A Multi-Objective Approach for Entity Refactoring Set Selection Problem

Camelia Chisăliţă–Creţu
Faculty of Mathematics and Computer Science
Babeş-Bolyai University
1, M. Kogalniceanu 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 approach to the Entity Refactoring Set Selection Problem (ERSSP) by treating the cost constraint as an objective and combining it with the effect objective. The results of the proposed weighted objective genetic algorithm on a experimental didactic case study are presented and compared with other previous results.

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 [4]. 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 [4]. The ERSSP is the identification problem of the optimal 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 (MOERSSP) and performs a proposed weighted objective genetic algorithm on an experimental didactic case study. Obtained results for our case study are presented and compared with other recommended solutions for similar problems [6].

The rest of the paper is organized as follows: Section 2 gives the definition of the Multi-Objective Optimization Problem (MOOP), while Section 3 presents the formal definition of the studied problem. A short description of the Local Area Network simulation source code used to validate our approach is provided in Section 4. The proposed approach and several details related to the genetic operators of the genetic algorithm are described in Section 5. The obtained results for the studied source code and for similar problems are presented and compared in Section 6. The paper ends with conclusions and future work.

2. MOOP Model

MOOP is defined in [8] as the problem of finding a decision vector $\vec{x} = (x_1, \ldots, x_n)$, which optimizes a vector of M objective functions $f_i(\vec{x})$ where $1 \leq i \leq M$, that are subject to inequality constraints $g_j(\vec{x}) \geq 0$, $1 \leq j \leq J$ and equality constraints $h_k(\vec{x}) = 0$, $1 \leq k \leq K$. A MOOP may be defined as:

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

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

There are several ways to deal with a multi-objective optimization problem. In this paper the weighted sum method [5] 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 $w_i, 1 \leq i \leq M$. 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:

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

3. ERSSP Definition

In order to state the ERSSP some notion and characteristics have to be defined. Let $SE = \{e_1, \ldots, e_m\}$ 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

978-1-4244-4457-1/09/$25.00 ©2009 IEEE

method or a local variable declared in the implementation of a method. They are considered to be low level components bounded thought dependency relations. The weight associated with each software entity $e_i, 1 \leq i \leq m$ is kept by the set $Weight = \{w_1, \ldots, w_m\}$, 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 [4] that may be applied to different types of software entities of SE is gathered up through $SR = \{r_1, \ldots, r_t\}$. There are various dependencies between such transformations when they are applied to the same software entity, a mapping emphasizing them being defined by:
$rd : SR \times SR \times SE \rightarrow \{\mathbf{B}efore, \mathbf{A}fter, \mathbf{A}lways\mathbf{B}efore, \mathbf{A}lways\mathbf{A}fter, \mathbf{N}ever, \mathbf{W}henever\}$,

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

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

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

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

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

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:
$res : SR \rightarrow Z$,

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

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

$$re(r_l) = \left\{ e_{l_1}, \ldots, e_{l_q} \quad | \text{ if } r_l \text{ is applicable to } e_{l_u}, 1 \leq u \leq q, 1 \leq q \leq m \right\},$$

where $re(r_l) = SE_{r_l}, SE_{r_l} \subseteq SE - \phi, 1 \leq l \leq t$.

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 $r_l, 1 \leq l \leq t$ is not relevant, since objective functions have to be optimized. This means there are subsets $ESet_l = \phi, ESet_l \subseteq SE, 1 \leq l \leq t$.

3.1. MOERSSP Formulation

Multi-objective optimization often means to compromise 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:

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

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

$$maximize\left\{f_2(\vec{r})\right\} = \sum_{l=1}^{t} res(r_l) = \sum_{l=1}^{t}\sum_{i=1}^{m} w_i * effect(r_l, e_i),$$

where $\vec{r} = (r_1, \ldots, r_t)$.

The goal is to select a subset of entities for each proposed refactoring that results in the minimum total cost and the maximum effect upon affected software entities. In order to convert the first objective function to a maximization problem in the MOERSSP, the total cost is subtracted from MAX, the biggest possible total cost, as it is shown below:

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

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

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

where $0 \leq \alpha \leq 1$.

4. 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 [2]. Figure 1 shows the class diagram of the studied source code. It contains 5 classes with 5 attributes and 13 methods, constructors included.

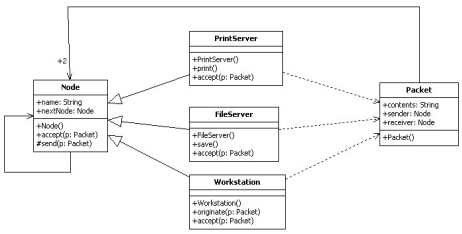


Figure 1. Class diagram for LAN simulation

Thus, for the studied problem the software entity set is defined as: $SE = \{c_1, \ldots, c_5, a_1, \ldots, a_5, m_1, \ldots, m_{13}\}$. The chosen refactorings that may be applied are: *renameMethod,*

extractSuperClass, pullUpMethod, moveMethod, encapsulateField, addParameter, denoted by the set $SR = \{r_1, \ldots, r_6\}$ in the following. The dependency relationship between refactorings is defined in what follows: $\{(r_1, r_3) = B,\ (r_1, r_6) = AA,\ (r_2, r_3) = B,\ (r_3, r_1) = A,\ (r_6, r_1) = AB,\ (r_3, r_2) = A,\ (r_1, r_1) = N,\ (r_2, r_2) = N,\ (r_3, r_3) = N,\ (r_4, r_4) = N,\ (r_5, r_5) = N,\ (r_6, r_6) = N\}$. 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.

5. Proposed Approach Description

The decision vector $\vec{S} = (S_1, \ldots, S_t), S_l \subseteq SE \cup \phi, 1 \leq l \leq t$ 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 $e_{l_u} \in SE_{r_l}, e_{l_u} \in S_l \subseteq SE \cup \phi, 1 \leq u \leq q, 1 \leq q \leq m, 1 \leq l \leq t$. 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.

5.1. Genetic Operators

The parameters used by the evolutionary approach are as follows: mutation probability 0.7 and crossover probability 0.7. Different number of generations and of individuals are used: number of generations 10, 50, 100, 200 and number of individuals 20, 50, 100, 200. The value of α used while aggregating the objectives was set to 0.5 which gives the same importance to both objectives.

5.1.1 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 $parent_2 = [g1[4,9,10,12],\ g2[7],\ g3[5,8,11],\ g4[10,11],\ g5[2,3,12],\ g6[5,9]]$ and the cutting point is 3, the two resulting offsprings are: $offspring_1 = [ga[1,7],\ gb[3,5,10],\ gc[8],\ g4[10,11],\ g5[2,3,12],\ g6[5,9]]$ and $offspring_2 = [g1[4,9,10,12],\ g2[7],\ g3[5,8,11],\ gd[2,3,6,9,12],\ ge[11],\ gf[13,4]]$.

5.1.2 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 11 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 $parent = [ga[1,7]\ gb[3,5,10]\ gc[8]\ gd[2,6,9,12]\ ge[\mathbf{10},12]\ gf[13,4]]$ by adding the *10*-th software entity to the *5*-th gene.

5.2. 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.

5.2.1 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 $max(|new\_v|) < 1$.

5.2.2 Min-max Normalization

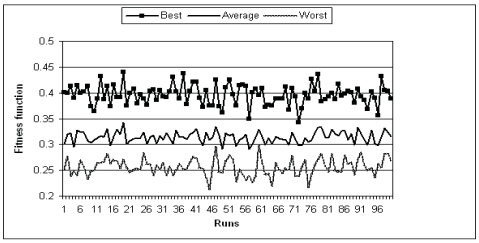
The min-max normalization performs a linear transformation on the original data values. Suppose that $minX$ and $maxX$ are the minimum and maximum of feature X. We would like to map interval $[minX, maxX]$ into a new interval $[new\_minX, new\_maxX]$. Consequently, every value v from the original interval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

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

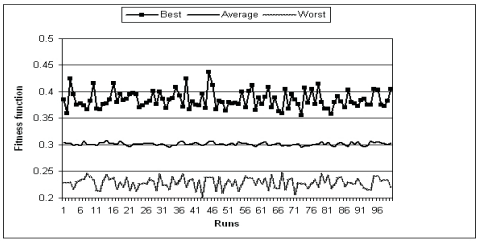
Min-max normalization preserves the relationships among the original data values.

6. Obtained Results by the Evolutionary Approach

The algorithm was run 100 times and the best, worse and average fitness values were recorded. Figure 2 presents the 10 generations evolution of the fitness function (best, worse and average) for 20 chromosomes populations (Figure 2(a)) and 200 chromosomes populations (Figure 2(b)).



(a) Experiment with 10 generations and 20 individuals with eleven mutated genes



(b) Experiment with 10 generations and 200 individuals

Figure 2. The evolution of fitness function (best, worse and average) for 20 and 200 individuals with 10 generations

It is easy to see that there is a strong struggle between 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 a few very weak solutions, worse than 0.35, but there are a lot of good solutions, i.e., 22 chromosomes with fitness better than 0.41. Compared to the former populations, the 200 chromosomes populations breed closer best individuals, since there is no chromosome with fitness value worse than 0.35, but the number of good chromosomes is smaller than the one for 20 individuals populations, i.e., 8 chromosomes with fitness better than 0.41 only. 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 is a large number of worse 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.

The number of chromosomes with fitness value better than 0.41 for the studied populations and generations is captured by Figure 3. It shows that smaller populations with poor diversity among chromosomes have a harder competition within them and more, the number of eligible chromosomes increases quicker for smaller populations than for the larger ones. Therefore, for the 20 chromosomes populations with 200 generations evolution all 100 runs have shown that the best individuals are better than 0.41, while for 200 individuals populations with 200 generations the number of best chromosomes better than 0.41 is only 53.

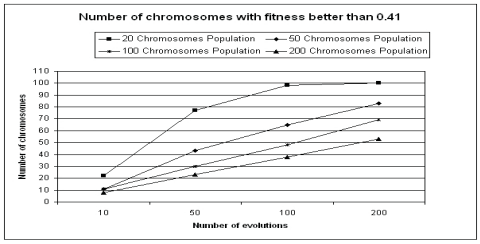


Figure 3. The evolution of the number of chromosomes with fitness better than 0.41 for the 20, 50, 100 and 200 individual populations

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). Various runs as number of generations, i.e., 10, 50, 100 and 200 generations, show the improvement of the best chromosome.

Thus, the best individual fitness value for 10 generations

is 0.43965 for 20 individuals populations and 0.43755 for 200 chromosomes populations. This means in small populations (with few individuals) the reduced diversity among chromosomes may induce a harsher struggle compared to large populations (with many chromosomes) where the diversity breeds weaker individuals. As it was said before, after several generations smaller populations produce better individuals (as number and quality) than larger ones, due to the poor populations diversity itself.

The best individual obtained allows to improve the structure of the class hierarchy. Therefore, a new `Server` class is the base class for `PrintServer` and `FileServer` classes. More, the signatures of the `print` method from the `PrintServer` class and the `save` method from the `FileServer` class are changed and then both renamed to `process`. The `accept` method is pulled up to the new `Server` class. The two refactorings applied to the `print` and `save` methods ensure their polymorphic behaviour. The correct access to the class fields by encapsulating them within their classes is enabled. The current solution representation allows to apply more than one refactoring to each software entity, i.e., method `print` from `PrintServer` class is transformed by two refactorings, *addParameter* and *renameMethod*.

6.1. 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 [1]. The decision vector $\vec{r} = (r_1, \ldots, r_m), r_i \in SR, 1 \leq i \leq m$ 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 $e_i \in SE_{r_i}, 1 \leq i \leq m$.

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.

While compared with the previous experiments it was noted that there was a lower number of different solutions while cumulating the results obtained in all the 100 runs, but the quality (and the number) of these solutions was improved much more. In a first experiment the greatest value of the fitness function is 0.3508 (with 69 individuals with the fitness better than 0.33) while in a second experiment this is not more than 0.3536 (55 individuals with the fitness better than 0.33). But the best chromosome was found in the experiment with 200 generations and 20 individuals having the value 0.3562.

The best individual obtained by the solution representation allowed to improve the structure of the class hierarchy too. Thus, for `PrintServer` and `FileServer` classes, a new base class `Server` is added. More, the signature of the `print` method from the `PrintServer` class is changed in order to allow the `accept` method to be pulled up to the new base class. The `save` method signature from the `FileServer` class should be changed and then renamed too. Next, the `accept` method should be pulled up to the `Server` class. But the studied best individual genes does not suggest the mentioned refactorings. The correct access to the class fields is accomplished by encapsulating them within their classes. Solution representation does not allow to apply more than one refactoring to each software entity. The best chromosome contains several refactorings that are not relevant for the associated software entities, though they improve the fitness value.

6.2. Obtained Results by Others

Fatiregun et al. [3] 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. [7] 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. [6] 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.

6.3. Discussion

The problem discussed in Section 6.1 represents a special case of the problem defined by the current paper. The former one identifies a single refactoring that changes a software entity that satisfies the established objectives in the most appropriate way. In order to improve the internal structure, the approach proposed here identifies a set of possible refactorings for each software entity. The best individual obtained for a 20 chromosomes population with 200 generations by the approach proposed in [1] was transposed to the current solution representation. Its fitness has the same value as in the original form, i.e., 0.3562. The best chromosome for the problem discussed here was obtained for a 20 chromosomes population with 200 generations too. But, it cannot be transposed to the solution representation presented in [1], since there are several refactorings suggested for each entity. Therefore, the

refactoring r_6 (*addParameter*) is applied for the methods m_8 (*print*) from c_3 (`PrintServer` class) and m_{11} (*save*) from c_4 (`FileServer` class). More, the refactoring r_1 (*renameMethod*) 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 in [1] without a rigorous analysis of the obtained best individual. Compared to Figure 1, Figure 4 shows that the solution proposed in [1] 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 that some relevant refactorings are applied to software entities, while others may be unfit, though they may improve the fitness value.

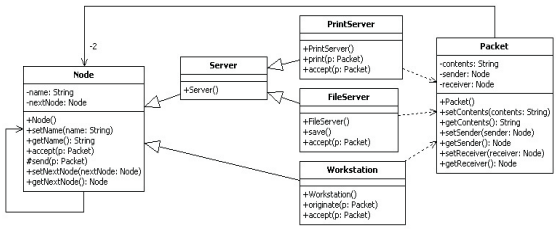


Figure 4. Class diagram for LAN simulation *after* applying the solution proposed in [1]

Figure 5 presents the result of the solution representation proposed by the current approach. Although, the fitness value of the best chromosome (0.4793) is better than the value of the approach discussed in [1], it suggests the possibility to apply more than one refactoring to each software entity. While Figure 4 highlight that all 5 class attributes from the class diagram are hidden within their classes, Figure 5 encapsulates 3 class fields only, though there are other relevant refactorings that are applied in order to improve the internal structure of the source code.

7. Conclusions and Future work

The paper discusses a new version of the MOORSP presented in [1]. 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.

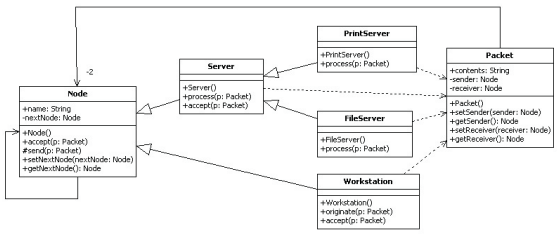


Figure 5. Class diagram for LAN simulation *after* applying the solution proposed by the current approach

References

[1] C. Chisăliţă–Creţu, A. Vescan, *The Multi-objective Refactoring Selection Problem*, in Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques (KEPT2009), Cluj-Napoca, Romania, July 24, 2009, *accepted paper*.

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

[3] D. Fatiregun, M. Harman, R. Hierons, *Evolving transformation sequences using genetic algorithms*, in 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04), Los Alamitos, California, USA, IEEE Computer Society Press, 2004, pp. 65-74.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Software*. Addison Wesley, 1999.

[5] Y. Kim, O.L. deWeck, *Adaptive weighted-sum method for bi-objective optimization: Pareto front generation*, in Structural and Multidisciplinary Optimization, MIT Strategic Engineering Publications, 29(2), 2005, pp. 149-158.

[6] T. Mens, G. Taentzer, O. Runge, *Analysing refactoring dependencies using graph transformation*, Software and System Modeling, 6(3), 2007, pp. 269-285.

[7] O. Seng, J. Stammel, D. Burkhart, *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., vol. 2, ACM Press, Seattle, Washington, USA, 2006, pp. 1909-1916.

[8] E. Zitzler, M. Laumanss, L. Thiele, *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*, Computer Engineering and Networks Laboratory, Technical Report, 103(2001), pp. 5-30.