Babeş–Bolyai University
Faculty of Mathematics and Computer Science
1, M. Kogălniceanu Street
400084, Cluj–Napoca, România
http://www.ubbcluj.ro

Refactoring in Object-Oriented Modeling

PhD Thesis Abstract

PhD Student: Maria-Camelia Chisăliţă-Creţu
Supervisor: Prof. Dr. Militon Frenţiu

2010

To my beloved husband Coni,

Your incessant love and support give me strength!

To our son David,

The time spent with you is our greatest achievement!

ACKNOWLEDGMENTS

This thesis would have never been finished without the help of many people, to whom I would like to express my sincere gratitude.

First of all, I want to express my deeply-felt thanks to my adviser Prof. Militon Frenţiu for his continued encouragement and patience. He gave me the opportunity to work in an inspiring environment with the freedom to experience many different aspects of software engineering. He trusted and guided me throughout this entire thesis process, believing in my abilities to complete this work.

I am thankful to my parents for their encouragement and understanding. I am especially thankful to my beloved husband Coni and our son David. You have always been with me, supporting me and inspiring me to bring out the best in myself during my PhD programme. Your endless love and patience gave me the strength and motivation to finish this work.

I want to thank to the Computer Science Department staff and all my colleagues. Many thanks go to Prof. Bazil Pârv for valuable suggestions and stimulating discussions. I thank to the reviewers for offering to assess this dissertation and for the positive observations advanced. Special thanks to my dear colleague Andreea Vescan for always helping me when I have needed it. Your ideas and encouragement gave impulse to my research activity. Many thanks to my colleagues Cristina Mihăilă and Andreea Mihiş for the advises and availability. Many thanks also to all of the colleagues who have helped me and collaborated with me (Crina Groşan, Camelia Şerban, Anca Gog) during my work on the PhD thesis.

Most importantly, I thank God for granting me the skills and opportunities that made this possible.

List of Abbreviations

The following table describes the significance of various abbreviations used throughout the thesis. The page/section on which each one is defined or firstly used is also given.

Abbreviation	Meaning	Page/Section
UoD	Universe of Discourse	7/2.1
AST	Abstract Syntax Tree	10/3.1
CBO	Coupling Between Objects	15/4.4.2
RFC	Response For a Class	15/4.4.2
DIT	Depth of Inheritance Tree	15/4.4.2
NOM	Number Of Methods	15/4.4.2
NOC	Number Of Children	15/4.4.2
$LCOM^*$	Lack Of Cohesion in Methods	15/4.4.2
NOD	Number Of Descendants	15/4.4.2
TCC	Tight Class Cohesion	15/4.4.2
NOC^*	Number Of Classes	15/4.4.2
LOCC	Line Of Code per Class	15/4.4.2
WMC	Weighted Method per Class	15/4.4.2
SBSE	Search-Based Software Engineering	6/1.9
LANSP	Local Area Network Simulation Problem	6/1.11
DSCHP	Data Structure Class Hierarchy Problem	6/1.11
DAMP	Didactic Activity Management Problem	6/1.11
fca	forward conceptual abstraction	7/2.1
cs	conceptual specialization	7/2.1
GMORSP	General Multi-Objective Refactoring Selection Problem	16/5.3
MORSSP	Multi-Objective Refactoring Set Selection Problem	17/5.4.1
MORSgSP	Multi-Objective Refactoring Single Selection Problem	18/5.4.2
MORSqSP	Multi-Objective Refactoring Sequence Selection Problem	19/5.4.3
MORPBP	Multi-Objective Refactoring Plan Building Problem	20/5.4.4
RSSGARef	Refactoring Set Selection Genetic Algorithm Refactoring-based	23/6.1
RSSGAEnt	Refactoring Set Selection Genetic Algorithm Entity-based	23/6.1
RSgSGAEnt	Refactoring Single Selection Genetic Algorithm Entity-based	28/6.7

Keywords: Refactoring, Search–Based Software Engineering, Multi–Objective Optimization, Software Metrics.

Contents

Introduction

This PhD thesis is the result of my researches in Software Engineering, particular in the domain of Software Systems Refactoring, started in 2002.

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 [Fow99, MT04]. 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 [Fow99]. 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. Refactoring fits naturally in different development processes (spiral model process [Boe88], eXtreme Programming [Bec99]).

The goal of this thesis is to investigate the appropriate refactoring techniques application within different contexts. Research within conceptual modeling reveals the possibility to integrate the refactoring process in the analysis development phase. A biological evolution-based model is proposed in order to cope with different types of conceptual modeling variability.

A formalism that indicates the refactoring impact is introduced and specific refactorings are described following the proposed notations. Refactoring impact assessment on internal program quality may be used as an indicator of the transformation necessity. A formal description of software metrics and the corresponding changes due to the refactoring application are investigated. A multi-step analysis strategy is proposed to address the glide within a software metric value range after specific refactoring applications.

Different multi-objective refactoring selection problems are formally introduced. Therefore, the refactoring set selection, refactoring sequence selection, and refactoring plan building problems are defined as multi-objective optimization problems with two conflicting objectives.

This thesis focuses on the activity of the appropriate refactoring selection and the refactoring impact assessment on the internal program quality. It contains over 150 bibliographical references and is divided in seven chapters as follows.

First chapter provides an introduction to the field of Software Engineering and anchors the refactoring process within different software development methodologies together with the motivation and the activities for the investigated process. General issues related to the software maintenance and evolution are stated. Previous formal approaches on refactoring are reminded too. Conceptual modeling and the existing variability types are shortly reminded. Specific refactorings investigated within this research are informally described. The used evolutionary-based approach for the refactoring selection is described in detail. The source code for the case studies and their difficulties are shortly presented.

Chapter 2, **Refactoring in Conceptual Modeling Variability**, describes the three possible types of conceptual variability, as: *construct*, *vertical abstraction*, and *horizontal abstraction*. Identified shifting strategies, like refactoring, forward conceptual abstraction, and conceptual specialization are presented. A biological evolution-based model is proposed to describe the changes within the studied models, as *ontogenic* and *phylogenic* processes.

Chapter 3, **Formal Representation of Refactoring Impact**, formalizes the refactoring impact on the internal program structure representation as affected node and edge numbers within the studied AST. Several relevant refactorings, as: *MoveMethod*, *MoveField*, *ExtractClass*, and *InlineClass* are rigorously approached and formally described based on the new impact-based refactoring proposed notation. The advanced formalism is used to study the refactoring impact for the described refactorings on a source code case study.

Chapter 4, **Formalizing Software Metrics. A Refactoring Impact-Based Approach**, highlights the possibility to link within a two way connection the refactoring process and the internal quality assessment process, through software metrics, within a formal context. A consistent set of software metrics defined in the literature in the context of object-oriented approach is formally described. A technique for the analysis of the refactoring impact on the internal program quality, assessed by software metrics is proposed. In order to achieve this, a set of goals chased by the developer within the refactoring impact assessment process, a list of refactoring impact categories, and a list of assessment rules applied to obtain the final refactoring impact on the internal program quality metrics were investigated.

Chapter 5, **The Refactoring Selection Problem. A Formal Multi-Objective Optimization Approach**, addresses the formal definition of the *General Optimal Refactoring Selection Problem* with several specialized multi-objective refactoring selection problems. Two compound and conflicting objective functions are defined, as the *refactoring cost* and the *refactoring impact* on software entities. Therefore, the *Multi-Objective Refactoring Set Selection Problem* (MORSSP) together with its special case, the *Multi-Objective Refactoring Single Selection Problem* (MORSgSP) are introduced. These are followed by the *Multi-Objective Refactoring Sequence Selection Problem* (MORSqSP). Finally, the *Multi-Objective Refactoring Plan Building Problem* (MORPBP) is defined.

Chapter 6, **The Refactoring Selection Problem - An Evolutionary Approach**, investigates the *Multi-Objective Refactoring Set Selection Problem* following an evolutionary-based solution approach. For the proposed steady-state genetic algorithm, two solution representations are studied: the *refactoring-based* and the *entity-based* solution representations. As a special case of MORSSP, the *Multi-Objective Refactoring Single Selection Problem* (MORSgSP) is addressed with the corresponding entity-based solution representation on a genetic algorithm. The obtained results are analyzed based on a proposed refactoring strategy.

Chapter 7, **Conclusions and Future Research**, draws the main conclusions about our approaches. Several potential improvements of our work, together with specific aspects concerning tool validation are also addressed.

The original contributions introduced by this thesis are contained in Chapters 2, 3, 4, 5, and 6. The main contributions target three of the six activity steps identified for a complete refactoring process, as: choosing the appropriate refactorings to be applied, refactoring effect assessment on quality characteristics of the software, and maintaining the consistency between the refactored program code and other software artifacts. They are as follows:

- on the **consistency maintenance** variability:
  - A new set of refactorings recommended for each type of conceptual modeling variability [CC05b] (Subsection 2.1);
  - A new description approach for the conceptual modeling variability types with specialized actions to switch between variants [CC07] (Subsections 2.1.1, 2.1.2, and 2.1.3);
  - A new biological evolution-based model to cope with different types of variability within conceptual modeling [CC10c] (Subsection 2.2);
- on the **refactoring impact formalization and assessment**:
  - A new formalism to describe node and edge types within the internal structure representation as AST [CC10b] (Subsection 3.1);
  - A new formalism for the refactoring impact description, based on the internal structure representation as AST [CC05a] (Subsection 3.1);

- A new impact based formal description of various refactorings, as: *MoveMethod*, *MoveField*, *ExtractClass*, and *InlineClass* [CC05a] (Subsection 3.2);
- The application of the proposed impact-based formal definitions of the discussed refactorings to the DAMP (see Subsection 1.11.3) [CC05a] (Subsection 3.2);
- A new formal description of the software metrics, based on the internal structure representation as AST [CC10b], for a substantial set of software metrics defined in the literature in the context of object-oriented approach, grouped in four categories (coupling, cohesion, complexity, and abstraction) (Subsection 4.3);
- A new formal description of the refactoring impact on software metrics [CC10b], for a set of software metrics grouped in four categories (coupling, cohesion, complexity, and abstraction) (Subsection 4.3);
- A new proposed technique for the analysis of the refactoring impact on the internal program quality through software metrics [CCcC06, CC10e] (Subsection 4.4);
- A new set of goals chased by the developer and a list of refactoring impact categories within the internal quality assessment process [CC10e] (Subsection 4.4.1);
- A new set of assessment rules applied to obtain the final refactoring impact on the internal program through quality metrics [CC10e] (Subsection 4.4.1);
- The analysis approach validation achieved by two experiments runs [CC10e] (Subsection 4.4.2);

- on the **refactoring selection**:
  - New formal descriptions of different refactoring selection problems as multi-objective optimization problems:
    - The General Multi-Objective Refactoring Selection Problem (GMORSP) [CC11] (Subsection 5.3);
    - The Multi-Objective Refactoring Set Selection Problem (MORSSP) [CC09c] (Subsection 5.4.1);
    - The Multi-Objective Single Refactoring Selection Problem (MORSgSP) [CCV09a, CCV09b] (Subsection 5.4.2);
    - The Multi-Objective Refactoring Sequence Selection Problem (MORSqSP) [CC10f] (Subsection 5.4.3);
    - The Multi-Objective Refactoring Plan Building Problem (MORPBP) [CC10f] (Subsection 5.4.4);
  - New strategic aspects faced by the project management leadership that rise up within the refactoring plan building process [CC10f] (Subsection 5.4.4);

- on the **evolutionary-based approaches for MORSSP and MORSgSP**:
  - A new evolutionary-based solution approach for the MORSSP, two different solution representations being used, as: refactoring-based [CC09c, CC09a] and entity-based [CC09b, CC09e, CC09d] solution representations for the proposed genetic algorithm (Subsections 6.2.1 and 6.2.2);
  - A new evolutionary-based solution approach for the MORSgSP, with an entity-based [CC10d] solution representation for the proposed genetic algorithm (Subsection 6.7);
  - A new strategy to assess the suggested solution improvements, based on the case study identified difficulties [CC10a, CC11] (Subsection 6.3.1).

1 Setting the context

The chapter explores the background of this research, with the aim of placing its contributions in the context. An introduction to the field of Software Engineering by focusing on the refactoring process is provided.

1.1 Software Maintenance and Evolution

A software system evolves from its entire life cycle. *Software maintenance* is one of the key issues in the overall software construction and management. Each software enhancement, bug correction or adapting the software to new requirements makes the software more complex, changes its original design and decreases the software quality. It became a part of the IEEE 1219 Standard for Software Maintenance [IEE99].

Definition 1.1.1([IEE99]) ***Software maintenance*** *is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.*

Swanson [Swa76] and Sommerville [Som96] describe three kinds of software maintenance and based on their work the ISO/IEC Standard 14764 [ISO99], maintenance is subdivided into four categories: corrective, adaptive, perfective, and preventive.

Preventive maintenance that does not alter the system functionality, is also referred to as *anti–regressive work*. The term was introduced by Lehman [LR00] to describe the work done in order to decrease the complexity of a program without altering the functionality of the system as it is perceived by the users. Anti-regressive work covers activities such as code rewriting, refactoring, reengineering, restructuring, redocumenting.

Nowadays more than 80% of total software life-cycle costs is devoted to its maintenance [Pig97] and it is known that software maintenance encompasses activities and processes involving existing software not only after its delivery but also during its development.

Within the well-known *waterfall model* [Roy70] software development process, the maintenance phase is the final phase of the life-cycle of a software system, after its deployment. According to Roy, only bug fixes and minor adjustments to the software are supposed to take place during this maintenance post-production phase.

Such a sequential decomposition as a single directed flow of activities prohibits the necessary interaction and feedback required by software development. This is addressed by more modern development models such as the *spiral model* [Boe88] and *eXtreme Programming* [Bec99].

1.2 Refactoring

1.2.1 Definitions

The term *refactoring* is originated from a Deutch's quote, who wrote "interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or recreate than code" [Deu89].

A software system may be *factored repeatedly*, calling it refactoring. Constantine [CY79] has introduced the term of *object factoring* in the context of object-oriented.

The first definition of refactoring mentioned by the literature was in Opdyke's PhD thesis [Opd92], though it was used well before this. Almost in the same time Chikofsky has defined the restructuring notion [CC90].

Definition 1.2.1([CC90]) ***Restructuring*** *is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject systems external behaviour (functionality and semantics).*

While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system [CC90].

Definition 1.2.2([Opd92]) ***Refactorings*** *are behaviour-preserving program restructurings specific to supporting the design, evolution and reuse of object-oriented application frameworks.* ***Refactorings*** *are transformations that do not change the behaviour of a program; that is, if the program is called twice (before and after refactoring) with the same set of inputs, the resulting set of output values will be the same.*

Restructuring and refactoring meaning have evolved from *very similar* to *different* terms. Restructuring was considered to be the term used for structured programming, while refactoring the term used for the object-oriented programming. Though, the term restructuring has a broader meaning, which does not demand the preservation of systems external behaviour, like refactoring does. Later, Fowler [Fow04] states that refactoring is a "very specific technique to do the more general activity of restructuring (...) founded on using small behaviour-preserving transformations (themselves called refactorings)."

In his PhD thesis, Roberts [Rob99] updates the refactoring definition. His definition covers both behaviour-preserving and non-behaviour-preserving transformations.

Definition 1.2.3([Rob99]) *A* ***refactoring*** *is an ordered triple $R = (pre, T, P)$ where pre is an assertion that must be true on a program for R to be legal, T is the program transformation, and P is a function from assertions to assertions that transforms legal assertions whenever T transforms programs.*

The most popular refactoring definition was provided by Fowler [Fow99].

Definition 1.2.4([Fow99]) ***Refactoring*** *(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.* ***Refactor*** *(verb): to restructure the software by applying a series of refactorings without changing its observable behaviour.*

1.2.2 Motivation and Activities

This subsection provides a motivation for the refactoring process as it was advocated by various researchers [Opd92, Fow99, Bec99, MT04]. The way the refactoring process may be integrated to different software processes is presented too (*spiral model* [Boe88], *eXtreme Programming* [Bec99]). Refactoring activities suggested by different research groups are approached by this subsection [MT04, KIAF02]. Several class-based object-oriented refactoring classifications based on various criteria are provided [Opd92, Rob99, Fow99, Ker04].

1.3 Studied Refactorings

This section shortly describes those refactorings that were used within our research. Refactorings from different categories have been selected in order to highlight aspects of the abstract syntax tree processing and refactoring selection.

1.4 Conceptual Modeling

Conceptual modeling is the first step in the requirements analysis phase and uses the information gathered and identified in the previous phase of the development cycle [Boe88, CY79]. The *variability* [Ver04] in the context of conceptual modeling means the possibility to build distinct and still correct conceptual models for the same set of requirements. Such conceptual model is called *variant*. A non-exhaustive framework of three types of variability was used in order to advocate the possibility to integrate refactoring in the conceptual modeling.

1.5 Formalizing Refactoring Using Graph Transformations

This section reminds a graph representation of an object oriented program presented in [MVEDJ05] required to formally describe the transformations applied through refactoring. Following this formalism, Chapter 3 introduces formal notations that express the refactoring impact on the program internal structure, while Chapter 4 advances a new assessing approach for the refactoring effect on the internal quality reflected by changes on software metrics.

1.6 Software Quality Attributes

Following the IEEE Standard definition for the *software (product) quality* [IEE92] and the *software quality characteristics* [ISO91] it may be stated that defining software quality for a system is equivalent to define a list of software quality attributes for that system.

Our research focuses on the quality attributes of the software products. A brief overview of the internal quality attributes is provided here as low coupling, high cohesion, manageable complexity, and appropriate data abstraction [Mar02].

1.7 Software Metrics

This section presents the metrics defined in the literature in the context of object-oriented approach, that were chosen for our research as coupling measures, class hierarchy coupling measures, cohesion measures, and size and complexity measures [Mar02].

1.8 Class Hierarchy Code Duplication Types

Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems [Joh93, Bak95]. Code clones are considered as one of the bad smells of a software system [Fow99]. This section shortly reminds a code duplication classification in class hierarchies [KN01] together with the recommended refactorings to remove them.

1.9 The Multi-Objective Refactoring Selection Problem

This section describes the main aspects of the *Multi-Objective Refactoring Selection Problem*. The general context of the Search-Based Software Engineering (SBSE) together with the refactoring selection subdomain and some related work results are advanced. Key concepts defining and bounding both multi-objective optimization problems and the corresponding optimization methods are presented here.

1.10 Artificial Intelligence-Based Solution Approach

There are many approaches within the SBSE domain related to the search-based refactoring selection. This section presents knowledge from the artificial intelligence domain needed to solve the *Multi-Objective Refactoring Selection Problem* defined by Chapter 5. Various aspects of the selected genetic algorithm and the used genetic operators are shortly approached. The *steady–state evolutionary model* used within the research is explicitly presented. The data normalization methods used are shortly reminded by this section.

1.11 Case Studies for the Discussed Problems

Three source code case studies problems are described in this section: Local Area Network Simulation Problem (LANSP), Data Structure Class Hierarchy Problem (DSCHP), and an extract from a Didactic Activity Management Problem (DAMP). They are used to emphasize various aspects related to the applied refactoring within our research.

2 Refactoring in Conceptual Modeling Variability

Refactoring application in the context of conceptual modeling variability was investigated in this chapter. Various types of changes that may consist of specific refactoring techniques are suggested. A formal biological evolution-based model to describe the comprising processes is introduced.

2.1 Conceptual Modeling Variability Types

The *variability* [Ver04] in the context of conceptual modeling means the possibility to build distinct and still correct conceptual models for the same set of requirements. Such conceptual model is called *variant*.

Within conceptual modeling variability, refactoring has proved to be a feasible technique to switch between variants. The identified changes, i.e., refactorings (r), conceptual specialization (cs), and forward conceptual abstraction (fca), are recommended for specific examples investigated for each variability situation.

2.1.1 Construct Variability

Definition 2.1.1([Ver04]) ***Construct variability*** *represents the possibility of modeling concepts in the UoD using different constructs in the same modeling language.*

Suggested Refactorings

Within construct variability, the concepts within UoD have the same semantics in all variants. They are represented by a class (entity), an attribute, a relationship. There are several types of refactorings that may be applied, corresponding to different types of construct variability studied:

- *use class (entity) instead of a set of attributes (properties)*;
- *use a method instead of an attribute*;
- *use derived classes instead of type codes.*

The effort required to switch between the variants is reduced by the application of a limited number of small refactorings. This type of variability is exploited in the shift from object-oriented analysis to design. As a consequence, it is expected that *construct variability* had been already used refactoring in current modeling activities. Other relevant results in refactoring conceptual models show that is unlikely that hard obstacles for this transformations between construct variants will be found [DDN00, DB04, DBM03].

2.1.2 Vertical Abstraction Variability

Definition 2.1.2([Ver04]) ***Vertical abstraction variability*** *refers to the possibility of modeling concepts in the UoD in a more or less generic (abstract) way.*

Suggested Refactorings

There are two ways to navigate over the vertical abstraction variability. The first one refers to the possibility to switch from a general conceptual model to a concrete model, while the other one increases the abstraction level by removing concrete aspects, or by adding various parameters. In [CC05a] refactoring categories needed to switch between models are identified and described. Following the two types of transformation that occur in vertical abstraction variability, appropriate solutions were provided when:

- *transforming to a more generic variant*;
- *transforming to a more specific variant.*

Forward conceptual abstraction is needed to identify and implement new concepts and specialized behaviour that transform the concrete models to more generic ones. Refactoring techniques have limited usage in this navigation way of the vertical abstraction variability. The new variants have the advantage of a raised adaptability and flexibility.

This type of variability was observed in the process of simplifying the design of already existing software systems, the so called over engineered systems [Par79, GP95, FS97, DDN00]. Changes to the system are made more easily if the conceptual model is more general and consequently, more difficult if the conceptual model is too simple or too concrete.

2.1.3 Horizontal Abstraction Variability

Definition 2.1.3([Ver04]) ***Horizontal abstraction variability*** *refers to the possibility of modeling concepts in the UoD based on different properties.*

Suggested Refactorings

The shift between a *cmA* variant and a *cmB* variant may be done using a *cmC* variant as an intermediate variant. In order to achieve it a two phases process has to be implemented:

- *Step 1:* establish an equivalency relationship between the two dimensions, by transforming the *cmA* variant to the *cmC* variant, consisting of the following aspects:
  1. *add new classes to specialize*;
  2. *responsibility reassignment*;
  3. *add new class to generalize*;
- *Step 2:* allow to keep the requested primary dimension only, by studying the following elements:
  1. *responsibility reassignment*;
  2. *safely removal of the specialized classes.*

In horizontal abstraction variability the concepts from the UoD are modeled using different semantic definitions, while the differences between variants bear upon concepts modeled based on different properties.

In the horizontal abstraction variability, the properties may be or not *visible* and *isolated*. They are classified as *primary dimension properties* (that can be visible and isolated from others) and *secondary dimension properties* (that are not visible and cannot be isolated from others)[Ver04].

2.2 A Model for Conceptual Modeling Variability Evolution

Variability within conceptual modeling outlines an evolutionary process among different models of a specific variability type. This process is similar to the biological evolutionary process presented by Maturana and Varela in [MV98].

For a software product, customers may require new functionalities to be implemented. This results in changes that serve as perturbation in the software product evolution. In order to achieve variability within conceptual modeling, changes provided by refactoring, forward conceptual abstraction, conceptual specialization or other evolutionary changes have to be applied. There are two types of evolution in biology: *phylogeny* and *ontogeny* [MV98]. The former refers to the evolution as species while the latter refers to the evolution of individual living beings.

For the already studied conceptual modeling variability with its three different types, i.e., construct, vertical abstraction, and horizontal abstraction, an evolution model was developed.

2.2.1 Conceptual Modeling Variability as Biological Evolution

Evolution in Construct Variability

The ontogenic evolution for conceptual models that are perturbated by small changes does not fundamentally affect the developed model. These changes correspond to switches between *attributes and entities* or *attributes and methods* approaches. The *multiple types definition* within construct variability expresses a phylogenic evolution process by the addition (forward conceptual abstraction) or removal (conceptual specialization) of types to the conceptual model.

Evolution in Vertical Abstraction Variability

Reducing the abstraction level for a conceptual model means to remove superfluous information in order to shape a more concrete conceptual model. The simplifying process consists of refactorings that remove the irrelevant information in the target model. *Raising the abstraction level* requires additional information gathered by forward conceptual abstraction.

Evolution in Horizontal Abstraction Variability

The phylogenic evolution within this type of variability appears at the first shifting step, by *the addition of a new visibility dimension* to the model, which drives the complexity of the development process to a higher level through forward conceptual abstraction. In order *to reduce the number of visible dimensions*, refactoring may be applied to a model within a conceptual specialization.

2.2.2 Formal Approach

This section formally introduces the basic aspects used to formalize the conceptual modeling variability as an ontogenic and phylogenic evolution. Therefore, the *conceptual model*, *refactoring*, *forward conceptual abstraction*, and *conceptual specialization* are formally defined.

Following the notions previously introduced, the *ontogenic* and *phylogenic* processes are formally described. The three types of conceptual modeling variability are formally defined as biological evolution processes, following the ontogenic and phylogenic principles.

The existing relations among various conceptual models within the same or different extension stages of the development process is formalized too. Consequently, the *ontogenic equivalence* and *phylogenic dominance* are formally defined. The notions defined for our conceptual modeling variability are summarized by the Table 1.

<table>
<tr><td rowspan="2">Type of evolution</td><td rowspan="2">Achieved by</td><td colspan="3">Type of conceptual modeling variability</td></tr>
<tr><td>Construct</td><td>Vertical abstraction</td><td>Horizontal abstraction</td></tr>
<tr><td rowspan="2">ontogenic</td><td rowspan="2">r</td><td>E – A</td><td rowspan="2">-</td><td rowspan="2">-</td></tr>
<tr><td>A – M</td></tr>
<tr><td rowspan="2">phylogenic</td><td>fca</td><td rowspan="2">Type Codes</td><td>Concrete to generic</td><td>Add primary dimension</td></tr>
<tr><td>cs</td><td>Generic to concrete</td><td>Remove secondary dimension</td></tr>
</table>

Table 1: Conceptual modeling variability as *ontogenic* and *phylogenic* evolution processes

2.3 Conclusions and Future Work

Variability occurs in almost every modeling activity and its exploitation may help modelers to switch between taken decisions and to validate the model equivalence. Refactoring techniques are dedicated to design and implementation phase, but the research shows that their applicability may be extended to the conceptual modeling level. The original results presented in this chapter have been reported in the papers [CC05b, CC07, CC10c].

3 Formal Representation of Refactoring Impact

The refactoring impact on the internal program structure representation is expressed as the affected node and edge number within the studied AST. New formalisms on refactoring description are approached.

3.1 Formal Representation of the Refactoring Impact on the Internal Structure of the Source Code

This section formally introduces new notions on the internal structure representation, reflected by changes on the AST. Based on the notations introduced in [DBM03], additional notations are needed to formalize the refactoring impact on the node and edge types within the corresponding AST of the source code.

3.1.1 Node and Edge Type-Based Formalisms on AST

Definition 3.1.1([CC10b]) *Let $\Sigma_c = \{M, A, P, L, E\}$ be a set of possible node types to whom a Class node type c may be connected through edges of different edge types and $T(c)$ the corresponding AST. Then, the* ***node type total number*** *for a node type $X, X \in \Sigma_c$, denoted by $\#X(T(c))$, is the total number of nodes sharing the node type X within $T(c)$.*

Definition 3.1.2([CC10b]) *Let $\Gamma_{cls} = \{i, t, c, a, u\}$ be the set of all possible edge types to whom a Class node type cls may be connected through nodes of different node types, ϵ a regular expression over the Γ_{cls} and $T(cls)$ be the corresponding AST. Then, the* ***edge type total number*** *for an edge type $\gamma, \gamma \in \Gamma_{cls}$, denoted by $\#ET(cls, \gamma)$, is the total number of edges sharing the edge type $\gamma, \gamma \in \Gamma_{cls}$ or the regular expressions ϵ over Γ_{cls} that contain the γ literal, incident to $T(cls)$.*

3.1.2 Node and Edge Type-Based Formalisms for the Refactoring Impact

Definition 3.1.3([CC10b]) *Let $\Sigma_c = \{M, A, P, L, E\}$ be a set of possible node types to whom a Class node type c may be connected through edges of different edge types, $T(c)$ the corresponding AST, r an applied refactoring to $T(c)$, $X(T(c), r+)$ the set of added nodes to the $T(c)$ after the r refactoring is applied, and $X(T(c), r-)$ the set of removed nodes from the $T(c)$ after the r refactoring is applied. Then, for a node type $X, X \in \Sigma_c$ are defined:*

i. ***addition node type refactoring impact number****, denoted by $\#X(T(c), r+)$, is an integer value that expresses the number of X type nodes added to the $T(c)$, after the r refactoring is applied;*
ii. ***removal node type refactoring impact number****, denoted by $\#X(T(c), r-)$, is an integer value that expresses the number of X type nodes removed from the $T(c)$, after the r refactoring is applied.*

Definition 3.1.4([CC10b]) *Let $\Gamma_{cls} = \{i, t, c, a, u\}$ be the set of all possible edge types to whom a Class node type cls may be connected through nodes of different node types, $T(cls)$ the corresponding AST, r an applied refactoring to $T(cls)$, $ET(cls, \gamma, r+)$ set of added γ type edges to the $T(cls)$ and $ET(cls, \gamma, r-)$ the set of removed γ type edges from the $T(cls)$, $\gamma \in T(cls)$. Then, for an edge type $\gamma, \gamma \in \Gamma_{cls}$, are defined:*

i. ***addition edge type refactoring impact number****, denoted by $\#ET(cls, \gamma, r+)$, is an integer value that expresses the number of γ type edges added to the $T(cls)$, after the r refactoring is applied;*
ii. ***removal edge type refactoring impact number****, denoted by $\#ET(cls, \gamma, r-)$, is an integer value that expresses the number of γ type edges removed from the $T(cls)$, after the r refactoring is applied.*

3.2 An Impact-Based Approach Refactoring Description

In order to formally describe refactorings, an impact-based approach was investigated. The tackled refactorings used within this research are: *MoveMethod*, *MoveField*, *ExtractClass* and *InlineClass*. The studied procedure follows the notations introduced in [DBM03] and the refactoring impact expressed as affected node and edge number on the internal structure representation as AST. Therefore, there are several aspects related to the refactoring impact that *cannot be computed* or expressed in AST terms.

Source code Experiment. The source code used for this study is presented in Subsection 1.11.1. The *MoveMethod*, *MoveField*, *ExtractClass*, and *InlineClass* refactorings applications were studied for it. For each of them a formal definition of the impact on the AST is given, based of the notations presented in Section 1.5. The identified formulas are applied for each affected class *before* and *after* each refactoring application.

Subsection 3.2.1 approaches the *MoveMethod* refactoring. Based on the proposed formalism on the program internal structure representation, the refactoring impact on the AST for the source class A and for the target class B was formalized as consequences and two propositions were defined. In Subsection 3.2.2 the *MoveField* refactoring impact for a moved attribute `attr` from a source class `A` to a target class `B` is formally described. The *ExtractClass* and the *InlineClass* refactorings impact are formally defined by Subsections 3.2.3 and 3.2.4.

3.3 Conclusions and Future Work

In order to formally describe refactorings, an impact-based approach was investigated. The studied procedure follows the notations previously introduced by [DBM03] on the internal structure representation as AST.

This chapter formally introduces the refactoring impact on the internal structure reflected by changes on the AST. Formal notations required to count the number of affected node and edge types and to formally assess the refactoring impact on the node and edge types within the AST were introduced, based on the papers [CC05a, CC10b].

Further research may be done within the following directions: a catalog with the formal refactoring impact on the internal structure representation as AST for other relevant refactorings, from different categories and a deeper analysis for the compound refactorings where some changes on the internal structure may be reversed by another subsequently refactorings.

4 Formalizing Software Metrics. A Refactoring Impact-Based Approach

Software metrics have become an essential instrument in some disciplines of software engineering. They are used to assess the quality and complexity of software systems, as well as to get a basic understanding and providing clues about sensitive parts of software systems. There are many software metrics informally defined in the literature. We give here a formal definition of the software metrics, even if we use a small number of them. Some of the defined software metrics are used in this chapter. A formal description of software metrics and the corresponding changes due to the refactoring application are introduced here. A multi-step analysis strategy is advanced to address the glide within a software metric value range after specific refactoring applications.

4.1 Object-Oriented Software Metrics in the Refactoring Process

Software metrics are typically used as internal quality factors [FP97]. Therefore, refactoring impact assessment on internal program quality may be used as an indicator of the transformation necessity through refactoring usage. Figure 1 shows a two way connection between the refactoring process and the quality assessment process, through software metrics, within a formal context, e.g., the source code AST.

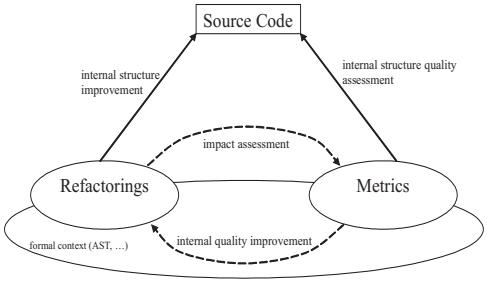

Figure 1: The connection between the *refactoring process* and the software assessment process (*metrics*)

Mens and Tourwe [MT04] suggest that within the six steps refactoring activities list, software metrics may be used *after* the refactoring process itself, as a means to assess the quality attributes of the changed software. Following the approach proposed by Kataoka et al. [KIAF02], the assessment of the refactoring effect should be carried out *before* the effective application of refactoring(s). Therefore, software metrics prove their relevance twice in the refactoring process.

The two way connection between refactorings and software metrics, the existing formalisms on internal structure representation, and the corresponding formal refactoring description suggest a bias towards formalizing software metrics.

4.2 Formalizing Object-Oriented Software Metrics

The formalism introduced to describe the node and edge type number of the internal program structure represented as an AST (see Section 3.1) will be used to express the object-oriented software metrics.

The software metrics defined in the literature in the context of object-oriented approach, were grouped in four category: coupling, cohesion, complexity, and abstraction, presented in [Mar02]. Based on the notation introduced in Section 3.1, the measures are formally defined.

Following the informally description for the coupling type measures in [Mar02] and the notation from [CC10b], Subsection 4.2.1 formalizes the access coupling, method coupling, service coupling, class hierarchy coupling, and system level coupling software metrics.

The research within object-oriented cohesion started with the lack of cohesion study covering later other aspects like tight and loose, directly and indirectly method connectness. Subsection 4.2.2 formally describes the cohesion measures presented in [Mar02].

The system and class level metrics used to assess the structural complexity and to find classes that are exceedingly large or complex are formally investigated in Subsection 4.2.3.

4.3 Refactoring Impact on Object-Oriented Software Metrics Formal Description

In [DBM03] it is suggested that object-oriented program quality metrics may be specified in terms of the extended AST of the program structure. Following the notations presented in [CC05a], the formal refactoring impact on the program structure (see Section 3.1) is translated into the refactoring impact on the object-oriented program quality metrics. The authors of [ML02] introduce a graph-based formalism on object-oriented software metrics, also used in [CC05a].

The refactoring formal description and the formal definition of a representative set of object-oriented program quality metrics (discussed in [Mar02]) provides feedback on the impact of application of a specific refactoring to any particular internal program quality metric, before effectively applying it. Defining metrics in terms of the entities of the AST, formal descriptions of refactorings are projected into impacts on the particular metrics.

For the formally introduced coupling type measures in [Mar02] in Subsection 4.2.1 and the notation from [CC10b], the refactoring impact on access coupling, method coupling, service coupling, class hierarchy coupling, and system level coupling measures are defined as consequences of the refactoring process in Subsection 4.3.1.

Following the formally cohesion measures in [Mar02], introduced by Subsection 4.2.2 and the notation from [CC10b], the Subsection 4.3.2 defines the refactoring impact on the cohesion measures as consequences of the refactoring process.

For the formally introduced size and complexity type measures in Subsection 4.2.3 and the notation from [CC10b], the refactoring impact on system size, class size, and structural complexity measures are defined as consequences of the refactoring process in Subsection 4.3.3.

4.4 Refactoring Impact on Software Metrics Analysis

The definitions provided by Section 3.1 suggests formal description of the changes on the AST after refactoring application. By translating the changes on the program structure into changes on different software metrics, the refactoring impact on the internal program quality metrics is obtained.

Analogous to the approach presented in [DBM03, TK03], the research in [CCcC06, CC10e] proposes a set of possible impacts on a software metric. Therefore, software metric computation results in three possible ranks: ***high*** ($\triangle$), ***low*** ($\triangledown$), and ***irrelevant*** ($\bigcirc$).

For each software metric formal definition (see Section 4.2) one or more defining terms may be identified. The immediate refactoring impact on these terms may be expressed in three categories: ***null*** (0), ***positive*** (+), and ***negative*** (−).

Extending over the two types of impact, i.e., on *the assumed rank* and on *the raw value*, the final refactoring impact on software metrics is depicted by the Table 2. Thus, there are three impact categories together with the expected value range for a specific metric.

Assumed rank	**Impact on software metric, Value Range**		
	null, 0	*positive*, $(0, \infty)$	*negative*, $(-\infty, 0)$
$\triangle$	$\triangle_{\bullet}$ *high value expected, but not affected* (***no impact***)	$\triangle_{+}$ *high value expected, and increases* (***positive impact***)	$\triangle_{-}$ *high value and decreases* (***negative impact***)
$\triangledown$	$\triangledown_{\bullet}$ *low value expected, but not affected* (***no impact***)	$\triangledown_{+}$ *low value expected and increases* (***negative impact***)	$\triangledown_{-}$ *low value expected and decreases* (***positive impact***)
$\bigcirc$	$\odot$ ***no impact***	$\oplus$ ***positive impact***	$\ominus$ ***negative impact***

Table 2: The impact categories on software metrics

4.4.1 Impact Assessment Phases

In order to achieve the impact analysis, there are several steps that have to be followed. The identified refactoring impact category (*null*, *positive*, *negative*) for the identified software metric terms are used together with some assessment rules to obtain the final refactoring impact on the internal program quality metric, as described by the Table 2. The evaluation rules are described in [CC10e].

The impact analysis consists of the following steps:

1. choose the source code that will be studied for the refactoring impact on the internal structure with internal quality software metrics, denoted by SC;
2. choose a relevant set of refactorings that serves to improve the internal structure program, denoted by $SR = \{r_1, r_2, \ldots, r_n\}$;
3. choose the appropriate set of software metrics that assess the requested internal program qualities, as $SM = \{sm_1, sm_2, \ldots, sm_m\}$, where $\forall sm_i, sm_i \in SM$ is defined as a set of terms $sm_i = \{t_{i_1}, t_{i_2}, \ldots, t_{i_k}\}, k \in \mathcal{N}$, that will be used to assess the corresponding $sm_i, i = \overline{1, m}$;
4. build an impact table, n rows and $k+1$ columns, for a software metric $sm_i, i = \overline{1, m}$, with the following items:
   (a) each table row contains a refactoring r_i intended to be applied to the source code SC, where $r_i \in SR,\ i = \overline{1, n}$;
   (b) each table column contains a term $t_{i_j}, j = \overline{1, k}$, of the current analyzed software metric $sm_i, i = \overline{1, m}$, filled with the immediate refactoring impact;
   (c) the last table column data ($col_{i_{k+1}}$) will be computed based on the assessment rules;
5. build a refactoring impact table, n rows and m columns, filled with the corresponding $col_{i_{k+1}}$ for the chosen set of software metrics from the SM set to reflect the final impact provided by each refactoring application from the SR set.

4.4.2 Impact Analysis Approach Validation

Based on the software metrics formal description presented in Subsection 4.4.1, the impact analysis approach may be validated. The goal is to prove that the software metric impact suggested by the current approach is similar to the one indicated by the corresponding formal definition. To achieve this validation two experiments were run as presented in [CC10e].

Experiment 1. The goal is to prove that for a set of chosen set of refactorings and a set of software metrics, the refactoring impact on the software metrics which is computed based

on the proposed approach is similar to the results advanced by the corresponding refactoring impact formal definitions on the software metrics. For the four refactoring techniques formally defined in Section 3.1, i.e., *MoveMethod, MoveField, ExtractClass*, and *InlineClass* refactorings, a set of relevant software metrics have been chosen: *NOM, NOC, CBO, RFC*, and $LCOM^*$, that were formally defined in Section 4.2. Tables E.1, E.2, E.3, E.4, and E.5 in **Appendix E** present the refactoring impact on the chosen software metrics for the source class A and the target class B, for a targeted rank on the value range ($\triangle$, $\bigtriangledown$, or $\bigcirc$).

Experiment 2. It validates the proposed approach studying a set of refactorings and their impact on a set of software metrics for the DSCH Problem (see Subsection 1.11.2). The refactoring opportunities are possible code clones from a class hierarchy (see Section 1.8). In [CCcC06] the validation for the proposed approach was achieved for the code redundancy removal problem. For the DSCH Problem were identified and removed six code clone types (see Section 1.8). The software metrics set aims several aspects of internal program quality, as: class hierarchy weight (NOC^* metric, applied to the studied system), size and complexity of a class (LOCC, NOM, WMC), coupling (CBO, RFC), cohesion (TCC), and class hierarchy coupling (DIT, NOC, NOD). The internal program quality assessment with software metrics has been achieved before and after the redundancy problem type had been removed. Tables F.1, F.2, F.3, F.4, F.5, and F.6 in **Appendix F** present the refactoring impact on the chosen software metrics for the affected classes.

4.4.3 Proposed Approach Limitations

For several software metrics no formal definition of the refactoring impact is available (CC, SIZE1, SIZE2). Therefore, the model representation for the internal program structure may be extended. The main advantage of such an intention facilitates the formal definition for different aspects related to the flow control and complexity, since the model becomes more weighted, resulting in a difficult management.

Another limitation is represented by the assessment difficulty when the composing terms of the software metric definition have different immediate impacts, i.e., few negative impact ($-$) and few positive impact ($+$). These cases require a more complex analysis since the studied source code AST has to be semantically evaluated if the addition or removal of certain typed edges has impact on the formal definition software metric term. This represents an important aspect of the approach, as the formal definition explicitly depends on typed edge sets.

4.5 Conclusions and Future Work

The refactoring impact assessment on the internal program quality may be used as an indicator of the transformation necessity. The refactoring formal description (see Chapter 3) and the formal definition of a representative set of object-oriented program quality metrics in [CC10b] provides feedback on the impact of the application of a specific refactoring on any particular internal program quality metric, before effectively applying it.

To achieve a refactoring impact analysis on software metrics, a five steps strategy was advanced. The analysis allows to establish the glide within a software metric value range after one or more specific refactoring applications.

The chapter is based on the published papers [CCcC06, CC10b, CC10e]. The aspects to be analyzed in the near future are: internal program structure model improvement like: flow control information, class feature complete specification; formal definition for the code duplication types within a class hierarchy; the automation of the refactoring impact analysis process, essential when dealing with the large amount of combinations between refactorings and software metrics.

5 The Refactoring Selection Problem. A Formal Multi-Objective Optimization Approach

Formal definitions of the *Refactoring Selection Problem* with several variants are approached in this chapter. The *Refactoring Selection Problem* is modeled as a multi-objective optimization problem here, while an evolutionary-based solution, where several criteria are aggregated in a single objective function is investigated in Chapter 6.

5.1 General Background

This section sets the general context of the refactoring selection research problem, by presenting the motivation and a working scenario.

5.2 Related Work

Related work to the studied problem is reminded in this section.

5.3 The Formal Multi-Objective Refactoring Selection Problem Definition

In order to state the *General Multi-Objective Refactoring Selection Problem* (*GMORSP*) some notion and characteristics were defined. $SE = \{e_1, \ldots, e_m\}$ is the 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. 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, defined as:
$SED = \{\mathbf{u}ses\mathbf{A}ttribute, \mathbf{c}alls\mathbf{M}ethod, \mathbf{s}uper\mathbf{C}lass, \mathbf{a}ssociatedwith\mathbf{C}lass, \mathbf{n}o\mathbf{D}ependency\}$,
$ed : SE \times SE \rightarrow SED$,

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

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

A set of possible relevant chosen refactorings applied to different types of software entities of SE is $SR = \{r_1, \ldots, r_t\}$. 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 on the set of software entities SE, is defined as: $ra : SR \times SE \rightarrow \{\mathbf{T}rue, \mathbf{F}alse\}$,

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

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

The various dependencies between refactorings when they are applied to the same software entity are emphasized by the following mapping:
$SRD = \{\mathbf{B}efore, \mathbf{A}fter, \mathbf{A}lways\mathbf{B}efore, \mathbf{A}lways\mathbf{A}fter, \mathbf{N}ever, \mathbf{W}henever\}$,

$rd : SR \times SR \times SE \to SRD$,

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

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

- $f_1,\ f_2, \ldots,\ f_M$ – M objective functions, where $f_i : DS \to \mathcal{R}$, $i = \overline{1, M}$, 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 $f_i : DS \to \mathcal{R}$, $i = \overline{1, M}$, $g_j(\vec{x}) \geq 0, j = \overline{1, J}$, $h_k(\vec{x}) = 0, k = \overline{1, K}$, $\vec{x} \in DS$.

Multi-objective optimization often means optimizing conflicting goals. For the GMORSP formulation there may be the possibility to blend different types of objectives, i.e., some of them to be maximized and some of them to be minimized.

5.4 Specific Multi-Objective Refactoring Selection Problem Definitions

5.4.1 The Multi-Objective Refactoring Set Selection Problem

The *Multi-Objective Refactoring Set Selection Problem* (*MORSSP*) is a special case of the refactoring selection problem. Its definition [CC09c] follows the *General Multi-Objective Refactoring Selection Problem* (see Section 5.3). Two compound and conflicting objective functions are defined: the *refactoring cost* and the *refactoring impact* on software entities. In order to state the MORSSP some additional notions and characteristics have to be defined.

Additional Notions

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 effort involved by each transformation is converted to cost, being described as $rc : SR \times SE \to Z$,

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

where the ra mapping is defined by the formula 2 (see Section 5.3), $1 \leq l \leq t$, $1 \leq i \leq m$.

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 as $effect : SR \times SE \to Z$,

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

where the ra mapping is defined by the formula 2 (see Section 5.3), $1 \leq l \leq t$, $1 \leq i \leq m$.

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

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

where $1 \leq l \leq t$ and w_i is the weight of the corresponding software entity e_i from SE.

Each refactoring $r_l, l = \overline{1,t}$, may be applied to a subset of software entities, defined as $re : SR \rightarrow \mathcal{P}(SE)$, $re(r_l) = \{ e_{l_1}, \ldots, e_{l_q} \mid \text{if } ra(r_l, e_{l_u}) = T, 1 \leq u \leq q, 1 \leq q \leq m \}$, where the ra mapping is defined by the formula 2 (see Section 5.3), $re(r_l) = SE_{r_l}, SE_{r_l} \in \mathcal{P}(SE) - \varnothing$, $1 \leq l \leq t$. In this way we have a function based description.

Output Data

The MORSSP is the problem of finding a subset of entities named $ESet_l$, $ESet_l \subseteq SE_{r_l} \subseteq SE$ for each refactoring $r_l \in SR, l = \overline{1,t}$ such that:

- the following objectives are 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.

In the MORSSP formulation, two objective functions are taken into consideration in order to 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 $\vec{r} = (r_1, \ldots, r_t)$ has to be optimized, as described below.

The first objective function defined 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 the 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 $\vec{r} = (r_1, \ldots, r_t)$.

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:

$$\begin{aligned} &maximize\left\{F(\vec{r})\right\} = maximize\left\{f_1^{'}(\vec{r}),\ f_2(\vec{r})\right\} = \\ &= maximize\left\{MAX - \textstyle\sum_{l=1}^{t}\sum_{i=1}^{m} rc(r_l, e_i),\ \sum_{l=1}^{t}\sum_{i=1}^{m} w_i \cdot effect(r_l, e_i)\right\}, \end{aligned} \quad (6)$$

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

5.4.2 The Multi-Objective Refactoring Single Selection Problem

The *Multi-Objective Refactoring Single Selection Problem* (*MORSgSP*) is a particular case of the MORSSP (see Section 5.4.1) with restraint requirements [CCV09a]. Therefore, the specific problem aspects are isolated within the *output data*.

Output Data

The MORSgSP is the multi-objective problem of finding a refactoring $r_l \in SR, l = \overline{1,t}$, for each entity $e_i \in SE, i = \overline{1,m}$, such that:

- the following objectives are 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.

5.4.3 The Multi-Objective Refactoring Sequence Selection Problem

The *Multi-Objective Refactoring Sequence Selection Problem* (*MORSqSP*) is based on the *General Multi-Objective Refactoring Selection Problem* (see Section 5.3). The goal is to find the sequences of refactorings that affect the entire software system based on the software entity dependencies and refactoring dependencies. The two compound and conflicting objective functions (the *refactoring cost* and the *refactoring impact*) defined for the MORSSP are used by this problem definition too.

Various specific elements are defined as *additional notations* required, while the problem requirements are stated as *output data* of the defined problem.

Additional Notations

Definition 5.4.1([CC10f]) *A* ***refactoring-entity pair*** *is a tuple* $\widehat{r_l\ e_i} = (r_l, e_i)$ *consisting of a refactoring* $r_l, 1 \le l \le t$, *applied to a software entity* $e_i, 1 \le i \le m$, *therefore* $ra(r_l, e_i) = T$.

Definition 5.4.2([CC10f]) *A* ***refactoring sequence*** *is a chain of refactoring-entity pairs* $rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2},\ \ldots, \widehat{r_s\ e_s})$, *where* $r_u \in SR,\ e_u \in SE,\ 1 \le u \le s$. *The followings stay:*

1. $ed(e_u, e_{u+1}) \in SED, \forall\ \widehat{r_u\ e_u},\ \widehat{r_{u+1}\ e_{u+1}} \in rs$, *where* $r_u,\ r_{u+1} \in SR,\ e_u,\ e_{u+1} \in SE,\ 1 \le u \le s-1$;
2. $rd(r_u, r_{u+1}, e_u),\ rd(r_u, r_{u+1}, e_{u+1}) \in SRD, \forall\ \widehat{r_u\ e_u},\ \widehat{r_{u+1}\ e_{u+1}} \in rs$, *where* $r_u,\ r_{u+1} \in SR,\ e_u,\ e_{u+1} \in SE,\ 1 \le u \le s-1$;
3. $rd(r_l, r_k, e_i) \in SRD, \forall\ \widehat{r_l\ e_i},\ \widehat{r_k\ e_i} \in rs$, *where* $r_l,\ r_k \in SR,\ e_i \in SE,\ 1 \le l < k \le t,\ 1 \le i \le m$;
4. $ed(e_i, e_k) \in SED, \forall\ \widehat{r_l\ e_i}, \widehat{r_l\ e_k} \in rs$, *where* $r_l \in SR, e_i, e_k \in SE, 1 \le i < k \le m, 1 \le l \le t$.

A *sequence of refactorings applied to a software entity* $e_i,\ 1 \le i \le m$, denoted by rs_{e_i}, is simply described as $rs_{e_i} = (r_1,\ r_2, \ldots, r_s),\ e_i \in SE,\ 1 \le i \le m,\ r_u \in SR,\ 1 \le u \le s,\ s \in \mathcal{N}^*$. The ra, ed, and rd mappings are formally described by the Section 5.3.

The *set of all refactoring sequences that may be applied to the software entities* $e_i,\ i = \overline{1,m}$, *using the refactorings* $r_l,\ l = \overline{1,t}$, is defined as:

$$SSR = \left\{\ rs\ \ \middle|\ rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, \widehat{r_s\ e_s}),\ r_u \in SR,\ e_u \in SE,\ 1 \le u \le s,\ s \in \mathcal{N}^*\ \right\}. \quad (7)$$

The *set of all refactoring sequences that may be applied to a software entity* $e,\ e \in SE$, is defined as:

$$SSR_e = \left\{\ rs_e\ \ \middle|\ rs_e = (r_1,\ r_2, \ldots, r_s),\ r_u \text{ is a refactoring applied to } e,\ 1 \le u \le s,\ s \in \mathcal{N},\ \right\}, \quad (8)$$

where the $SSR_e \in \mathcal{P}(SSR),\ e \in SE$.

The *set of distinct refactorings that compose a refactoring sequence* $rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, \widehat{r_s\ e_s}),\ s \in \mathcal{N}$, is denoted by $SR_{rs} = \{r |\ \exists\ \widehat{r\ e}\ in\ rs,\ e \in SE,\ r \in SR\}$, $SR_{rs} \in \mathcal{P}(SR)$.

The *set of distinct refactoring-entity pairs that compose a refactoring sequence* $rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, \widehat{r_s\ e_s}),\ s \in \mathcal{N}$, is denoted by $SEP_{rs} = \{\widehat{r\ e} |\ \exists\ \widehat{r\ e}\ in\ rs,\ r \in SR,\ e \in SE\}$, $SEP_{rs} \in \mathcal{P}(SSR)$.

Output Data
The MORSqSP is a two folded problem, as:

1. it finds a refactoring sequence $rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, \widehat{r_s\ e_s})$ applied to the software system SS, where $e_u \in SE,\ r_u \in SR,\ 1 \leq u \leq s,\ s \in \mathcal{N}$;
2. it finds a refactoring sequence $rs_e = (r_1,\ r_2, \ldots, r_s)$ applied to a given software entity e, where $e \in SE,\ r_u \in SR,\ 1 \leq u \leq s,\ s \in \mathcal{N}$.

The MORSqSP multi-objectiveness aspects are:

- the objectives to be optimized are:
  - the overall refactoring cost is minimized;
  - the overall refactoring impact on software entities is maximized;
- the software entity dependencies constraints defined by the mapping 1 (see Section 5.3) are satisfied;
- the refactoring dependencies constraints defined by the mapping 3 (see Section 5.3) are satisfied.

The overall objective function definition for the MORSqSP is analogous to the one in the formula 6. The decision vector $\vec{r}$ has a goal-based description within an appropriate DS. Therefore, for the intended aims in the output data, we are interested in:

- searching for *the refactoring sequence rs applied to the software system SS*:
  - $DS = SSR$;
  - $\vec{r} = (\widehat{r_1, e_1},\ \widehat{r_2, e_2}, \ldots, \widehat{r_s, e_s})$, where $e_u \in SE,\ r_u \in SR,\ 1 \leq u \leq s,\ s \in \mathcal{N},\ s \leq n$;
- searching for *the refactoring sequence rs_{e_i} applied to a given software entity e_i*, where $e_i \in SE,\ 1 \leq i \leq m$:
  - $DS = SR^n,\ n \in \mathcal{N}$;
  - $\vec{r} = (r_1,\ r_2, \ldots, r_s)$, where $e_i \in SE,\ r_u \in SR, 1 \leq i \leq m, 1 \leq u \leq s, s \leq n, s \in \mathcal{N}$.

5.4.4 The Multi-Objective Refactoring Plan Building Problem

This subsection presents the *Multi-Objective Refactoring Plan Building Problem* (*MORPBP*), based on the *General Multi-Objective Refactoring Selection Problem* (*GMORSP*) (see Section 5.3) and the *Multi-Objective Refactoring Sequence Selection Problem* (*MORSqSP*) (see Subsection 5.4.3). A motivation together with a problem working scenario is provided here too. Input data, additional terms and notations have been defined to completely state the *MORPBP*.

Input Data
Let $SE = \{e_1, \ldots, e_m\}$ be a set of software entities as it was defined by Section 5.3. The software entity set SE together with different types of dependencies among its items form a software system named SS. The set of software entity dependency types SED and the dependency mapping ed are similar to the one described by the formula 1 (see Section 5.3).

A set of signifying chosen refactorings that may be applied to the software entities of SE is gathered up through $SR = \{r_1, \ldots, r_t\}$. The ra mapping that sets the applicability for the chosen set of refactorings SR on the set of software entities SE is defined by the formula 2 (see Section 5.3).

The set of refactoring dependencies SRD together with the mapping rd that highlight the dependencies among different refactorings when applied to the same software entity are stated by the formula 3 (see Section 5.3).

Definitions 5.4.1 and 5.4.2 (see Subsection 5.4.3) introduce the *refactoring-entity pair* and *refactoring sequence* terms, respectively. The set of all refactoring sequences that may be applied to the software entities $e_i,\ i = \overline{1,m}$, using the refactorings $r_l,\ l = \overline{1,t}$, denoted by $SSR = \{rs_1,\ \ldots, rs_p\}$, where $rs_k = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, r_{s_{rs_k}}\widehat{\ e_{s_{rs_k}}})$, $k = \overline{1,p},\ e_u \in SE,\ r_u \in SR,\ 1 \leq u \leq s_{rs_k},\ p, s_{rs_k} \in \mathcal{N}$, is analogous defined by the formula 7 (see Subsection 5.4.3).

Each refactoring sequence $rs_k,\ 1 \leq k \leq p$, has a *building weight* within the overall set of refactoring sequences SSR, that suggests the refactoring sequence priority when combining to each other, being expressed by the set $SSRWeight = \{rsw_1, \ldots, rsw_p\}$, where $rsw_k \in [0,1]$ and $\sum_{k=1}^{p} rsw_k = 1$.

Each refactoring-entity pair $\widehat{r_u\ e_u},\ u = \overline{1, s_{rs_k}}$, from the participating refactoring sequences $rs_k = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, r_{s_{rs_k}}\widehat{\ e_{s_{rs_k}}})$, where $rs_k \in SSR,\ k = \overline{1,p},\ s_{rs_k} \in \mathcal{N}$, within the plan building process, has an associated *integration status*. This is attached to each refactoring-entity pair at the refactoring sequence building time.

The two possible refactoring statuses within a refactoring sequence composition form the set $RStatus = \{\mathbf{M}andatory,\ \mathbf{O}ptional\}$. A mapping that links the status to a refactoring-entity pair is defined as $rstatus_{rs} : SEP_{rs} \rightarrow RStatus$,

$$rstatus_{rs}(\widehat{r\ e}) = \begin{cases} \text{M}, & \text{if } \widehat{r\ e} \text{ is } \textit{mandatory} \text{ in the plan building} \\ \text{O}, & \text{otherwise} \end{cases},$$

where the $\widehat{r\ e} \in SEP_{rs}$.

Additional Notations

Refactoring sequence-based plan building means combining multiple refactoring sequences together into a single refactoring sequence. The resulting refactoring sequence contains connection (junction) points between each two refactoring-entity pairs, inside and outside the same or different refactoring sequences.

Definition 5.4.3([CC10f]) *Let $rs = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, \widehat{r_s\ e_s}), rs \in SSR$, be a refactoring sequence, $\widehat{r_u\ e_u}$,$r_{u+1}\widehat{\ e_{u+1}}$ two consecutive refactoring-entity pairs in rs, where $1 \leq u \leq s-1$ and $\widehat{r\ e}$ a refactoring-entity pair, where $e \in SE, r \in SR, ra(r,e) = T$ and $\widehat{r\ e}$ not in rs. Then:*

1. *the refactoring-entity pair $\widehat{r_u\ e_u},\ 1 \leq u \leq s$, has a **before junction point**, denoted by ${}_b(\widehat{r_u\ e_u})$, if exists a refactoring-entity pair $\widehat{r\ e}$ that may be inserted before $\widehat{r_u\ e_u}$ within the refactoring sequence rs, where rs improved by the $\widehat{r\ e}$ is $rs' = (\ldots, \widehat{r\ e},\ \widehat{r_u\ e_u}, \ldots)$ and the followings hold:*
   (a) *$ed(e, e_u) \in SED$, where $e,\ e_u \in SE,\ 1 \leq u \leq s$;*
   (b) *$rd(r, r_u) \in \{B,\ AB\} \subseteq SRD$, where $r,\ r_u \in SR,\ 1 \leq u \leq s$.*
2. *the two refactoring-entity pairs $\widehat{r_u\ e_u}$ and $r_{u+1}\widehat{\ e_{u+1}},\ 1 \leq u \leq s-1$, have a **middle junction point**, denoted by $(\widehat{r_u\ e_u})_m(r_{u+1}\widehat{\ e_{u+1}})$, if exists a refactoring-entity pair $\widehat{r\ e}$ that may be inserted between $\widehat{r_u\ e_u}$ and $r_{u+1}\widehat{\ e_{u+1}}$ within the refactoring sequence rs, where rs improved by the $\widehat{r\ e}$ is $rs' = (\ldots, \widehat{r_u\ e_u}, \widehat{r\ e}, r_{u+1}\widehat{\ e_{u+1}}, \ldots)$ and the followings hold:*
   (a) *$ed(e_u, e),\ ed(e, e_{u+1}) \in SED$, where $e_u,\ e,\ e_{u+1} \in SE,\ 1 \leq u \leq s-1$;*
   (b) *$rd(r_u, r),\ rd(r, r_{u+1}) \in \{B,\ AB\} \subseteq SRD$, where $r_u,\ r,\ r_{u+1} \in SR,\ u = \overline{1, s-1}$.*
3. *the refactoring-entity pair $\widehat{r_u\ e_u},\ 1 \leq u \leq s$, has an **after junction point**, denoted by $(\widehat{r_u\ e_u})_a$, if exists a refactoring-entity pair $\widehat{r\ e}$ that may be added after $\widehat{r_u\ e_u}$ within the refactoring sequence rs, where rs improved by the $\widehat{r\ e}$ is $rs' = (\ldots,\ \widehat{r_u\ e_u},\ \widehat{r\ e},\ \ldots)$ and the followings hold:*
   (a) *$ed(e_u, e) \in SED$, where $e_u,\ e \in SE,\ 1 \leq u \leq s$;*
   (b) *$rd(r_u, r) \in \{B,\ AB\} \subseteq SRD$, where $r_u,\ r \in SR,\ 1 \leq u \leq s$.*

A *refactoring-entity pair junction points set for a refactoring sequence* rs is defined as: $REP_{rs}^{jp} = \{_b(\widehat{r_u\ e_u})|\widehat{r\ e}$ *is inserted before* $\widehat{r_u\ e_u},\ 1 \leq u \leq s\} \cup \{(\widehat{r_u\ e_u})_m(r_{u+1}\widehat{\ e_{u+1}})|\widehat{r\ e}$ *is inserted between* $\widehat{r_u\ e_u}$ *and* $r_{u+1}\widehat{\ e_{u+1}},\ 1 \leq u \leq s-1\} \cup \{(\widehat{r_u\ e_u})_a|\widehat{r\ e}$ *is inserted after* $\widehat{r_u\ e_u},\ 1 \leq u \leq s\}$, being provided at the refactoring sequence building time.

Definition 5.4.4([CC10f]) *Let* $SSR_{rp} = \{rs_1,\ \ldots, rs_q\}$ *be a set of refactoring sequences,* $REP_{rs_1}^{jp},\ REP_{rs_2}^{jp}, \ldots,\ REP_{rs_q}^{jp}$ *the corresponding junction points sets for the refactoring sequences* rs_k*, where* $1 \leq k \leq q$*, where* $SSR_{rp} \in \mathcal{P}(SSR),\ rs_k = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, r_{s_{rs_k}}\widehat{\ e_{s_{rs_k}}}),\ k = \overline{1,q},\ e_u \in SE,\ r_u \in SR,\ 1 \leq u \leq s_{rs_k},\ s_{rs_k}, q \in \mathcal{N}$.

A **refactoring plan** is a refactoring sequence $rp = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, r_{s_{rp}}\widehat{\ e_{s_{rp}}})$ formed by navigating among the refactoring-entity pairs $\widehat{r_u\ e_u},\ 1 \leq u \leq s_{rs_k},\ k = \overline{1,q},\ q \in \mathcal{N}$ of the composing refactoring sequences rs_k of the set SSR_{rp}. Then, for each refactoring-entity pair $\widehat{r_u\ e_u} \in rp$, where $1 \leq u \leq s_{rp}$, it exists a junction point jp, where $jp \in REP_{rs_k}^{jp}$, such that jp is the refactoring-entity pair junction point that introduced $\widehat{r_u\ e_u}$ within the rp refactoring plan, where $1 \leq k \leq q$.

Output Data

The MORPBP is the problem of finding a refactoring plan $rp = (\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots,\ r_{s_{rp}}\widehat{\ e_{s_{rp}}})$, $s_{rp} \in \mathcal{N}$, from a refactoring sequence set SSR_{rp}, such that:

- the following objectives are optimized:
  - the overall refactoring cost is minimized;
  - the overall refactoring impact on software entities is maximized;
- software entity dependencies constraints defined by the mapping 1 are satisfied;
- refactoring dependencies constraints defined by the mapping 3 are satisfied.

In the MORPBP, two objectives are optimized: the *refactoring cost* and the *refactoring impact* on the affected software entities. The multi-objective function

$$optimize\{F(\vec{rp})\} = optimize\{f_1(\vec{rp}),\ f_2(\vec{rp})\}, \tag{9}$$

where $\vec{rp}$= $(\widehat{r_1\ e_1},\ \widehat{r_2\ e_2}, \ldots, r_{s_{rp}}\widehat{\ e_{s_{rp}}})$, $s_{rp} \in \mathcal{N}$, to be optimized, is similarly described as in Section 5.3.

The current MORPBP formulation studies the refactoring cost as an objective instead of a constraint, as the software entities dependencies defined by the formula 1 (see Section 5.3) and refactoring dependencies described by the mapping 3 are investigated.

The problem decision space is $DS = SSR$ while the decision vector is $\vec{rp}$= $(\widehat{r_1\ e_1},$ $\widehat{r_2\ e_2}, \ldots, r_{s_{rp}}\widehat{\ e_{s_{rp}}}),\ s_{rp} \in \mathcal{N}$, contains the refactoring sequences obtained by cross-navigating the set of the proposed refactoring sequences $SSR_{rp},\ SSR_{rp} \in \mathcal{P}(SSR)$.

5.5 Conclusions and Future Work

The appropriate refactoring selection within different sized software is a stimulating research problem investigated by this chapter. Software entity dependencies and refactoring dependencies are the basic intriguing elements that drive the research within this domain.

Some of the original results presented in this chapter have been reported in the papers [CC09b, CC09a, CC10a, CC10d]. Current chapter is based on the original work published in the papers [CCV09a, CCV09b, CC09c, CC09e, CC10f].

Some further work may be done in the following directions: the refactoring plan building problem definition following the approach with a parallel refactoring application with new notations such as multi-junction points which provides multi-connection points between different refactoring sequences, resulting in a parallel refactoring selection; new multi-objective problem formulations that use the cost as a constraint instead of an objective; new multi-objective problem formulations that may use more than two objectives to optimize.

6 The Refactoring Selection Problem - An Evolutionary Approach

The *Multi-Objective Refactoring Set Selection Problem* and its special case, the *Multi-Objective Refactoring Single Selection Problem*, are investigated in this chapter. Different genetic algorithms solution representations are proposed. The *Multi-Objective Refactoring Set Selection Problem* is modeled as a single aggregated objective function, where two conflicting objectives are optimized. Comparisons between the proposed approaches are discussed.

6.1 Evolutionary-Based Solution Representations for the Refactoring Set Selection Problem

The MORSSP is approached here by exploring the various existing refactoring dependencies. Two conflicting objectives were studied, i.e., minimizing the refactoring cost and maximizing the refactoring impact, together with some constraints to be kept, as the refactoring dependencies.

The *weighted sum method* [Kd05] was adopted to solve the MORSSP. The overall objective function to be maximized $F(\vec{r})$ 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, within a decision space.

The steady–state evolutionary model is advanced by the proposed evolutionary computation technique. An adapted genetic algorithm to the context of the investigated problem, with weighted sum fitness function, was proposed in [CC09b, CC10a].

The proposed genetic algorithm that uses a *refactoring-based* solution representation for the refactoring set selection problem is denoted by *RSSGARef*, while the corresponding *entity-based* genetic algorithm is denoted by *RSSGAEnt*.

6.2 Studied Solution Representations

6.2.1 Refactoring-Based Solution Representation

For the *RSSGARef* algorithm the solution representation is presented in [CC09c]. 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 any $e_{l_u} \in SE_{r_l}, e_{l_u} \in S_l \in \mathcal{P}(SE), 1 \leq u \leq q, 1 \leq q \leq m, 1 \leq l \leq t$. This means it is possible to apply more than once the same refactoring to different software entities. The genetic operators used by this approach are crossover and mutation.

6.2.2 Entity-Based Solution Representation

The *RSSGAEnt* algorithm uses the solution representation presented in [CC09e], where the decision vector $\vec{S} = (S_1, \ldots, S_m), S_i \in \mathcal{P}(SR), 1 \leq i \leq m$ 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 $e_{l_u} \in SE_{r_l}, S_{r_l} \in \mathcal{P}(SR), 1 \leq u \leq q, 1 \leq q \leq m, 1 \leq l \leq t$. It means it is possible to apply more than once the same refactoring to different software entities. The genetic operators used by this approach are crossover and mutation too.

6.3 Case Study: The LAN Simulation Problem

The adapted genetic algorithm proposed in [CC09b, CC10a] is applied to a simplified version of the *LAN Simulation* source code (see Subsection 1.11.1). 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 on 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 is presented by the Table 3 , where $\sum_{i=1}^{23} w_i = 1$.

The dependency relationship between refactorings, described by the rd mapping and the final impact of each refactoring stated by the res mapping are defined by the Table 3. 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 5.3. The refactoring applicability (ra mapping) to each software entity from the SE set is expressed as a non null value of the refactoring cost as presented by the Table 3. Therefore, for the software entity c_3 and the refactoring r_2, the notations $\surd/2$ means the refactoring may be applied to the specified software entity ($\surd$, i.e., $ra(r_2, c_3) = T$) with the transformation cost of 2.

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

6.3.1 Proposed Refactoring Strategy

A possible refactoring strategy for the *LAN Simulation Problem* is presented below. Based on the *difficulties* 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).

6.4 Practical Experiments for the *RSSGARef* and *RSSGAEnt* Algorithms

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, and 200 and number of individuals 20, 50, 100, and 200.

(a) Refactoring dependencies (rd) and final impact (res) of their application to the software 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 to the 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 3: The input data for the *LAN Simulation Problem* case study

6.4.1 Experiment 1: Equal Weights on the Refactoring Cost and Impact ($\alpha = 0.5$)

This subsection presents the results obtained for a first experiment with equal weights, i.e., $\alpha = 0.5$, run for the *RSSGARef* and *RSSGAEnt* Algorithms [CC09c, CC09b, CC09a].

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 was obtained for the *RSSGARef Algorithm* through a 200 generations evolution with a 20 chromosomes population, having the fitness value of 0.4793. This means in small populations (with fewer individuals) the reduced diversity among chromosomes may induce a harsher struggle compared to large populations (with many chromosomes) where the diversity breeds near quality individuals.

For both solution representations small populations keep their good chromosome quality, by breeding all 100 individuals with the fitness better than the reference value, i.e., 0.41, for the refactoring-based solution representation and 0.15 for the entity-based solution representation. The results provided by the refactoring-based solution representation are better as fitness value quality. Impact on the *LAN Simulation* source code after the *RSSGARef* and *RSSGAEnt* Algorithms were run are presented too.

6.4.2 Experiment 2: Higher Weight on the Refactoring Impact ($\alpha = 0.3$)

This subsection shows the results of a first 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 in [CC09b, CC09a]. The weighted sum fitness function used by both *RSSGARef* and *RSSGAEnt* Algorithms is $F(\vec{r}) = 0.3 \cdot f_1(\vec{r}) + 0.7 \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_t)$.

The experiment for the *RSSGARef Algorithm* shows good results in all 100 runs as quality and number for the studied individuals populations and number of generations. A tendency to breed better individuals in smaller population size has been observed, as it was noticed for the $\alpha = 0.5$ experiment too.

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 individual population recorded by the experiment. Impact on the *LAN Simulation* source code after the *RSSGARef* and *RSSGAEnt* Algorithms were run are presented too.

6.4.3 Experiment 3: Lower Weight on the Refactoring Impact ($\alpha = 0.7$)

This subsection presents the second experiment run with different weights on the two objectives, for $\alpha = 0.7$. This means the cost (rc mapping) of the applied refactorings is more important than the implied final effect (res function) on the affected software entities [CC09a]. The corresponding fitness function for the *RSSGARef* and *RSSGAEnt* Algorithms is $F(\vec{r}) = 0.7 \cdot f_1(\vec{r}) + 0.3 \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_t)$.

The best individual obtained by the *RSSGARef* Algorithm was for a 200 generations evolution with 20 individuals population, where the best fitness value was 0.61719 (all individuals with the fitness > 0.53). For the *RSSGAEnt* Algorithm the highest fitness value was 0.16862, being recorded for a 50 generations evolution run with 20 chromosomes population (with 98 chromosomes with fitness > 0.155).

For the *RSSGARef* Algorithm, the experiment results assessment reveals that they are very much alike to the experiment with $\alpha = 0.3$. This way, the best chromosomes for the 20 and 200 individuals populations runs cover the value ranges $(0.54, 0.63)$, while for the *RSSGAEnt* Algorithm the best fitness value range is smaller $((0.15, 0.17))$, where the fitness is distributed within all value ranges with some exceptions.

Unlike previous run experiments for the *RSSGAEnt* Algorithm, the diversity among chromosome populations is kept for the best and the worst individual. The worst individual recorded for the 20 individuals populations with 50 generations evolution has the fitness value of 0.12515 (27 chromosomes with fitness value < 0.13).

The *RSSGARef* and *RSSGAEnt* Algorithms applications to the *LAN Simulation* source code are discussed too.

6.5 Discussion on the Applied Algorithms for the Refactoring Set Selection Problem

Current section summarizes the results of the proposed *RSSGARef* and *RSSGAEnt* Algorithms in Section 6.4 for three different values for the α parameter, i.e., 0.3, 0.5, and 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.

6.6 Results Analysis for the Multi-Objective Refactoring Set Selection Problem

This section analyzes the proposed solutions for the refactoring-based and entity-based solution representations (see Subsections 6.2.1 and 6.2.2). 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 studied refactorings, 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), for a 200 generations evolution with 20 chromosomes population, having the fitness value of 0.4793, while for the entity-based approach (*RSSGAEnt* algorithm) the recorded best chromosome was obtained for a 20 generations evolution with 20 individuals, with a fitness value of 0.17345. These solutions may be transposed from a representation to another, which means their structure may be compared and their efficiency evaluated.

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 populations itself. Large number of genes of the individual structure induces poor quality to the current entity-based solution representation. The former one may be perceived as a large sized population within the entity-based solution representation experiments.

Table 4 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)**			**Class hierarchy** *(3a)*
						(2a)	*(2b)*	*(2c)*	
Refactoring based	0.3	20c/200g	0.33587	32secs	0	0	0	50	100
	0.5	20c/200g	0.4793	36secs	60	50	50	50	50
	0.7	20c/200g	0.61719	37secs	0	0	50	100	50
Entity based	0.3	20c/200g	0.19023	61secs	80	50	100	100	50
	0.5	20c/200g	0.17345	75secs	40	0	50	100	100
	0.7	20c/50g	0.16862	19secs	0	50	100	100	100

Table 4: The best chromosomes obtained for the refactoring-based and entity-based solution representation, for different α parameter values: 0.3, 0.5, and 0.7

6.7 Evolutionary-Based Solution for the Refactoring Single Selection Problem

6.7.1 Genetic Algorithm-Based Approach

Similar to the MORSSP, the MORSgSP investigates the existing refactoring dependencies to identify a single most appropriate refactoring for each software entity. The two conflicting objectives are the same, i.e., to minimize the refactoring cost and to maximize the refactoring impact. The constraints to be satisfied are the refactoring dependencies. Current approach uses the weighted sum method together with a steady–state evolutionary model.

The proposed genetic algorithm approaches an entity-based solution representation for the studied problem, denoted by *RSgSGAEnt*. The adapted genetic algorithm proposed in [CC10d] was applied to a simplified version of the *LAN Simulation* source code too.

6.7.2 Entity-Based Solution Representation

Within the *RSgSGAEnt* algorithm solution representation presented in [CC10d], the decision vector $\vec{r} = (r_1, \ldots, r_m)$, $r_i \in SR, 1 \leq i \leq m$, determines the refactorings that may be applied in order to transform the considered set of software entities SE. The item r_i on the i-th position of the solution vector represents the refactoring that may be applied to the i-th software entity from SE, where $e_i \in SE_{r_i}, 1 \leq i \leq m$. The genetic operators used by this approach are crossover and mutation operators.

6.8 Practical Experiments for the *RSgSGAEnt* Algorithm

The *RSgSGAEnt* algorithm was run 100 times and the best, worse, and average fitness values were recorded. The parameters used by the evolutionary approach were: mutation probability 0.7 and crossover probability 0.7. The experiments include runs for 10, 50, 100, and 200 number of generations with 20, 50, 100, and 200 as number of individuals. Each following subsection shortly presents the raw results and emphasize the impact on the class diagram. The run experiments have worked with different values for the α parameter: 0.3, 0.5, and 0.7. Therefore, the fitness function is rewritten within the formula $F(\vec{r}) = \alpha \cdot f_1(\vec{r}) + (1-\alpha) \cdot f_2(\vec{r})$, where $\vec{r} = (r_1, \ldots, r_m)$.

6.8.1 Experiment 1: Equal Weights on the Refactoring Cost and Impact ($\alpha = 0.5$)

A first experiment that proposes equal weights, i.e., $\alpha = 0.5$, for the studied fitness function [CC10d] is presented by this subsection.

In the 50 generations evolution experiments for 200 chromosomes populations the greatest value of the fitness function was 0.3455 (with 38 individuals with the fitness > 0.33) while in the 200 evolutions experiments for 20 individuals populations the best fitness value was 0.3562 (96 individuals with the fitness > 0.33), which is the best fitness value in the current experiment.

The worst chromosome in all runs was recorded for a 200 individuals population for 50 generations evolution with a fitness value of 0.27005 (87 chromosomes with fitness < 0.283), while in the 20 chromosomes populations with 200 generations evolution the worst individual had the fitness value 0.2772 (11 individuals having the fitness value < 0.283 only). The impact on the *LAN Simulation* source code after the *RSgSGAEnt Algorithm* application is presented.

6.8.2 Experiment 2: Higher Weight on the Refactoring Impact ($\alpha = 0.3$)

A first experiment that works with different weights, i.e., $\alpha = 0.3$, where the final effect (res function) has a higher relevance than the implied cost (rc mapping) of the applied refactorings [CC10d] is presented by this subsection.

The best chromosome in the entire experiment was obtained within a 100 generations run within a 200 chromosomes population with the fitness value of 0.25272 (with 69 individuals with the fitness > 0.233), while the best chromosome obtained for the 10 generations evolution with 200 individuals populations had the fitness value of 0.24462 (34 individuals with the fitness > 0.233).

Worst individual obtained by the current experiment has near values to the average individuals. For the 200 chromosomes populations with 10 generations evolution the worst individual has the fitness value of 0.18495 (15 individuals with fitness value worse than 0.195), while for the 200 generations evolution runs with 100 individuals populations the worst chromosome has the fitness value of 0.18907 (33 individuals having the fitness worse than 0.195).

The grouping of the eligible chromosomes for the 50, 100, and 200 individuals populations for small numbers of generations is visible. The solutions for the 20 individuals populations for each studied number of evolutions keep their good quality, with a higher number of eligible chromosomes. The *RSgSGAEnt Algorithm* application impact to the *LAN Simulation* source code is presented too.

6.8.3 Experiment 3: Lower Weight on the Refactoring Impact ($\alpha = 0.7$)

The second experiment with different weights 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 [CC10d].

In the 200 generations runs for 100 chromosomes populations the greatest value of the fitness function was 0.44919 (75 individuals with the fitness > 0.425), while in the 50 generations evolutions experiments for 50 individuals populations the best fitness value was 0.45757 (65 individuals with the fitness > 0.425) which is the best fitness value obtained within the current experiment. The *RSgSGAEnt Algorithm* application impact to the *LAN Simulation* source code is presented too.

6.8.4 Discussion on *RSgSGAEnt* Algorithm for the Refactoring Single Selection Problem

The results of the proposed approach from Section 6.7 for three different value of the α parameter, i.e., 0.3, 0.5, and 0.7, are summarized and discussed by the current subsection.

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

Table 5 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)*			Class hierarchy *(3a)*
						(2a)	*(2b)*	*(2c)*	
Entity based	0.3	100c/200g	0.25272	64secs	100	50	50	50/100	100
	0.5	20c/200g	0.3562	14secs	100	50	50	100	100
	0.7	50c/50g	0.45757	9secs	100	50	0	50/100	100

Table 5: The best chromosomes obtained for the entity based solution representation, for different α parameter values: 0.3, 0.5, and 0.7

6.9 The MORSSP and MORSgSP Solutions Analysis

The MORSgSP represents a special case of the MORSSP, where the corresponding artificial intelligence-based solutions are approached by Sections 6.3 and 6.7. The former one identifies a single refactoring that changes a software entity that satisfies the established constraints in the most appropriate way, while the latter identifies a set of possible refactorings for each software entity.

The best individual obtained for the run experiments for the MORSgSP, i.e., a 20 chromosomes population with 200 generations evolution, was transposed to the refactoring-based solution representation of the MORSSP. The resulted individual has the same fitness as in the original form (0.3562). The best chromosome recorded for the MORSSP experiments, is obtained for a 20 chromosomes population with 200 generations evolution too. But, it cannot be transposed to the solution representation presented in Section 6.7, since there are several refactorings suggested for each entity.

First, a refactoring may be applied to more than one software entity, as the r_6 (*AddParameter* refactoring) which is applied to the m_8 (`print` method) from c_3 (`PrintServer` class) and m_{11} (`save` method) from c_4 (`FileServer` class). Second, r_1 (*RenameMethod* refactoring) is then applied for the same methods in order to highlight the polymorphic behaviour of the new renamed method `process`. This means there are at least two refactorings that have to be applied to the methods referred here (`print` and `save`). Thus, the multiple transformations of software entities cannot be coded by the solution representation proposed by Section 6.7.

The *RSgSGAEnt* algorithm (see Subsection 6.8.1) allows information hiding by suggesting the refactoring for field encapsulation. But the solution representation does not allow to apply more than one refactoring to each software entity. This results in the lack of possibility to apply some relevant refactorings to software entities [CC11].

For the solution proposed by the *RSSGARef* Algorithm with $\alpha = 0.5$, the fitness value of the best chromosome (0.4793) is better than the value of the approach discussed in Section 6.8, while it suggests to apply more than one refactoring to a single software entity.

6.10 Conclusions and Future Work

This chapter has advanced the evolutionary-based solution approach for the MORSSP and the MORSgSP. Adapted genetic algorithms have been proposed in order to cope with a weighted-sum objective function for 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 representation were studied and the various results of the run experiments were discussed and compared.

The chapter is based on the following papers [CCV09a, CCV09b, CC09c, CC09b, CC09a, CC09e, CC09d, CC10a, CC10d, CC11]. Further work may be done in the following directions: different and adapted to the refactoring selection area crossover operators may be investigated; the Pareto front approach may be studied further.

7 Conclusions and Future Research

The intent of the present PhD thesis was to advocate the idea that refactoring techniques play an increasing part in software engineering, taking benefit on an active research area. The aimed goal and objectives of this research are met as supported by this thesis.

The main contributions of this thesis target three of the six activity steps identified for a complete refactoring process, as: choosing the appropriate refactorings to be applied, refactoring effect assessment on quality characteristics of the software, and maintaining the consistency between the refactored program code and other software artifacts. Possible developments and new research directions were suggested for each original approach addressed.

Refactoring Selection

The thesis approaches the activity of appropriate refactorings identification within various contexts. The appropriate refactoring selection within different sized software is a stimulating research problem. Software entity dependencies and refactoring dependencies are the basic intriguing elements that drive the research within this domain.

Different multi-objective refactoring selection problem are formally investigated, as: the refactoring set selection, single refactoring selection, refactoring sequence selection and refactoring plan building problems are defined. New multi-objective formulations using conflicting objectives to optimize were addressed, as: to minimize the required refactoring cost and to maximize the refactoring impact on software entities, within the corresponding problems. Moreover, new strategically aspects related to the refactoring plan building process based on the management leadership decision were advanced too.

For the *Refactoring Set Selection Problem* and its special case the *Refactoring Single Selection Problem* an evolutionary-based approach was investigated here. Refactoring-based and entity-based solution representations for different genetic algorithms were proposed. In order to compare the various results of the run experiments, a new goal-based assessment strategy for the selected refactorings was proposed. A steady-state genetic algorithm applied used the weighted sum method to aggregate the conflicting objectives and adapted genetic operators to the refactoring selection.

Future work. For the various refactoring selection problems formalized within this thesis, there is a number of directions where the research may follow. An important aspect of the refactoring plan building problem is to follow the approach with the parallel refactoring sequence composition. New notations such as multi-junction points that provide multi-connection points between different refactoring sequences, resulting in parallel refactoring selection have to be defined. A thoroughly study on the junction points may be advanced further, as they may represent refactoring sequences intersection points.

Based on the different criteria used in refactoring plan building process by the management leadership, different refactoring strategies may be shaped. Therefore, refactoring plan building principles may be enounced and described. Research on the strategic refactoring plans configuration may address the possibility to setup a model that may highlight different scenarios, relations, and building methods for different refactoring stories.

Different and adapted to the refactoring selection area crossover operators may be inquired. The Pareto front may be studied further in order to analyze the best chromosomes for the addressed multi-criteria and conflicting objective problems. Other aspects to achieve would be to run different experiments on relevant and real world software systems. Moreover, different solution approaches to the *MORSqSP* and *MORPBP* have to be tackled.

A tool setup that will gather the required input data for the addressed software system represents, a key step within refactoring-based search software engineering field, will be developed.

Refactoring Impact Formalization and Assessment

The refactoring impact on the internal structure representation as AST was investigated by this thesis too. Therefore, the refactoring impact on the internal program structure representation was expressed as the affected node and edge number within the studied AST, following the proposed formalisms. The new impact-based refactoring description was applied for several relevant refactorings, as: *MoveMethod*, *MoveField*, *ExtractClass*, and *InlineClass*.

Refactoring impact assessment on internal program quality may be used to indicate the transformation necessity through refactoring application. Software metrics have become an essential instrument in some disciplines of software engineering. They are used to assess the quality and complexity of software systems, as well as getting a basic understanding and providing clues about sensitive parts of software systems.

A formal description of a consistent set of software metrics was introduced based on the refactoring impact previously advanced. The addressed software metrics were grouped in four categories, as: coupling, cohesion, complexity, and abstraction.

A proposed technique for the analysis of the refactoring impact on the internal program quality through software metrics was introduced. It consists of a five steps strategy applied to address the glide within a software metric value range after specific refactoring application. It investigates the set of the goals chased by the developer within the refactoring impact, the list of refactoring impact categories, and the assessment rules applied to obtain the final refactoring impact on the internal program quality through metrics. Two didactic experiments were run in order to validate the approach.

The proposed classification of the refactoring impact based on the developer intention provides a prior to application feedback to the developers, on the chosen refactorings. Thus, they are able to predict the internal quality slide determined by the refactoring applications. Two limitations of the approach were identified and immediate solutions were advanced.

Future work. The research focused on the refactoring impact and its assessment may be driven by the several aspects in the future. A new catalog with the formal refactoring impact on the internal structure representation as AST for a set of relevant refactorings, from different categories may be build. It may be used as a starting element to build a refactoring strategy based on different internal quality attributes.

Different compound refactorings may be further analyzed in order to identify the basic refactoring impact for its components. Moreover, such analysis may reveal that primitive refactorings that are a part of a larger one may reverse the impact on another refactorings applied. The current AST representation limitations in describing complex refactorings may be removed by extending it with appropriate notations.

The refactoring impact-based analysis for the internal program quality needs further validation. It may be achieved by following the already addressed approach of validating it at theoretical level and practical level as well. Therefore, additional analysis on real world case studies is required, in order to verify the refactoring impact for the set of refactorings and the set of software metrics investigated.

A future step in this area research is to build a tool that encloses all the approach characteristics. A validation approach using the build approach is a subsequent track that will be achieved. Comparison with similar existing tools will be addressed. The automation of the refactoring impact analysis process may be bounded to the developed tool.

Other aspects that need further detailed examination are the possible limitations of this analysis approach, as how they may reduce the number of internal program quality metrics on which the refactoring impact can be assayed. Thereafter, catalogs of software metrics for which the proposed approach may not be applied due to the insufficient model information have to be build. Formal definitions for the identified code duplication types within a class hierarchy may be introduced in the future.

Consistency Maintenance

Another aspect addressed here is related to the consistency maintenance between the refactored program code and other software artifacts. Research within conceptual modeling reveals the possibility to integrate the refactoring process in the analysis development phase.

In order to cope with different variability types, a biological evolution-based model was proposed. Specific refactorings were suggested to shift between ontogenic conceptual models, while forward conceptual abstraction and conceptual specialization are advanced to achieve phylogenic evolution between conceptual models.

Various aspects like the conceptual model and the transformations between variants as refactoring, forward conceptual abstraction, and conceptual specialization are formalized. The biological evolution processes identified when shifting between conceptual models are formally defined. Finally, the three types of conceptual variability are modeled as ontogenic and phylogenic processes.

Future work. The research within the conceptual modeling variability domain may be guided by several directions. The effort required to switch between conceptual models may be estimated and a model to assess the migration process between different variants based on established criteria may be developed. The appraisal criteria may follow the two folded way of quality assessment, as the internal quality and external quality for the studied variants.

Another direction is related to the refactoring application contribution within the switching process between different conceptual models. Subsequently, the external quality attributes may be assessed, starting from transformations imposed by the applied refactorings.

A thoroughly study of the switching process between horizontal abstraction variants may be addressed in the future too. The ontology-based conceptual modeling in order to achieve the migration in the horizontal abstraction variability represents a new research field connected to the refactoring process.

The *Local Area Network Simulation Problem*, the *Data Structure Class Hierarchy Problem* and an extract from a *Didactic Activity Management Problem* are used as case studies to emphasize different aspects related to the applied refactoring within our research.

Bibliography

[Bak95] B. Baker. On finding duplication and near-duplication in large software systems. In *Proceeding of the Second Working Conference on Reverse Engineering (WCRE'95), Toronto, Ontario, Canada*, pages 86–95, 1995.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Boe88] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CC05a] **M.C. Chisăliţă-Creţu**. Efecte ale refactorizării asupra structurii interne a codului. *"Analele Facultăţii", Seria Ştiinţe Economice, Universitatea Creştină "Dimitrie Cantemir" Bucureşti, Facultatea de Ştiinţe Economice Cluj-Napoca, ISSN:1584-5621*, 13(1):214–230, 2005.

[CC05b] **M.C. Chisăliţă-Creţu**. General aspects of refactoring applicability to conceptual models. In *Proceedings of the Symposium "Colocviul Academic Clujean de INFORMATICĂ"(CACI2005)*, pages 99–104, 2005.

[CC07] **M.C. Chisăliţă-Creţu**. Describing low level problems as patterns and solving them via refactorings. *"Studii şi Cercetări Ştiinţifice", Seria Matematică, ISSN: 1224-2519*, 1(17):29–48, 2007.

[CC09a] **M.C. Chisăliţă-Creţu**. The entity refactoring set selection problem - practical experiments for an evolutionary approach. In *Proceedings of the World Congress on Engineering and Computer Science (WCECS2009), October 20-22, 2009, San Francisco, USA*, pages 285–290. Newswood Limited, ISBN: 978-988-17012-6-8, 2009.

[CC09b] **M.C. Chisăliţă-Creţu**. First results of an evolutionary approach for the entity refactoring set selection problem. In *Proceedings of the 4th International Conference "Interdisciplinarity in Engineering" (INTER-ENG 2009), November 12-13, 2009, Târgu Mureş, România*, pages 303–308. ISSN: 1843-780x, 2009.

[CC09c] **M.C. Chisăliţă-Creţu**. A multi-objective approach for entity refactoring set selection problem. In *Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2009), August 4-6, 2009, London, UK*, pages 790–795. ISBN: 978-1-4244-4456-4, 2009.

[CC09d] **M.C. Chisăliţă-Creţu**. Search-based software entity refactoring - a new solution representation for the multi-objective evolutionary approach of the entity

set selection refactoring problem. In *Proceedings of the International Scientific and Professional Conference XXII. DidMatTech 2009, September 10-11, 2009, Trnava, Slovakia*, pages 36–41, 2009.

[CC09e] **M.C. Chisăliţă-Creţu**. Solution representation analysis for the evolutionary approach of the entity refactoring set selection problem. In *Proceedings of the 12th International Multiconference "Information Society" (IS2009), October 12th-16th, 2009, Ljubljana, Slovenia*, pages 269–272. Informacijska družba, ISBN: 978-961-264-010-1, 2009.

[CC10a] **M.C. Chisăliţă-Creţu**. An evolutionary approach for the entity refactoring set selection problem. *Journal of Information Technology Review, ISSN: 0976-2922*, pages 107–118, 2010.

[CC10b] **M.C. Chisăliţă-Creţu**. Formalizing the refactoring impact on internal program quality. In *Proceedings of the Symposium "Zilele Academice Clujene" (ZAC2010)*, pages 86–91, 2010.

[CC10c] **M.C. Chisăliţă-Creţu** and A. Mihiş. A model for conceptual modeling evolution. In *The 7th International Conference on Applied Mathematics (ICAM 2010), September 1-4, 2010, Baia-Mare, România*, 2010.

[CC10d] **M.C. Chisăliţă-Creţu**. The optimal refactoring selection problem - a multi-objective evolutionary approach. In *The 5th International Conference on virtual Learning (ICVL 2010), October 29-31, 2010, Târgu Mureş, România*, pages 410–417, 2010.

[CC10e] **M.C. Chisăliţă-Creţu**. A refactoring impact based approach for the internal quality assessment. In *The 7th International Conference on Applied Mathematics (ICAM 2010), September 1-4, 2010, Baia-Mare, România*, 2010.

[CC10f] **M.C. Chisăliţă-Creţu**. The refactoring plan configuration. a formal model. In *The 5th International Conference on virtual Learning (ICVL 2010), October 29-31, 2010, Târgu Mureş, România*, pages 418–424, 2010.

[CC11] **M.C. Chisăliţă-Creţu**. *Advances in Computer Science and Engineering*, chapter The Entity Refactoring Set Selection Problem - A Solution Representation Analysis. IN-TECH, *accepted book chapter*, 2011.

[CCcC06] **M.C. Chisăliţă-Creţu** and C.A. Şerban. Impact on design quality of refactorings on code via metrics. In *Proceedings of the Symposium "Zilele Academice Clujene" (ZAC2006)*, pages 39–44, 2006.

[CCV09a] **M.C. Chisăliţă-Creţu** and A. Vescan. The multi-objective refactoring selection problem. *Studia Universitatis Babes-Bolyai, Series Informatica, ISSN:2065-9601*, Special Issue KEPT-2009: Knowledge Engineering: Principles and Techniques(July 2009):249–253, 2009.

[CCV09b] **M.C. Chisăliţă-Creţu** and A. Vescan. The multi-objective refactoring selection problem. In *Proceedings of the 2nd Internaltional Conference Knowledge Engineering: Principles and Techniques (KEPT2009)*, pages 291–298. Presa Universitară Clujeană, 2009.

[CY79] L. Constantine and E. Yourdon. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.

[DB04] B. Du Bois. Opportunities and challenges in deriving metric impacts from refactoring postconditions. In *In Proceedings of the Fifth International Workshop on Object Oriented Reengineering (WOOR2004), ECOOPworkshop*, 2004.

[DBM03] B. Du Bois and T. Mens. Describing the impact of refactoring on internal program quality. In *In Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), ICSM-workshop*, 2003.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *In Proceedings of OOPSLA 2000, ACM SIGPLAN Notices*, pages 166–178, 2000.

[Deu89] L.P. Deutsch. Design reuse and frameworks in the smalltalk-80 system. *Software Reusability: Applications and Experience*, II(1):57–72, 1989.

[Fow99] M. Fowler. *Refactoring Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fow04] M. Fowler. Refactoringmalapropism, 2004.

[FP97] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* 2nd Edition, PWS Publishing Company, 1997.

[FS97] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[GP95] D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.

[IEE92] IEEE. *Standard 1061-1992 for a Software Quality Metrics Methodology.* New York: Institute of Electrical and Electronics Engineers, 1992.

[IEE99] IEEE. *Standard IEEE Std 1219-1999 on Software maintenance Volume 2.* IEEE Press, 1999.

[ISO91] ISO/IEC. *9126 Standard, Information technology. Software product evaluation. Quality characteristics and guidelines for their use.* Switzerland: International Organization For Standardization, 1991.

[ISO99] ISO. *Standard 14764 on Software Engineering. Software Maintenance.* ISO/IEC, 1999.

[Joh93] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON'93), Toronto, Canada*, pages 171–183, 1993.

[Kd05] Y. Kim and O.L. deWeck. Adaptive weighted-sum method for bi-objective optimization: Pareto front generation. *Structural and Multidisciplinary Optimization*, 29(2):149–158, 2005.

[Ker04] J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, 2004.

[KIAF02] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings International Conference on Software Maintenance*, pages 576–585, 2002.

[KN01] G. G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Master's thesis, University of Bern, Diploma thesis, 2001.

[LR00] M.M. Lehman and J.F. Ramil. Towards a theory of software evolution and its practical impact. In *Invited Talk, Proceedings of International Symposium on Principles of Software Evolution (ISPSE2000)*, pages 2–11. Press, 2000.

[Mar02] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timisoara, 2002.

[ML02] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):–, 2002.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[MV98] H. R. Maturana and F.J. Varela. *The Tree of Knowledge: The Biological Roots of Human Understanding*. Shambhala Publiccations,Inc., Boston, MA., USA, 1998.

[MVEDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks, PhD thesis*. Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[Par79] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–128, 1979.

[Pig97] T.M. Pigoski. *Practical Software Maintenance. Best Practices for Managing Your Software Investment*. John Wiley and Sons, 1997.

[Rob99] D.B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[Roy70] W.W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON, IEEE Computer Society Press (August 1970) Reprinted in Proc. International Conf. Software Engineering (ICSE) 1989, ACM Press*, pages 328–338, 1970.

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.

[Swa76] E.B Swanson. The dimensions of maintenance. In *Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society*, pages 492–497, 1976.

[TK03] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceeding of the European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE Computer Society Press, 2003.

[Ver04] J. Verelst. The influence of the level of abstraction on the evolvability of conceptual models of information systems. In *Proceedings of the International Symposium on Empirical Software Engineering (IIESE04), Los Angeles, IEEE CS Press*, pages 17–26, 2004.