We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

23

The Multi-Objective Refactoring Set Selection Problem - A Solution Representation Analysis

Camelia Chisăliţă-Creţu
Babeş-Bolyai University
Romania

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. 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 (Fowler, 1999). The Multi-Objective Refactoring Set Selection Problem (MORSSP) is the identification problem of the set of refactorings that may be applied to the software entities, such that some specified constraints are kept and several objectives optimized.

This work is organized as follows: The motivation and a possible working scenario for the proposed refactoring selection problem is presented by Section 2. Section 3 reminds the existing work related to the studied domain. The *General Multi-Objective Refactoring Selection Problem* is formally stated by Section 4. Section 5 defines the *Multi-Objective Refactoring Set Selection Problem* as a two conflicting objective problem. The case study used within the research is shortly reminded by Section 6. The evolutionary approach with a proposed weighted objective genetic algorithm and the different solution representations studied are addressed by Section 7. A proposed refactoring strategy together with the input data for the advanced genetic algorithms are presented by Section 8. The results of the pratical experiments for the *entity based* and *refactoring based* solution representations for the multi-objective approach are summarized and analyzed by Section 9. Section 10 lists the conclusions and future research direction of the presented work.

2. Background

2.1 Motivation

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 (Fowler, 1999; Mens & Tourwe, 2004). 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 (Fowler, 1999). While some useful refactorings can be easily identified, it is difficult to determine those refactorings that really improve the internal structure of the program. It is a fact that many

useful refactorings, whilst improving one aspect of the software, make undesirable another one.
Refactorings application is available for almost all object-oriented languages and programming environments. Though, there are still a number of problems to address in order to raise the refactoring automation level.

2.2 Working scenario

Assuming a tool that detects opportunities for refactoring is used (Mens & Tourwe, 2003), it will identify badly structured source code based on code smells (van Emden & Moonen, 2002; Fowler, 1999), metrics (Marinescu, 1998; Simon et al., 2001) or other techniques. The gathered information is used to propose a set of refactorings that can be applied in order to improve the software internal 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. It means that applying any of the suggested refactorings may cancel the application of other refactorings that have been already proposed by the developer, but not selected and applied yet.
In (Mens et al., 2007) are presented three kinds of such dependencies: mutual exclusion, sequential dependency, asymmetric conflict. Therefore, the goal is to explore the possibility of identifying the refactorings that optimize some objectives, like costs or 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.

3. Related work

A closely related previous work is the Next Release Problem (NRP) studied by several authors (Bagnall et al., 2001; Greer & Ruhe, 2004; Zhang et al., 2007), where the goal was to find the most appropriate set of requirements that equilibrate resource constraints to the customer requests, in this way the problem was defined as a constrained optimization problem.
The corresponding refactoring selection problem is an example of a Feature Subset Selection (FSS) search problem. Other FSS problems in previous work on SBSE include the problem of determining good quality predictors in software project cost estimation, studied by Kirsopp et al. (Kirsopp et al., 2002), choosing components to include in different releases of a system, studied by Harman et al. (Harman et al., 2005) and Vescan et al. (Vescan & Pop, 2008).
Previous work on search-based refactoring problems (Bowman et al., 2007; Harman & Tratt, 2007; O'Keefe & O'Cinneide, 2006; Zhang et al., 2007) in SBSE has been concerned with single objective formulations of the problem only. Much of the other existing work on SBSE has tended to consider software engineering problems as single objective optimization problems too. But recent trends show that multi-objective approach has been tackled lately, which appears to be the natural extension of the initial work on SBSE.
Other existing SBSE work that does consider multi-objective formulations of software engineering problems, uses the weighted approach to combine fitness functions for each objective into a single objective function using weighting coefficients to denote the relative importance of each individual fitness function. In the search based refactoring field, Seng

et al. (Seng et al., 2006) and O'Keeffe and O'Cinneide (O'Keefe & O'Cinneide, 2006) apply a weighted multi-objective search, in which several metrics that assess the quality of refactorings are combined into a single objective function. Our approach is similar to those presented in (O'Keefe & O'Cinneide, 2006; Seng et al., 2006) but the difference is the heterogeneity of the weighted fitness functions that are combined together. Thus, we gather up the cost aspect of a refactoring application, the weight of the refactored software entity in the overall system and the effect or impact of the applied transformation upon affected entities as well.

4. General refactoring selection problem

4.1 GMORSP statement

In order to state the *General Multi-Objective Refactoring Selection Problem* (GMORSP) some notion and characteristics have to be defined. Let $SE = \{e_1, \ldots, e_m\}$ be a set of software entities, e.g., a class, an attribute from a class, a method from a class, a formal parameter from a method or a local variable declared in a method implementation. They are considered to be low level components bounded through dependency relations.

A software system SS consists of a software entity set SE together with different types of dependencies between the contained items. A dependency mapping ed is defined as:

$SED = \{\mathbf{u}ses\mathbf{A}ttribute, \mathbf{c}alls\mathbf{M}ethod, \mathbf{s}uper\mathbf{C}lass, \mathbf{a}ssociatedwith\mathbf{C}lass, \mathbf{n}o\mathbf{D}ependency\}$,

$ed : SE \times SE \rightarrow SED$,

$$ed(e_i, e_j) = \begin{cases} \text{uA, if the } \textit{method } e_i \textit{ uses} \text{ the } \textit{attribute } e_j \\ \text{cM, if the } \textit{method } e_i \textit{ calls} \text{ the } \textit{method } e_j \\ \text{sC, if the } \textit{class } e_i \text{ is a } \textit{direct superclass} \text{ for the } \textit{class } e_j \\ \text{aC, if the } \textit{class } e_i \text{ is } \textit{associated} \text{ with the } \textit{class } e_j \\ \text{nD, otherwise} \end{cases}, \quad (1)$$

where $1 \leq i, j \leq m$.

If a *class* e_i, $1 \leq i \leq m$, is an *indirect superclass*, for the *class* e_j, $1 \leq j \leq m$ then $ed(e_i, e_j) = sC*$ and $\exists$ *class* e_k, $1 \leq k \leq m$ such that $ed(e_i, e_k) = sC$, where $1 \leq i \leq m$, $1 \leq j \leq m$. The association relationship between two classes may be expressed as: aggregation, composition or dependency. If a *class* e_i, $1 \leq i \leq m$, has an aggregation relationship with a *class* e_j, $1 \leq j \leq m$, the association multiplicity is nested within the simple class association notation, i.e., $ed(e_i, e_j) = aC*_1^n$.

A set of possible relevant chosen refactorings (Fowler, 1999) that may be applied to different types of software entities of SE is gathered up through $SR = \{r_1, \ldots, r_t\}$. Specific refactorings may be applied to particular types of software entities, i.e., the *RenameMethod* refactoring may be applied to a method entity only, while the *ExtractClass* refactoring has applicability just for classes. Therefore a mapping that sets the applicability for the chosen set of refactorings SR to the set of software entities SE, is defined as:

$ra : SR \times SE \rightarrow \{\mathbf{T}rue, \mathbf{F}alse\}$,

$$ra(r_l, e_i) = \begin{cases} \text{T, if } r_l \text{ may be applied to } e_i \\ \text{F, otherwise} \end{cases}, \quad (2)$$

where $1 \leq l \leq t, 1 \leq i \leq m$.

There are various dependencies between refactorings when they are applied to the same software entity, a mapping emphasizing them being defined by:
$SRD = \{\mathbf{B}efore, \mathbf{A}fter, \mathbf{A}lways\mathbf{B}efore, \mathbf{A}lways\mathbf{A}fter, \mathbf{N}ever, \mathbf{W}henever\}$,
$rd : SR \times SR \times SE \rightarrow SRD$,

$$rd(r_h, r_l, e_i) = \begin{cases} \text{B}, & \text{if } r_h \text{ may be applied to } e_i \text{ only before } r_l,\ r_h < r_l \\ \text{A}, & \text{if } r_h \text{ may be applied to } e_i \text{ only after } r_l,\ r_h > r_l \\ \text{AB}, & \text{if } r_h \text{ and } r_l \text{ are both applied to } e_i \text{ and } r_h < r_l \\ \text{AA}, & \text{if } r_h \text{ and } r_l \text{ are both applied to } e_i \text{ and } r_h > r_l \\ \text{N}, & \text{if } r_h \text{ and } r_l \text{ cannot be both applied to } e_i \\ \text{W}, & \text{otherwise, i.e., } r_h \text{ and } r_l \text{ may be both applied to } e_i \end{cases}, \quad (3)$$

where $ra(r_h, e_i) = T$, $ra(r_l, e_i) = T$, $1 \leq h, l \leq t$, $1 \leq i \leq m$.

Let $DS = (SE^m, SR^t)$ be the *decision domain* for te GMORSP and $\vec{x} = (e_1, e_2, \ldots, e_m, r_1, r_2, \ldots, r_t)$, $\vec{x} \in DS$ a *decision variable*. The GMORSP is defined by the followings:

- $f_1, f_2, \ldots, f_M - M$ objective functions, where $f_i : DS \rightarrow \mathcal{R}^{m+t}$, $i = \overline{1, M}$ and $F(\vec{x}) = \{f_1(\vec{x}), \ldots, f_M(\vec{x})\}$, $\vec{x} \in DS$;
- $g_1, \ldots, g_J - J$ inequality constraints, where $g_j(\vec{x}) \geq 0$, $j = \overline{1, J}$;
- $h_1, \ldots, h_K - K$ equality constraints, where $g_k(\vec{x}) = 0$, $k = \overline{1, K}$.

The GMORSP is the problem of finding a decision vector $\vec{x} = (x_1, \ldots, x_{m+t})$ such that:

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

where $f_i : DS \rightarrow \mathcal{R}^{m+t}$, $\vec{x} \in DS$, $g_j(\vec{x}) \geq 0, j = \overline{1, J}$, $h_k(\vec{x}) = 0, k = \overline{1, K}$, $i = \overline{1, M}$.
Multi-objective optimization often means optimizing conflicting goals. For the GMORSP formulation there may be the possibility to blend different types of objectives, like: some of them to be maximized and some of them to be minimized.
For those cases where the conflicting objectives exist, they must be converted to meet the optimization problem requirements. Therefore, for an objective f_i, $0 \leq i \leq M$, that needs to be converted, where MAX is the highest value from the objective space of the objective mapping f_i, $0 \leq i \leq M$, $MAX \in \mathcal{R}^{m+t}$, there are two ways to switch to the optimal objective, as:

- $MAX - f_i(\vec{x})$, when MAX can be computed;
- $-f_i(\vec{x})$, when MAX cannot be computed.

4.2 Specific multi-objective refactoring selection problems

In the context of appropriate refactoring selection research domain there are many problems that may be defined as multi-objective optimization problems. Section 5 states the *Multi-Objective Refactoring Set Selection Problem* as a particular refactoring selection problem. A more restraint problem definition for the case when a single refactoring is searched is discussed too, as *Multi-Objective Single Refactoring Selection Problem* in (Chisăliţă-Creţu & Vescan, 2009), (Chisăliţă-Creţu & Vescan, 2009).
Specific conflicting objectives are studied in order to identify the optimal set of refactorings. Therefore, the *refactoring cost* has to be minimized, while the *refactoring impact* on the affected software entities needs to be maximized.

5. The multi-objective refactoring set selection problem

The *Multi-Objective Refactoring Set Selection Problem* (MORSSP) is a special case of refactoring selection problem. Its definition in Chisăliţă-Creţu (2009) follows the *General Multi-Objective Refactoring Selection Problem* (see Section 4). The two compound and conflicting objective functions are defined as the *refactoring cost* and the *refactoring impact* on software entities.
In order to state the *Multi-Objective Refactoring Set Selection Problem* (MORSSP) some notions and characteristics have to be defined.

Input data
Let

$$SE = \{e_1, \ldots, e_m\}$$

be a set of software entities, e.g., 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.
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 $\sum_{i=1}^{m} w_i = 1$.
The set of possible relevant chosen refactorings Fowler (1999) that may be applied to different types of software entities of SE is

$$SR = \{r_1, \ldots, r_t\}.$$

Dependencies between such transformations when they are applied to the same software entity are expressed by the formula 3 (see Section 4).
The effort involved by each transformation is converted to cost, described by the following function:
$rc : SR \times SE \rightarrow Z$,

$$rc(r_l, e_i) = \begin{cases} > 0, \text{ if } ra(r_l, e_i) = T \\ = 0, \text{ otherwise} \end{cases},$$

where the ra mapping is defined by the formula 2 (see Section 4), $1 \leq l \leq t, 1 \leq i \leq m$.
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:
$effect : SR \times SE \rightarrow Z$,

$$effect(r_l, e_i) = \begin{cases} > 0, \text{ if } ra(r_l, e_i) = T \text{ and has the requested effect on } e_i \\ < 0, \text{ if } ra(r_l, e_i) = T; \text{ has } not \text{ the requested effect on } e_i \\ = 0, \text{ otherwise} \end{cases},$$

where the ra mapping is defined by the formula 2 (see Section 4), $1 \leq l \leq t, 1 \leq i \leq m$.
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 it follows:
$res : SR \rightarrow Z$,

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

where $1 \leq l \leq t$.

Additional notations
Each refactoring $r_l, l = \overline{1,t}$ may be applied to a subset of software entities, defined as a function:
$re : SR \rightarrow \mathcal{P}(SE)$,

$$re(r_l) = \left\{ e_{l_1}, \ldots, e_{l_q} \mid \text{if } ra(r_l, e_{l_u}) = T, 1 \leq u \leq q, 1 \leq q \leq m \right\},$$

where the ra mapping is defined by the formula 2 (see Section 4), $re(r_l) = SE_{r_l}, SE_{r_l} \in \mathcal{P}(SE) - \emptyset, 1 \leq l \leq t$.

Output data
The MORSSP is the problem of finding a subset of entities named $ESet_l$, $ESet_l \in \mathcal{P}(SE) - \emptyset$ for each refactoring $r_l \in SR, l = \overline{1,t}$ such that:

- the following objectives have to be optimized:
  - the overall refactoring cost is minimized;
  - the overall refactoring impact on software entities is maximized;
- refactoring dependencies constraints defined by the mapping 3 are satisfied.

The solution $S = (ESet_1, \ldots, ESet_t)$ consists of the $ESet_l$ elements for a specific refactorings $r_l, 1 \leq l \leq t$, where $ESet_l \in \mathcal{P}(SE) - \emptyset, 1 \leq l \leq t$.

5.1 Multi-objective optimization problem formulation

The MORSSP optimizes the required cost minimize required cost for the applied refactorings and to maximize the refactoring impact on software entities. Therefore, the multi-objective function $F(\vec{r}) = \{f_1(\vec{r}), f_2(\vec{r})\}$, where $\vec{r} = (r_1, \ldots, r_t)$ is to be optimized, as described below. Current MORSSP treats cost as an objective instead of a constraint, like the refactoring dependencies described by the mapping 3 (see Section 4).
The first objective function minimizes the total cost for the applied refactorings:

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

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

$$maximize\left\{ f_2(\vec{r}) \right\} = maximize\left\{ \sum_{l=1}^{t} res(r_l) \right\} = maximize\left\{ \sum_{l=1}^{t} \sum_{i=1}^{m} w_i \cdot effect(r_l, e_i) \right\}, \quad (4)$$

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. The objective that does not follow the maximizing approach needs to be converted in a suitable way.
In order to convert the first objective function to a maximization problem in the MORSSP, the total cost is subtracted from MAX, the biggest possible total cost, as it is shown below:

$$maximize\left\{ f_1(\vec{r}) \right\} = maximize\left\{ MAX - \sum_{l=1}^{t} \sum_{i=1}^{m} rc(r_l, e_i) \right\}, \quad (5)$$

where $\vec{r} = (r_1, \ldots, r_t)$. The overall objective function for MORSSP is defined by:

$$\begin{aligned} &maximize\left\{F(\vec{r})\right\} = maximize\left\{f_1(\vec{r}),\ f_2(\vec{r})\right\} = \\ &= maximize\left\{MAX - \sum_{l=1}^{t}\sum_{i=1}^{m} rc(r_l, e_i),\ \sum_{l=1}^{t}\sum_{i=1}^{m} w_i \cdot effect(r_l, e_i)\right\}, \end{aligned} \tag{6}$$

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

6. Case study: LAN simulation

The proposed algorithm was applied on a simplified version of the *Local Area Network* (LAN) simulation source code, that was presented in (Demeyer et al., 2005). Figure 1 shows the class diagram of the studied source code. It contains 5 classes with 5 attributes and 13 methods, constructors included.

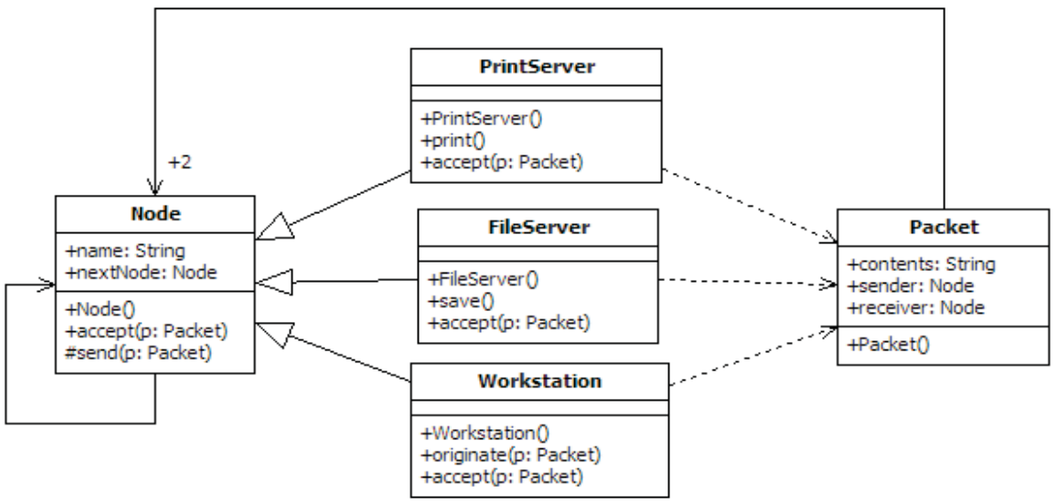


Fig. 1. Class diagram for the LAN Simulation source code

The *LAN Simulation* source code has been successfully used in several introductory programming courses to illustrate and teach good object-oriented design. Moreover, a large number of refactoring domain research papers have used it to present the effect of different refactoring applications.

The *LAN Simulation Problem* is sufficiently simple for illustrative purposes, by covering most of the interesting constructs of the object-oriented programming paradigm, like: inheritance, late binding, super calls, method overriding. While it has been implemented in several object-oriented programming languages, within our research a *Java* implementation is used. The *LAN Simulation* source code has proved its relevance within the research since it follows an incremental development style and it suggests several typical refactorings. Therefore, it is accepted as a suitable example that serves as a basis for different refactoring feasibility studies. The source code version used within our experiments 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. The `PrintServer`, `FileServer`, and `Workstation` classes refine

the behaviour of the `accept` method (and perform a super call) to achieve specific behaviour for printing or saving the `Packet` and avoiding its endless cycling. A `Packet` object can only originate (`sender` attribute) from an `Workstation` object and sequentially visits every `Node` object in the network until it reaches its receiver (`receiver` attribute) that `accepts` the `Packet`, or until it returns to its `sender` workstation, indicating that the `Packet` cannot be delivered. The corresponding initial class diagram is depicted by Figure 1.

Difficulties. This *LAN Simulation* version has several aspects denoting lack of data hiding and flexibility. The instance variables are visible from outside causing possible unauthorized accesses or changes. Intuitively, shielding the attribute from direct references reduces data coupling. This allows the attribute to change its value without affecting clients using the attribute value. Another issue is related to the small capability of code reuse within the class hierarchy. Generally speaking, generalization may increase code reuse and reduce the impact of changes. Thus, any change is done within a single place and all possible clients remain unchanged.

Solutions. The *EncapsulateField* refactoring can be performed in order to guard the attribute `nextMachine` of the class `Node` from direct access. It results in the introduction of two methods in `Node` class, as: the `getNextNode` method which accesses the attribute and returns its value to the caller and the `setNextNode` method which takes the new value of the attribute as a parameter and updates the attribute. Then, the attribute itself is made `private`.
Generalization degree may be raised by applying the *RenameMethod* refactoring to the `print` method from the `PrintServer` class and to the `save` method from the `FileServer` class to an unique name `process`, while its signature remains unchanged. Then each call of the former methods will be replaced by a call to the corresponding `process` method from the `PrintServer` or `FileServer` classes. The generalization process may go thoroughly. The *PullUpMethod* refactoring may be applied to the `process` methods from the `PrintServer` and `FileServer` classes. This requires the introduction of a new empty body method in the `Node` superclass. By pulling up a method to a base class, its specific behaviour is generalized, making possible for subclasses to reuse and specialize the inherited behaviour.

7. Proposed approach description

The MORSSP is approached here by exploring the various existing refactoring dependencies. Two conflicting objectives are studied, i.e., minimizing the refactoring cost and maximizing the refactoring impact, together with some constraints to be kept, as the refactoring dependencies.
There are several ways to handle a multi-objective optimization problem. The *weighted sum method* (Kim & deWeck, 2005) was adopted to solve the MORSSP. The overall objective function to be maximized $F(\vec{r})$, defined by the formula 6 (see Section 4), is shaped to the weighted sum principle with two objectives to optimize.
Therefore, $maximize\left\{F(\vec{r})\right\} = maximize\left\{f_1(\vec{r}), f_2(\vec{r})\right\}$, is mathematically rewritten to:

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

where $0 \leq \alpha \leq 1$ and $\vec{r}$ is the decision variable.

A steady–state evolutionary model is advanced by the proposed evolutionary computation technique (Chisăliţă-Creţu, 2010). Algorithm 1 is the adapted genetic algorithm to the context of the investigated MORSSP, proposed in (Chisăliţă-Creţu, 2009; 2010).

Algorithm 1 the adapted evolutionary algorithm for the MORSSP Chisăliţă-Creţu (2009)

Input:
- SR – the set of refactorings;
- SE – the set of entities;
- rd – the mapping of refactoring dependencies;
- $Weight$ – the set of entity weights;
- rc – the mapping of refactoring costs;
- $effect$ – the mapping of refactoring impact on entities;
- $NumberOfGenerations$ – the number of generations to compute;
- $NumberOfIndividuals$ – the number of individuals within a population;
- $CrossProbability$ – the crossover probability;
- $MutProbability$ – the mutation probability.

Output:
- the solution.

Begin
@ Randomly create the initial population $P(0)$;
for t := 1 to $NumberOfGenerations$ **do**
  for k := 1 to $NumberOfIndividuals/2$ **do**
    @Select two individuals p_1 and p_2 from the current population;
    @OnePointCutCrossover for the parents p_1 and p_2, obtaining the two offsprings o_1 and o_2;
    @Mutate the offsprings o_1 and o_2;
    if (Fitness(o_1) $<$ Fitness(o_2)) **then**
      if (Fitness(o_1) $<$ the fitness of the worst individual) **then**
        @Replace the worst individual with o_1 in P(t);
      else
        if (Fitness(o_2) $<$ the fitness of the worst individual) **then**
          @ Replace the worst individual with o_2 in P(t);
        end if
      end if
    end if
  end for
end for
@Select the best individual from $P(NumberOfGenerations)$;
End.

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.
The proposed genetic algorithm approaches two solution representations for the studied problem. The genetic algorithm that uses a *refactoring-based* solution representation for the

MORSSP is denoted by *RSSGARef*, while the corresponding *entity-based* genetic algorithm is denoted by *RSSGAEnt*.

7.1 Refactoring-based solution representation

For the *RSSGARef* algorithm the solution representation is presented in (Chisăliţă-Creţu, 2009), with the decision vector $\vec{S} = (S_1, \ldots, S_t)$, where $S_l \in \mathcal{P}(SE), 1 \leq l \leq t$, determines the entities that may be transformed using the proposed refactoring 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 for each $e_{l_u}, e_{l_u} \in SE_{r_l}, e_{l_u} \in S_l, S_l \in \mathcal{P}(SE), 1 \leq u \leq q, 1 \leq q \leq m, 1 \leq l \leq t$. 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.

7.1.1 Genetic operators

Crossover and mutation operators are used by this approach, being described in the following.

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 instance, 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

$parent_2 = [g1[4,9,10,12],\ g2[7],\ g3[5,8,11],\ g4[10,11],\ g5[2,3,12],\ g6[5,9]]$, for the cutting point 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

Mutation operator used here exchanges the value of a gene with another value from the allowed set. Namely, mutation of the i-th gene consists of adding or removing a software entity from the set that denotes the i-th gene.

For example, if the individual to be mutated is

$parent = [ga[1,7],\ gb[3,5,10],\ gc[8],\ gd[2,6,9,12],\ ge[12],\ gf[13,4]]$ and

if the 5-th gene is to be mutated, the obtained offspring is $offspring = [ga[1,7],\ gb[3,5,10],\ gc[8],\ gd[2,6,9,12],\ ge[\mathbf{10},12]\ gf[13,4]]$ by adding the 10-th software entity to the 5-th gene.

7.2 Entity-based solution representation

The *RSSGAEnt* algorithm uses the solution representation presented in (Chisăliţă-Creţu, 2009), where the decision vector (chromosome) $\vec{S} = (S_1, \ldots, S_m), S_i \in \mathcal{P}(SR), 1 \leq i \leq m$ determines the refactorings that may be applied in order to transform the proposed set of software entities SE.

The item S_i on the i-th position of the solution vector represents a set of refactorings that may be applied to the i-th software entity from SE, where each entity $e_{l_u} \in S_{r_l}, S_{r_l} \in \mathcal{P}(SR), 1 \leq u \leq q, 1 \leq q \leq m, 1 \leq l \leq t$. It means it is possible to apply more than once the same refactoring to different software entities, i.e., distinct gene values from the chromosome may contain the same refactoring.

7.2.1 Genetic operators

The genetic operators used by the *RSSGAEnt* algorithm are crossover and mutation as described by Section 7.1.1. The *crossover operator* uses a simple one point cut scheme, randomly chosen. All the data beyond the cut point from the parent strings is swapped between the two parents.

The mutation operator used here exchanges the value of a gene with another value from the allowed set. The mutation of the i-th gene consists of adding or removing a refactoring from the set that denotes the i-th gene.

8. Input data

The adapted genetic algorithm proposed in (Chisăliţă-Creţu, 2009; 2010) is applied to a simplified version of the *LAN Simulation* source code (see Section 1). Relevant data about the source code is extracted and the software entity set is defined as: $SE = \{c_1, \ldots, c_5, a_1, \ldots, a_5, m_1, \ldots, m_{13}\}$, $|SE| = 23$. The chosen transformations are refactorings that may be applied to classes, attributes or methods, as: *RenameMethod, ExtractSuperClass, PullUpMethod, MoveMethod, EncapsulateField, AddParameter*. They will form the refactoring set $SR = \{r_1, \ldots, r_6\}$ in the following. The entity weights are gathered within the set $Weight$, that presented by Table 1 , where $\sum_{i=1}^{23} w_i = 1$.

The dependency relationship between refactorings, described by the mapping *rd* and the final impact of each refactoring stated by the *res* mapping are defined by the Table 1. The *res* mapping value computation for each refactoring is based on the weight of each possible affected software entity, as it was defined in Section 5.

Each software entity allows specific refactorings to be applied to, otherwise the cost mapping values are 0. E.g., the r_1, r_3, r_4, r_6 refactorings may be applied to the $m_1, m_4, m_7, m_{10}, m_{13}$ methods. For special methods, i.e., constructors, refactorings like $pullUpMethod$ (r_3) and $moveMethod$ (r_4) cannot be applied. Here, the cost mapping *rc* is computed as the number of transformations needed in order to apply the refactoring. Therefore, refactoring applications to related entities may have different costs.

Intermediate data for the $effect$ mapping was used to compute the *res* mapping values. The $effect$ mapping values were considered numerical data, denoting an estimated impact of refactoring application, e.g., a software metric assessment.

8.1 Proposed refactoring strategy

A possible refactoring strategy for the *LAN Simulation Problem* is presented below. Based on the *difficulties* (see Section 6) presented for the corresponding class hierarchy, three transformation categories may be identified. For each of them several improvement targets that may be achieved through refactoring are defined.

1. *information management* (data hiding, data cohesion):
   (a) control the attribute access (*EncapsulateField* refactoring);
2. *behaviour management* (method definition, method cohesion):
   (a) adapt the method signature to new context (*AddParameter* refactoring);
   (b) increase the expressiveness of a method identifier by changing its name (*RenameMethod* refactoring);
   (c) increase method cohesion within classes (*MoveMethod* and *PullUpMethod* refactorings);
3. *class hierarchy abstraction* (class generalization, class specialization):
   (a) increase the abstraction level within the class hierarchy by generalization (*ExtractSuperClass* refactoring).

(a) Refactoring dependencies (rd) and final impact (res) of their applying to the entity set (SE)

rd	r_1	r_2	r_3	r_4	r_5	r_6
r_1	N		B			AA
r_2		N	B			
r_3	A	A	N	N		
r_4			N	N		
r_5					N	
r_6	AB					N
res	0.4	0.49	0.63	0.56	0.8	0.2

(b) Refactoring costs (rc) and their applicability on software entities. The weight for each software entity ($Weight$)

rc	r_1	r_2	r_3	r_4	r_5	r_6	$Weight$
c_1		$\surd/1$					0.1
c_2		$\surd/1$					0.08
c_3		$\surd/2$					0.08
c_4		$\surd/2$					0.07
c_5		$\surd/1$					0.07
a_1					$\surd/4$		0.04
a_2					$\surd/5$		0.03
a_3					$\surd/5$		0.03
a_4					$\surd/5$		0.05
a_5					$\surd/5$		0.05
m_1	$\surd/1$		$\surd/0$	$\surd/0$		$\surd/1$	0.04
m_2	$\surd/3$		$\surd/1$	$\surd/1$		$\surd/3$	0.025
m_3	$\surd/5$		$\surd/1$	$\surd/1$		$\surd/5$	0.025
m_4	$\surd/1$		$\surd/0$	$\surd/0$		$\surd/1$	0.04
m_5	$\surd/1$		$\surd/1$	$\surd/1$		$\surd/1$	0.025
m_6	$\surd/1$		$\surd/1$	$\surd/1$		$\surd/1$	0.025
m_7	$\surd/1$		$\surd/0$	$\surd/0$		$\surd/1$	0.04
m_8	$\surd/2$		$\surd/1$	$\surd/1$		$\surd/2$	0.025
m_9	$\surd/1$		$\surd/1$	$\surd/1$		$\surd/1$	0.025
m_{10}	$\surd/1$		$\surd/0$	$\surd/0$		$\surd/1$	0.04
m_{11}	$\surd/2$		$\surd/1$	$\surd/1$		$\surd/2$	0.025
m_{12}	$\surd/1$		$\surd/1$	$\surd/1$		$\surd/1$	0.025
m_{13}	$\surd/1$		$\surd/0$	$\surd/0$		$\surd/1$	0.04
							$\sum_{i=1}^{23} w_i = 1$

Table 1. Input Data for the *LAN Simulation Problem* case study

9. Practical experiments

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 subsections reveal the obtained results for different values of the α parameter: 0.3, 0.5 and 0.7, presented in Chisăliţă-Creţu (2009); Chisăliţă-Creţu (2009) (Chisăliţă-Creţu, 2009; 2010).

9.1 Refactoring-based solution representation experiments

Two types of experiments were run: with equal objective weights and different objectives weights. The equal objective weights uses $\alpha = 0.5$, while the two different weight experiments uses $\alpha = 0.7$ and $\alpha = 0.3$. In the former experiment the refactoring cost has a greater relevance than the refactoring impact, while in the last one, the refactoring impact drives the chromosome competition.

9.1.1 Equal weights ($\alpha = 0.5$)

The current experiment run on the *LAN Simulation Problem* proposes equal weights, i.e., $\alpha = 0.5$, for the studied fitness function Chisăliţă-Creţu (2009); Chisăliţă-Creţu (2009). That is,

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

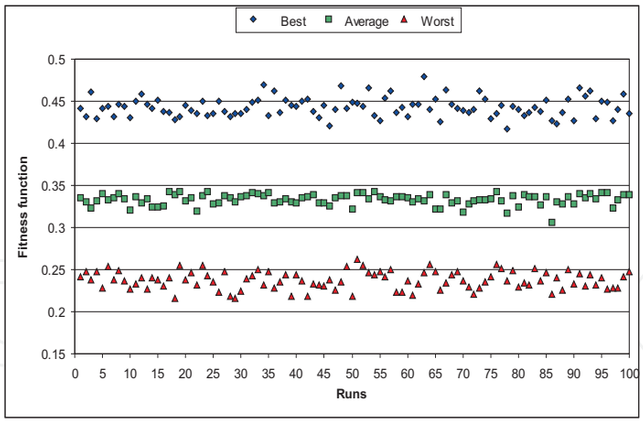
where $\vec{r} = (r_1, \ldots, r_t)$. Figure 2 presents the 200 generations evolution of the fitness function (best, worse and average) for 20 chromosomes populations (Figure 2(a)) and 200 chromosomes populations (Figure 2(b)).

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 individuals have fitness better than 0.41.

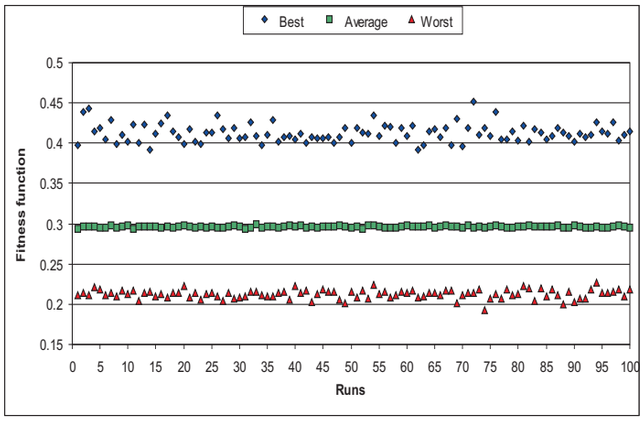
Compared to the former populations, the 200 chromosomes populations breed closer best individuals. The number of good chromosomes is smaller than the one for 20 individuals populations, i.e., 53 chromosome with fitness better than 0.41 only.

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 harder competition than within large populations (with many chromosomes) where the diversity breeds weaker individuals. As the Figure 2 shows it, after several generations smaller populations produce better individuals (as number and quality) than larger ones, due to the poor populations diversity itself.



(a) Experiment with 200 generations and 20 individuals



(b) Experiment with 200 generations and 200 individuals

Fig. 2. The evolution of fitness function (best, worse and average) for 20 and 200 individuals with 200 generations, with 11 mutated genes, for $\alpha = 0.5$

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 involve 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.

Impact on the *LAN simulation* 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. Moreover, the signatures of the `print` method from the `PrintServer` class is changed, though the method renaming to `process` identifier was not suggested. Opposite to this, for the `save` method in the `FileServer` class was recommended to change the method name to `process`, while the signature changing was not suggested yet. The two refactorings (*addParameter* and

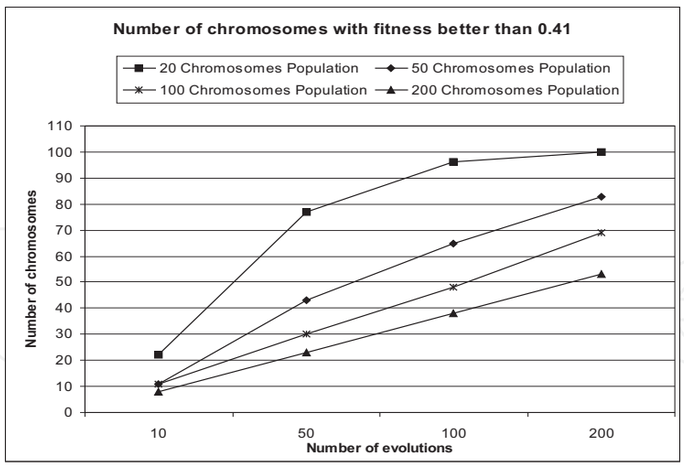


Fig. 3. The evolution of the number of chromosomes with fitness better than 0.41 for the 20, 50, 100 and 200 individual populations, with $\alpha = 0.5$

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 of the latter. The correct access to the class fields by encapsulating them within their classes is enabled for three of five class attributes.

The refactoring cost and refactoring impact on software entity have been treated with the same importance within the refactoring process ($\alpha = 0.5$).

The current solution representation allows to apply more than one refactoring to each software entity, i.e., the `print` method from the `PrintServer` class is transformed by two refactorings, as the *AddParameter* and *RenameMethod* refactorings.

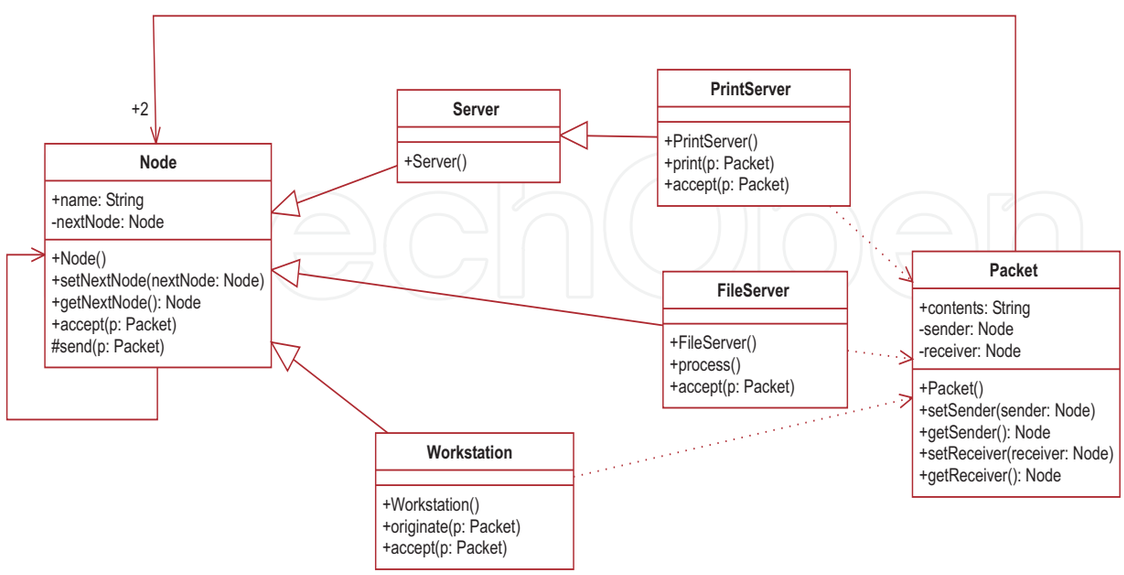


Fig. 4. The class diagram for the *LAN Simulation* source code, after applying the *RSSGARef Algorithm* solution for $\alpha = 0.5$

9.1.2 Discussion on *RSSGARef algorithm* experiments

Current subsection summarizes the results of the proposed *RSSGARef Algorithm* for three different values of 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 Chisăliţă-Creţu (2009). A chromosome summary of the obtained results for all run experiments as it is presented in Chisăliţă-Creţu (2009); Chisăliţă-Creţu (2009) (Chisăliţă-Creţu, 2009) is given below:

- $\alpha = 0.3$, $bestFitness = 0.33587$ 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]]
- $\alpha = 0.5$, $bestFitness = 0.4793$ 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]]
- $\alpha = 0.7$, $bestFitness = 0.61719$ 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 $\alpha = 0.3$ should identify those refactorings for which the cost has a lower relevance than the overall impact on the applied software entities. But, the best chromosome obtained has the fitness value 0.33587, lower than the best fitness value for the $\alpha = 0.5$ chromosome, i.e., 0.4793. This shows that an unbalanced aggregated fitness function with a higher weight for the overall impact on the applied refactorings, promotes the individuals with a lower cost and small refactorings. Therefore, there are not too many key software entities to be refactored by such an experiment.

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.

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. Moreover, there is a missing link in the same experiment, due to the fact the `AddParameter` refactoring was not recommended for the `save` method from the `FileServer` and the `print` method from the `PrintServer` class.

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 4 highlights the changes in the class hierarchy for the $\alpha = 0.5$ following the suggested refactorings from the recorded best chromosome.

9.2 Entity-based solution representation experiments

Similar to the *RSSGARef Algorithm*, the *RSSGAEnt Algorithm* was run 100 times and the best, worse and average fitness values were recorded. The algorithm was run for different number of generations and of individuals, as: number of generations 10, 50, 100, 200, and number of individuals 20, 50, 100, 200.

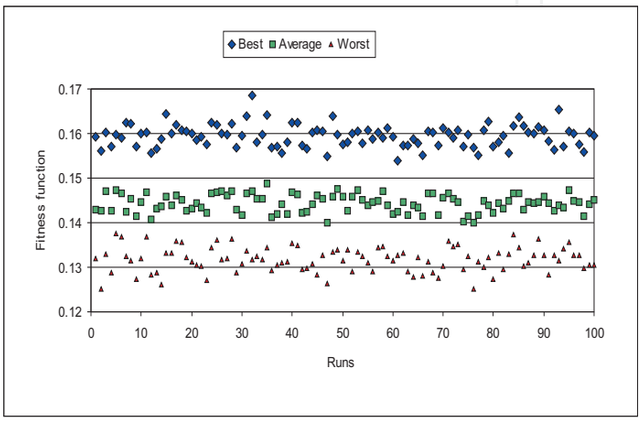
The parameters used by the evolutionary approach were the same as the ones used in the refactoring-based approach, like: mutation probability 0.7 and crossover probability 0.7. The

run experiments Chisăliţă-Creţu (2009) have used different values for the α parameter (0.3, 0.5 and 0.7).

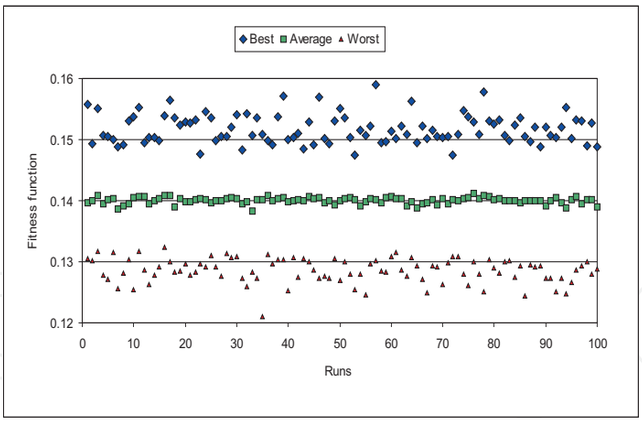
9.2.1 Different weights ($\alpha = 0.7$)

One of the different weighted experiments 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 Chisăliţă-Creţu (2009).

The results of the this experiment for the 20 individual populations with 50 generations evolution (Figure 5(a)) and 200 chromosome populations with 10 generations evolution (Figure 5(b)) are depicted by the Figure 5 with the fitness function (best, worse and average) values.



(a) Experiment with 50 generations and 20 individuals



(b) Experiment with 10 generations and 200 individuals

Fig. 5. The evolution of the fitness function (best, worst and average) for 20 and 200 individuals with 50 and 10 generation evolutions, with 11 mutated genes, for $\alpha = 0.7$

The best individual was obtained for a 50 generations run with a 20 chromosomes population with the fitness 0.16862 (with 98 chromosomes with fitness > 0.155), while the greatest fitness value of the 200 chromosomes populations with 10 generations evolution was 0.15901 (11 individuals only with fitness value > 0.155).

The worst individual was recorded for a 200 chromosomes population with a 10 generations evolution with the fitness value 0.121 (72 individuals having the fitness < 0.13), while for the 20 individuals population for a 50 generations evolution the worst chromosome had the fitness value 0.12515 (27 chromosomes with fitness value < 0.13).
The number of chromosomes better than 0.155 for the 20, 50, 100 and 200 individuals populations with 10, 50, 100 and 200 generations is captured by Figure 6. The solutions for the 20 individuals populations for each studied number of evolutions keep their good quality, but the 50, 100 and 200 chromosomes populations carry a more intense chromosome competition compared to previously run experiments.

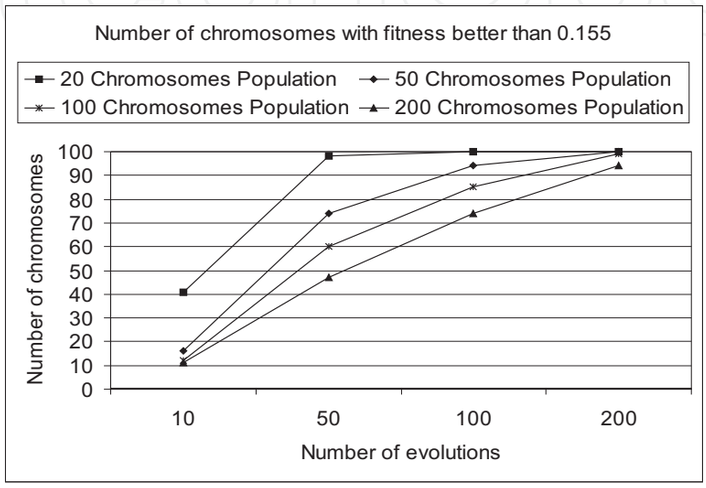


Fig. 6. The evolution of the number of chromosomes with fitness better than 0.155 for the 20, 50, 100 and 200 individuals populations, with $\alpha = 0.7$

Impact on the *LAN simulation* source code
The best chromosome obtained within this experiment suggests several refactorings, but there are some that have to be interpreted by the programmer as well. A new base class for the `PrintServer` and the `FileServer` classes is recorded by the obtained solution. The signature for the `save` method from the `FileServer` class is suggested to be changed by the best chromosome, though the similar change for the `print` method from the `PrintServer` class is not included by the studied best chromosome. The *renameMethod* refactoring was recommended for the `save` method from the `FileServer` class and for the `print` method from the `PrintServer` sibling class yet. Another improvement 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` but not for the same method within the `FileServer` class. No appearance of the *EncapsulatedField* refactoring was recorded in order to protect public class attributes from unauthorized access.

9.2.2 Discussion on *RSSGAEnt algorithm* experiments

The results of the proposed *RSSGAEnt Algorithm* for three different values of 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 the affected software entities Chisăliţă-Creţu (2009) are discussed by this section. A best chromosome summary for all run experiments as it is presented in Chisăliţă-Creţu (2009) is given below:

- $\alpha = 0.3$, $bestFitness = 0.19023$ for 20 *chromosomes and* 200 *generations*
  - $bestChrom$ = [[1], [∅], [1], [∅], [1], [4], [∅], [4], [4], [4], [3], [0, 3, 5], [3, 0, 5], [3] , [3, 5, 0], [3], [5], [2, 3, 0], [3], [5, 0, 3, 2], [0, 5, 2], [2, 3], [3]]
- $\alpha = 0.5$, $bestFitness = 0.17345$ for 20 *chromosomes and* 200 *generations*
  - $bestChrom$ = [[∅], [1], [1], [1], [1], [4], [∅], [∅], [∅], [4], [0, 2], [2, 0], [0], [5, 2] , [5, 3], [2, 5], [2, 0, 3], [0, 3, 2], [5, 0, 3, 2], [3, 2, 5], [3], [3, 0], [3, 5]]
- $\alpha = 0.7$, $bestFitness = 0.16862$ for 20 *chromosomes and* 50 *generations*
  - $bestChrom$ = [[∅], [1], [1], [1], [∅], [∅], [∅], [∅], [∅], [∅], [2, 3], [3, 2, 0], [3], [5, 0, 2] , [0, 2], [2, 3], [2], [2, 0, 3], [2, 3], [0, 5, 3, 2], [0, 2, 5], [3, 0], [2, 3, 0]]

The experiment for $\alpha = 0.5$ should identify those refactorings for which the refactoring cost and impact on the applied software entities have the same relevance within the overall maximization problem. Though, this best chromosome is lower than the best fitness value obtained for $\alpha = 0.3$, i.e., 0.19023.

Moreover, the analysis for the obtained best individuals suggests that an unbalanced aggregated fitness function (with a higher weight for the overall impact of the applied refactorings) advances low cost refactorings, bringing a higher benefit for the structure and the quality of the suggested solution.

The refactorings suggested by the $\alpha = 0.5$ experiment are not connected one to another, such that a coherent strategy may be drawn. The main achievement suggested by the analyzed best chromosome of this experiment is related to the *EncapsulateField* refactoring for the public class attributes, not suggested for all five of them yet.

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 of the best chromosome for this experiment is 0.16862, lower than the $\alpha = 0.5$ experiment best fitness value.

The experiment for $\alpha = 0.7$ gets near to the $\alpha = 0.3$ experiment as quality of the proposed solution. The best chromosome obtained within the former experiment suggests several refactorings, but there are some that have to be interpreted by the programmer. The achieved improvements cover two of the aspects to be improved within the class hierarchy, i.e., common behaviour (refactorings for methods), and class hierarchy abstraction (refactorings for classes). The information hiding aspects by suggesting refactorings for attributes was not recorded at all.

Compared to the other run experiments ($\alpha = 0.5$ and $\alpha = 0.3$) the achievements are more important as quality, though the effective overall fitness value is not the biggest.

The proposed solution by the $\alpha = 0.3$ experiment is more homogeneous, touching all the improvement categories. The drawback of this solutions is the ambiguity in several suggested refactoring sets for behaviour improvement. Therefore, the `save` method from the `FileServer` class and the `print` method from the `PrintServer` class may contain refactorings that belong to different refactoring strategies, i.e., *MoveMethod* and *PullUpMethod* refactorings.

9.3 Results analysis

This section analyzes the proposed solutions for the refactoring-based and entity-based solution representations. Both solution representations identify a set of refactorings for each software entity to which it may be applied to.

The chromosome size within the refactoring-based approach is 6, i.e., the number of possible refactorings to be applied, while the individual for the entity-based approach has 23 genes.

The recommended refactorings proposed by different runs and experiments does not shape a fully homogeneous refactoring strategy for none of the studied solution representations.
The best individual was obtained by the refactoring-based approach (*RSSGARef Algorithm*) was for a 200 generations evolution with 20 chromosomes population, having the fitness value of 0.4793, while by the entity-based approach (*RSSGAEnt Algorithm*) the recorded best chromosome was obtained for 200 generations and 20 individuals, with a fitness value of 0.19023. These solutions may be transposed from a representation to another, which means their structure may be compared and their efficiency assessed.
The idea that emerge from the run experiments was that smaller individual populations produce better individuals (as number, quality, time) than larger ones, that may be caused by the poor diversity within the population itself. Large number of genes of the individual structure induces poor quality to the current entity-based solution representation.
Table 2 summarizes the solutions obtained for the studied solution representation together with the goals reached by each of them. The number of achieved targets is computed based on the recommended refactoring presence within the studied chromosomes genes.

Solution representation	α **value**	**Best chrom.** (pop. size/ no. gen.)	**Best Fitness**	**Execution Time**	**Number of achieved targets (%)**				
					Data *(1a)*	**Method (2)** *(2a)*	*(2b)*	*(2c)*	**Class hierarchy** *(3a)*
Refactoring based	0.5	20c/200g	0.4793	36secs	60	50	50	50	50
	0.3	20c/200g	0.33587	32secs	0	0	0	50	100
	0.7	20c/200g	0.61719	37secs	0	0	50	100	50
Entity based	0.5	20c/200g	0.17345	75secs	40	0	50	100	100
	0.3	20c/200g	0.19023	61secs	80	50	100	100	50
	0.7	20c/50g	0.16862	19secs	0	50	100	100	100

Table 2. The best chromosomes obtained for the refactoring and entity based solution representations, with the α parameter values 0.5, 0.3, and 0.7

10. Conclusions

This work has advanced the evolutionary-based solution approach for the MORSSP. Adapted genetic algorithms have been proposed in order to cope with the multi-objectiveness of the required solution. Two conflicting objectives have been addressed, as to minimize the *refactoring cost* and to maximize the *refactoring impact* on the affected software entities. Different solution representations were studied and the various results of the run experiments were presented and compared.
The main contributions and results of the current work are:

- new genetic algorithms were proposed and different solution representations were studied for the MORSSP;
- adapted genetic operators to the refactoring selection area were tackled;
- a new goal-based assessment strategy for the selected refactorings was proposed in order to analyze and compare different achieved solutions;
- different experiments on the *LAN Simulation Problem* case study were run in order to identify the most appropriate refactoring set for each software entity such that the refactoring cost is minimized and the refactoring impact is maximized.

Further work may be done in the following directions:

- different and adapted to the refactoring selection area crossover operators may be investigated;
- the Pareto principle approach may be studied further;
- other experimental run on other relevant and real-world software systems case studies.

11. References

Bagnall, A., Rayward-Smith, V. & Whittley, I. (2001). The next release problem, *Information and Software Technology* Vol. 43(No. 14): 883–890.

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

Chisăliţă-Creţu, C. (2009). A Multi-Objective Approach for Entity Refactoring Set Selection Problem, *Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2009)*, August 4- 6, 2009, London, UK, pp. 790–795.

Chisăliţă-Creţu, M.C. (2009). First Results of an Evolutionary Approach for the Entity Refactoring Set Selection Problem, *Proceedings of the 4th International Conference "Interdisciplinarity in Engineering" (INTER-ENG 2009)*, Editura Universităţii Petru Maior din Târgu Mureş, November 12-13, 2009, Târgu Mureş, România, pp. 303–308.

Chisăliţă-Creţu, M.C. (2009). The Entity Refactoring Set Selection Problem - Practical Experiments for an Evolutionary Approach, *Proceedings of the World Congress on Engineering and Computer Science (WCECS2009)*, Newswood Limited, October 20-22, 2009, San Francisco, USA, pp. 285–290.

Chisăliţă-Creţu, M.C. (2009). Solution Representation Analysis For The Evolutionary Approach of the Entity Refactoring Set Selection Problem, *Proceedings of the 12th International Multiconference "Information Society" (IS2009)*, Informacijska družba, October 12-16, 2009, Ljubljana, Slovenia, pp. 269–272.

Chisăliţă-Creţu M.C. (2009). An Evolutionary Approach for the Entity Refactoring Set Selection Problem, *Journal of Information Technology Review accepted paper*.

Chisăliţă-Creţu M.C. & Vescan, A. (2009). The Multi-objective Refactoring Selection Problem, *Studia Universitatis Babes-Bolyai, Series Informatica* Special Issue KEPT-2009: Knowledge Engineering: Principles and Techniques (July 2009)(No.): 249–253.

Chisăliţă-Creţu, M.C. & Vescan, A. (2009). The Multi-objective Refactoring Selection Problem, *Proceedings of the 2nd Internaltional Conference Knowledge Engineering: Principles and Techniques (KEPT2009)*, Presa Universitară Clujeană, July 1-3, 2009, Cluj-Napoca, Romania, pp. 291–298.

Demeyer, S., Van Rysselberghe, F., Gǐrba, T., Ratzinger, J., Marinescu, R., Mens, T., Du Bois, B., Janssens, D., Ducasse, S., Lanza, M., Rieger, M., Gall, H. & El-Ramly, M. (2005). The LAN-simulation: a refactoring teaching example, *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE05)*, September 05-06, 2005, Lisbon, Portugal, pp. 123–131.

van Emden, E. & Moonen, L. (2002). Java quality assurance by detecting code smells, *Proceedings of 9th Working Conference on Reverse Engineering*, IEEE Computer Society Press, October 29 - November 01, 2002, Richmond, Virginia, USA, pp. 97–107.

Greer, D. & Ruhe, G. (2004). Software release planning: an evolutionary and iterative approach, *Information and Software Technology* Vol. 46(No. 4): 243–253.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Software*, Addison Wesley.

Harman, M., Swift, S. & Mahdavi, K. (2005). An empirical study of the robustness of two module clustering fitness functions, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, IEEE Computer Society Press, 25-29 June 2004, Washington DC, USA, pp. 1029–1036.

Harman, M. & Tratt, L (2007). Pareto optimal search based refactoring at the design level, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO2007)*, ACM Press, July 7-11, 2007, London, UK, pp. 1106–1113.

O'Keefe, M. & O'Cinneide, M. (2006). Search-based software maintenance, *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, IEEE Computer Society, 22-24 March 2006, Bari, Italy, pp. 249–260.

Kirsopp, C., Shepperd, M. & Hart, J. (2002). Search heuristics, case-based reasoning and software project effort prediction, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, Morgan Kaufmann Publishers, 9-13 July 2002, San Francisco, CA, USA, pp. 1367–1374.

Kim, Y. & deWeck, O.L. (2005). Adaptive weighted-sum method for bi-objective optimization: Pareto front generation, *IEEE Transactions on Software EngineeringStructural and Multidisciplinary Optimization* Vol. 29(No. 2): 149–158.

Marinescu, R. (1998). Using object-oriented metrics for automatic design flaws in large scale systems, *Lecture Notes in Computer Science* Vol. 1543(No.): 252–253.

Mens, T. & Tourwe, T. (2003). Identifying refactoring opportunities using logic meta programming, *Proceedings of 7th European Conference on Software Maintenance and Re-engineering (CSMR2003)*, IEEE Computer Society Press, 26-28 March 2003, Benevento, Italy, pp. 91–100.

Mens, T. & Tourwe, T. (2004). A Survey of Software Refactoring, *IEEE Transactions on Software Engineering* Vol. 30(No. 2): 126–129.

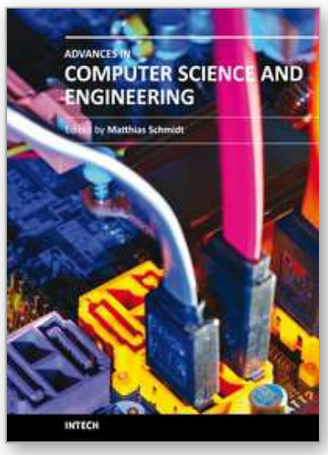
Mens, T., Taentzer, G. & Runge, O. (2007). Analysing refactoring dependencies using graph transformation, *Software and System Modeling* Vol. 6(No. 3): 269–285.

Simon, F., Steinbruckner, F. & Lewerentz, C. (2001). Metrics based refactoring, *Proceedings of European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, March 14-16, 2001, Lisbon, Portugal, pp. 30–38.

Seng, O., Stammel, J. & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of objectoriented systems, *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ACM Press, Seattle, Washington, USA, 2006, pp. 1909–1916.

Vescan, A. & Pop, H.F (2008). The Component Selection Problem as a Constraint Optimization Problem, *Proceedings of the Work In Progress Session of the 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques (Software Engineering Techniques in Progress)*, IEEE Computer Society Press, Wroclaw, Poland, pp. 203–211.

Zhang, Y., Harman, M. & Mansouri, S.A. (2007). The multi-objective next release problem, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO2007)*, ACM Press, London, UK, 2006, pp. 1129–1136.



Advances in Computer Science and Engineering

Edited by Dr. Matthias Schmidt

ISBN 978-953-307-173-2
Hard cover, 462 pages
Publisher InTech
Published online 22, March, 2011
Published in print edition March, 2011

The book Advances in Computer Science and Engineering constitutes the revised selection of 23 chapters written by scientists and researchers from all over the world. The chapters cover topics in the scientific fields of Applied Computing Techniques, Innovations in Mechanical Engineering, Electrical Engineering and Applications and Advances in Applied Modeling.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Camelia Chisăliță-Crețu (2011). The Multi-Objective Refactoring Set Selection Problem - A Solution Representation Analysis, Advances in Computer Science and Engineering, Dr. Matthias Schmidt (Ed.), ISBN: 978-953-307-173-2, InTech, Available from: http://www.intechopen.com/books/advances-in-computer-science-and-engineering/the-multi-objective-refactoring-set-selection-problem-a-solution-representation-analysis

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen