An Evolutionary Approach for the Entity Refactoring Set Selection Problem

Camelia Chisăliță-Crețu
Faculty of Mathematics and Computer Science
Babeş-Bolyai University
1, M. Kogălniceanu Street
RO-400084 Cluj-Napoca, Romania
cretu@cs.ubbcluj.ro

ABSTRACT: *Refactoring is a commonly accepted technique to improve the structure of object oriented software. The paper presents a multi-objective evolutionary approach for the Entity Refactoring Set Selection Problem (ERSSP) by treating the cost constraint as an objective and combining it with the effect objective. The refactoring selection problem is discussed here as a multi-objective optimization within the Search-Based Software Engineering (SBSE) field. Different results of the proposed weighted objective genetic algorithm on a experimental didactic case study are presented and discussed.*

Keywords: Refactoring, Search based engineering, Object-oriented programming

Received: 14 August 2009, Revised 28 September 2009, Accepted 3 October 2009

1. Introduction

Software systems continually change as they evolve to reflect new requirements, but their internal structure tends to decay. Refactoring is a commonly accepted technique to improve the structure of object oriented software [9]. Its aim is to reverse the decaying process in software quality by applying a series of small and behaviour-preserving transformations, each improving a certain aspect of the system [9]. Entity Refactoring Set Selection Problem (ERSSP) is the identification problem of the set of refactorings that may be applied to software entities, such that several objectives are kept or improved.

The paper introduces a first formal version definition of the Multi-Objective Entity Refactoring Set Selection Problem (MO-ERSSP) and performs a proposed weighted objective genetic algorithm on an experimental didactic case study. Different solution representation for our case study are used and the corresponding solutions are presented and compared.

The rest of the paper is organized as follows. A working scenario is presented in Section 2. In Section 3 the definition of the ERSSP is given, while Section 4 shortly reminds the principle of multi-objective optimization and introduces the formal definition for the MOERSSP. A short description of the Local Area Network simulation source code used to validate our approach is provided in Section 5. The proposed approach and several details related to the genetic operators of the genetic algorithm and data normalization are described in Section 6. The obtained results for the studied source code using different solution representation are presented and discussed in Section 7. The paper ends with conclusions and future work.

2. Working Scenario

There are still a number of problems to address if someone wants to raise the automation level for refactoring applying. Assuming a tool that detects opportunities for refactoring is used [15], it will identify badly structured code based on code smells [9, 16], metrics [14, 11] or other techniques. The gathered information is used to propose a set of refactorings that can be applied in order to improve the software structure. The developer chooses which refactorings he would consider more appropriate to apply, and use a refactoring tool to apply them. There are several problems that rise up within the considered context. The first one that hits the developer is the large number of refactorings proposed to him, thus the most useful ones to be applied have to be identified. Another aspect is represented by the possible types of dependencies that may exist between the selected refactorings. In means that applying any of the suggested refactorings may cancel the application of other refactorings that have been already selected by the developer. In [12] are presented three kinds of such dependencies: mutual exclusion, sequential dependency, asymmetric conflict, that may be used to drive the refactoring selection process. Therefore, the goal of this paper is to explore the possibility of identifying the optimal sequences of refactorings that keep or improve some objectives, like cost and impact on software entities. Thus, the developer is helped to decide which refactorings are more appropriate and in which order the transformations must be applied, because of different types of dependencies existing between them. ERSSP is an example of a Feature Transformation Subset Selection (FTSS) search problem in SBSE field.

3. ERSSP Definition

In order to state the ERSSP some notion and characteristics have to be defined. Let $SE = \{e_1,...,e_m\}$ be a set of software entities, i.e., a class, an attribute from a class, a method from a class, a formal parameter from a method or a local variable declared in the implementation of a method. They are considered to be low level components bounded through dependency relations. The weight associated with each software entity $e_i, 1 \leq i \leq m$ is kept by the set $Weight = \{w_1,...,w_m\}$, where $w_i \in [0,1]$ and $\sum_{i=1}^{m} w_i = 1$. A software system SS consists of a software entity set SE together with different types of dependencies between the contained items.

A set of possible relevant chosen refactorings [9] that may be applied to different types of software entities of SE is gathered up through $SR = \{r_1,...,r_t\}$. There are various dependencies between such transformations when they are applied to the same software entity, a mapping emphasizing them being defined by:

$$rd : SR \times SR \times SE \rightarrow \{Before, After, AlwaysBefore, AlwaysAfter, Never, Whenever\},$$

$$rd(r_h, r_l, e_i) = \begin{cases} B, if\ r_h\ may\ be\ applied\ to\ e_i\ only\ before\ r_l, r_h < r_l \\ A, if\ r_h\ may\ be\ applied\ to\ e_i\ only\ after\ r_l,\ r_h > r_l \\ AB, if\ r_h\ and\ r_l\ are\ both\ applied\ to\ e_i\ then\ r_h < r_l \\ AA, if\ r_h\ and\ r_l\ are\ both\ applied\ to\ e_i\ then\ r_h > r_l \\ N, if\ r_h\ and\ r_l\ cannot\ be\ both\ applied\ to\ e_i \\ W, otherwise,\ i.e.,\ r_h\ and\ r_l\ may\ be\ both\ applied\ to\ e_i \end{cases},$$

where $1 \leq h, l \leq t, 1 \leq i \leq m$. The effort involved by each transformation is converted to cost, described by the following function:

$$rc : SR \times SE \rightarrow Z,$$

$$rc(r_l, e_i) = \begin{cases} > 0, if\ r_l\ may\ be\ applied\ to\ e_i \\ = 0, otherwise \end{cases},$$

where $1 \leq l \leq t$, $1 \leq i \leq m$. Changes made to each software entity e_i, $i = \overline{1,m}$ by applying the refactoring $r_l, 1 \leq l \leq t$ are stated and a mapping is defined:

$$effect : SR \times SE \rightarrow Z,$$

$$effect(r_l, e_i) = \begin{cases} > 0, if\ r_l\ is\ applied\ to\ e_i\ and\ has\ the\ requested\ effect\ on\ it \\ < 0, if\ r_l\ is\ applied\ to\ e_i;\ has\ not\ the\ requested\ effect\ on\ it, \\ = 0, otherwise \end{cases}$$

where $1 \leq l \leq t$, $1 \leq i \leq m$. The overall effect of applying a refactoring $r_l, 1 \leq l \leq t$ to each software entity e_i, $i = \overline{1,m}$ is defined as:

$$res : SR \rightarrow Z,$$

$$res(r_l) = \sum_{i=1}^{m} w_i \cdot effect(r_l, e_i),$$

where $1 \leq l \leq t$. Each refactoring r_l, $l = \overline{1,t}$ may be applied to a subset of software entities, defined as a function:

$$re : SR \to P(SE),$$

$$re(r_l) = \{ \ e_{l_1}, \ldots, e_{l_q} \ \mid \ if \ r_l \ is \ applicable \ to \ e_{l_u}, 1 \le u \le q, 1 \le q \le m \ \},$$

where $re(r_l) = SE_{r_l}$, $SE_{r_l} \subseteq SE - \phi$, $1 \le l \le t$. The purpose is to find a subset of entities $ESet_l$ for each refactoring $r_l \in SR$, $l = \overline{1,t}$ such that the fitness function is maximized. The solution space may contain items where a specific refactoring applying r_l, $1 \le l \le t$ is not relevant, since objective functions have to be optimized. This means there are subsets $ESet_l = \phi$, $ESet_l \subseteq SE$, $1 \le l \le t$.

4. MOOP Model

MOOP is defined in [17] as the problem of finding a decision vector $\vec{x} = (x_1,...,x_n)$, which optimizes a vector of M objective functions $f_i(\vec{x})$ where $1 \le i \le M$, that are subject to inequality constraints $g_j(x) \ge 0$, $1 \le j \le J$ and equality constraints $h_k(\vec{x}) = 0$, $1 \le k \le K$. A MOOP may be defined as:

$$maximize\{F(\vec{x})\} = maximize\{f_1(\vec{x}), \ldots, f_M(\vec{x})\},$$

with $g_j(\vec{x}) \ge 0$, $1 \le j \le J$ and $h_k(\vec{x}) \ge 0$, $1 \le k \le K$ where $\vec{x}$ is the vector of decision variables and $f_i(\vec{x})$ is the i-th objective function; and $g(x)$ and $h(x)$ are constraint vectors.

There are several ways to deal with a multi-objective optimization problem. In this paper the weighted sum method [10] is used.

Let us consider the objective functions $f_1, f_2, \ldots, f_m$. This method takes each objective function and multiplies it by a fraction of one, the ''weighting coefficient'' which is represented by w_i, $1 \le i \le M$. The modified functions are then added together to obtain a single fitness function, which can easily be solved using any method which can be applied for single objective optimization. Mathematically, the new mapping may be written as:

$$F(\vec{x}) = \sum_{i=1}^{M} w_i \cdot f_i(\vec{x}), \ 0 \le w_i \le 1, \ \sum_{i=1}^{M} w_i = 1.$$

4.1. MOERSSP Formulation

Multi-objective optimization often means compromising conflicting goals. For our MOERSSP formulation there are two objectives taken into consideration in order to maximize refactorings effect upon software entities and minimize required cost for the applied transformations. Current research treats cost as an objective instead of a constraint. Therefore, the first objective function defined below minimizes the total cost for the applied refactorings, as:

$$minimize\left\{ f_1(\vec{r}) \right\} = minimize\left\{ \sum_{l=1}^{t} \sum_{i=1}^{m} rc(r_l, e_i) \right\},$$

where $\vec{r} = (r_1,...,r_t)$. The second objective function maximizes the total effect of applying refactorings upon software entities, considering the weight of the software entities in the overall system, like:

$$maximize\left\{ f_2(\vec{r}) \right\} = maximize\left\{ \sum_{l=1}^{t} res(r_l) \right\},$$

where $\vec{r} = (r_1,...,r_t)$. The goal is to identify those solutions that compromise the refactorings costs and the overall impact on transformed entities. In order to convert the first objective function to a maximization problem in the MOERSSP, the total cost is subtracted from MSX, the biggest possible total cost, as it is shown below:

$$maximize\left\{ f_1(\vec{r}) \right\} = maximize\left\{ MAX - \sum_{l=1}^{t} \sum_{i=1}^{m} rc(r_l, e_i) \right\},$$

where $\overrightarrow{r} = (r_1,...,r_t)$. The final fitness function for MOERSSP is defined by aggregating the two objectives and may be written as:

$$F(\overrightarrow{r}) = \alpha \cdot f_1(\overrightarrow{r}) + (1-\alpha) \cdot f_2(\overrightarrow{r}),$$

where $0 \le \alpha \le 1$.

5. Case Study: LAN Simulation

The algorithm proposed was applied on a simplified version of the Local Area Network (LAN) simulation source code, that was presented in [7]. The example has been successfully used in several programming courses to illustrate and teach good practices in object-oriented design. It covers most of the interesting constructs of the object-oriented programming paradigm, like: inheritance, late binding, super calls, method overriding. It has been implemented in Java and Smalltalk and it follows an incremental development style, including several typical refactorings. Therefore, the example is suitable as a feasability study.

The source code version used here consists of 5 classes: Packet, Node and its three subclasses Workstation, PrintServer and FileServer. The Node objects are linked together in a token ring network, using the nextNode attribute; they can send or accept a Packet object. PrintServer, FileServer and Workstation refine the behaviour of accept (and perform a super call) to achieve specific behaviour for printing the Packet and avoiding its endless cycling. A Packet object can only originate from an Workstation object, and sequentially visits every Node object in the network until it reaches its receiver that accepts the Packet, or until it returns to its sender workstation, indicating that the Packet cannot be delivered. Figure 1 shows the class diagram of the studied source code. It contains 5 classes with 5 attributes and 13 methods, constructors included.

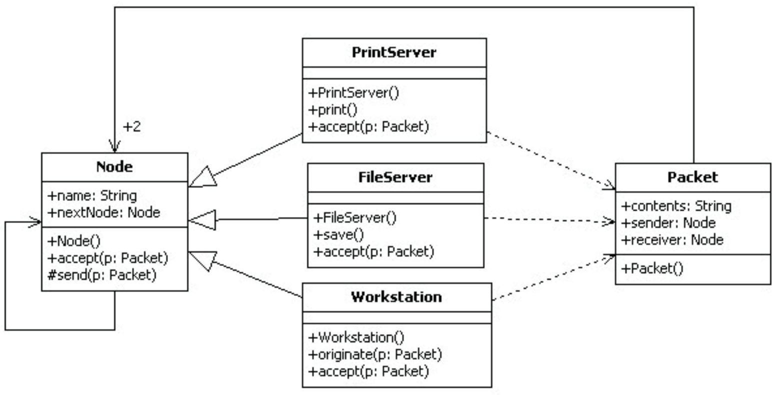


Figure 1. Class diagram for LAN simulation

Thus, for the studied problem the software entity set is defined as: $SE = \{c_1,...,c_5, a_1,...,a_5, m_1,..., m_{13}\}$. The chosen refactorings that may be applied are: *renameMethod, extractSuperClass, pullUpMethod, moveMethod, encapsulateField, addParameter*, denoted by the set $SR = \{r_1,...,r_6\}$ in the following. The dependency relationship between refactorings is defined in what follows: $\{(r_1, r_3) = B, (r_1, r_6) = AA, (r_2, r_3) = B, (r_3, r_1) = A, (r_6, r_1) = AB, (r_3, r_2) = A, (r_1, r_1) = N, (r_2, r_2) = \mathrm{N}, (r_3, r_3) = N, (r_4, r_4) = \mathrm{N}, (r_5, r_5) = \mathrm{N}, (r_6, r_6) = N\}$. For the *res* mapping, values were computed for each refactoring, by using a specified weight for each existing and possible affected software entity, as it was defined in Section 3. The value of the *res* function for each refactoring is: 0.4, 0.49, 0.63, 0.56, 0.8, 0.2.

Here, the cost mapping *rc* is computed as the number of the needed transformations, therefore related entities may have different costs for the same refactoring. Each software entity has a weight within the entire system, but $\sum_{i=1}^{23} w_i = 1$. Due to the space limitation, intermediate data for other mapping (e.g., *effect*) was not included. For *effect* mapping, values were considered to be numerical data, denoting estimated impact of refactoring applying.

6. Proposed Approach Description

The decision vector $\vec{S} = (S_1,...,S_t)$, $S_l \subseteq SE \cup \phi$, $1 \le l \le t$ determines the entities that may be transformed using the proposed refactorings set *SR*. The item S_l on the *l*-th position of the solution vector represents a set of entities that may be refactored by applying the *l*-th refactoring from *SR*, where each entity $e_{l_u} \in SE_{r_l}$, $e_{l_u} \in S_l \subseteq \text{SE} \cup \phi$, $1 \le u \le q, 1 \le q \le m, 1 \le l \le t$. This means it is possible to apply more than once different refactorings to the same software entity, i.e., distinct gene values from the chromosome may contain the same software entity.

A steady-state evolutionary algorithm was applied here, a single individual from the population being changed at a time. The best chromosome (or a few best chromosomes) is copied to the population in the next generation. Elitism can very rapidly increase performance of GA, preventing to lose the best found solution. A variation is to eliminate an equal number of the worst solutions, i.e. for each best chromosome kept within the population a worst chromosome is deleted.

6.1 Genetic Operators

The genetic operators used are *crossover* and *mutation*. Each of them is presented below.

6.1.1 Crossover Operator

A simple one point crossover scheme is used. A crossover point is randomly chosen. All data beyond that point in either parent string is swapped between the two parents.

For example, if the two parents are: $parent_1$ = [*ga*[1,7], *gb*[3,5,10], *gc*[8], *gd*[2,3,6,9,12], *ge*[11], gf[13,4]] and $parent_2$ = [*g*1[4,9,10,12], *g*2[7], *g*3[5,8,11], *g*4[10,11], *g*5[2,3,12], *g*6[5,9]] and the cutting point is 3, the two resulting offsprings are: $offspring_1$ = [*ga*[1, 7], *gb*[3,5,10], *gc*[8], *g*4[10,11], *g*5[2,3,12], *g*6[5,9]] and $offspring_2$ = *g*1[4,9,10,12], *g*2[7], *g*3[5,8,11], *gd*[2,3,6,9,12], *ge*[11], gf[13,4]].

6.1.2. Mutation Operator

Mutation operator used here exchanges the value of a gene with another value from the allowed set. In other words, mutation of *i*-th gene consists of adding or removing a software entity from the set that denotes the *i*-th gene. We have used 1 mutations for each chromosome, number of genes being 6.

For instance, if we have the chromosome *parent* = [[*ga*[1,7], *gb*[3,5,10], *gc*[8], *gd*[2,6,9,12], *ge*[12], gf[13,4]] and we chose to mutate the fifth gene, then a possible offspring may be *parent* = [[*ga*[1,7], *gb*[3,5,10], *gc*[8], *gd*[2,6,9,12], *ge*[10,12], gf[13,4]] by adding the *10*-th software entity to the *5*-th gene.

6.2. Algorithm description

In a steady-state evolutionary algorithm a single individual from the population is changed at a time. The best chromosome (or a few best chromosomes) is copied to the population in the next generation. Elitism can very rapidly increase performance of GA, because it prevents to lose the best found solution to date. A variation is to eliminate an equal number of the worst solutions, i.e. for each *best chromosome* kept within the population a *worst chromosome* is deleted.

The general pseudocode of the evolutionary algorithm used in this paper is given in the Algorithm 1.

```
Algorithm 1: Evolutionary algorithm
Require: SR - set of refactoringsş
SE - set of entitiesş
Ensure: Solution obtained
 1: Population Initialization;
 2: for ( t = 1 to nEvolutionCount) do
 3: for ( k=0 to lstPopulationCount; k += 2)) do
 4: Randomly choose two individuals, p1 and p2;
 5: OneCutPointCrossover for p1 and p2, resulting child c1 and child c2;
 6: Mutation(c1); Mutation(c2);
 7: Memorize in c1 and c2 the best individual from p1 and c1 and from p2 and c2;
 8: Replace the two worst individuals form population with c1 and c2.
 9: end for
10: end for
```

6.3 Data Normalization

Normalization is the procedure used in order to compare data having different domain values. It is necessary to make sure that the data being compared is actually comparable. Normalization will always make data look increasingly similar. An attribute is normalized by scaling its values so they fall within a small-specified range, e.g., 0.0 to 1.0.

As we have stated above we would like to obtain a subset of refactorings to be applied to each software entity from a given set of entities, such that we obtain a minimum cost and a maximum effect. The cost for an applied refactoring to an entity is between 0 and 100. At each step of the selection the *res* function is considered. We must normalize the cost of applying the refactoring, i.e., *rc* mapping, and the value of the *res* function too. Two methods to normalize the data: *decimal scaling* for the *rc* mapping and *min-max normalization* for the value of the *res* function have been used here.

6.3.1 Decimal Scaling

The decimal scaling normalizes by moving the decimal point of values of feature *X*. The number of decimal points moved depends on the maximum absolute value of *X*. A modified value *new_v* corresponding to *v* is obtained using:

$$new\_v = \frac{v}{10^n},$$

where *n* is the smallest integer such that max(|*new_v*|) < 1.

6.3.2 Min-max Normalization

The min-max normalization performs a linear transformation on the original data values. Suppose that *minX* and *maxX* are the minimum and maximum of feature X. We would like to map interval [*minX*, *maxX*] into a new interval [*new_minX*, *new_maxX*]. Consequently, every value *v* from the original interval will be mapped into value *new_v* using the following formula:

$$new\_v = \frac{v - minX}{maxX - minX}.$$

Min-max normalization preserves the relationships among the original data values.

7. Practical Experiments for the Proposed Approach

The algorithm was run 100 times and the best, worse and average fitness values were recorded. The parameters used by the evolutionary approach were as follows: mutation probability 0.7 and crossover probability 0.7. Different number of generations and of individuals were used: number of generations 10, 50, 100, 200 and number of individuals 20, 50, 100, 200. The following subsection reveals the obtained results for the α parameter set to 0.5. The summary subsection shortly reminds all the results for the run experiments, including those for 0.3 and 0.7 values for the α parameter, as presented in [3, 4, 5].

7.1 Equal Weights ($\alpha = 0.5$)

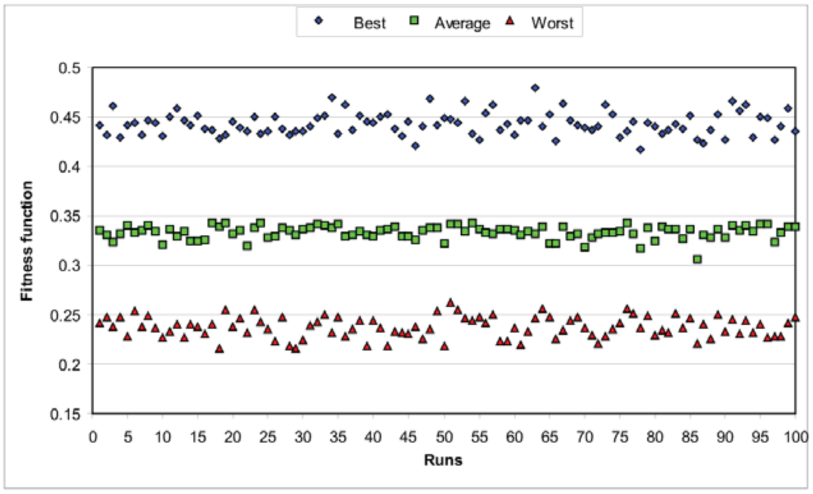
A first experiment run on the *LAN Simulation Problem* proposes equal weights, i.e., $\alpha = 0.5$, for the studied fitness function [3, 5]. That is,

$$F(\vec{r}) = 0.5 \cdot f_1(\vec{r}) + 0.5 \cdot f_2(\vec{r}),$$

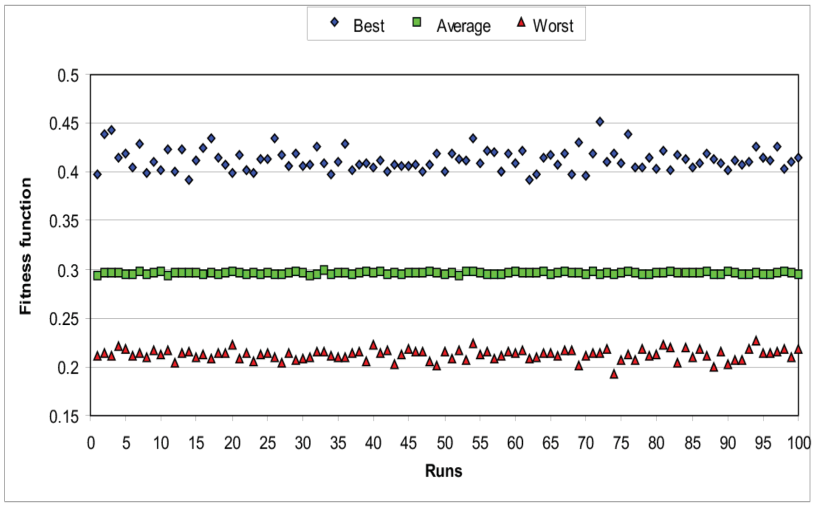
where $\vec{r} = (r_1,...,r_t)$. Figure 2 presents the 200 generations evolution of the fitness function (best, worse and average) for 20 chromosomes populations (Figure 1) and 200 chromosomes populations (Figure 1).

There is a strong competition among chromosomes in order to breed the best individual. In the 20 individuals populations the competition results in different quality of the best individuals for various runs, from very weak to very good solutions. The 20 individuals populations runs have few very weak solutions, better than 0.25, while all the best chromosomes are good solutions, i.e., all 100 best individuals for the 100 runs have fitness better than 0.41.

Compared to the former populations, the 200 chromosomes populations breed closer best individuals. The number of good chromosomes is smaller than the one for 20 individuals populations, i.e., 53 chromosome with fitness better than 0.41 only.



(a) [Experiment with 200 generations and 20 individuals]



(b) [Experiment with 200 generations and 200 individuals]

Figure 2. The evolution of fitness function (best, worse and average) for 20 and 200 individuals with 200 generations, with eleven mutated genes, for $\alpha = 0.5$

The data for the worst chromosomes reveals similar results, since for the 200 individuals populations there is no chromosome with fitness better than 0.25, while for the 20 chromosomes populations there are 12 worst individuals better than 0.25. This situation outlines an intense activity in smaller populations, compared to larger ones, where diversity among individuals reduces the population capability to quickly breed better solutions.

Various runs as number of generations, i.e., 10, 50, 100 and 200 generations, show the improvement of the best chromosome. For the recorded experiments, the best individual for 200 generations was better for 20 chromosomes populations (with a fitness value of 0.4793) than the 200 individuals populations (with a fitness value of just 0.4515). This means in small populations (with fewer individuals) the reduced diversity among chromosomes may induce a stronger competition than in large populations (with many chromosomes) where the diversity breeds weaker individuals. As the Figure 2 shows its, after several generations smaller populations produce better individuals (as number and quality) than larger ones, due to the poor populations diversity itself.

The number of chromosomes with fitness value better than 0.41 for the studied populations and generations is captured by Figure 3.

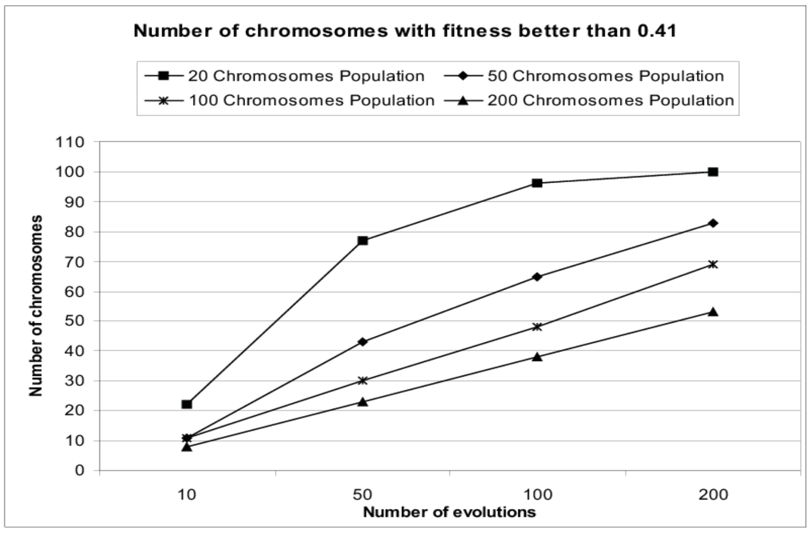


Figure 3. The evolution of the number of chromosomes with fitness better than 0.41 for the 20, 50, 100 and 200 individual populations, with $\alpha = 0.5$

Impact on the *LAN Simulation Problem* source code

The best individual obtained allows to improve the structure of the class hierarchy. Therefore, a new Server class is the base class for PrintServer class. More, the signatures of the print method from the PrintServer class is changed, though the renaming to process was not suggested. Opposite to this, for the save method in the FileServer class was recommended to rename the method to process and the changing signature was not suggested yet. The two refactorings (*addParameter* and *renameMethod*) applied to the print and save methods would had been ensured their polymorphic behaviour.

The accept method is moved to the new Server class for the FileServer class, though the former was not suggested to be added as a base class for the latter. The correct access to the class fields by encapsulating them within their classes is enabled for three of five class attributes. Figure 4 presents the class diagram for the *LAN Simulation Problem* after the obtained solution is applied to.

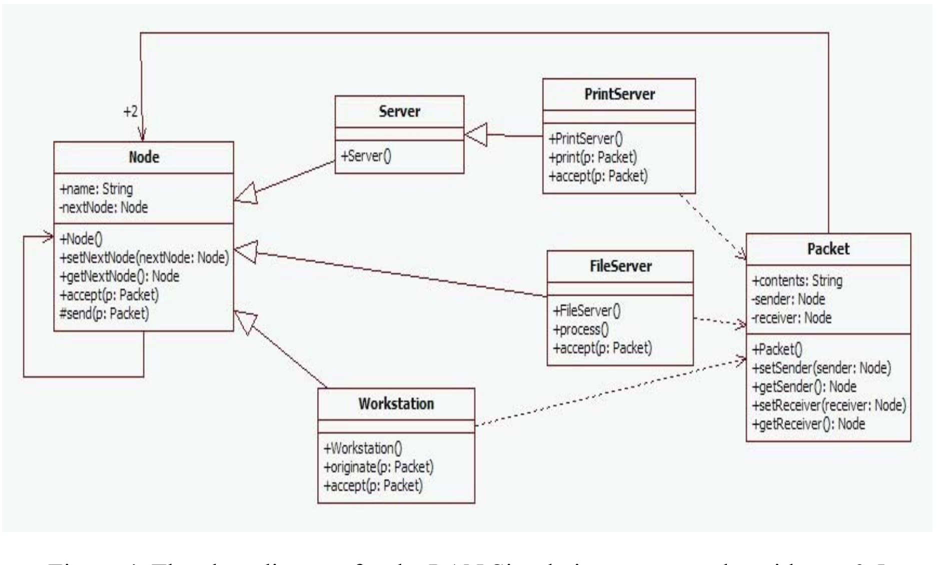


Figure 4. The class diagram for the LAN Simulation source code, with $\alpha = 0.5$

7.2. Summary of the Experiment Results

Current subsection summarizes the results of the proposed algorithm in Section 6.2 for three different value for the α parameter, i.e., 0.3, 0.5, 0.7, in order to maximize the weighted sum fitness function that optimizes the refactoring cost and the refactoring impact on affected software entities [5]. A chromosome summary of the obtained results for all run experiments as it is presented in [3, 4, 5] is given below:

- α = 0.3, *bestFitness* = 0.33587 for 20 *chromosomes and* 200 *generations*

  - *bestChrom* = [[10, 22, 21, 19, 15], [3, 2], [21, 19, 10, 16, 17, 13, 11, 14, 12], [19, 10, 22, 11, 13, 16], [∅], [21, 22]]

- α = 0.5, *bestFitness* = 0.4793 for 20 *chromosomes and* 200 *generations*

  - *bestChrom* = [[20, 13, 19, 11], [1, 2], [15, 10, 20, 17, 19, 13, 12], [12, 11, 15, 14, 21], [6, 8, 9], [22, 12, 18, 17, 13, 14, 15]]

- α = 0.7, *bestFitness* = 0.61719 for 20 *chromosomes and* 200 *generations*

  - *bestChrom* = [[20, 16], [3], [15, 18, 14, 21, 16, 13, 22, 10], [20, 10, 22, 16, 17], [∅], [16, 10, 11]]

The experiment for α = 0.3 should identify those refactorings for which the cost has a lower relevance than the overall impact on the applied software entities. But, the obtained best chromosome obtained has the fitness value 0.33587, lower than the best fitness value for the α = 0.5 chromosome, i.e.,0.4793. This shows that an unbalanced agreggated fitness function with a higher weight for the overall impact on the applied refactorings, promotes the individuals with more low cost and small refactorings. Therefore, there are not too many key software entities to be refactored by a such an experiment.

The α = 0.7 experiment should identify the refactorings for which the cost is more important than the final effect of the applied refactorings. The fitness value for the best chromosome for this experiment is 0.61719, while for the α = 0.5 experiment the best fitness value is lower than this one.

The experiment for α = 0.7 gets near to the α = 0.5 experiment. The data shows similarities for the structure of the obtained best chromosomes for the two experiments. A major difference is represented by the encapsulatedField refactoring that may be applied to the public class attributes from the class hierarchy. This refactoring was not suggested by the solution proposed by the α = 0.7 experiment. Moreover, there is a missing link in the same experiment, due to the fact the addParameter refactoring was not recommended for save method from FileServer and print method from PrintServer class.

Balancing the fitness values for the studied experiments and the relevance of the suggested solutions, we consider the α = 0.5 experiment is more relevant as quality of the results than the other analyzed experiments. Figure 4 (see Section 7.1) highlights the changes in the class hierarchy for the α = 0.5 following the suggested refactorings from the recorded best chromosome.

7.3 Obtained Results by Another Solution Representation

For the problem presented in Section 3 the aspect of the most appropriate refactoring for an entity is studied in [2, 6]. The decision vector $\overrightarrow{r} = (r_1,\ldots,r_m)$, $r_i \in SR$, $1 \le i \le m$ determines such refactorings that may by applied in order to transform the considered set of software entities *SE*. The item r_i on the *i*-th position of the solution vector represents the refactoring that may be applied to the *i*-th software entity from *SE*, where $e_i \in SE_{r_i}$, $1 \le i \le m$.

Therefore, a chromosome is represented as a string of size equal to the number of entities from *SE*. The value of the *i*-th gene represents the refactoring that may be applied to the *i*-th entity. The values of these genes are not different from each other, i.e, the same refactoring may be applied to multiple entities.

The results of the proposed approach in [2] for three different value for the α parameter, i.e., 0.3, 0.5, 0.7, are summarized and discussed by the current section [6]. A best chromosome list of the obtained results for all experiments is given below:

- α = 0.3, *bestFitness* = 0.25272 for 100 *chromosomes and* 200 *generations*

  - *bestChrom* = [1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 2, 2, 3, 3, 3, 2, 3, 2, 0, 2, 3]

- α = 0.5, *bestFitness* = 0.3562 for 20 *chromosomes and* 200 *generations*

  - *bestChrom* = [1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2, 3, 2, 5, 2, 2, 0, 3, 2]

- α = 0.7, *bestFitness* = 0.45757 for 50 *chromosomes and* 50 *generations*

  - *bestChrom* = [1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 3, 2, 3, 2, 2, 3, 3, 5, 2, 2, 3, 2, 3]

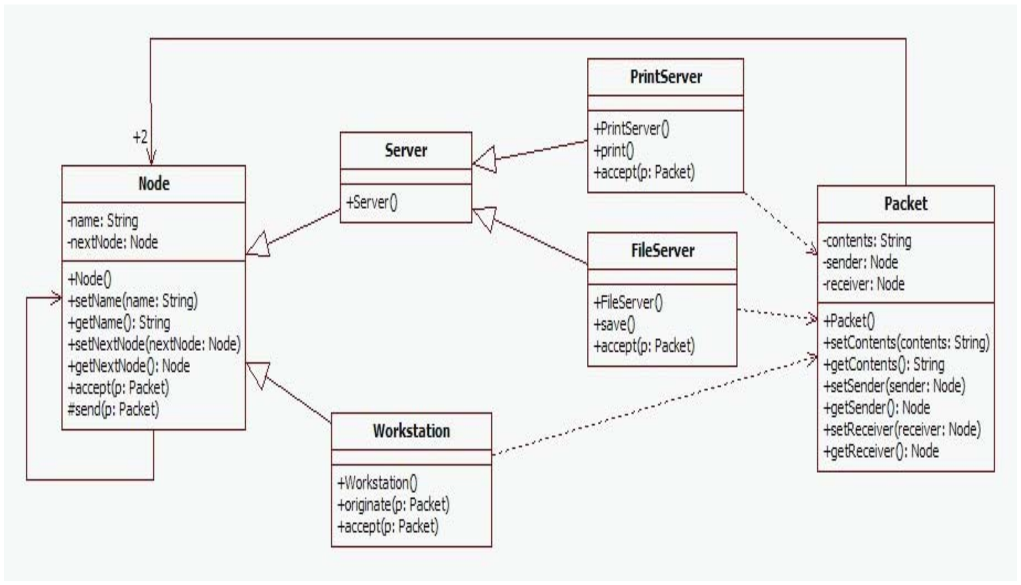


Figure 5. The class diagram for the LAN Simulation source code, with $\alpha = 0.5$

The data shows similar results for the structure of the best chromosome. A major difference is represented by the possible refactoring that may be applied to the save method from FileServer and accept method from PrintServer and FileServer classes. The suggested solutions by $\alpha = 0.3$ and $\alpha = 0.5$ experiments recommend a second refactoring that may be applied to the save method, i.e., the *renameMethod* refactoring, while for $\alpha = 0.7$ the suggested refactoring is not appropriate, i.e., the moveMethod refactoring. Figure 5 highlights the changes in the class hierarchy for the $\alpha = 0.5$.

The experiments for 20 chromosomes populations have good results in each of the three runs with different values for the α parameter, bringing a better solution quality for the eligible individuals.

This means in small populations (with few individuals) the reduced diversity among chromosomes may induce a harsher competition compared to large populations (with many chromosomes) where the diversity breeds weaker and closer individuals as fitness quality. As the run experiments revealed it, after several generations smaller populations produce better individuals (as number and quality) than larger ones, due to the poor populations diversity itself.

7.4 Discussion

The Multi-Objective Refactoring Single Selection Problem (MPRSgSP) shortly reminded in Section 7.3 represents a special case of the MOERSSP. The former one identifies a single refactoring that changes a software entity that satisfies the established objectives in the most appropriate way, while the latter identifies a set of possible refactorings for each software entity.

The best individual obtained for the run experiments of the MORSgSP, i.e., a 20 chromosomes population with 200 generations evolution, was transposed to the refactoring-based solution representation of the MOERSSP. The resulted individual has the same fitness as in the original form, i.e., 0.3562.

The best chromosome recorded for the MOERSSP experiments, is obtained for a 20 chromosomes population with 200 generations evolution too. But, it cannot be transposed to the solution representation presented in Section 7.3, since there are several refactorings suggested for each entity.

First, a refactoring may be applied to more than one software entity, as the r_6 (*addParameter* refactoring) which is applied to the m_8 (*print* method) from c_3 (PrintServer class) and m_{11} (*save* method) from c_4 (FileServer class). Second, r_1 (*renameMethod* refactoring) is then applied for the same methods in order to highlight the polymorphic behaviour of the new renamed method process. This means there are at least two refactorings that may be applied to the methods referred here (print and save). Thus, the multiple transformations of software entities cannot be coded by the solution representation proposed by Section 7.3.

Compared to Figure 4, Figure 5 shows that the Section 7.3 allows information hiding by suggesting refactorings for field encapsulation. But the solution representation does not allow to apply more than one refactoring to each software entity. This results in the lack of possibility to apply some relevant refactorings to software entities.

Figure 4 presents the transformed class diagram based on the solution proposed by the run experiments on MOERSSP. Although, the fitness value of the best chromosome (0.4793) is better than the value of the approach discussed in Section 7.3, it suggests the possibility to apply more than one refactoring to a single software entity. While Figure 5 highlight that all 5 class attributes from the class diagram are hidden within their classes, Figure 4 encapsulates 3 class fields only, though there are other relevant refactorings that are applied in order to improve the internal structure of the studied source code.

7.5 Obtained Results by Others

Fatiregun et al. [8] applied genetic algorithms to identify the transformation sequences for a simple source code, with 5 transformation array, whilst we have applied 6 distinct refactorings to 23 entities. Seng et al. [13] applied a weighted multi-objective search, in which metrics were combined into a single objective function. An heterogeneous weighed approach was applied in our approach, because of the weight of software entities in the overall system and refactorings cost being applied. Mens et al. [12] propose techniques to detect the implicit dependencies between refactorings. Their analysis helped to identify which refactorings are most suitable to LAN simulation case study. Our approach considers all relevant applying of the studied refactorings to all entities. Bowman et al. [1] discuss the class responsibility assignment using a multi-objective optimization approach. The goal is to optimize the coupling and cohesion of a given class diagram based on five distinct measures [1]. Although a single objective optimization problem suggests a unique optimal solution, the approach proposed by the authors offers a large set of solutions that, when evaluated, produces vectors whose components represent tradeoffs in the objective space. Similar to this we focus on two objectives. We study the final effect of the applied refactorings but we pay attention to the involved costs too. In this way we try to obtain a solution that is acceptable, with a positive impact on the internal structure of the source code and with low costs too.

8. Conclusions and Future work

The paper discusses a new version of the MORSgSP presented in [2]. The results of a proposed weighted objective genetic algorithm on the same experimental didactic case study are presented and compared with other previous results. Furthermore, another solution representation of the chromosome for the same problem is compared with the approach proposed by the current paper.

The weighted multi-objective optimization is discussed here, but the Pareto approach may prove to be more suitable when it is difficult to combine fitness functions into a single overall objective function. Thus, a further step would be to apply the Pareto front approach in order to prove or deny the superiority of the second possibility. Here, the cost is described as an objective, but it can be interpreted as a constraint, with the further consequences.

References

[1] Bowman, M., Briand, L.C., Labiche, Y (2007). Multi-Objective Genetic Algorithm to Support Class Responsibility Assignment, *In*: Proceedings of the IEEE International Conference on Software Maintenance (ICSM1007), IEEE, October 2-5. Paris, France, p.124–133.

[2] Chisăliță–Crețu, C., Vescan, A. (2009). The Multi-objective Refactoring Selection Problem, *In*: Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques (KEPT2009), Cluj-Napoca, Romania, July 24. Studia Universitatis Babes-Bolyai, Series Informatica. p. 249-253.

[3] Chisăliță-Crețu, C. (2009). The entity refactoring set selection problem - practical experiments for an evolutionary approach, *In*: Proceedings of the World Congress on Engineering and Computer Science (WCECS2009), October 20-22. San Francisco, USA, p. 285–290. Newswood Limited.

[4] Chisăliță-Crețu, C. (2009). First results of an evolutionary approach for the entity refactoring set selection problem, *In*: Proceedings of the 4th International Conference "Interdisciplinarity in Engineering" (INTER-ENG 2009), November 12-13. Târgu Mureş, România, p. 303–308. Editura Universității Petru Maior din Târgu Mureş.

[5] Chisăliță-Crețu, C. (2009). A multi-objective approach for entity refactoring set selection problem, *In*: Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2009), August 4-6. London, UK, p. 790–795.

[6] Chisăliță-Crețu, C. (2010). The Optimal Refactoring Selection Problem - A Multi-Objective Evolutionary Approach, *In*: The 5th International Conference on Virtual Learning (ICVL2010), October 29-31. Târgu Mureş, România, submitted paper.

[7] Demeyer, S., Janssens, D., Mens, T. (2002). Simulation of a LAN, Electronic Notes in Theoretical Computer Science, 72. p. 34-56.

[8] Fatiregun, D., Harman, M., Hierons, R. (2004). Evolving transformation sequences using genetic algorithms, *In*: 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04), Los Alamitos, California, USA, IEEE Computer Society Press, p. 65-74.
[9] Fowler, M. (1999). Refactoring: Improving the Design of Existing Software. Addison Wesley.
[10] Kim, Y., deWeck, O.L (2005). Adaptive weighted-sum method for bi-objective optimization: Pareto front generation, in Structural and Multidisciplinary Optimization, MIT Strategic Engineering Publications, 29 (2) p. 149-158.
[11] Marinescu, R.(1998). Using object-oriented metrics for automatic design flaws in large scale systems, *In*: Object-Oriented Technology (ECOOP98 Workshop Reader), S. Demeyer, J. Bosch, eds., Lecture Notes in Computer Science, Springer-Verlag, 1543. p. 252-253.
[12] Mens, T., Taentzer, G., Runge, O. (2007). Analysing refactoring dependencies using graph transformation, *Software and System Modeling*, 6 (3) 269-285.
[13] Seng, O., Stammel, J., Burkhart, D (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems, *In*: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, M. Keijzer, M. Cattolico, eds., V. 2, ACM Press, Seattle, Washington, USA. p. 1909-1916.
[14] Simon, F., Steinbruckner, F. Lewerentz, C.(2001). Metrics based refactoring, *In*: Proceeding of European Conf. Software Maintenance and Reengineering, IEEE Computer Society Pres, 2001, pp. 30-38.
[15] Tourwe, T., Mens, T (2003). Identifying refactoring opportunities using logic meta programming, *In*: Procedeengs of 7th European Conference on Software Maintenance and Re-engineering (CSMR2003), IEEE Computer Society Press, p. 91-100.
[16] van Emden, E., Moonen, L. (2002). Java quality assurance by detecting code smells, *In*: Procedeeing of 9th Working Conference on Reverse Engineering, IEEE Computer Society Press. p. 97-107.
[17] Zitzler, E., Laumanss, M., Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm, Computer Engineering and Networks Laboratory, Technical Report, 103. p. 5-30.

Camelia Chisăliţă-Creţu is an Assistant Professor in Computer Science Department, Faculty of Mathematics and Computer Science, Babe S-Bolyai University in Romania. She obtained a M.Sc. degree in Component-Based Programming and she is currently pursuing the Ph.D. degree in Software Engineering at Babe S-Bolyai University of Cluj-Napoca, Romania. Her research domains include object-oriented programming, software engineering, refactoring, search-based engineering and formal methods, with recent focus on identifying appropriate refactorings for multi-objective transformation problems, refactoring assessment using new defined software metrics, multi-refactoring strategies development and integration. She has published many papers in various international conferences and journals. She is a member of IEEE, International Association of Engineers (IAEng) and International Association of Computer Science and Information Technology (IACSIT). Her e-mail address is cretu@cs.ubbcluj.ro.