \frac{CBE_i}{CMI_i} & if\ CBE_i < CMI_i \end{cases} \quad (2)$$

Where ***m*** is the number of metrics considered in this project. CIM_i is the i^{th} metric value of the class CIM in the initial model while CBE_i is the i^{th} metric value of the class CBE in the base of examples. Using the similarity between classes, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f = \frac{1}{n}\sum_{j=1}^{n} Similarity(CMI_{Bj}, CBE_{Bj}) \quad (3)$$

Where ***n*** is the number of blocks in the solution and CMI_{Bj} and CBE_{Bj} are the classes composing the first two parts of the j^{th} block of the solution. To illustrate how the fitness function is computed, we consider a system containing two classes as shown in Table 1 and a base of examples containing two classes shown in Table 2. In this example we use six metrics and these metrics are given for each class in the model in Table 1 and each class of the base of examples in Table 2.

Table 1. Classes from the initial model and their metrics values

CMI	*NPvA*	*NPbA*	*NPbMeth*	*NPvMeth*	*NAss*	*NGen*
LineOrder	4	1	3	1	1	1
Product	2	2	6	0	1	0

Table 2. Classes from the base of examples and their metrics values

CBE	*NPvA*	*NPbA*	*NPbMeth*	*NPvMeth*	*NAss*	*NGen*
Student	2	1	3	0	3	0
Plane	5	1	4	0	1	0

Consider an individual/solution I_1 composed by two blocks (*LineOrder*/*Student* and *Product*/*Plane*). The fitness function of I_1 is calculated as follows:

$$\boldsymbol{f_{I_1}} = \frac{1}{12}\left[\left(\frac{2}{4} + 1 + 1 + 0 + \frac{1}{3} + 0\right) + \left(\frac{2}{5} + \frac{1}{2} + \frac{4}{6} + 0 + 1 + 0\right)\right] = 0{,}45$$

3.5 Collecting and Integrating the Feedbacks from Designers

Model refactoring is a design operation that is context-sensitive. In addition, depending on the semantics of the system under analysis and the system's evolution as foreseen by different designers, a refactoring proposed by the classic GA can be considered as mandatory by a designer and as acceptable by another. Even if a sequence of refactorings optimizes the fitness function (as defined in the previous section), that does not ensure that these refactorings conform to and preserve the semantics of the system. Consequently, we use Interactive GA (IGA) to partly tackle this problem by interacting with designers and getting their feedbacks on a number of the proposed refactoring sequences. To do so, we adopted a five level scale to rate the proposed refactorings; i.e., we distinguish five types of rating that a designer can assign to a proposed refactoring. The meaning and the value of each type of rating are as follows:

- Critical (value = 1): it is mandatory to apply the proposed refactoring;
- Desirable (value = 0.8): it is useful to apply the refactoring to enhance some aspect of the model but it's not mandatory;
- Neutral (value = 0.5): the refactoring is applicable but the designer does not see it as necessary or desirable;
- Undesirable (value = 0.3): the refactoring is applicable but it is not useful and could alter the semantics of the system;
- Inappropriate (value = 0): the refactoring should not be applied because it breaks the semantics of the system.

As described in section 3.1., during the execution of IGA, the designer is asked to rate a percentage of the best solutions found by the classic GA after a defined number of iterations. For each of the selected solutions, the designer assigns a rating for each refactoring included in the solution. Depending on the values entered by the designer, we re-evaluate the global fitness function of the solution as follows. For each block of the solution, we compute the block rating as an average of the ratings of the refactorings in the block. Then we compute the overall designer's rating as an average of all blocks ratings. Finally, the new fitness function of the solution is computed as an average of its old fitness function and the overall designer's rating. The new values of the fitness functions of the selected solutions are injected back into the IGA process to form a new population of individuals.

4 Experiments

The goal of the experiment is to evaluate the efficiency of our approach for the generation of the refactorings' sequences. In particular the experiment aimed at answering the following research questions:

- **RQ1:** To what extent can the interactive approach generate correct refactorings' sequences?
- **RQ2:** What types of refactorings are correctly suggested?

To answer these questions we implemented and tested the approach on open source projects. In particular, to answer RQ1, we used an existing corpus of known models refactorings to evaluate the precision and recall of our approach, and to answer RQ2, we investigated the type of refactorings that were suggested by our tool. In this section, we present the experimental setup and discuss the results of this experiment.

4.1 Supporting Tool and Experimental Setup

We implemented our approach as a plugin within the Eclipse™ development environment. Fig. 5 shows a screenshot of the model refactoring plugin perspective. This plugin takes as input a base of examples of refactored models and an initial model to refactor. The user specifies the population size, the number of iterations, the individual size, the number of mutations, the number of interactions, and the percentage of the solutions shown in each interaction. It generates as output an optimal sequence of refactorings to be applied on the analyzed system.

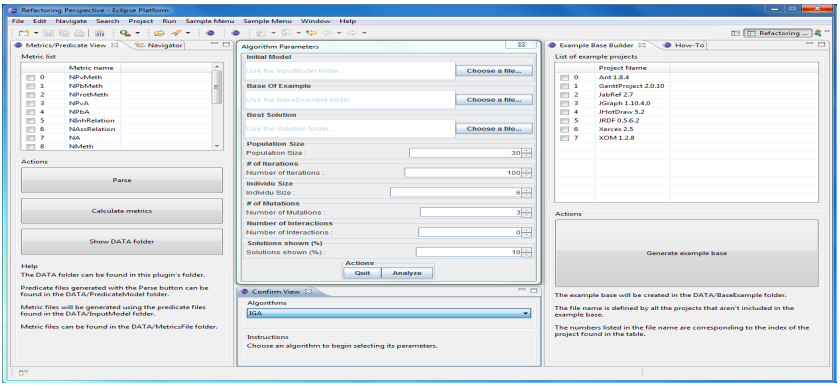

Fig. 5. Model Refactoring Plugin

To build the base of examples, we used the Ref-Finder tool [43] to collect the refactoring that were applied on six Java open source projects (Ant, JabRef, JGraphx, JHotDraw, JRDF, and Xom). Ref-Finder helps retrieving the refactorings that a system has undergone by comparing different versions of the system. We manually validated the refactorings returned by Ref-finder before including them in the base of examples. To answer the research questions reported above, we analyzed two open-source Java projects in our experiment. We have chosen these open source projects because they are medium-sized open-source projects and they have been actively developed over the past 10 years. The participants in the experiment were three Ph.D students enrolled in Software Engineering and all of them are familiar with the two analyzed systems and have a strong background in object-oriented refactoring.

4.2 Results and Discussions

To assess the accuracy of the approach, we compute the precision and recall of our IGA algorithm when applied to the two projects under analysis. In the context of our

study, the precision denotes the fraction of correctly proposed refactorings among the set of all proposed refactorings. The recall indicates the fraction of correctly proposed refactorings among the set of all actually applied refactorings in the subsequent versions of the analyzed projects. To assess the validity of the proposed refactorings, we compare them to those returned by Ref-Finder when applied to the two projects and their subsequent versions. The precision and recall results might vary depending on the refactorings used, which are randomly generated, though guided by a meta-heuristic. Fig. 6 and Fig. 7 show the results of 23 executions of our approach on Xerces and GanttProject, respectively. Each of these figures displays the precision and the recall values for each execution.

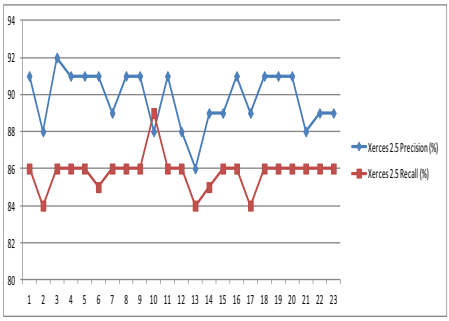

Fig. 6. Multiple execution results for Xerces

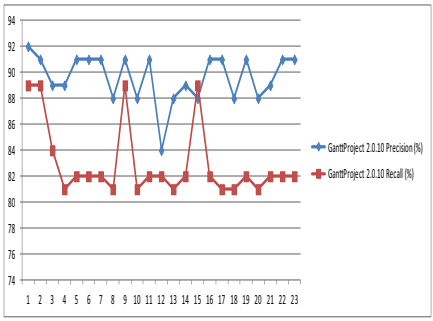

Fig. 7. Multiple Execution results for GanttProject

Generally, the average precision and recall (around 88%) allows us to positively answer our first research question RQ1 and conclude that the results obtained by our approach are very encouraging. The precision in the two projects under analysis (on average 90% of all executions) proves that a big number of the refactorings proposed by our approach were indeed applied to the system's model in its subsequent version (i.e., the proposed refactorings match, in most cases, those returned by Ref-Finder when applied on the system's model and its subsequent version). To ensure that our results are relatively stable, we compared the results of the multiple executions (23) of the approach on the two analyzed projects shown in Fig. 6 and Fig. 7. The precision and recall scores are approximately the same for different executions in the two considered projects. We also compared the sequences of refactorings returned by different executions of our algorithm on the same project. We found that when a class (from the model under analysis) is part of two different returned sequences, the refactoring operations proposed for this class within these sequences are similar. We consequently conclude that our approach is stable.

Our experiment through the interactions with designers allowed us to answer the second research question RQ2 by inferring the types of refactorings they recognized as good refactorings. Fig. 8 shows that 82% of the the *Move_method* and *Pull_up_method* refactorings proposed during the executions are recognized as good refactoring versus only 70% of the *Rename_method* refactorings. We noticed also,

that only 9 of 12 refactorings used in the approach are considered in this analysis. This may result from the quality of the base of examples or from the random factor which characterizes genetic algorithm. We made a further analysis to understand the causes of such results. We found out that through the interactions, the designers have to recognize the meaningless refactorings and penalize them by assigning them a 0 as a rating value; this has significantly reduced the number of these types of refactorings in the optimal solution.

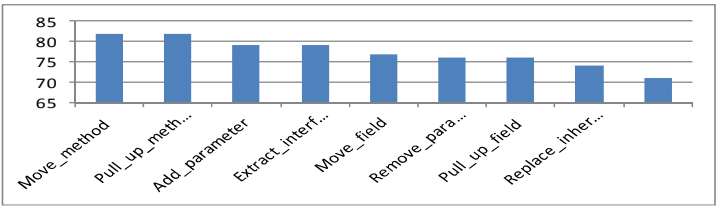

Fig. 8. Distribution of refactorings recognized as correct refactorings through intercations

Despite the good results, we noticed a very slight decrease in recall versus precision in the analyzed projects. Our analysis pointed out towards two factors. The first factor is the project domain. In this study we tried to propose refactorings using a base of examples which contains different projects from different domains. We noticed that some projects focus on some types of refactorings compared to others (i.e., some projects in the base of examples has a big frequency of «*pull_up_Attribute*» and «*pull_up_method*»). The second factor is the number and types of refactorings considered in this experimentation. Indeed, we noticed that some refactorings (e.g., «*pull_up_method*», «*pull_up_Attribute*», «*add_parameter*») are located correctly in our approach. We have no certainty that these factors can improve the results but we consider analyzing them as a future work to further clarify many issues.

4.3 Threats to Validity

We have some points that we consider as threats to the generalization of our approach. The most important one 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. Other threats can be related to the IGAs parameters setting. Although we applied the approach on two systems, further experimentation is needed. Also, the reliability of the proposed approach requires an example set of applied refactoring on different systems. It can be argued that constituting such a set might require more work than these examples. In our study, we showed that by using some open source projects, the approach can be used out of the box and will produce good refactoring results for the studied systems. In an industrial setting, we could expect a company to start with some few open source projects, and gradually enrich its refactoring examples to include context-specific data. This is essential if we consider that different languages and software infrastructures have different best/worst

practices. Finally, since we viewed the model refactorings' generation problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed the plugin on a standard desktop computer (i7 CPU running at 2.67 GHz with 8GB of RAM). The number of interactions was set to 50. The execution time for refactorings' generation with a number of iterations (stopping criteria) fixed to 1000 was less than seventy minutes. This indicates that our approach is reasonably scalable from the performance standpoint.

5 Conclusion and Future Work

In this article, we presented a new approach that aims to suggest appropriate sequences of refactorings that can be applied on a given design model and in particular on a UML class diagram. To do so, we adapted Interactive Genetic Algorithms (IGAs) to build an algorithm which exploits both existing model refactoring examples and the designer's knowledge during the search process for opportunities of model refactorings. We implemented the approach as a plugin integrated within the Eclipse platform and we performed multiple executions of the approach on two open source projects. The results of our experiment have shown that the approach is stable regarding its correctness, completeness and the type and number of the proposed refactorings per class. IGA has significantly reduced the number of meaningless refactorings in the optimal solutions for these executions. While the results of the approach are very promising, we plan to extend it in different ways. One issue that we want to address as a future work is related to the base of examples. In the future we want to extend our base of examples to include more refactoring operations. We also want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, we kept the random aspect that characterizes genetic algorithms even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis. Finally, we want to apply the approach on other open source projects and further analyze the type of refactorings that are correctly suggested.

References

1. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of application software maintenance. Commun. ACM 21(6), 466–471 (1978)
2. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
3. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Softw. Eng. 30(2), 126–139 (2004)
4. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, U. Illinois at Urbana-Champaign (1992)
5. Moha, N.: DECOR: Détection et correction des défauts dans les systèmes orientés objet, p. 157. UdeM & USTdeLille, Montréal (2008)
6. Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of the 9th Annual GECCO 2007, pp. 1106–1113. ACM, London (2007)
7. O'Keeffe, M.: Search-based refactoring: an empirical study. J. Softw. Maint. Evol. 20(5), 345–364 (2008)

8. O'Keeffe, M., Cinneide, M.O.: Search-based software maintenance. In: CSMR (2006)
9. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
10. El-Boussaidi, G., Mili, H.: Detecting Patterns of Poor Design Solutions Using Constraint Propagation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 189–203. Springer, Heidelberg (2008)
11. Takagi, H.: Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. Proceedings of the IEEE 89(9), 1275–1296 (2001)
12. Fenton, N.E., Pfleeger, A.S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn., p. 656. PWS Pub., Boston (1998)
13. Genero, M., Piattini, M., Calero, C.: Empirical validation of class diagram metrics. In: Proceedings of the International Symposium on ESE (2002)
14. Pearl, J.: Heuristics: intelligent search strategies for computer problem solving, p. 382. Addison-Wesley Longman Publishing Co., Inc. (1984)
15. Mitchell, M.: An Introduction to Genetic Algorithms, p. 209. MIT Press (1998)
16. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, p. 372. Addison-Wesley Longman Publishing Co., Inc. (1989)
17. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, p. 680. MIT Press (1992)
18. Dawkins, R.: The BlindWatchmaker, 1st edn., p. 358. Longman, Essex (1986)
19. Kim, H.S., Cho, S.B.: Application of interactive genetic algorithm to fashion design. In: Engineering Applications of Artificial Intelligence (2000)
20. Chen, Y.-P.: Interactive music composition with the CFE framework. SIGEVOlution 2(1), 9–16 (2007)
21. Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M., Oliveto, R.: Putting the developer in-the-loop: an interactive GA for software re-modularization. In: Fraser, G., Teixeira de Souza, J. (eds.) SSBSE 2012. LNCS, vol. 7515, pp. 75–89. Springer, Heidelberg (2012)
22. Van Der Straeten, R., Jonckers, V., Mens, T.: A formal approach to model refactoring and model refinement. J. SoSyM 6(2), 139–162 (2007)
23. Van Kempen, M., et al.: Towards proving preservation of behaviour of refactoring of UML models. In: Proceedings of the annual SAICSIT 2005, pp. 252–259. White River, South Africa (2005)
24. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. J. SoSyM 6(3), 269–285 (2007)
25. Biermann, E.: EMF model transformation based on graph transformation: formal foundation and tool environment. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 381–383. Springer, Heidelberg (2010)
26. El Boussaidi, G., Mili, H.: Understanding design patterns — what is the problem? Software: Practice and Experience (2011)
27. Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proc. of the 8th Annual GECCO 2006, pp. 1909–1916. ACM, Seattle (2006)
28. Jensen, A.C., Cheng, B.H.C.: On the use of genetic programming for automated refactoring and the introduction of design patterns. In: Proc. of the 12th Annual GECCO 2010, pp. 1341–1348. ACM, Portland (2010)
29. Kessentini, M., et al.: Search-based model transformation by example. J. SoSyM 11(2), 209–226 (2012)