The Entity Refactoring Set Selection Problem - Practical Experiments for an Evolutionary Approach

Camelia Chisăliţă–Creţu *

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.**

Keywords: refactoring, object-oriented programming, multi-objective optimization

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 presents the formal definition of the studied problem, while Section 3 gives the definition of the Multi-Objective Optimization Problem (MOOP). A short description of the Local Area Network simulation source code used to validate our approach is provided in Section **??**. 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.

*Manuscript submission date: 26 July, 2009, Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania, 1, M. Kogalniceanu Street, RO-400084, Tel: 40-264-405.300/5240 Email: cretu@cs.ubbcluj.ro

2 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 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:

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

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

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 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:

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

3.1 MOORSP Formulation

Multi-objective optimization often means compromising conflicting goals. For our MOORSP 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 $\vec{r} = (r_1, \ldots, r_t)$. 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 $\vec{r} = (r_1, \ldots, r_t)$. 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 MOORSP, the total cost is subtracted from MAX, 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 $\vec{r} = (r_1, \ldots, r_t)$. The final fitness function for MOORSP 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.

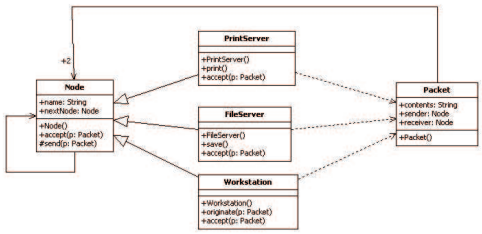


Figure 1: Class diagram for LAN simulation

The current version of the source code lacks of hiding information for attributes since they are directly accessed by clients. The abstraction level and clarity may be increased by creating a new superclass for `PrintServer`

and `FileServer` classes, and populate it by moving up methods in the class hierarchy.

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 $SR = \{r_1, \ldots, r_6\}$ in the following. The dependency relationship between refactorings is defined as follows: $\{(r_1, r_3) = B,\ (r_1, r_6) = AA,\ (r_2, r_3) = B,\ (r_3, r_1) = A,\ (r_6, r_1) = AB,\ (r_3, r_2) = A,\ (r_1, r_1) = N,\ (r_2, r_2) = N,\ (r_3, r_3) = N,\ (r_4, r_4) = N,\ (r_5, r_5) = N,\ (r_6, r_6) = N\}$.

The values of the final effect were computed for each refactoring, but using the weight for each existing and possible affected software entity, as it was defined in Section **??**. Therefore, the values of the res function for each refactoring are: 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, so related entities may have different costs for the same refactoring. Each software entity has a weight within the entire system, but $\sum_{i=1}^{23} w_i = 1$. For the $effect$ mapping, values were considered to be numerical data, denoting the estimated impact of applying a refactoring. Due to the space limitation, intermediate data for these mappings was not included. An acceptable solution denotes lower costs and higher effects on transformed entities both objectives being satisfied.

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.

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.

5.1 Genetic Operators

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 $offspring_2 = [g1[4,9,10,12],\ g2[7],\ g3[5,8,11],\ gd[2,3,6,9,12],\ ge[11],\ gf[13,4]]$.

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.

In order to compare data having different domain values the normalization is applied firstly. We have used two methods to normalize the data: decimal scaling for the refactorings *cost* and min-max normalization for the value of the res function.

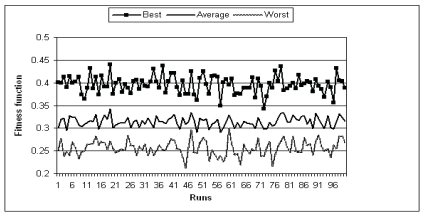
6 Practical Experiments for the Proposed Approach

The algorithm was run 100 times and the best, worse and average fitness values were recorded. Following subsections reveal the obtained results for different values of the α parameter.

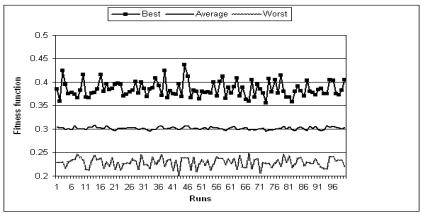
6.1 Equal Weights: $\alpha = 0.5$

A first experiment proposes equal weights, i.e., $\alpha = 0.5$, for the studied fitness function. Thus, $F(\vec{r}) = 0.5 \cdot f_1(\vec{r}) + 0.5 \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_m)$. 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)).

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,



(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

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.

For the recorded experiments, the best individual for 200 generations was better for 20 chromosomes populations

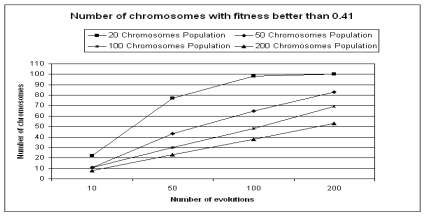


Figure 3: The evolution of the number of chromosomes with fitness better than 0.41 for the 20, 50, 100 and 200 individual populations

(with 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*.

6.2 Different Weights: $\alpha = 0.3$

An experiment with different weights, i.e., $\alpha = 0.3$, where the final effect (res function) has a greater relevance than the implied cost (rc mapping) of the applied refactorings is presented below. Therefore, the new fitness function may be rewritten as: $F(\vec{r}) = 0.3 \cdot f_1(\vec{r}) + 0.7 \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_m)$.

Figure 4 shows the the number of chromosomes better than 0.298 for the 20, 50, 100 and 200 individuals populations with 10, 50, 100 and 200 generations. It shows the grouping of the eligible chromosomes for the 100 and 200

individuals populations for each number of generations. The solutions for the 20 individuals populations for the studied number of generations keep their good quality, since the number of eligible chromosomes remains higher than any individuals population recorded by the experiment.

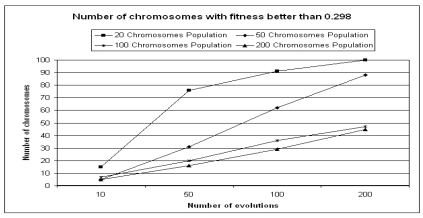


Figure 4: The evolution of the number of chromosomes with fitness better than 0.298 for the 20, 50, 100 and 200 individuals populations, with $\alpha = 0.3$

The experiment shows good results in all 100 runs as quality and number for the studied individuals populations and number of generations. In the 200 generations runs for 200 chromosomes populations the greatest value of the fitness function was 0.33426 (with 45 individuals with the fitness > 0.298) while in the 200 evolutions experiments for 50 individuals populations the best fitness value was not more than 0.32772 (88 individuals with the fitness > 0.298). But the best chromosome was found in the experiment with 200 generations and 20 individuals having the value 0.33587 (with all individuals with the fitness > 0.298).

The best individual obtained by this solution representation makes only small changes to the structure of the class hierarchy. Its analysis allows to extract a base class for the `PrintServer` and `FileServer` classes. Therefore, a new class named `Server` is added to the source code. The `addParameter` refactoring was not suggested such that the signature for `print` method from `PrintServer` class and for `save` method from `FileServer` class are changed in order to allow the corresponding `accept` methods from the `PrintServer` and `FileServer` classes to be pulled up. More, no appearance for the `encapsulatedField` refactoring have been recorded.

6.3 Different Weights: $\alpha = 0.7$

The last experiment was run for $\alpha = 0.7$, where the cost (rc mapping) of the applied refactorings is more important than the implied final effect (res function) on the affected software entities. Thus, the new fitness function is: $F(\vec{r}) = 0.7 \cdot f_1(\vec{r}) + 0.3 \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_m)$.

The number of chromosomes better than 0.527 for the 20, 50, 100 and 200 individuals populations with 10, 50, 100 and 200 generations is depicted in Figure 5. The solutions for the 20 individuals populations for each studied number of evolutions keep their good quality, the number of eligible chromosomes remaining raised.

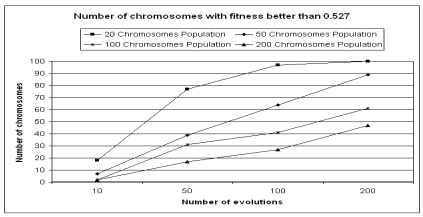


Figure 5: The evolution of the number of chromosomes with fitness better than 0.527 for the 20, 50, 100 and 200 individuals populations, with $\alpha = 0.7$

In the 200 generations runs for 50 chromosomes populations the greatest value of the fitness function was 0.60772 (with 89 individuals with the fitness > 0.527) while in the 200 evolutions experiments for 200 individuals populations the best fitness value was not more than 0.617 (47 individuals with the fitness > 0.527). But the best chromosome was found in the experiment with 200 generations and 20 individuals having the value 0.61719 (with all individuals with the fitness > 0.527).

The best chromosome obtained for $\alpha = 0.7$ experiment suggests several refactorings, but there are some that have to be interpreted by the programmer. A new base class for the `PrintServer` and `FileServer` classes is added too. The signature for the `print` method from `PrintServer` class and for `save` method from `FileServer` class is not suggested to be changed by the best chromosome. More, the `renameMethod` refactoring was recommended for the `save` method from `FileServer` class, but not for the `print` method from the sibling `PrintServer` class. Another improvment suggested by the current experiment is to apply the `pullUpMethod` refactoring in order to highlight the polymorphic behaviour of the `accept` method from the `PrintServer` and `FileServer` classes. But, no appearance for the `encapsulatedField` refactoring have been recorded such that all public class attributes become protected from unauthorized access.

6.4 Discussion

Current paper presents the results of the proposed approach in Section 5 for three different value for the α parameter, i.e., 0.3, 0.5, 0.7. A chromosome summary of the obtained results for all experiments is given below:

- $\alpha = 0.3$, $bestFitness = 0.33587$ for 20 *chromosomes and* 200 *generations*
- $\alpha = 0.5$, $bestFitness = 0.4793$ for 20 *chromosomes and* 200 *generations*
- $\alpha = 0.7$, $bestFitness = 0.61719$ for 20 *chromosomes and* 200 *generations*

The experiment for $\alpha = 0.3$ 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, loer than the best fitness value for the $\alpha = 0.5$ chromosome, i.e.,0.4793. This shows that an agreggated fitness function with a higher weight for the overall impact of the applied refactorings unbalance the fitness function. Therefore, there are not too many key software entities to be refactored by a such an experiment. The experiment for $\alpha = 0.7$ gets near to the $\alpha = 0.5$ 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 $\alpha = 0.7$ experiment. More, 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. The $\alpha = 0.7$ 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 $\alpha = 0.5$ experiment the best fitness value is lower than this one.

Balancing the fitness values for the studied experiments and the relevance of the suggested solutions, we consider the $\alpha = 0.5$ experiment is more relevant as quality of the results, than the other analyzed experiments. Figure 6 highlights the changes in the class hierarchy for the $\alpha = 0.5$.

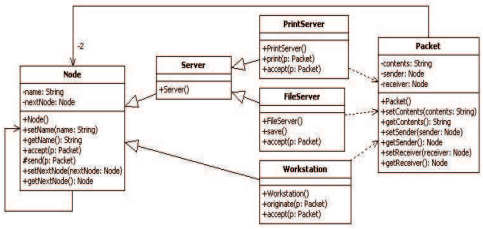


Figure 6: The class diagram for the LAN Simulation source code, with $\alpha = 0.5$

7 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.

8 Conclusions and Future work

The paper presents three experiments of the MOERSSP with different α values. The results for a proposed weighted objective genetic algorithm on a experimental didactic case study are presented and analized. Different α values may strongly unbalance the aggregated fitness function, making it either too poor or too expensive.

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] C. Chisăliţă–Creţu, A. Vescan, "The Multi-objective Refactoring Selection Problem", *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.