A Computational Solution for the Software Refactoring Problem: From a Formalism Toward an Optimization Approach

David Alberto Nader Palacio
Ingeniero, Ms.(c)
Code: 2702283

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
May 2017

A Computational Solution for the Software Refactoring Problem: From a Formalism Toward an Optimization Approach

David Alberto Nader Palacio
Ingeniero, Ms.(c)
Code: 2702283

This dissertation is submitted for the degree of
Master in Computer Engineer

Advisor
Jonatan Gómez Perdomo, Ph.D.
Doctor in Mathematics

Research line
Evolutionary Computation & Software Maintenance

Research group
Research Group on Artificial Life

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
May 2017

Title in English

A Computational Solution for the Software Refactoring Problem: From a Formalism Toward an Optimization Approach

Abstract: Software Refactoring consists in reconstructing the code design of an object-oriented system without affecting the external functionality. Common refactoring tasks, in the initial steps, detect the source code components to be affected and recommend accurate Refactoring Operations to be applied to those components (an estimation problem). In this document, the author defines, develops and evaluates an Artificial Refactoring Hybrid Adaptive Technique (ArHaT) to estimate refactorings as a combinatorial model. The author investigated whether Refactoring Impact Prediction yields sets of artificial refactoring operations before implementing them into the code. ArHaT estimates the sets of Refactoring Operations according to some predefined weight values of the quality metrics. The weight values or coefficients capture the developer error-prone knowledge of the code in the objective function. The author conducted several fitness performance evaluations in two open software systems and organized them as a set of experiments. The goal of the experiments is to minimize a ratio between impacted quality metrics and actual quality metrics. Hill-Climbing, Simulated Annealing, and Hybrid Adaptive Evolutionary Algorithm assembled feasible refactorings, yet the latter accomplished the best performance. This research establishes the fundamentals of the automation of the refactoring problem and contributes to reducing the gap between the software defect prediction and the software refactoring.

Keywords: Evolutionary Computation , Refactoring, Evolutionary Algorithm, Software Engineering

Acceptation Note

Thesis Work
Aprobado

"Meritoria o Laureada mention"

Jury
Mario Linares., PhD.

Jury
Jairo Aponte., PhD.

Advisor
Jonatan Goméz., PhD.

Bogotá, D.C., June 31 of 2017

Dedication

To my mom who always demonstrates her restless love.

Acknowledgement

The author gratefully acknowledge the support of the thesis director Jonatan Gomez for his paternal advice, unflagging effort and determination for teaching during this project. The author thanks the participants to the experimental study; Daniel Rodriguez, Oscar Chaparro and Miguel Ramirez who contributed to this research with the approach's implementation and statistical analysis; Carlos A. Sierra for his continue support during the last four years of the project (and the last eight years of friendship); the former members of ALIFE and SEMERU research groups and the anonymous reviewers for their valuable feedback. The author wants to thank all institutions who contributed directly or indirectly to this project: Universidad Nacional de Colombia, Technical University of Munich, Center for Development of Advance Computing and KSMTI (Sandra, German & William). Finally, the author thanks his mom, aunts, cousins and closest friends (Andre, Diego, Pao, Jorge, Carlos, Miguel) for all the emotional support.

Contents

List of Tables

List of Figures

Introduction

At the 2012 International Conference on Software Engineering (ICSE 2012), industrial software practitioners discussed the state of the art in data mining and software engineering [21]; panelists stated that "prediction is all well and good, but what about decision making?" Ergo, practitioners are more interested in the interpretation that follows data mining, rather than just about the mining process. For instance, the interpretation of software defect data is a time-consuming and expensive task. B. Murphy (from Microsoft, UK) states that the high cost of interpretation implies that there is never sufficient time to review all software defect data or every model generated by data miners [4]. Generating actionable recommendations (to automatically modify the code) from software defect data is an open research problem.

Professors Menzies [1] and Marcus [2] proposed a REcommenDation for REducing Software defectS (REDRESS). REDRESS is a solution to generate actionable recommendations from software defect data that provides developers with decision support when changing software to make it less error-prone (a probability of finding errors/defects in the code). The specific goal is to offer software engineers "actionable refactoring recommendations" that shall change the software properties related to error-proneness.

REDRESS is based on producing new technology that shall transform defect prediction and software change practices. The solution is composed of two automatically recommendation techniques: REDRESS1 that recommends how software properties should be changed to reduce error-proneness (currently REDRESS1 is proposed, but not developed) and REDRESS2 that recommends a set of refactorings that achieves the property changes recommended by REDRESS1. The first technique tells the developer what software properties need to change, while the second technique tells the developer how to change the software.

Basically, REDRESS2 is the study of automatic techniques for generating or constructing refactoring. Providing refactoring recommendations is a widespread practice that assists developers in ameliorating the maintainability and readability of their code [8]. However, constructing massive refactoring recommendations is an arduous task for maintenance groups [19, 18]. Researchers reported considerable approaches that focus on combinatorial techniques to solve the construction of refactoring recommendations [17].

[1] Professor Menzies has extensive experience in defect prediction https://scholar.google.com/citations?user=7htTUTgmLtUC

[2] Professor Marcus has expertise on refactoring recommendation tools https://scholar.google.com/citations?user=ZZiaPdYAAAAJ

This document introduces an implementation for the second component of REDRESS original research.

Refactoring is the process of modifying unstructured code from an object-oriented software by improving the system quality without altering its external behavior [8, 37]. The research community has made advances related to automated detection of bad smells and refactorings, although the detection remains complex [17].

Current research in software maintenance improves the determination of refactoring solutions by considering some design features [28], semantic coherence of the refactoring [39], and recorded code changes [42] in different software approaches. Even though, ignores the defect information, hence those approaches do not help developers to decide how to change the code to make such code less error-prone. For instance, Mkaouer et al.[26] propose a robust multi-objective approach for dealing with refactoring configuration. Three objectives are intended to maximize: the quality improvements, the severity of bad designs and the importance of the affected classes. In addition, Ouni et al. [41] introduce a multi-objective optimization based on the use of development history for the automation of refactoring process. Besides, Seng et al. [50] present a primitive form of single objective solution, a search-based approach for reconditioning the class structure of a system. The core of the proposed model is refactoring simulations and violations of design patterns. Likewise, there are approaches based on genetic programming, as proposed by Jensen & Cheng [11], that use design patterns and software engineering metrics in order to generate sets of refactorings.

Approaches that suggest refactoring operations must be very clear on how those refactorings were generated. Any software researcher ought to reproduce proposed approaches on the same analyzed data sets and expect the same reported results. Besides, each approach must give a sound, concise and justified answer to the following questions: Why do researchers use combinatorial optimization? Which are the variables of the model? Which are the parameters or the constraints? How are the refactorings built and performed? Do the reported optimization models explain the automatized refactoring steps?

Unfortunately, current research [24, 26, 25, 25, 43, 38, 41, 39, 42] proposes informal optimization models for the refactoring problem, making the experiments difficult to compare and reproduce. The implementation and execution of refactorings, which are described in reported models, are ambiguous; for instance, Ouni *et al*'s "DefectCorrection" algorithm did not clarify how the refactoring sequences were computed on the code to assemble final recommendations [40].

Using a mathematical approximation to the refactoring problem, software researchers can reproduce and report combinatorial optimization approaches in a standardized manner. A formalization development for finding refactorings is a proper hypothesis about modeling techniques for building artificial refactorings. The formalism allows researchers to tackle the software refactoring problem in a new way. For example, a researcher might introduce a model that can estimate the effect of proposed refactoring sequences before these sequences are implemented on the code. Rather than starting from scratch, the research community would substantially benefit if adopting a mathematical representation of the refactoring problem.

In this document, I introduce an approach to **estimate** refactorings that impacts the quality metrics before any change on the source code. Artificial Refactoring GENeration (ARGen) tells developers how to change the software, but it does not tell what software

properties need to change. This research hypothesized that techniques based on evolutionary computation constitute appropriate approaches for building artificial refactorings. The proposed approach solves this problem by using a math model and then generating massive artificial refactorings with combinatorial optimization. One of the research purposes is to compare baseline combinatorial algorithms (Hill Climbing and Simulated Annealing) performance to an evolutionary algorithm. The behavior of fitness values, during fewer and larger evaluations, were investigated on two different open systems (written in Java).

Besides, software quality metrics allow this research to guide the construction of refactoring operations. Such metrics do not suggest any refactoring opportunities by themselves; however, the metrics aid fitness function in recommending a suitable sequence of refactorings. Refactoring parameters (e.g. methods, classes or attributes) and some weights/metric coefficients (for ensuring the decrease of the error-proneness) are the inputs for the fitness function. Moreover, the refactoring fitness constraints take into account the object-oriented style guidelines.

Artificial Refactoring GENeration (ARGen) allows researchers to approach software refactoring in a new way: estimating the effect of proposed changes before such changes are implemented and assessing a large number of alternative changes. The research work presented in this document is expected to contribute to the decision making process within software companies and to produce high quality code by reducing error-proneness [3].

In summary, our research contributes to both theory and practice in Software Engineer as follows:

- a definition, a development, and an evaluation of a technique that estimates a set of refactoring operations, which achieve parametric software properties recommended by developers or any other approaches;
- a mathematical definition to the refactoring problem;
- a combinatorial problem for the refactoring generation definition;
- a computational complexity classification for the refactoring problem;
- a refactoring combinatorial solution through a Hybrid Adaptive Evolutionary Algorithm (HaEa);
- an empirical study that evaluate the performance of the refactoring problem between baseline algorithms (Hill Climbing and Simulated Annealing) and HaEa;
- a time complexity analysis of the approaches:
- a paper that describes a computational solution for the software refactoring problem [4] and;
- a paper that generalizes the theory to estimate artificial software refactorings [5].

The remainder of this document is organized into six chapters: Chapter 1 is a reproducibility analysis of reported studies of automatic refactorings. Chapter 2 is a mathematical formalism for dealing with refactoring concepts used in Chapter 3. Chapter 3 is

[3] Regarding the impact on the research community and industry, all the artifacts of the project are available for any researcher or developer who needs to perform refactoring analysis: `https://github.com/anonypeople/argen`

[4] A Computational Solution for the Software Refactoring Problem: From a Formalism Toward an Optimization Approach

[5] Theory and Combinatorial Models to Estimate Artificial Software Refactoring Operations with Impact Metric Prediction

an explanation of using combinatorial optimization for the software refactoring. Chapter 4 & 5 is the implementation and evaluation of the proposed approach. Chapter 6 is an evolutionary computation proposal for the Artificial Refactoring Generation. Chapter 7 is a complete discussion of the results. General conclusions is the last chapter. Therefore, this documents presents two parts: Model Refactoring Definition (chapter 2; chapter 3) and Model Refactoring Implementation and Evaluation (chapter 4).

CHAPTER 1

A Reproducibility Analysis of The Refactoring Optimization Models

Refactoring is one of the most common solutions developers adopt to improve software quality by changing vulnerable code while preserving software's functionality [8]. Developers, based on their experience, usually perform refactoring operations to reduce "error-proneness" in an organized fashion.

The experimental component in software maintenance research should be reproducible in order to assess introduced approaches. Some studies have reported with such characteristic, allowing other researchers to generate verified analysis.

The aim of this chapter is to make a reproducibility analysis about the most relevant research in automatic generation of refactorings. Therefore, this chapter covers the search techniques, applied to the refactoring problem, reproducible for tasks that require further analysis (e.g. proposing new techniques for automating the software refactoring).

The search strategy, which was employed in this analysis, is composed of the identification of databases and the exclusion and inclusion criteria. Scopus, IEEE, Science Direct, Springer and ACM were used because such databases are the most relevant for software engineering. Exclusion criteria includes: i) papers that do not describe an optimization model for solving the refactoring problem; ii) different languages than English; and iii) any abstract, technical report, doctoral symposium, patents or conference review. Inclusion criteria includes any paper or study that applies optimization approaches to solve the software refactoring problem.

The previous strategy helped to configure the State-Of-The-Art of the refactoring optimization models. From that State-Of-The-Art, Mens & Tourwe's survey analysis was used for generate a conceptual map (Figure 1.1). The map not only introduces major topics in software refactoring, but also shows three aspects: in red "Unclear State-Of-The-Art", in blue "Analyzed topics", and in green "Contributions" in this research work. In addition, "Analyzed topics" includes the definition of refactoring, the software refactoring activities (specially those which are implemented in automatic approaches), software metrics as a formalism, and some object oriented guidelines for preserved program behaviour.

The map is divided into four areas. The first area (Figure 1.1 - 1) is the *refactoring definition* embedded into the *restructuring definition*; refactoring is the term employed

when restructuring software in the context of object-oriented development. The second area (Figure 1.1 - 2) is the *context perspective* or other software fields that show interest about software refactoring. Software evolution and re-engineering are the main fields that support refactoring. Even though, the motivation of making a software restructuring comes from requirements more than bad code smells -software requirement field has shown interest in software refactoring [1]. The third area (Figure 1.1 - 3) is *software refactoring activities*; there are six activities that must be done when performing a software refactoring (from detection of source code components to synchronization of the source code modifications with other artifacts). A full automatic approach must consider such six activities altogether. The fourth area (Figure 1.1 - 4) describes *techniques and formalism* for treating the refactoring problem. The techniques include concepts about refactoring representation and mechanism for automatic implementations.

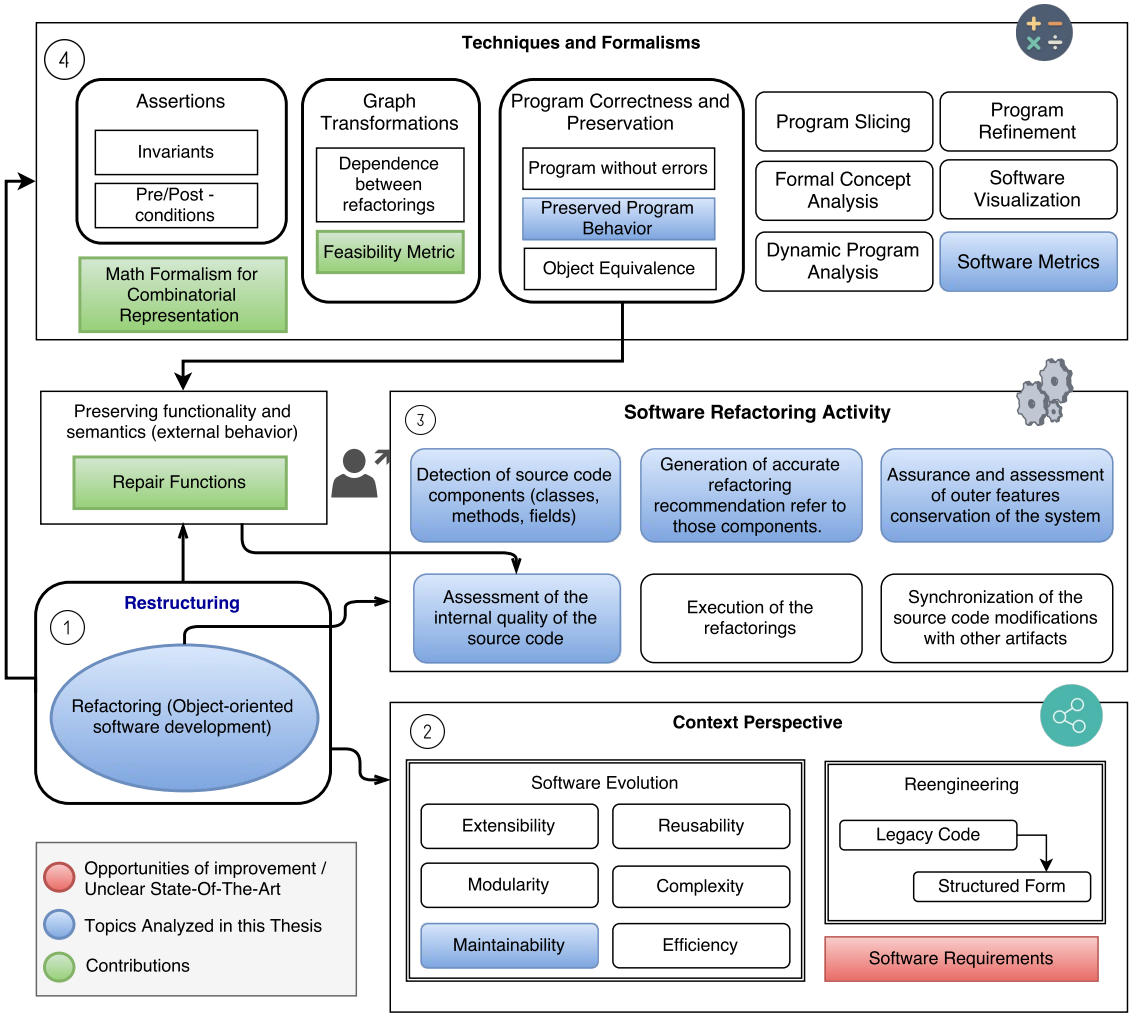


FIGURE 1.1. Refactoring Conceptual Map

1.1 Software Refactoring Background

Software deteriorates during its life-cycle because of continuous changes performed by developers; the software drifts away from its original design and becomes more difficult

and costly to maintain. Those changes often introduce faults and lead to software decay over time. When dealing with a specific piece of software, an instrument or tool is required to notify developers *how the software must be changed to reduce its defects* [10, 5]. State-of-the-art defect predictors estimate the error-proneness of a code component (packages, classes or methods). However, defect predictors' capacity to inform *how to change code component properties to make such code less error-prone*, is still questionable [46, 20].

The software refactoring is a process composed of certain activities introduced in figure 1.1 third area. Figure 1.2 shows the order in which the activities should be executed [17, 19, 18]. The first activity is the detection of source code components. Any automatic approach should identify which are the classes, methods or packages that should be refactored. The second activity is the recommendation of correct refactoring operation to be applied on these code components. Any automatic approach should suggest which are the refactoring operations that ought to be applied on the identified components in the previous stage. The third activity is assessment and assurance of the preservation of external behavior. Any automatic approach should preserve the functionality of the system after applying the refactoring (which conditions should be taken into account when dealing with an specific refactoring?). The fourth activity is the implementation of the refactoring. Any automatic approach should implement the sequence of refactorings or the recommendation on the code. The fifth activity is the assessment of the internal quality of source code. After applying the refactoring, the automatic approach should ensure the quality of the code. The sixth activity is the synchronization of the source code modification with other artifacts. Any automatic approach should preserve the traceability and consistency between the artifacts of the same software. Therefore, the State-Of-The-Art approaches are meant to automatize these activities.

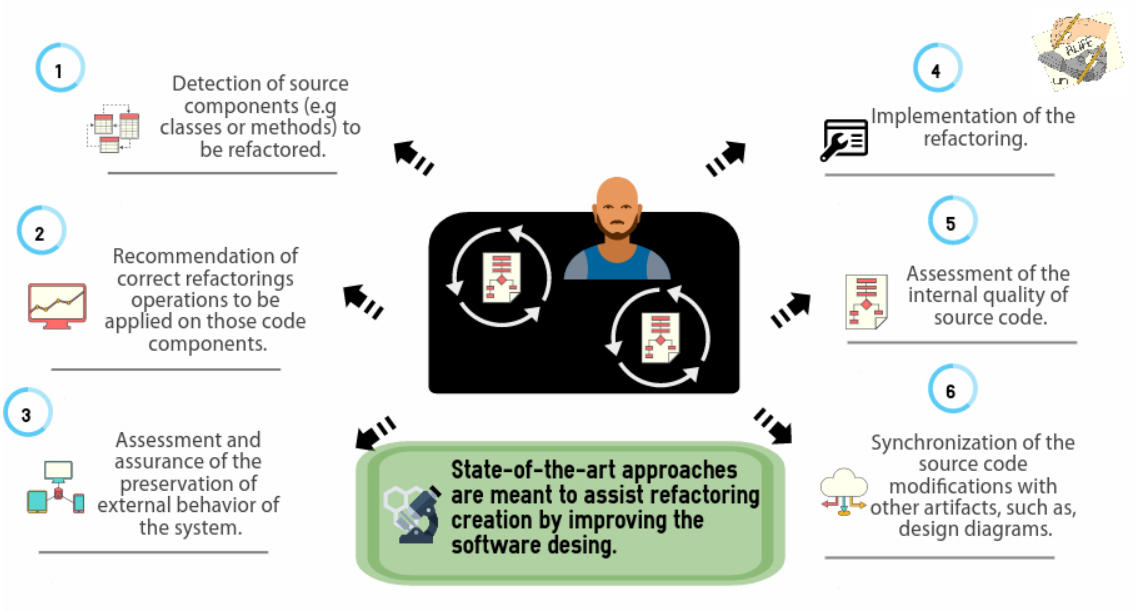


FIGURE 1.2. The Software Refactoring Process

The refactoring problem is a set of steps that represents how a refactoring is performed by developers. The problem is represented as a Cycle in figure 1.3. The first stage is when the software deteriorates during its life-cycle. The second stage is when developers perform changes on the code to preserve functionality (e.g. any modification for maintaining

the external functionality when implementing requirements). The third stage is when developers introduce undesired faults when performing the previous changes. Then, in the fourth stage, the software decays even more. The solution for that decay [1] is to implement proper refactoring sequences, in the fifth stage, according to the expertise of developers. In the last stage, the code changes and might reduce the error-proneness.

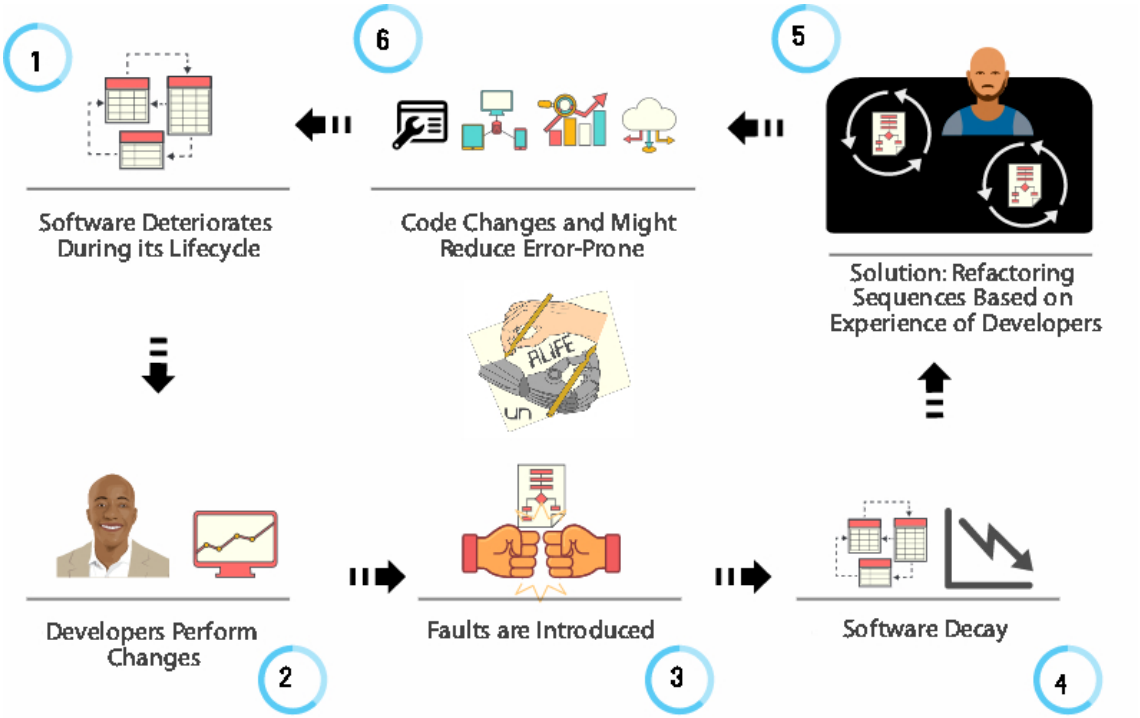


FIGURE 1.3. The Refactoring Problem

1.1.1 Search-Based Approaches

These types of approaches are represented by a cluster of researches that have significantly contributed to the refactoring problem and generated collaborations among them. Approaches are compared to other reported search-based solutions, manual refactoring detection and non-heuristic tools (e.g. JDeodorant[52]).

In order to preserve class level refactoring, Seng *et al.* [50] propose an evolutionary algorithm for optimizing class structure in an open source system. The value of several quality metrics and the number of violations of object-oriented design guidelines support the workflow to produce refactorings. However, if only considers only one refactoring operation (move method) and a restricted number of classes, the search space is significantly reduced.

Harman *et al.* [10] introduce a search-based refactoring approach based on pareto optimality by extending the initial idea of Seng *et al* [50]. They use one refactoring (move method) and two metrics (coupling between objects and standard deviation of methods per class) for determining a *sequence of refactoring*. The procedure for applying the sequence to the system is unclear (as well as the importance of choosing those metrics).

[1] Different from other techniques such as re-modularization, modernization, or architecture re-design.

Ouni *et al.* [40, 43] suggest using development history to assemble refactoring solutions in similar context. NSGA-II is used for minimizing the number of bad smells and maximizing the semantic coherence and *the development history use*. Such approach finds refactoring solutions composed of sequences of refactoring operations where the order of the operations is taken into account. Vector representation of candidate solution is used for preserving pre and post conditions. Crossover and mutation are the only genetic operators implemented for NSGA-II approximation. Furthermore, authors test a chemical reaction optimization approach to find solutions of refactorings that maximize the number of fixed riskiest code smells[38] and use genetic programming to generate rules that detect defects when minimizing software effort.

Mksover *et al.* [24] orient their research to the high dimensional problem [25] in refactoring determination. Their extensive research in multi-objective optimization, includes the following features: quality improvements (number of fixed code smells), severity of code smells and importance of the classes that contain bad smells (leading to lack of robustness) [26]. Moreover, they propose a tool that accepts developer feedback and adapts to suggest refactoring operations [25]. Even though, any computational execution of refactoring needs not only exhaustive code validations, but also a deep knowledge of the system under analysis; thus, increasing computational time complexity.

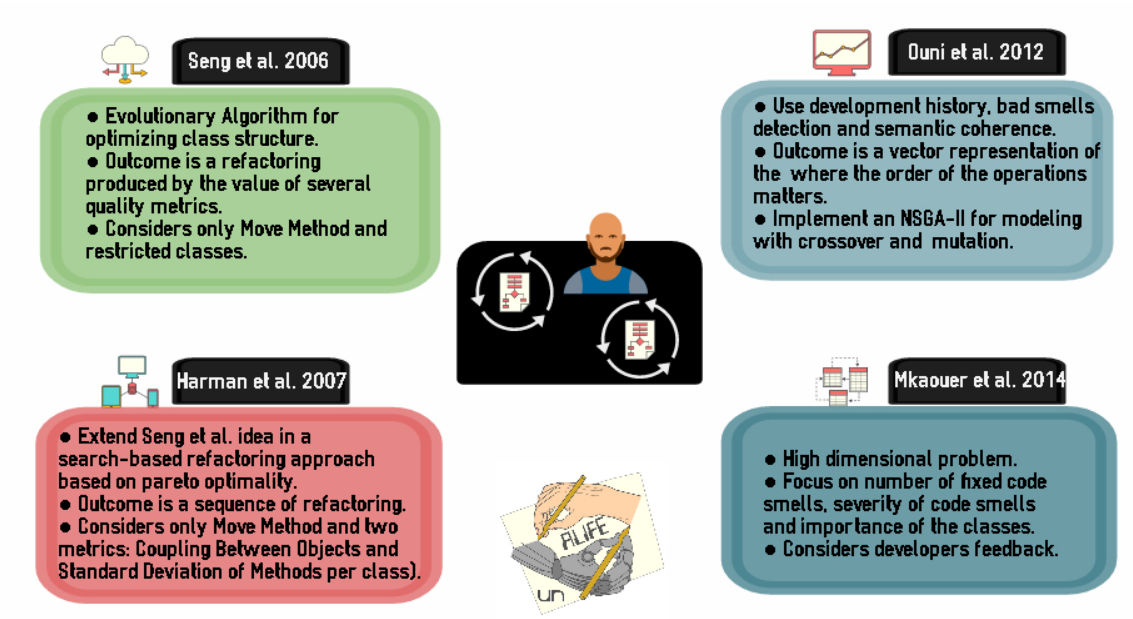


FIGURE 1.4. State-of-the-art Approaches

1.2 A Discussion of Search-Based Techniques

Search-based approaches share the same framework described in figure 1.5 [17]. The first box depicts the inputs, the search techniques and the outputs concepts; the inputs could be a representation of an artifact (e.g. a graph or a metaphor) or the artifact itself (e.g. source code). A search technique could be any optimization algorithm (e.g hill climbing, ant colony or NSGA-II) that has an individual and fitness definition. Furthermore, the search technique receives a refactoring (e.g. Fowler's Catalog), some metrics (e.g. number

of methods, cyclomatic complexity or coupling) and additional parameters (e.g. development history, semantic information or bad code smell information). Finally, the solution set is generally a sequence of refactorings, which are the solutions or refactoring recommendations for a system. However, none of the search-based approaches has shown any concrete example of those sequences or vectors.

The second box, in figure 1.5, depicts five important topics when performing a study design and evaluation of search-based approaches: solution, behaviour preservation, search technique, refactorings and metrics. Each topic contains subtopics that one can easily identify in the conceptual map, like "the techniques used for behavior presentation are Opdyke's functions, Cinneide's functions or domain specific". The red color indicates that the behavior preservation topic is unclear in the existing state of the art. The reason for unclear states are detailed in the reproducibility analysis (subsection 1.2.1).

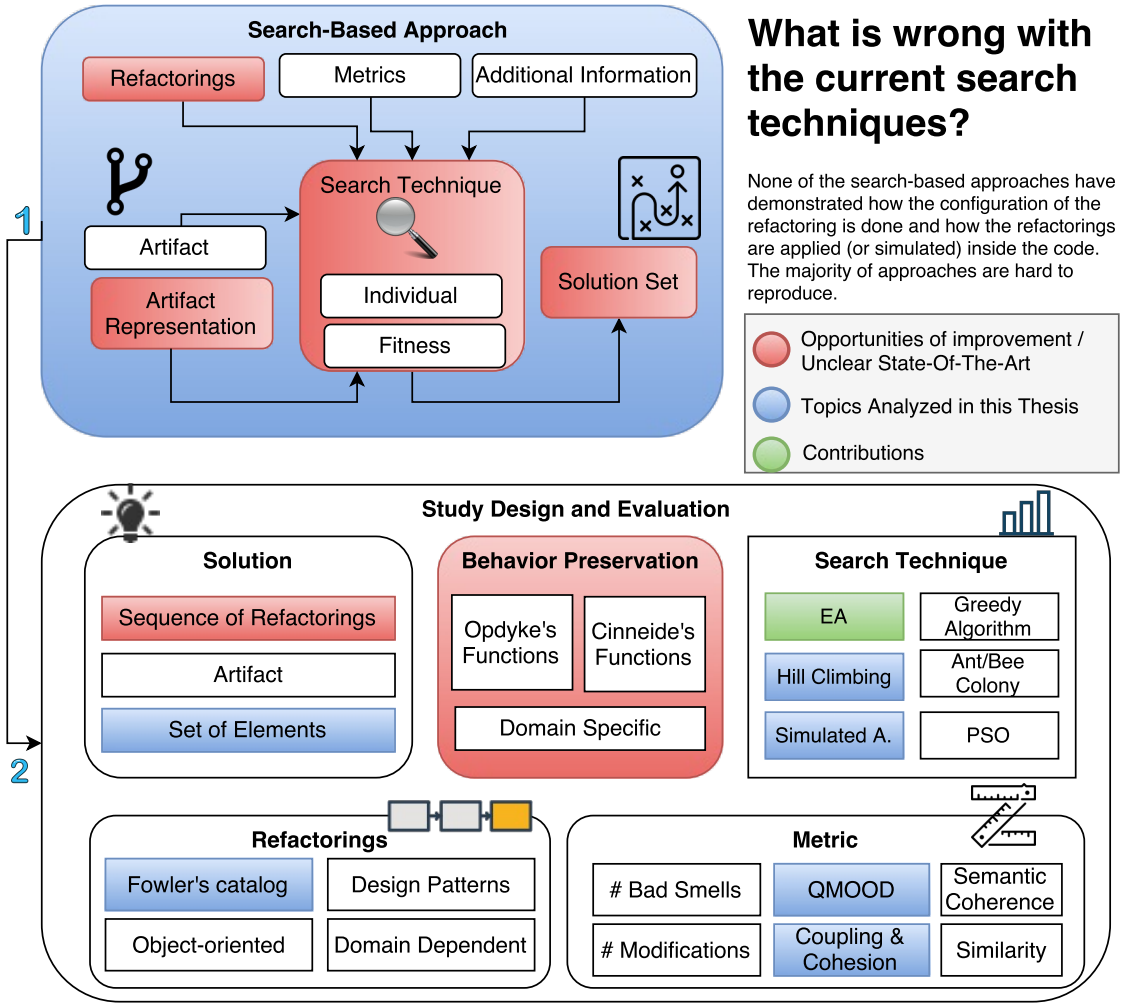


FIGURE 1.5. Search-Based Conceptual Map

Regardless of the efforts to present a model for generating refactoring recommendations, the published search-based models have weak consensus on how to represent a refactoring. Experiments with such models are hard to reproduce and information for configuring them is usually omitted. Besides, when comparing with previous approaches, results are biased due to hard reproducibility of reported data as evidenced in Ouni *et al*'s.

research [41]. The problem of hard reproducibility of empirical studies in software maintenance is well described by Bogdan *et al.*[7]; Bogdan *et al* explain that the researchers' interpretation of applying approaches from original authors could have a significant impact on the results when original authors take for granted their assumptions in "well known" techniques (e.g. NSGA-II, Vector Space Model, Genetic Algorithm).

Some researches are oriented to fix somehow the occurrence of code smells [26, 43]. Two recent studies [1, 51] rebut the idea of including code smell resolutions in determining refactoring operations, yet these studies suggest the incorporation of the code changes in the refactoring problem as evidenced in the development history research of Ouni *et al.*

According to a study of Silva *et al.*[51], a manual refactoring is often applied than any automated refactoring approach; the most frequent reason to manual application is "lack of trust" (developers argue that they do not trust automated support for complex refactorings). Additionally, the refactoring process is highly motivated by changes in the requirements (new features or bug fix requests) and slightly motivated by code smell resolution. Hence, the analysis of error-proneness is relevant for identifying refactoring opportunities.

Figure 1.6 states the gaps found in the State-Of-The-Art. Search-based techniques present four gaps: 1) weak consensus on how to represent a refactoring, 2) hard reproducibility of the experiments, 3) omitted parametric information and 4) oriented to fix the occurrence of code smells without harnessing error-proneness information.

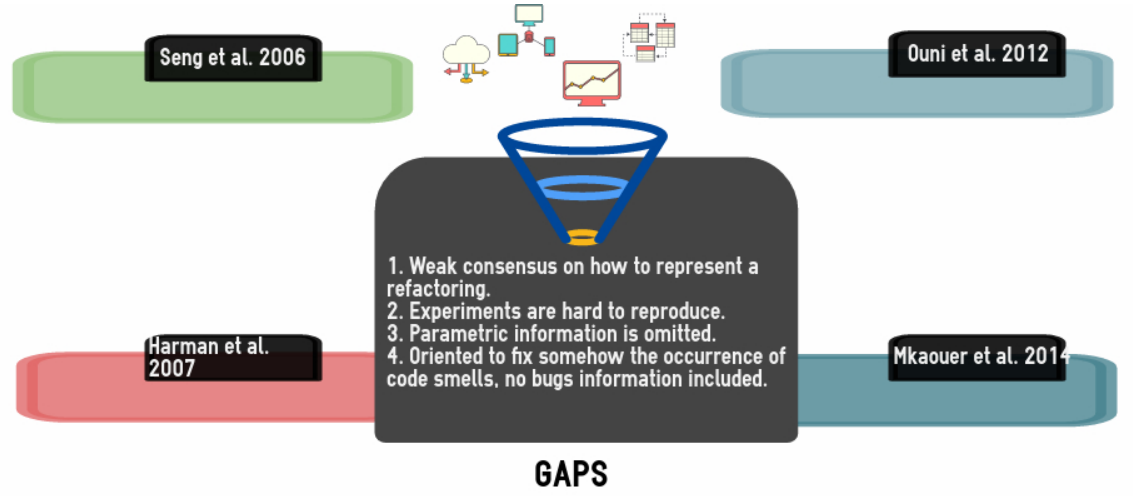


FIGURE 1.6. State-of-the-art Gaps

1.2.1 The reproducibility analysis

This analysis was performed and traced in a seven dimension conceptual matrix. Each dimension is defined by the following characteristics: optimization technique, objective(s), type of solution, evaluation method, analyzed systems, supported refactoring and behaviour preservation.

The "optimization technique" dimension describes the used approach and the approach's implementation.

In the "objective(s)" dimension, the idea was to identify which fitness functions were employed, numerate each fitness function and identify if the function is maximizing or minimizing. The most important part is to identify where in the fitness function the refactoring is executed.

The "type of solution" dimension describes how the configuration of the final solution is represented (general description). If an evolutionary algorithm is considered, then the solution is the representation of the individual. This dimension embodies a general idea of how the refactoring looks.

In the "evaluation method" dimension, the study design is exposed and analyzed. This dimension contributes to understand how the approaches are tested and to what extent are biased (qualitative assessment).

In the "analyzed systems" dimension, used open software are listed. Besides, the "supported refactoring" dimension lists the refactoring operations used from Fowler's Catalog [8] (or other refactoring classification). For researchers, it is important to know what systems are evaluated and which refactorings are used because the reproducibility highly depends on both dimensions.

The last dimension is "behavior preservation". The description of how the State-Of-The-Art approaches approximate to preservation of the system functionality is one of the most relevant activities. Basically, non-reproducible approach occurs when such approach does not fulfill the purpose of a refactoring (e.g. changes without affecting functionality). Therefore, approaches are non-reproducible whether such approaches do not describe how to guarantee/assess behaviour preservation.

TABLE 1.1. Comparison of Combinatorial Techniques Related to The Refactoring Problem

Authors	Journal/Venue	Google Scholar Citation Count	Optimization Technique	Objective(s)	Type of solution	Dimension Analysis Evaluation Method	Analyzed Systems	Supported Refactorings	Behavior Preservation	Reproducibility
Seng & Stammel (2006)	GECCO 2006	232	Single Objective EA (unclear technique).	Maximize features such as complexity, stability, coupling and cohesion by means of a weighted sum of seven metrics values. The input is the source code.	A list of model refactorings. The list is ordered according to established pre/post conditions of refactorings.	Fitness convergence analysis and an example code development.	JHotDraw	Move Method	Domain specific preconditions based on the object oriented structure and design patterns.	Moderately Hard
Keeffe, Mark O Cinnéide, Mel Ó (2007)	Journal of Software Maintenance and Evolution	101	Multiple-ascent hill climbing, simulated annealing and genetic algorithms.	An implementation of the unserstandability function from QMOOD model. The input is an Abstract Syntax Tree (AST).	The refactorings are applied to the AST, then the outputs are the refactored input code (unclear solution example).	Mean analysis on fitness values and execution time for each search technique.	SpecCheck, Beaver, EAOP, Mango and Grammatica.	14 Refactorings from Fowler's Catalog.	Static program analysis on an unknown pre/post-conditions. There is no evidence of how the refactoring is executed on the AST.	Hard
Harman & Tratt (2007)	GECCO 2007	173	Variant of hill climbing (unclear).	Two objectives: metric coupling between classes (CBO) and the standard deviation of methods per class (SDMPC).	Sequences of refactorings, although, it is unclear how the solution is configured.	Pareto front analysis. However, without concrete type solutions or outcome examples, the analysis is hard to interpret.	JHotDraw, Maven and Xom	Move Method	NA	Hard
Jensen & Cheng (2010)	GECCO 2010	52	Genetic programming.	A proposed fitness function composed of the QMOOD model, specific penalties,and number of modifications.	The individual is composed of a pair that includes a transformation tree and design graph (UML class diagram).	Exploratory analysis on fitness values. However, the analysis are very informal and no concrete solution is described.	The use of RE-MODEL on published models and real-world case study.	Gamma design patterns and mini-transformations.	NA	Hard
Ouni & Kessentini (2013)	GECCO 2013	24	NSGA-II (unclear how the refactorings were executed).	Maximize design quality, semantic coherence and the re-use of the history of changes.	Vector-based representation. A set of refactorings is configured and sorted using dominance principle and a comparison operator based on crowding distance. The order of the operations inside the vector is important.	DCR (defect correction ratio) and RP (refactoring precision). It is unclear how they reproduce previous approaches. The use of RefFinder (Eclipse Tool) for refactoring comparison.	Xerces-j, JFreeChart.	9 Refactorings from Fowler's Catalog.	Pre and post conditions described by Opdyke (not clear how authors performed behavior preservation).	Hard
Mkaouer & Kessentini (2014)	ASE 2014	13	NSGA-II (unclear how authors execute the 23 refactorings).	Improve software quality, reduce the number of refactorings and increase semantic coherence.	A set of Non dominated refactoring solutions. The set of refactorings are ranked.	Execution time analysis and a manual validation of the refactorings. There are comparison with other reported refactorings.	GanttProject, JHotDraw, JFreeChart, JDI-Ford.	23 Refactorings from Fowler's Catalog.	NA	Hard
Ouni & Kessentini (2015)	Software Quality Journal	6	Chemical Reaction Optimization. Although, there is no evidence or concrete example of how the refactorings are executed on the code.	One objective that minimize the number of bad smells.	A sequence of refactorings in a vector representation (unclear how it is the appearance of the suggested refactoring).	Execution time analysis and comparison to the number of bad-code smells with an old version of the systems.	Xerces-J, GanttProject, JHotDraw, JFreeChart.	11 Refactorings from Fowler's Catalog.	Opdyke's functions. However, these conditions are unclear in the implementation of the approach.	Hard
Mkaouer & Kessentini (2016)	Empirical Software Engineering	NA	NSGA-II (unclear how the refactorings are executed).	Maximize quality improvements and the respectively uncertainties associated with severity and importance of refactorings opportunities.	Vector-based representation of refactoring operations. The order depends on the position of the refactoring inside the array.	Hypervolume, Inverse Generational Distance, Number of Fixed Code-Smells, Severity of Fixed Code-Smells, Correctness of the suggested Refactorings and Computational Time.	Xerces-J, JFreeChart, GanttProject, ApacheAnt, JHotDraw, Rhino, Log4J, Nutch and JDI-Ford.	10 Refactorings from Fowler's Catalog.	Opdyke's functions (defining pre/post conditions). However, these conditions are unclear in the implementation of the approach.	Hard

The goal of the analysis is to assign one qualification to each analyzed approach according to reproducibility characteristics (or dimensions). Each dimension adds one point to the specific approach qualification when the description is unclear: zero points indicates that the approach is **easy** to reproduce; one or two points that the approach is **moderately hard** to reproduce; three or more than three points that the approach is **hard** to reproduce. Furthermore, whether the behavior preservation dimension is unclear, the approach is *hard to reproduce* (no matter how many points have been added); a major characteristic of Software Refactoring is the ability to preserve the functionality in an inspected system. Besides, the issue is not related to refactoring representation, but how authors use that representation to perform the automatic process and to obtain the reported results (it is not clear how pre/post conditions of behavior preservation are implemented inside the approach).

From table 1.1, it is implied that there is no evidence of a real refactoring outcome in any of the approaches. How does a real refactoring outcome look? How is the sequence of refactoring used to implement a manual software refactoring? Search-based techniques reported so far fail in clearly explaining how to reproduce such techniques. Besides, none of the papers reported algorithm analysis convergence -except for one (Seng, et al. [50]).

Actually, the paper published by Seng, et al.[50] is the only one that could be reproduced assuming several parameters -taking into account that the used optimization technique is not described in the paper as well. Nevertheless, the idea is clear when researchers need to tune the remaining parameters. Consequently, Seng et al.'s approach is a milestone in this context and *this document is an improvement from that approach.*

1.3 Toward a new paradigm

The research community has contributed to automatic detection of bad smells and refactorings, but the problem remains complex [17]. Existing research work consider several design features [25], semantic coherence [39] or recorded code changes [42] to determine refactoring solutions. However, researchers ignore the defect information which helps developers to decide how to change the code to make it less error-prone.

The milestones of refactoring automatic generation and analysis are depicted in figure 1.7. A timeline is introduced for localizing major contributions in the field. The timeline is based on Mariani & Vergilio's review paper [17] and further interpretations for organizing the milestones. It is clear that Opdyke [37] introduced the term in 1992 and Fowler [8] built the catalog for the refactorings. The first years, from 1992 to 2004, software research community adopted some formalism. Since 2006, with the research of Seng & Stammel [50], the search-based approaches has started a new line of research for software refactoring: evolutionary techniques with convergence analysis.

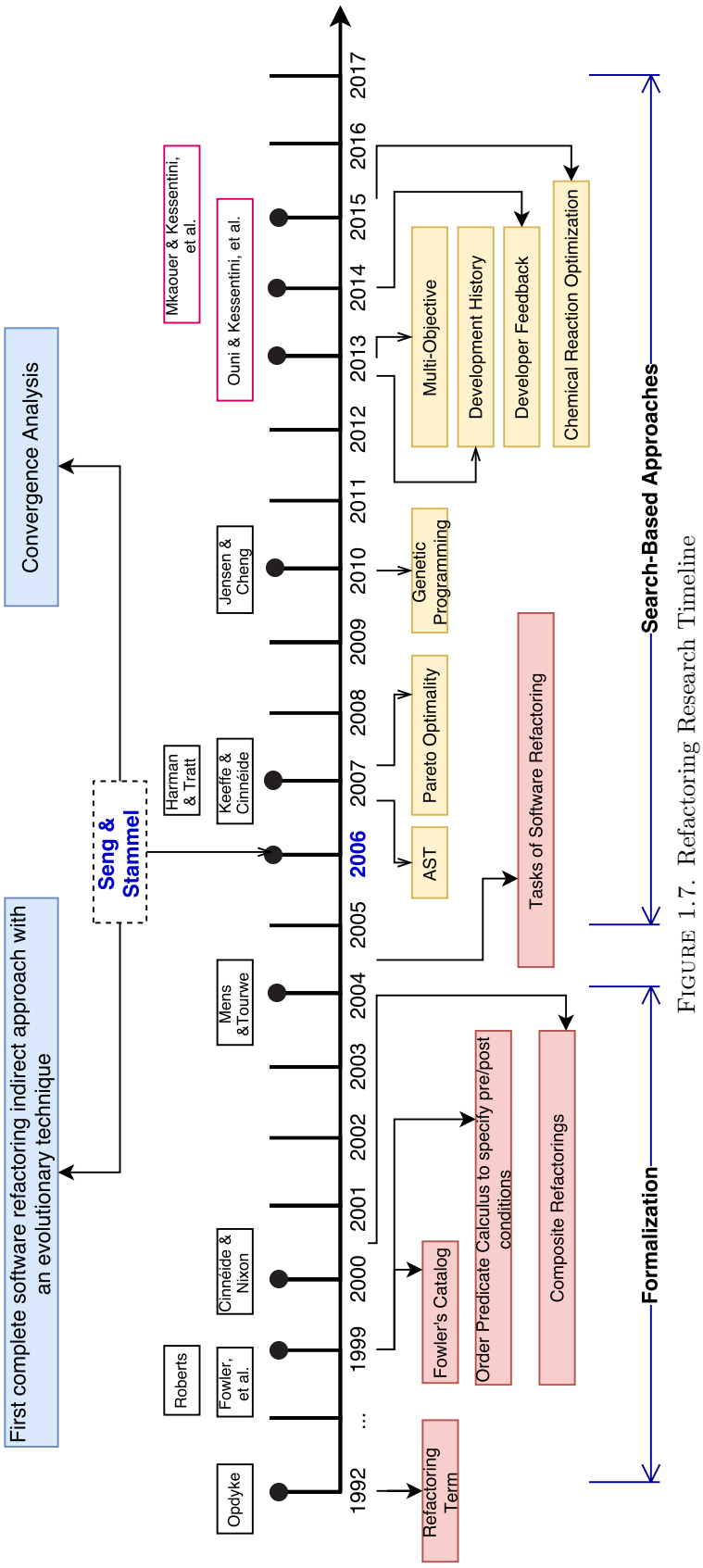


FIGURE 1.7. Refactoring Research Timeline

As evidenced throughout this section, there are some gaps between defect prediction theory (error-proneness) [20] and Software Refactoring (to provide developers with decision support when changing software to make such software less error-prone). In addition, research community has not currently reported reviews in software refactoring field that include *reproducibility analysis of the proposed approaches*.

1.3.1 Proposed Approach

This document presents a solution for the Software Refactoring Problem, defined as the artificial generation of refactoring sequences thorough a systematic formal approach: **Artificial Refactoring GENeration (ARGen)**. Moreover, this document contributes to both software theoretical study (represented by the model refactoring definition) and software empirical study (represented by model refactoring implementation and evaluation).

Figure 1.8 depicts the proposed approach (ARGen) and unifies the mathematical approximation and the combinatorial modeling for generating the refactoring sets. Besides, the approach includes six relevant aspects: 1) expected quality attributes, 2) mathematical approximation, 3) tuning, 4) search-technique or combinatorial model, 5) recommendation sets, and 6) outcome serialization. Each aspect is described as follows:

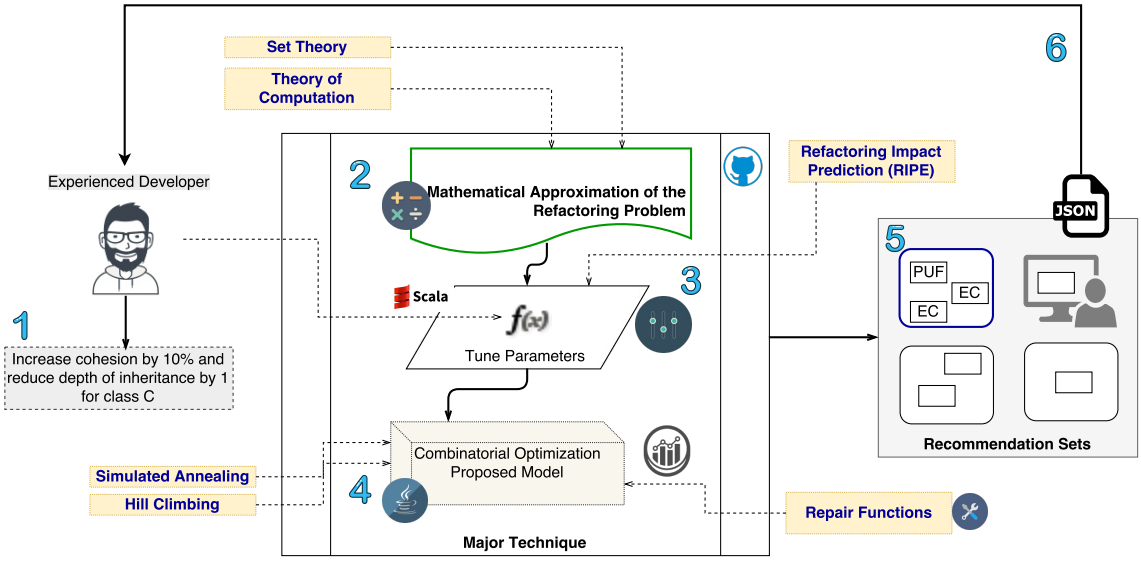


FIGURE 1.8. Artificial Refactoring GENeration (Approach Overview)

Expected Quality Attributes. The objective function is highly dependent of the parameters that represent the software properties (e.g coupling or cohesion). Therefore, the input should embody specific values that represent high level recommendations such as *increase cohesion by 20%* or *reduce depth of inheritance by 1 for class C*.

Mathematical approximation. The formalism introduced in ahead chapters is a standardization for dealing with the refactoring construction. It includes concepts from "set theory" (representation of parameters) and "theory of computation" (representation of refactorings).

Tuning. The objective function has some associated weights that can be used for introducing developer's specific information. The objective function was developed in Scala to enhance concurrency.

Search-technique. Hill Climbing and Simulated Annealing are the combinatorial algorithms for assessing this approach. It includes the repair functions based on object-oriented guidelines for behavior preservation.

Recommendation sets. The recommendation sets are the solutions of the major technique. They are composed of refactoring sequences that contain refactoring operations with their associated parameters.

Outcome serialization. The technique serialized the recommendations sets into a JSON format for an easy representation. The developer could inspect the JSON file in order to apply the recommendations in any order.

CHAPTER 2

A Mathematical Approximation to Software Refactoring

Figure 2.1 depicts, in step two, a formalism to treat object-oriented software for the refactoring problem. The formalism includes *theory of computation concepts and set theory representations*. The object-oriented elements such as classes, fields (attributes) or methods are defined as sets by means of Kleene star operation. Likely, refactoring relations and parameters are represented as Cartesian products.

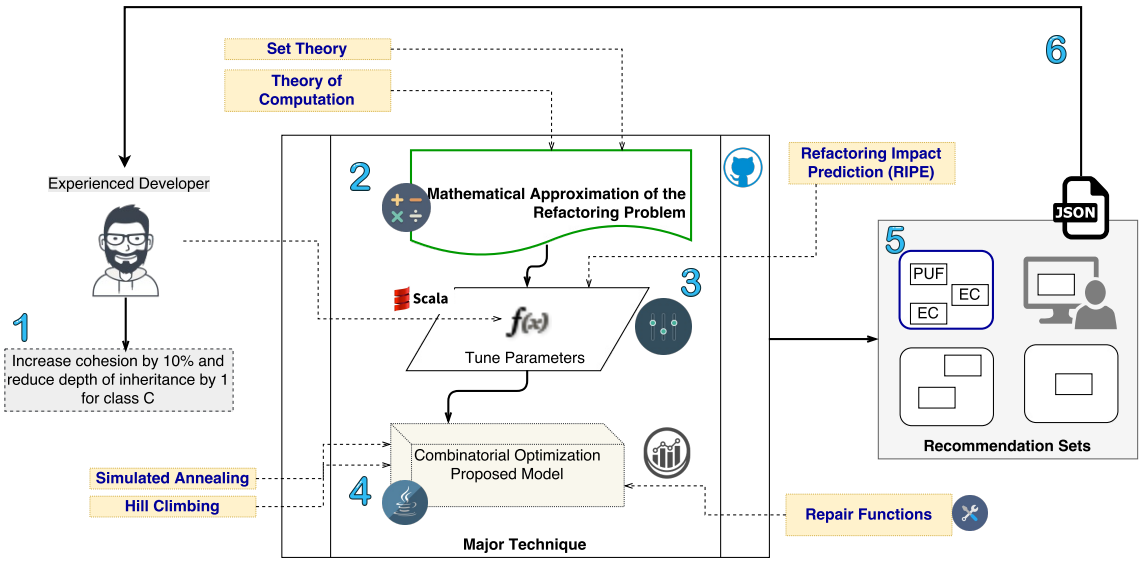


FIGURE 2.1. Artificial Refactoring GENeration (Approach Overview): Model Refactoring Definition

This chapter introduces an **object-oriented programming** and a **software refactoring** formalism to establish the mathematical concepts when dealing with the refactoring construction as a combinatorial problem. Therefore, it is the first part of the *Model Refactoring Definition*.

2.1 Why does one propose a mathematical approximation?

The advantages of using a mathematical approximation are related to the reproducibility analysis in previous chapter. Search-based approaches lack proper representation for combinatorial problems by making their approaches hard to reproduce and understand. Besides, a formalism for refactoring is appropriated whether the research community wants to extend any technique or introduce other formalism (e.g. using a hyper-heuristic or memetic algorithm for the software refactoring problem).

Figure 2.2 depicts a high level explanation of all relevant concepts to take into account when dealing with refactoring recommendations: 1) composition of parameters, which include a combination of a class, field and/or method that are going to be altered, 2) improvement of software design, which is the final goal of the recommendations, 3) representation of refactoring operations, which are the set of refactoring operations that are going to modify the software, 4) impact quality of software metrics, which are quality representation of the source code that are going to be altered due to sequence of refactorings, and 5) reduction of error-proneness, which is a desired behavior that expresses a probability of failure or error bias of the system.

This document links the refactoring formalism with the combinatorial optimization. Classes, methods, fields, metrics, refactoring operations and metric predictions should be properly represented for being included in a combinatorial framework. The idea behind of using a mathematical representation is to include terms like parameters composition, software quality metrics, refactoring sets, error-proneness and design improvement.

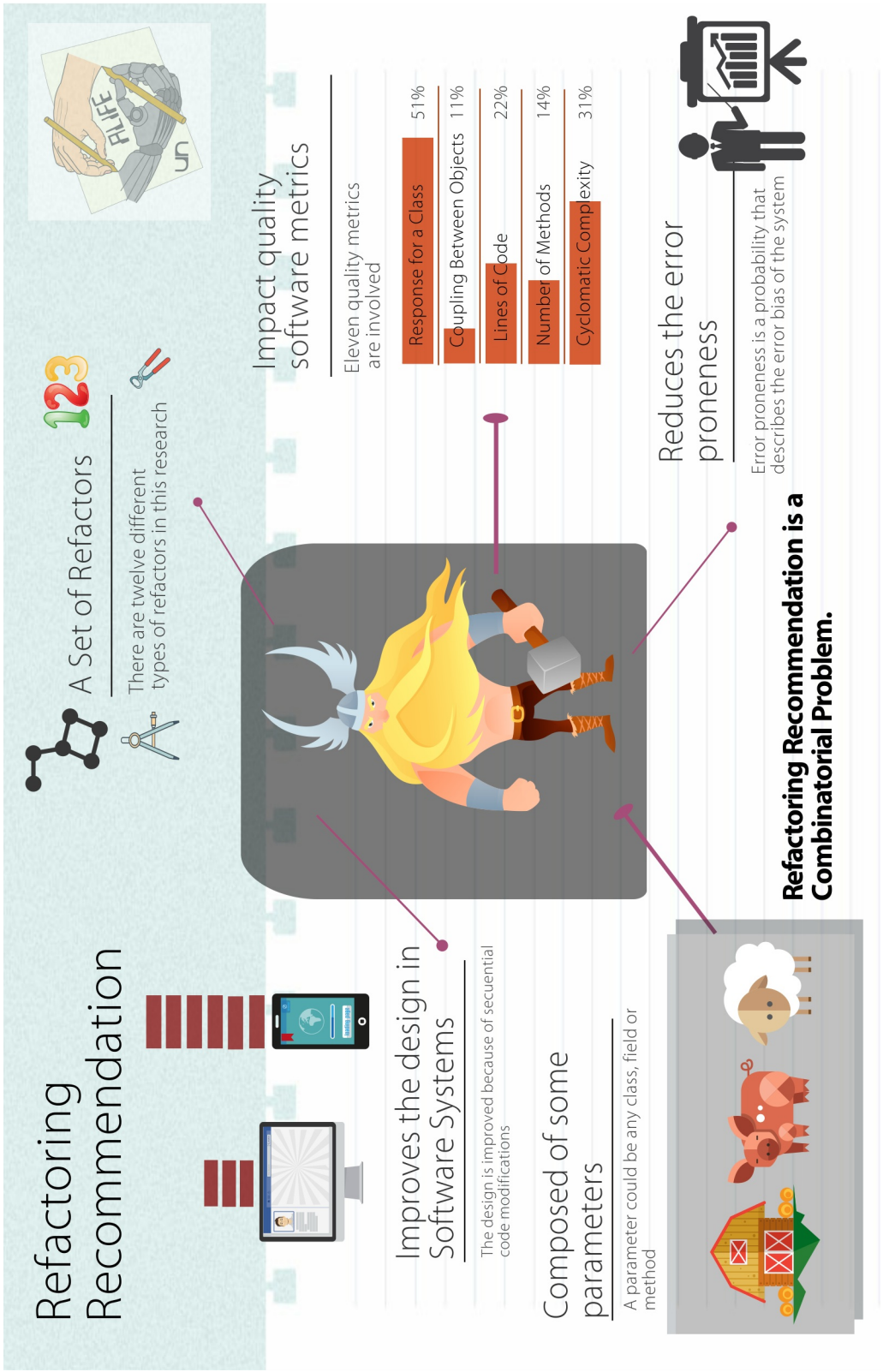


FIGURE 2.2. High Level Combinatorial Problem

2.2 An Object-Oriented Programming Formalism

Object-oriented programming (OOP) is a software development style, organized around objects, which can be used in paradigms, like imperative, functional or logic programming [33]. Purpose of OOP is to reach high modularity through a reduction of complexity in large scale systems. An object is defined as a run-time entity composed of: 1) a state represented by a set of internal objects named attributes and 2) a set of subroutines named methods [44].

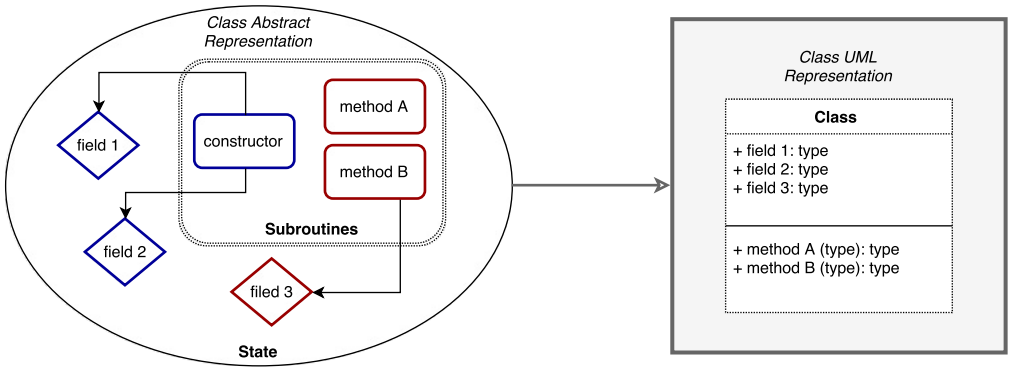


FIGURE 2.3. Class Definition: an abstract and UML representation.

Definition 1. *System Under Analysis (SUA) definition. SUA is an information system or program composed of classes, methods, and attributes, where objects instantiate classes and communicate with each other through messages.*

Definition 2. *Class definition. A class, prototype or template is a set of similar objects including collections of similar attributes and methods Figure 2.3. In fact, classes are the programming-time counterparts of objects [44]. Assume a single class c_α is a Cartesian product represented by:*

$$c_\alpha = identifier \times Attribute(s) \times Method(s) \tag{2.1}$$

$$c_\alpha = str \times str^* \times str^* \tag{2.2}$$

where str^* is a Kleene on str string abstraction or chain. A set of classes is a power set of c_α:

$$C \subseteq \wp(c_\alpha) \tag{2.3}$$

$|C| = k$ where k is the total number of System Under Analysis (SUA) classes and $c = \{c_\alpha \in C | 1 \leqslant \alpha \leqslant k\}$.

Definition 3. *Method Definition. Methods are subroutines which define the set of valid messages the object may accept. Methods are represented by Kleene star. Set:*

$$M = str^* = \bigcup_{n=0} (str)^n \tag{2.4}$$

Method property: $\forall_\alpha$ among $1 \leqslant \alpha \leqslant k$ states $|M| = \beta_\alpha$ is finite. So $M \subseteq \mathbb{N}$ and $M(c_\alpha) = M_\alpha$.

Definition 4. *Attribute Definition. Attributes (or fields) are sets of internal objects that represent a state. Attributes are represented by Kleene star. Set:*

$$A = str^* = \bigcup_{n=0} (str)^n \tag{2.5}$$

Attribute property: $\forall_\alpha$ *among* $1 \leqslant \alpha \leqslant k$ *states* $|A| = \gamma_\alpha$ *is finite. So* $A \subseteq \mathbb{N}$ *and* $A(c_\alpha) = A_\alpha$.

2.3 Software Refactoring Formalism

Refactoring is the process of modifying vulnerable code from an object-oriented software, by improving the system quality without altering its external behavior [8, 37]

Definition 5. *Refactoring Definition. The refactoring process consists in re-constructing the code design of a SUA without affecting the behavior functionality [8].*

Table 2.1. Fowler's Catalog Refactoring Operations Used

Refactoring Operation	**Abbreviation**	**Category**
Extract Method	EM	Composing Method
Inline Method	IM	Composing Method
Replace Method w. Method Obj	RMMO	Composing Method
Pull Up Field	PUF	Dealing w. Generalization
Pull Up Method	PUM	Dealing w. Generalization
Push Down Field	PDF	Dealing w. Generalization
Push Down Method	PDM	Dealing w. Generalization
Replace Delegation w. Inheritance	RDI	Dealing w. Generalization
Replace Inheritance w. Delegation	RID	Dealing w. Generalization
Extract Class	EC	Moving Features
Move Field	MF	Moving Features
Move Method	MM	Moving Features

A refactoring is a function $R_\delta : \Omega \longrightarrow$ *(Code Modification) where* δ *represents a specific refactoring operation (see used operations in Table 2.1) and* $\Omega = c_s \times A_s \times M_s \times c_t$ *is a Cartesian product parameter, where* c_s *is the source class,* c_t *is the target class,* A_s *are the attributes of the source class and,* M_s *the methods of the source class.*

Refactoring properties are: 1) a solution set of all the possibilities of refactorings functions with refactoring parameters, coined as $R(\Omega)$, so $RI \in R(\Omega)$, is a specific solution or Refactoring Instance, 2) a refactoring recommendation S_i is a set of Refactoring Instances RI and a subset of $R(\Omega)$.

Software metrics measure the degree of possession of specific properties in the systems [31]. Eleven quality metrics (see Table 2.2) were considered for our model, based on previous reports [5].

Definition 6. *Code Quality Metric Definition. A quality metric is a function* $\eta_j : c_\alpha \longrightarrow \mathbb{R}$. *Each class* c_α *of the SUA has a set of metric values:*

TABLE 2.2. Quality Metrics Used

Metric	Name	Coupling
RFC	Response for a Class	Coupling
CBO	Coupling Between Objects	Coupling
DAC	Data Abstraction Coupling	Coupling
MPC	Message Passing Coupling	Coupling
LOC	Lines of Code	Size
NOM	Number of Methods	Size
CYCLO	Cyclomatic Complexity	Complexity
LCOM2	Lack of Cohesion of Methods2	Cohesion
LCOM5	Lack of Cohesion of Methods5	Cohesion
NOC	Number of Children	Inheritance
DIT	Depth of Inheritance	Inheritance

$$H_\alpha = \{\eta_1(c_\alpha), \eta_2(c_\alpha), \ldots, \eta_J(c_\alpha)\} \tag{2.6}$$

where J is the total number of metrics and $H_\alpha \in \mathbb{R}^J$.

Refactoring Impact PrEdiction (RIPE) is an approach that predicts the impact of 12 refactoring operations on 11 code metrics [5]. RIPE implements 89 impact prediction functions that show developers the variation of code metrics before the application of a refactoring operation. We utilized *prediction function* because a solution set of refactoring recommendation is too expensive to perform in a SUA (it may require several resources that could exceed the budget of a project), yet an *estimation* of the refactoring on the metrics could be calculated. For instance:

$$LOC_p(c_s) = LOC_b(c_s) - LOC(m_k) \tag{2.7}$$

$$LOC_p(c_t) = LOC_b(c_t) + LOC(m_k) \tag{2.8}$$

Where LOC_p is the predicted lines of codes, LOC_b is the lines of codes before the prediction, c_s is a source class, c_t is a target class, and m_k is an specific method k from the source class. For each Refactoring Operation r_δ and quality metric η_j there is a prediction function $Prediction_{\delta,j}(c_\alpha)$, which *estimates* the impact of the refactoring on the metric of a class (iff the refactoring affects the metric):

$$Prediction_{\delta,j}(c_\alpha) = \widetilde{\eta}_{\delta,j}(c_\alpha) \tag{2.9}$$

The previous equation is a formalization we proposed to link RIPE with our definitions.

Definition 7. *Impacted Code Quality Metric Definition. An impacted quality metric is a function $\widetilde{\eta_{\delta,j}} : c_\alpha \longrightarrow \mathbb{R}$. Each class $\widetilde{c_\alpha}$, impacted by a refactoring operation r_δ of the SUA, has a set of metric values:*

$$\widetilde{H_\alpha} = \{\widetilde{\eta}_{(\delta,1)}(c_\alpha), \widetilde{\eta}_{(\delta,2)}(c_\alpha), \ldots, \widetilde{\eta}_{(\delta,J)}(c_\alpha)\} \tag{2.10}$$

where J is the total number of metrics and $\widetilde{H_\alpha} \in \mathbb{R}^J$.

CHAPTER 3

The Artificial Refactoring Generation as a Combinatorial Estimation Problem

Figure 3.1 represents, in steps three and four, a modelling to the software refactoring problem as a combinatorial estimation model. ARGen recommends how to reconstruct the code based on eleven different quality metrics (see Table 2.2) that can be adjusted according to the experience of the developers. Therefore, it is the second part of the *Model Refactoring Definition*.

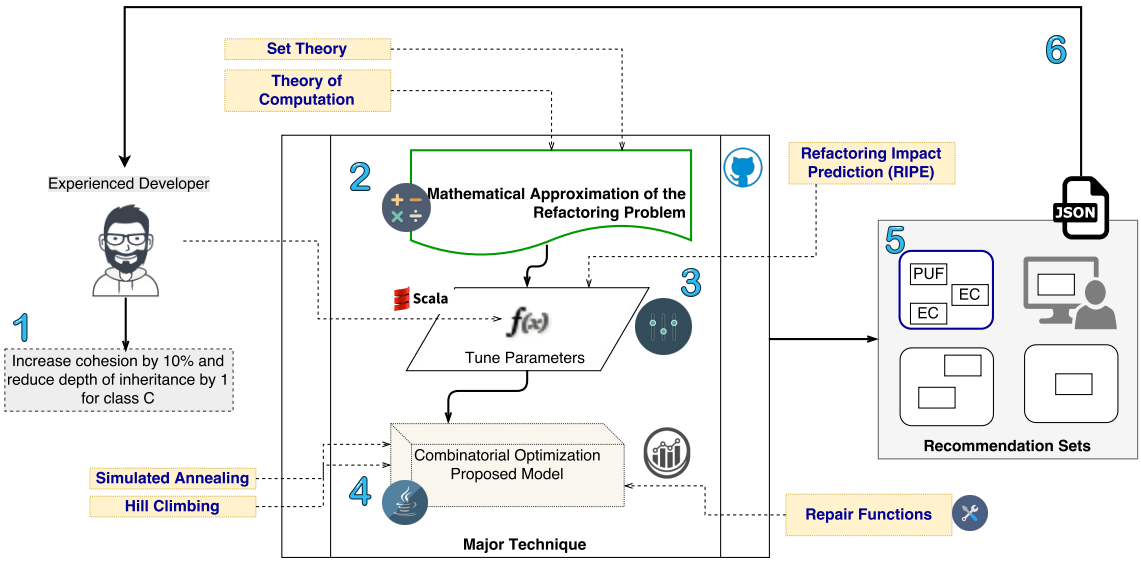


FIGURE 3.1. Artificial Refactoring GENeration (Approach Overview): Model Refactoring Definition

The purpose of the technique is to estimate several recommended refactoring operations that fulfill the proposed parametric objective function. The recommendations are composed of twelve different refactoring operations (see Table 2.1). ARGen considers all possible combinations of refactoring operations for a given object-oriented system, which implies operating in a large solution space. Therefore, whichever algorithm used would require high computation resources.

The refactoring generation is a *"combinatorial problem"*. The solution represents a set of refactoring operations, instead of a sequence of refactoring operations. Each refactoring operation is independent from the others in the same set. Therefore, one refactoring operation does not affect another refactoring operation.

The ARGen does not guarantee that obtained refactoring solutions might reduce the error-proneness of an inspected piece of code. Although, ARGen is able to explore the search space in actionable areas, if and only if, the developers/researchers tune the objective function accordingly. The software developers can adjust the fitness parameters according to their experience with the source code analyzed.

3.1 Characterization of the Search Space

Figure 3.2 shows the search space of the Artificial Refactoring Generation composed of four parts.

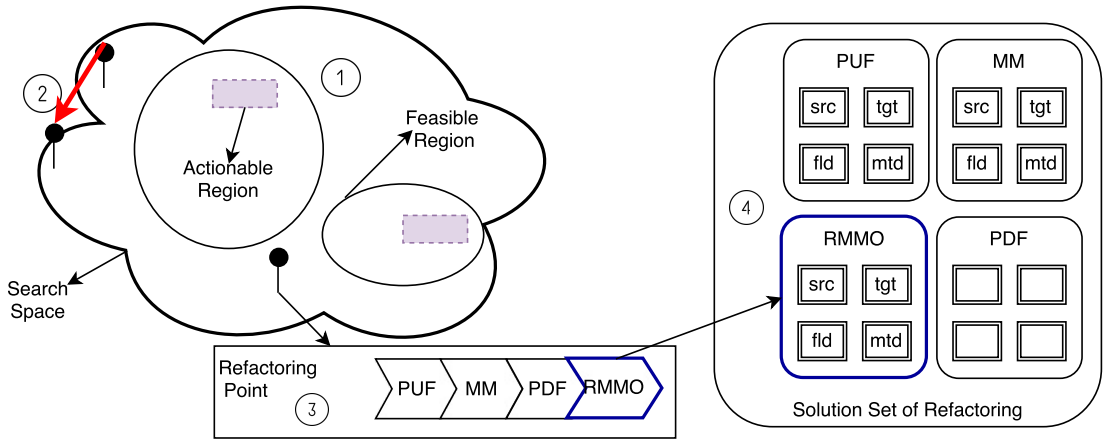


FIGURE 3.2. Refactoring Problem Search Space. The abbreviations of refactorings can be consulted in Table 2.1

The first part of the graphic is an overview of the search space. It includes two relevant regions: feasible and actionable. A **feasible region** is all the refactoring solutions that fulfill the defined constraints of the proposed objective function. The **actionable region** is where solutions reduces the error-proneness.

The size of the search space depends upon the number of classes, attributes and methods of the inspected system: increased by the expression $(C^2 * A * M)^r$ where C is the number of classes[1], A the number of attributes in the selected source class, M the number of methods in the selected source class and r the number of refactoring operations in a sequence (e.g. a system with 10 classes, 10 attributes and 10 methods would have a size of 10,000 whether the sequence is composed of one refactoring). A huge space is analyzed; however, the idea is to inspect a specific piece of code. The document demonstrates that the classes involved in the optimization affect the time complexity.

[1] C^2 represents that each class serves as source and target in whichever suggested refactoring.

It is assumed that the search space is not convex (see Figure 3.2) because such space is discrete; nobody can guarantee, only with the provided refactoring discrete points, that whichever two point in the search space are connected through a line that is contained inside the same space. The third part presents a refactoring solution point that would be a sequence of refactoring operations from the Fowler's Catalog. This specific solution point is amplified in the fourth part where a source class (src), a target class (tgt), a field from the source class (fld) and a method from the source class (mtd) are included within each refactoring operation.

3.2 Objective Function

Figure 3.3 explains the proposed objective function for ARGen at a high level. To understand the function, the explanation is divided into six topics: candidate solution set, objective function representation, bias, code quality additive impacted metric, code quality additive metric and bias quality system ratio.

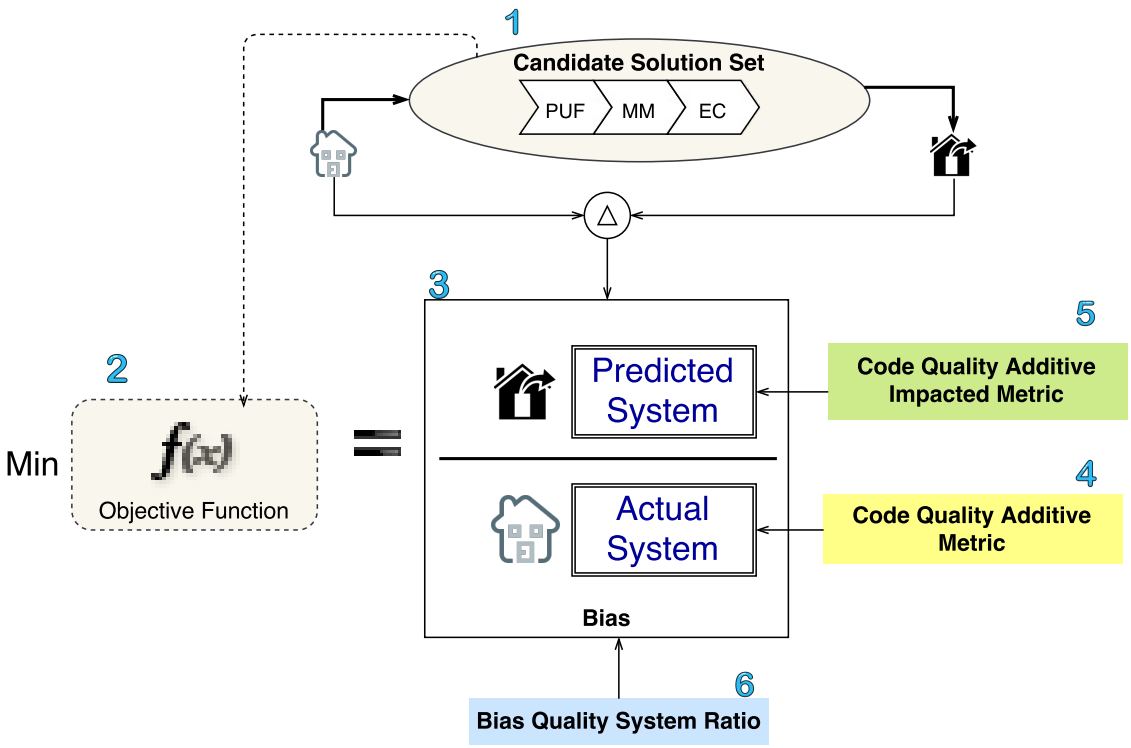


FIGURE 3.3. High Level Objective Function

The first topic is the "candidate solution set". The candidate solution set is any point of the search space and the input of the second topic (objective function); it contains the suggested refactoring operations to be assessed. The proposed model sets the objective function for minimizing.

The third topic is the "bias" which represents a simple ratio. The idea of the ratio is to compare the predicted system to the actual system (the fourth and fifth topics respectively). The predicted system is the computation of the "code quality additive impacted metric", which is a normalized prediction of the system after applying suggested

refactorings for a specific metric. The actual system is the computation of the "code quality additive metric", which is a regular normalized measurement for a specific metric.

Lastly, the sixth topic is the "bias quality system ratio", which is the formal representation of the objective function. Whether the "bias" includes more than one metric, then the ratio is a general quality representation of the system.

3.2.1 Code Quality Metrics Simplification

The use of quality metrics is not only oriented to identify refactoring opportunities, but to support the configuration of refactoring solutions that fulfill the initial setting. Then according to desire software properties, the technique computes the adequate refactorings.

Definition 8. *Code Quality Additive Metric. The elements of the refactoring solution set are independent among them. That indicates, one refactoring is taken isolated from others in the same solution set.*

$$SUM_H(j \in J) = \sum_{\alpha=1}^{k} (\eta_j(c_\alpha)) \tag{3.1}$$

Then, this formula can be calculated for describing a general value on one specific metric for the SUA processed.

The parameter Φ_j of the impacted sum function $SUM_{\widetilde{H}}(\Phi_j)$ is composed of a metric $j \in J$ and the solution set S_i.

$$\Phi_j = (j \in J, S_i) = (j, \{RI_1, RI_2, ..., RI_i, ..., RI_T\})$$

where T is the total number of Refactoring Instances in S_i.

Definition 9. *Code Quality Additive Impacted Metric.*

$$SUM_{\widetilde{H}}(\Phi_j) = \sum_{i=1}^{|S_i|} \sum_{\alpha=1}^{|c \in \Omega_i|} max_{1 \leq j \leq \widetilde{H_\alpha}} ((\widetilde{\eta_{\delta,j}}(c_\alpha))) \tag{3.2}$$

This formula can be calculated for describing a general estimated value on one specific metric for the SUA processed according to a solution set.

3.2.2 The Bias Quality System Ratio

The Bias Quality System Ratio BQR is introduced here as a general quality metric that measures the bias ratio between the quality of the SUA and the predicted quality of the impacted classes according to a refactoring recommendation S_i.

$$BQR(\Phi_j) = \frac{SUM_{\widetilde{H}}(\Phi_j)}{SUM_H(j)} \tag{3.3}$$

The objective function is an adaptable optimization function which can be changed via user feedback or include user-defined goals. The metric measures the difference between predicted quality metrics and real quality metrics. Minimizing the metric indicates that

whether the predicted quality metric is less than the real metric, then the quality of the specific code ameliorates.

Definition 10. *The objective function is a parametric optimization function that includes developer-defined goals represented in a normalized version of BQR. We use min-max normalization to put the metrics in a positive scale:*

$$ObjectiveFunction(\Phi) = \frac{\sum_{j=1}^{J}\left(w_j \frac{SUM_{\widetilde{H}}(\Phi_j) - min(SUM_{\widetilde{H}}(\Phi_j))}{max(SUM_{\widetilde{H}}(\Phi_j)) - min(SUM_{\widetilde{H}}(\Phi_j))}\right)}{\sum_{j=1}^{J}\left(w_j \frac{SUM_H(\eta_j) - min(SUM_H(\eta_j))}{max(SUM_H(\eta_j)) - min(SUM_H(\eta_j))}\right)} \tag{3.4}$$

Where the weights are a number between $w \in \{-1, 1\}$.

When the numerator is less than the denominator (proper function), the predicted system improves their quality metrics. Whether the objective function is an improper fraction, the predicted system is worse than the actual.

The weights (and their sign) are the channel where developers can convey their point-of-view for obtaining meaningful suggestions of classes to be refactored [1].

The input of the objective function is a set Φ of refactoring recommendations. The output is a value that represents the estimation of the metrics after applying all the refactorings operations in Φ. The prediction formulas of quality metrics are helpful in bringing forward objective function(s). Besides, the objective function does not ensure refactoring opportunities in the code system [1], but an estimation of how the refactoring is impacted by the change of a quality metric [5].

3.2.3 The Overlap Problem

This subsection explains a common problem when performing bias computations. The overlap problem occurs when two different solution points impacted the same class. Which metrics should the object function select whether two solution points alter the same class?

Consider the following example, two different refactoring operations impact the LOC (lines of code) of the same class. One possible scenario is that both first and second refactorings reduce the LOC, second scenario is that both first and second refactorings augment the LOC, third scenario is when the first refactoring reduces and the second augments. The last scenario is when the first augments and second reduces. Because of the refactoring operation independence, the quality metrics among classes are not related. Trying to approximate with a mean value would be wrong because anybody can ensure how the final metrics are represented, the mean is a highly sensible statistic. However, the maximum value of a quality metric is a much more stable outcome.

Figure 3.4 represents the mechanism for selecting the metrics. The best representation of a system after refactoring is with the maximum value of an impacted metric due to the refactoring independence nature.

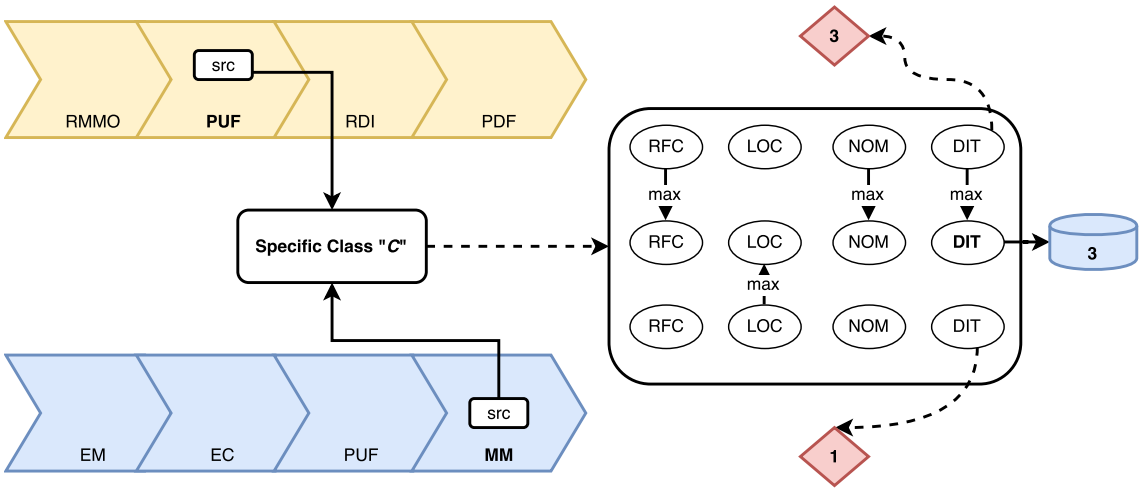


FIGURE 3.4. Refactoring Overlap Predicted Values

3.3 Computational Complexity Classification

The set $SubsetSum$ is a NP-complete problem that consists in a set of pairs: a finite sequence $\vec{B} = \{B_1, B_2, ..., B_n\} \in N^n$, and a target number $v \in N$ such that $(\vec{B}, v) \in SubsetSum$ iff $\exists c = \{c_1, c_2, \ldots, c_n\} \in 0, 1^n$ such that $\sum_{i=1}^{n} c_i B_i = v$. In other words, the sum of some sub set of numbers of $\vec{B}$ is equals to v.

For example, given a finite set of numbers $B = \{113, 45, 87, 32, 67, 12, 56\}$ and $v = 156$, one should find a sub set of $B|\langle B, v\rangle \in SubsetSum$. Each number B_n represents a refactoring RI_n and the instance $c = \{0, 1, 0, 1, 1, 1, 0\}$ represents the set C that activate or deactivate a RI_n refactoring.

$$S_i = RI_2(c_2) + RI_4(c_4) + RI_5(c_5) + RI_6(c_6) \text{ iff C is a solution of SubsetSum.}$$

Estimating a refactoring sequence with ARGen is a combinatorial problem (NP-Complete) since the search space is discrete and the subset sum problem can be reduced to the refactoring estimation problem. Φ is the refactoring set applied to a SUA, each element into Φ is an impacted metric value [5]. Therefore, Φ is the set of values and one wants to search a subset of Φ that fulfills the summation in the objective function.

3.4 Refactoring Constraints

This section introduces a catalog of constraints that depends of the refactoring operation structure and the general object-oriented guidelines. Table 3.1 shows all the constraints and the refactorings that apply the constraint. Nine constraints were identified for the behavior preservation:

Empty Methods. This constraint occurs when assessing a refactoring on a solution that does not define an specific method. "Empty Methods" applies for all refactoring that involves method movement or modification.

MM Constructor. This constraint occurs when a "Move Method" refactoring tries to modify a constructor in a class. The approach should not alter any constructor because of behavior preservation.

Empty Fields. This constraint occurs when assessing a refactoring on a solution that does not define an specific field. "Empty Fields" applies for all refactorings that involves field movement or modification.

SRCSubClassTGT. This constraint verifies that a source class is a subclass of a target class. The violation of this rule matters to refactorings that involve movement of methods among hierarchy related classes.

SRCSupClassTGT. This constraint verifies that a source class is a superclass of a target class. The violation of this rule matters to refactorings that involve movement of methods among hierarchy related classes.

Confirm Delegation. This constraint verifies that a delegation exists between source and target class.

Override Method. This constraint occurs when assessing a refactoring on a solution where at least one of its methods is overridden.

Hierarchy Verification. This constraint searches for any hierarchy relationship between the source and target classes. For instance, when parents and children are the same selected class as target and source.

Methods of SRC. This constraint verifies that the method belongs to the source class.

TABLE 3.1. Catalog of Refactoring Constraints For Behavior Preservation

Constraint/Refactoring	MF	PUF	PDF	EM	IM	MM	PDM	PUM	RMMO	RDI	RID	EC
Empty Methods				x	x	x	x	x	x			x
MM Constructor				x	x	x	x	x	x			x
Empty Fields	x	x	x									x
SRCSubClassTGT		x						x			x	
SRCSupClassTGT			x				x					
Confirm Delegation										x		
Override Method		x		x	x	x	x	x	x			x
Hierarchy Verification					x	x				x	x	
Methods of SRC Class				x								

3.4.1 Refactoring Constraint Examples

Figure 3.5 shows a class abstract representation without methods, except for the constructor. Because the constructor should often be stable and nothing should modify it, the class does not have methods for performing refactorings that involve methods as parameters. However, this class owns two fields, then one can apply any field refactoring operation(PUF, PDF or MF).

Figure 3.6 shows a typical configuration of a solution after applying any mutation, genetic operator, or new generation. Consider a class C1 a subclass of C2. Whether one tries to apply a "Pull Up Field" on a solution where C1 is the source and C2 the target, then the operation violates the constraint **SRCSubClassTGT**. In "Pull Up Field", the source class must be a subclass of the target.

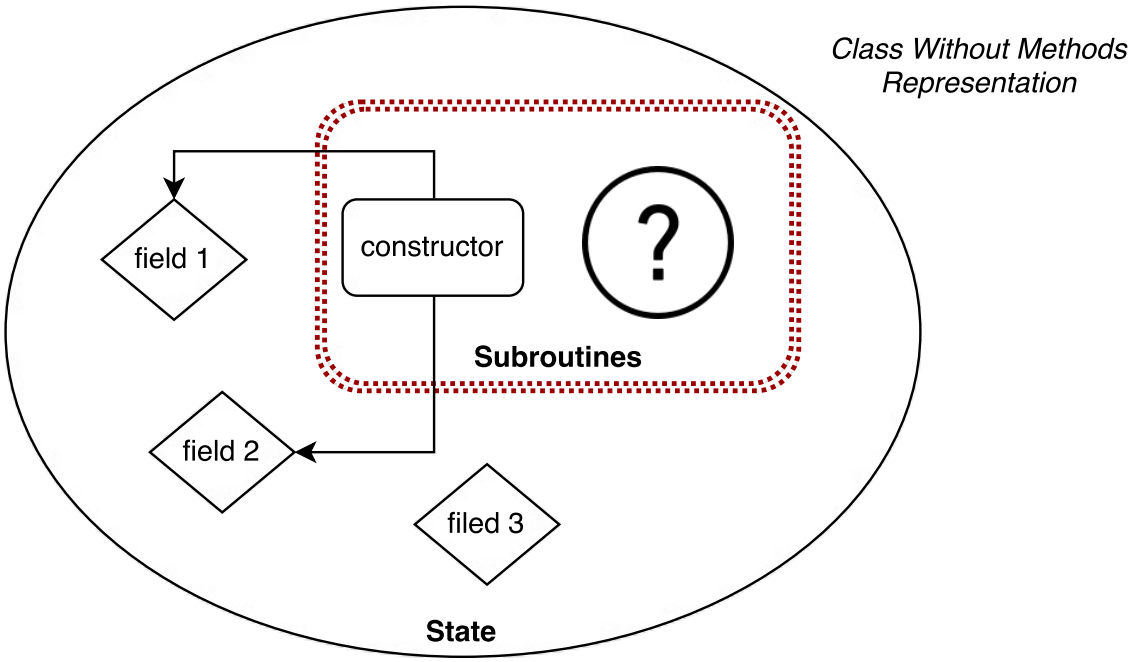


FIGURE 3.5. Refactoring Constraint Class Without Methods

3.5 Refactoring Generating & Repair Functions

The generating and repair functions are based on the previous catalog of constraints and grouped by the refactoring operation. Figure 3.7, for instance, represents a violation of a PUF refactoring with its possible repair. The repair consisted of a succession of steps systematically performed: first, looking for a different target (another class from the system); second, looking for a different source whether the previous step does not work and; third, generating a new refactoring from scratch (calling a generating function) whether the first and second steps do not work. In Figure 3.7, the repair function applies the first step and replaces C2 with C3 which is a superclass of C2. Now, the solution is feasible and the PUF could be applied successfully.

3.5.1 Generating Functions

The refactoring generating functions are the mechanism for assembling a refactoring set solution. It contains the steps for configuring each refactoring operation from scratch and adapts independently to a specific search technique. The following description embodies all the systematic steps for generating a refactoring.

PUM (1) Includes breakpoint[2] equal to the size of the system. (2) Search for tgt randomly. (3) Get the child classes of tgt. A child class is a src. (4) Pick up one child class randomly. (5) Extracting methods from the child class. (6) Pick up a method randomly. (7) Verification that the method is not a constructor. (8) Verification that the method is going to override a parent. (9) Verification that the method is going to be overridden by children. (10) Count an iteration. The program continues

[2]The breakpoint in the whole section refers to a limit of iterations when searching for parameters (classes, methods or fields).

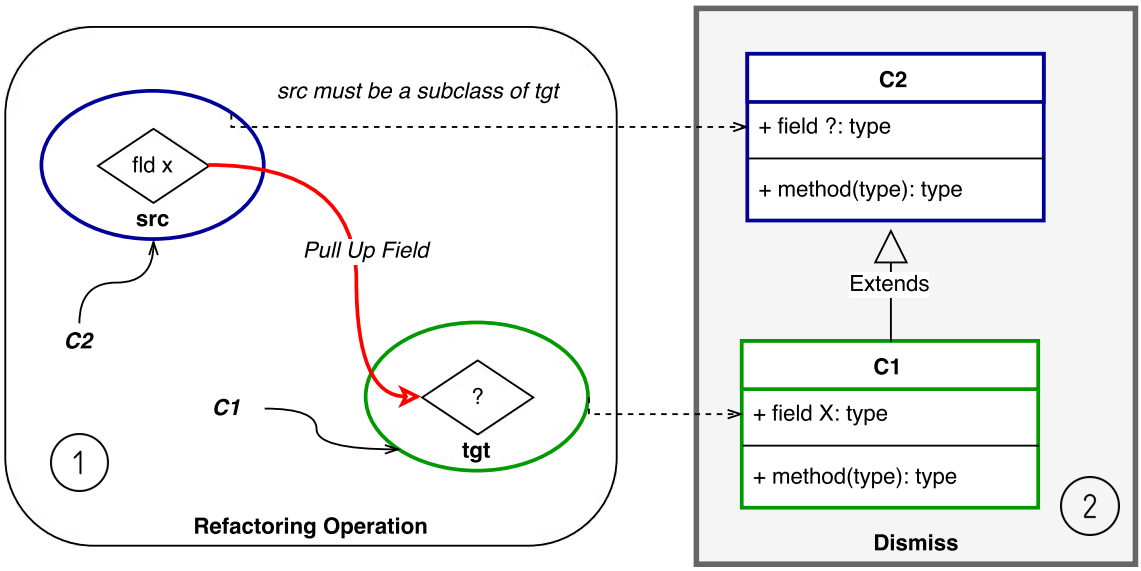


FIGURE 3.6. Refactoring Constraint Pull Up Field

whether the number of the count is less than the breakpoint. Otherwise, the loop is broken. (11) Feasible verification. Whether the created ref is not feasible, then the program starts in the step number 1. Otherwise it continues returning a refactoring object.

PUF (1) Includes breakpoint equal to the size of the system. (2) Search for tgt randomly. (3) Whether the tgt has no children, then the refactor is not feasible and restart from step 2 . (4) Pick up a random src from the children classes detected. (5) Whether the src class has no fields, then the refactor is not feasible and restart from step 2. (6) Pick up a random fld from the selected src class. (7) Identifying other src classes that contain the same fld.

PDF (1) Includes breakpoint equal to the size of the system. (2) A src class is picked up randomly. (3) From the src of step 2 a fld is extracted randomly. Whether fld does not exist, then another src is selected. Whether the breakpoint is overpassed, then the refactoring is not feasible. (4) According to the src, all childs are identified. At least one of the children are added to tgt. Whether no children are identified, then the refactoring is not feasible. (5) A tgt is selected according to the src. Whether the tgt does not exist, the refactoring is not feasible and restart in step 2.

PDM (1) Includes breakpoint equal to the size of the system. Generates a random src. (2) Pick Up a method from src. (3) Verify that the mtd do not override any parent or children. Otherwise, the refactoring would be not feasible and starts again from step 2. (4) Verify that the method is not a constructor. Otherwise, the refactoring would be not feasible and starts again from step 2. (5) Whether the counter is greater than the breakpoint value inside the src do-while, then the refactoring is not feasible due to sufficient times. (6) Identify all children classes from src and select a tgt.

EM (1) Only Feasible Refactorings (2) Select one src randomly. (3) Select one mtd from the src. Whether the src has no methods, then the process starts from step 2. (4) Verify that mtd is not constructor or overrides any parents or children. Otherwise, the process start from step 2.

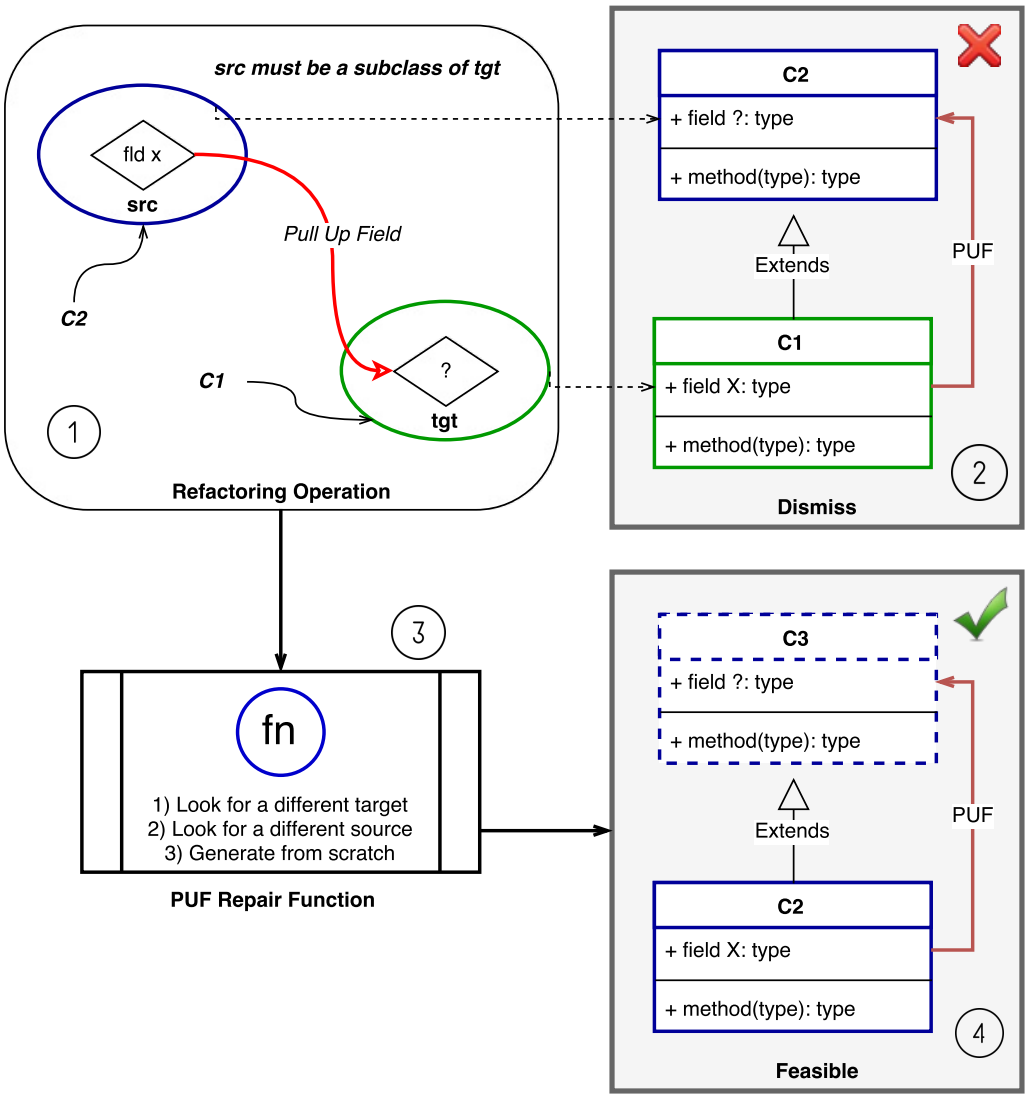


FIGURE 3.7. Refactoring Repair Pull Up Field

IM (1) Only Feasible Refactorings. (2) Select one src randomly. (3) Select one mtd from the src. Whether the src has no methods, then the process starts again (step 2). (4) Verify that the selected method is not a constructor or overrides parent or children.

MF (1) Only feasible refactorings. (2) Select one src randomly. (3) Select one fld from the src. Whether the src has no fields, the the process starts again (step 2). (4) Select one target class randomly.

MM (1) Only feasible refactorings. (2) Select one src randomly. (3) Select one method from the selected source class. Whether the src has no methods, then the process starts again from step 2. (4) Verify that the selected method is not a constructor. If so, the process starts again from step 2. (5) Select a target class randomly. (6) Verify that the method does not override any parent or children and verify that there is no hierarchy conflicts.

RDI (1) Includes breakpoint equal to the size of the system. (2) Select a source and a target class. (3) Verify that the source and target are not the same class. Otherwise

the process restart in step 2. (4) Verify that the source and the target are not hierarchy related. Otherwise the process restart in step 2. (5) The refactoring is not feasible whether the counter exceeds the breakpoint.

RID (1) Includes breakpoint equal to the size of the system. (2) Select a target class. (3) Verify that the source and the target are hierarchy related. Otherwise the process restart in step 2. (4) The refactoring is not feasible whether the counter exceeds the breakpoint.

RMMO (1) Only feasible refactorings. (2) Select one source class randomly. (3) Pick up a method from source class. If there are no methods, then the refactoring is not feasible and restarts in step 2. (4) Verify that the method is not a constructor or overrides any parent or children. Otherwise, the refactoring would not be feasible and process restarts in step 2. (5) Generate the new target class.

EX (1) Only feasible refactorings. (2) Generate a source class randomly. (3) Pick up a method from the source class. If there are no methods or the selected method is a constructors or overrides any parent or child, then the process restarts in step 2. (4) Pick up a field from the source class. If there are no fields, then the process restarts in step 2. (5) Create a new target class.

3.5.2 Repair Functions

The repair functions are the other mechanism for verifying the consistency of a generated or modified (e.g. by a genetic operator) solution set and for correcting those solutions that violates any restriction. The repair functions should be less complex than a generation function of refactorings. The following description embodies all the systematic steps for repairing a refactoring.

PUM (1) Includes a local breakpoint equal to children of the tgt. (2) A tgt is extracted from the refactoring to be repaired. If the tgt is null, empty or a new class, then it is replaced with a class from the metaphor. (3) Get the children from tgt. If getting the children is null or empty, then the refactoring is not feasible. (4) Pick up one child (src). (5) Get a method (mtd) from the source class (src). (6) Verify that the method is not a constructor. Otherwise, the feasibility is false and start from step 4. (7) Identify other src classes that contain the method from step 4. (8) Whether the counter is greater than the breakpoint value, then the loop is broken and the repair is left to a complete new generation from scratch. Otherwise, the repair is completed.

PUF (1) Includes a local breakpoint equal to the children of the tgt. (2) A tgt is extracted from the refactoring to be repaired. Whether the tgt is null, empty or a new class, then it is replaced with a class from the metaphor. (3) Get the children from tgt. If getting the children is null or empty, then the refactoring is not feasible. (4) Pick up one child (src) randomly. (5) Whether the src has no fields, then the refactoring is not feasible and restart in the step 3. (6) Pick up one field (fld) from the source class from step 4. (7) Identify other source classes with the same field. (8) If the counter is greater than the breakpoint value, then the loop is broken and the repair is left to complete a new generation from scratch. Otherwise, the repair is completed.

PDF (1) Extract src from parameters. Whether the src is null, then the system picks up one randomly. (2) Select a fld from the src class. If there is no fld, the refactoring is

not feasible and the repair is left to the generation. (3) Identifying all the children of the src class. If there are no children, then the refactoring is not feasible. (4) Pick up a tgt from the children list.

PDM (1) Extract src class from the given refactoring. Whether the src does not exist, then the repair function selects a src randomly. (2) Pick up a method from the src class. If there are no methods, then the refactoring is not feasible and require a new generation. (3) Verify whether the method overrides parents or children. If so, then the refactoring is not feasible. (4) Verify whether the method is not a constructor. If so, then the refactoring is not feasible. (5) Retrieve children classes from src.

EM (1) Extract src from the given refactoring. Whether the src does not exist, the process selects one randomly. (2) Get the methods from the src class. If there are no methods, then the refactoring is not feasible and generate from scratch. (3) Verify that the selected method is not a constructor or overrides any parent or children.

IM (1) Extract the source class from the given refactoring. Whether no src is provided, then the process selects one class randomly. (2) Get the methods from the source class. If there are no methods, then the refactoring is not feasible and requires to be generated from scratch. (3) Verify that the selected mtd is not a constructor or overrides any parent or children.

MF (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Select one field from the src. If there are no fields, then the refactoring is not feasible and requires to be generated from scratch. (3) Extract the target class from the refactoring. If there is no tgt, then the process selects one randomly.

MM (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Select one method from the source class. If there are no methods, then the refactoring is not feasible and required to be generated from scratch. Verify that the method is not a constructor. Otherwise, the refactoring is not feasible. (3) Extract the target class from the refactoring. If there is no target class, the process selects one randomly. (4) Verify that there is no hierarchy conflicts or that the method does not overrides any parent or child.

RDI (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Extract the target class from the refactoring. If there is no target class, the process selects one randomly. (3) Verify that the source class and the target class are not the same. Otherwise the refactoring is not feasible and requires to be generated from scratch. (4) Verify that the source and the target are not hierarchy related. Otherwise the refactoring is not feasible and requires to be generated from scratch.

RID (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Extract the target class from the refactoring. If there is no target class, the process selects one randomly. (3) Verify that the source class and the target class are not the same. Otherwise the refactoring is not feasible and requires to be generated from scratch. (4) Verify that the source and the target are hierarchy related. Otherwise the refactoring is not feasible and requires to be generated from scratch.

RMMO (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Pick up a method from the source class. If there

are no methods, then the refactoring is not feasible and requires to be generated from scratch. (3) Verify that the method is not a constructor or overrides the parents or children. (4) Create a new target class.

EX (1) Extract the source class from the refactoring. If there is no source class, the process selects one randomly. (2) Pick up a method from the source class. If there are no methods or the selected method is a constructor or overrides any parent or child, then the refactoring is not feasible and requires to be generated from scratch. (3) Pick up a field from the source class. If there are no fields, then the refactoring is not feasible and requires to be generated from scratch. (4) Create a new target class.

CHAPTER 4

Approach Implementation

This chapter introduces the design of computational techniques to solve the software refactoring problem.

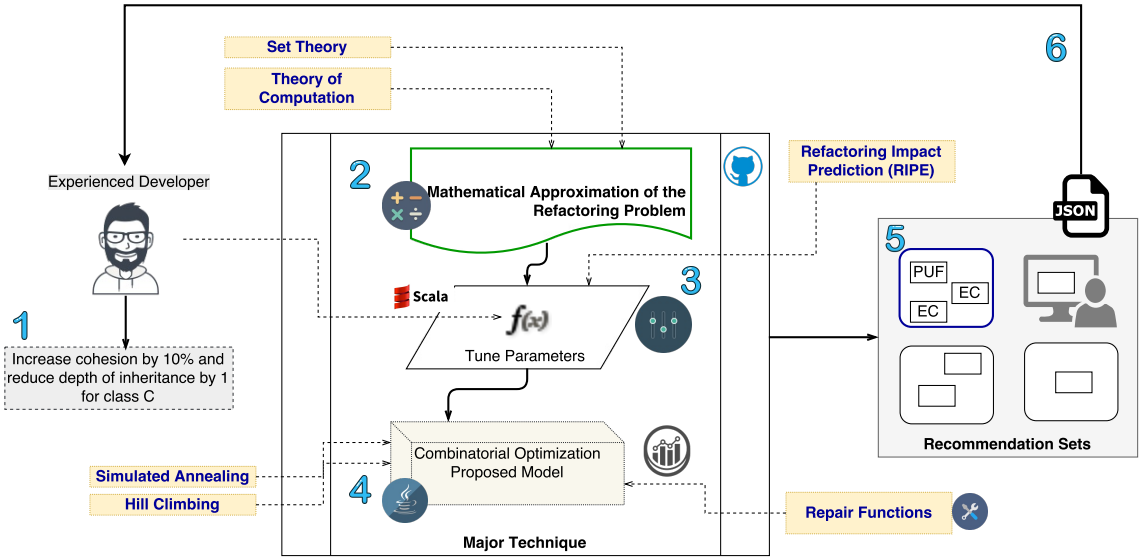


FIGURE 4.1. Artificial Refactoring GENeration (Approach Overview): Model Implementation

Figure 4.1 depicts, in step four, the techniques to obtain artificial refactorings. In this research, ARGen is the baseline for proposing any construction of refactorings focused on combinatorial optimization techniques. In this case, the search techniques are Hill Climbing (HB) and Simulated Annealing (SA). Furthermore, the approach requires the definition of specific refactoring data structures (lists, sets, maps and metaphors) and a clear definition of the objective function based on the application deployment.

4.1 Refactoring Data Structures

Figure A.2 describes the refactoring data structures used in this approach. This work proposes special data structures due to the complexity of the manipulation of refactoring parameters. The description is divided into three parts.

Refactoring Data Structure Outside Fitness (objective function). Each refactoring operation is the individual or solution representation of the approach. An object represents the refactoring operation outside the fitness function. "RefactoringOperation" is the class for a candidate solution. The class contains the parameters as HashMap structures, the refactoring types as special object "RefactoringType", some sub-refactoring (in case of composed refactorings) as a List of "RefactoringOperations", the feasibility of the operation as a Boolean and the penalty value as a double vector whether a penalty function is defined. Some getter and setter functions are available for consulting the fields. This object was developed in Java and the migration process transforms the object into a Scala immutable structure.

Refactoring Data Structure Inside Fitness (objective function). Three data structures are used for keeping the information related to the objective function. RefMetric, ClassMap and Metric are all maps structures that represents the estimated refactoring recommendations. The individual representation is a Java Object with name "RefactoringOperation" The data structures for controlling the computations are in Scala environment.

Source Code Metaphor. The source code metaphor is one of the most important structures throughout the entire program. It keeps the information of the analyzed systems. The metaphor is composed of five important objects: a "TypeDeclaration", a "HierarchyBuilder", the system path, the system name and a refactoring cache.

On the one hand, the "TypeDeclaration" object is the representation of all classes from a specific analyzed system. This object contains five lists: all system classes, only classes with inheritance, only classes with fields, only classes without inheritance and only classes with methods. On the other hand, "HierarchyBuilder" is a general representation of the system after being processed by RIPE [5] with an XML parser. The most important methods are depicted in Figure A.2. Both structures support the generating and repair functions in their computations.

The refactoring cache is a cache from Google Guava library. The fitness functions use the cache when retrieving information about metrics that have already been predicted.

4.2 Refactoring Fitness (objective function)

Figure 4.2 depicts the six steps when executing the objective function. Because of the use of Scala language, the program used Future objects for concurrency and high performance, and the flat function for easily converting data structures.

Step 1: Penalty Computation. If the approach considers a penalty, the fitness computes an accumulated penalty for each refactoring operation in the solution. Penalties are vectors and the final penalty is a mean value of those vectors.

Step 2: Metric Prediction. The first step consists in mapping composed refactorings into primitive forms (Extract Method = Move Method + Move Field) when predicting

metrics. The second step is when the technique chooses two possibilities to obtain prediction values: from RIPE [5] ("memorization") or Cache ("recalling"). Whether the memorization is active, then RIPE predicts the values and the results are migrated to an immutable list, which the program concurrently saves in a database. Otherwise, the predicted values are retrieved from the cache [1].

Step 3: Traversable. This step consist in processing `Future[ List[ RefMetric ] ]` and converting such List into `List[ RefMetric ]`. Therefore, the program computes each refactoring operation independently (distinct threads).

Step 4: get System Under Analysis. This step implements the function getSUA. That function is a set of related recursive functions that flattens the data structure `RefMetric` into `ClassMap` by systematically visiting each register (from the refactoring to the metric).

Step 5: Metric Map-Reduce. In this step, the fitness awaits for each refactoring operation (future structures disappears). Actual metrics from the SUA are calculated by using the `HierarchyBuilder`. Lastly, the program implements a map-reduce by using additions to obtain the "code quality additive impacted metric" and the "code quality additive metric".

Step 6: Mathematical Model. The last step calculates the ratio between the actual system and the predicted one. When computing the ratio, the fitness may use a defined penalty.

[1] The fitness functions use the cache when retrieving information about metrics that have already been predicted.

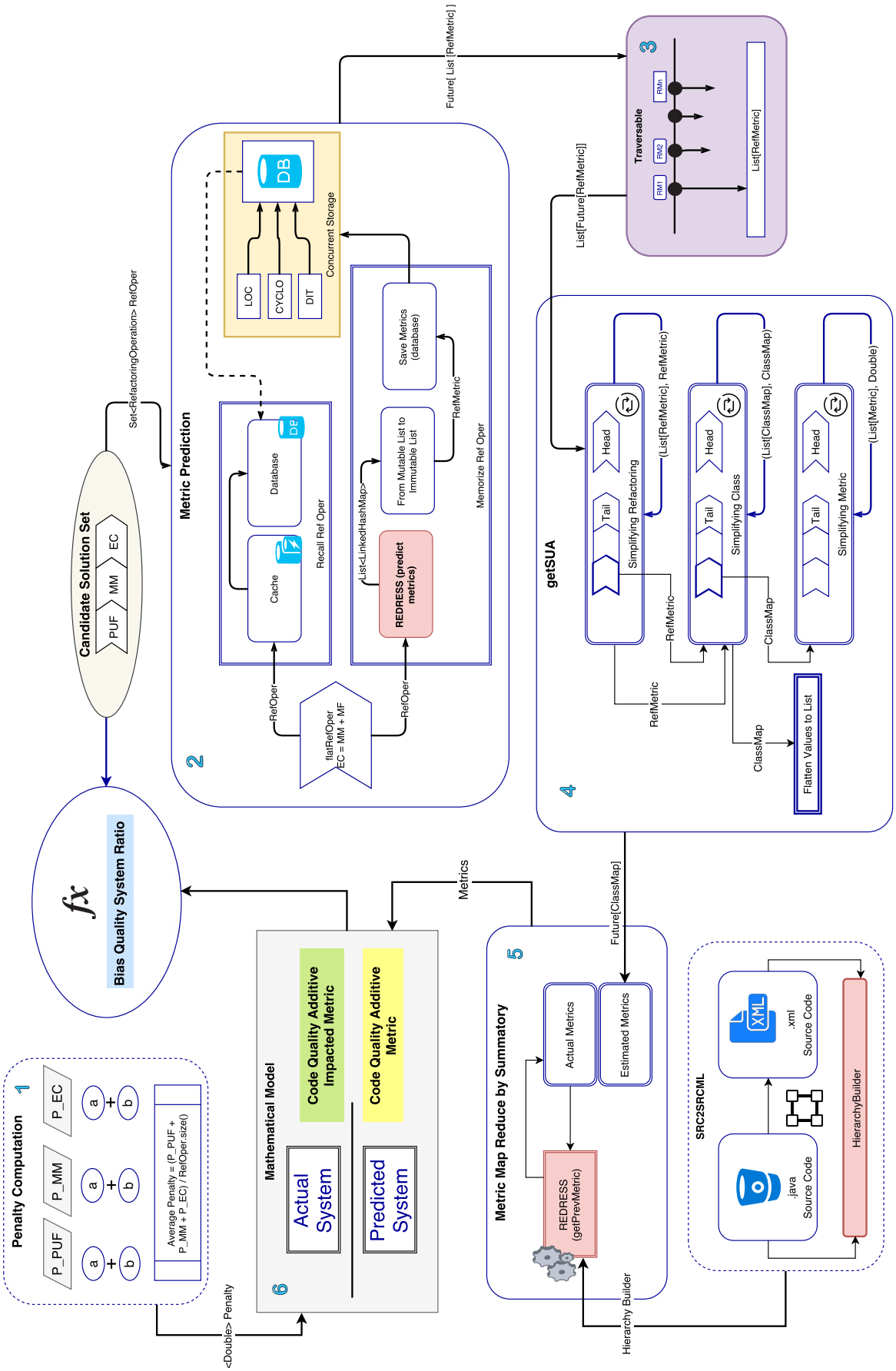


FIGURE 4.2. Refactoring Fitness Map

CHAPTER 5

Approach Evaluation

In this section, the experiments outcomes were organized according to the proposed study design (Figure 5.1: steps 5 and 6). This section shows data related to *baseline search techniques (Hill Climbing and Simulated Annealing)* with proper statistical analysis. Although, the preliminary experiment includes a Shapiro-Will test that took into account the Hybrid Adaptive Evolutionary Algorithm (HaEa) for further comparative analysis (chapter 6).

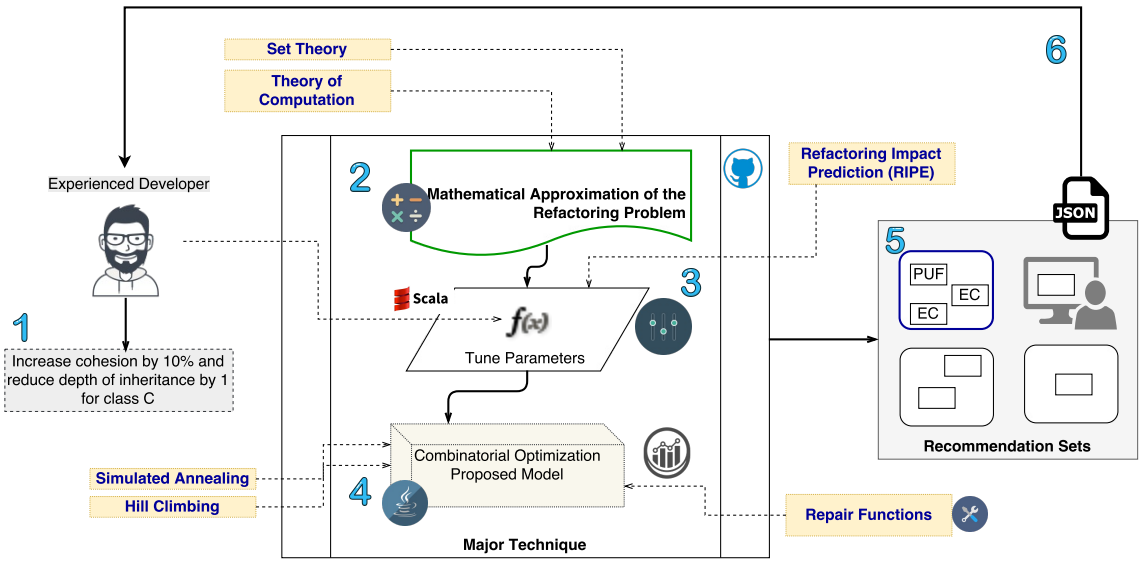


FIGURE 5.1. Artificial Refactoring GENeration (Approach Overview): Model Evaluation

5.1 A study to validate performance and time complexity of the proposed approach

5.1.1 Context

The experiments used three open software systems for analysis: Commons Codec v.1.10 (123 number of classes), Acra v.4.6.0 (59 number of classes) and JFreeChart v.1.0.9. (558 number of classes) These systems were chosen because any researcher could access their source code and subsequent versions (useful for future validation experiments).

Therefore, two datasets were configured according to the expected test. The first dataset contains all the source classes from Commons Codec and the second dataset all the source classes of Acra. Additionally, this document presents six datasets from JFreeChart classes. These datasets were used in time complexity analysis. The classes inside each dataset are hierarchy related, that is the reason why each dataset has different number of classes.

A preliminary experiment and a formal experiment were performed. The preliminary experiment evaluated the do-ability and the course of the study. The formal experiment validated which model configuration was obtaining the best performance. Two sub-experiments were proposed on small and large fitness evaluations.

Both the specification of the combinatorial model and the experiments were constructed and performed within nine months. The necessary intervals between experiments constituted a feedback of the general procedure in each step and spent between two and four weeks before starting new tests.

All the experiments were deployed in cloud services with virtual environments. Thirty machine instances were used with [4-7] Gb of RAM and 2 cores. The generated records were kept in a relational database and the refactoring recommendations outputs were saved in plain text traces and JSON format (Figure A.1).

5.1.2 Procedure Overview

The goal of the study is to evaluate the performance of the approach in generating artificial refactoring recommendations. Based on previous reports of automated refactoring [50, 26], this document proposes the following experiments:

5.1.2.1 Preliminary Experiment: Do-ability

The goal of this experiment is to assess the baseline algorithms and the consistency of the solutions. The preliminary experiment was executed with 10,000 iterations for each algorithm proposed, but machines had been still processing after seven weeks. It was decided to adjust the settings and re-deploy the architecture solution in order to carry out a new experiment. Furthermore, a Shapiro-Wilk test was performed on the data to determine if the data follow a normal distribution.

5.1.2.2 Formal Experiment: Assess the refactoring formalism and the combinatorial model

Each dataset generated a sample of fitnesses. The fitnesses were organized sequentially one by one. The existing position with the previous one was compared by keeping the least value in each evaluation (steady state). As all the experiments were run 30 times, the author organized the results in a matrix where each row represented the evaluation and the column represented each experiment (of the 30). For each row, the median, the median deviation, maximum and minimum values were calculated. Several plots depict previous values to determine the behavior of the fitness of each algorithm during evaluations. The author configured in R all the algorithms for the statistical test.

- Experiment I. Algorithm's Performance by validating which approach is getting the best fitness behavior results.
- Experiment II. Time Complexity of the recommendations by assessing the relationship between the number of classes and the time.

5.1.3 Algorithm's Performance Evaluation

The proposed approach is executed by using two baseline optimization algorithms for two open software systems: Commons Codec v.1.10 and Acra v.4.6.0. The following research question limits the study:

RQ1: What is the technique's performance in generating feasible recommendations on source code?

To answer **RQ1**; first, the source code datasets are organized according to heritage and design relationships; and second, the test performance includes the Hill Climbing and Simulated Annealing algorithms for convergence analysis.

This experiment was performed on two datasets with different settings configurations depicted in Table 5.1. The purpose of this experiment determined which algorithm configuration presented the best fitness performance. The datasets were processed by baseline and evolutionary algorithms with the combinatorial model proposed [12]. The evaluation context presents the following points:

- The experiments were run 30 times and the results organized in a matrix: each row represented a fitness evaluation and each column an experiment.
- The author calculated the median, the median deviation, maximum and minimum values for each row.
- The results of these analyses were presented as value $a \pm b[c]$ format which represents a maximum median performance a with median deviation b through c fitness evaluations.

1. Algorithm Performance

What is the technique's performance in generating feasible recommendations on source code?

TABLE 5.1. Formal Experiment Settings for Algorithm's Performance

Sub-experiment	System	Opt. Algorithm	Genetic Operators	Iterations	Population Size	Refactoring Size
Sub-experiment I: this sub-experiment is oriented to convergence of the algorithms performance with less evaluations.	Commons Codec	SA	Mutation	1000	1	7
				2000	1	7
		HC	Mutation	1000	1	7
				2000	1	7
Sub-experiment II: this sub-experiment is oriented to convergence of the algorithms performance with large evaluations.	Acra	SA	Mutation	10000	1	7
				60000	1	7
		HC	Mutation	10000	1	7
				60000	1	7

5.1.4 Time Complexity Evaluation

The time complexity of the proposed approach is analyzed. The JFreeChart system datasets are used for performing a complete evaluation of Hill Climbing. The following reasearch question limits the study:

RQ2: How is the relation between the variables n classes and t time?

To answer **RQ2**; first, the source code datasets were organized according to design relationships and heritage; and second, the time analysis includes HC and SA techniques for establishing the trend line between classes and time. The experiments were run 30 times and the results organize in a matrix: each row represented a time value and each column an experiment.

2. Time Complexity

How is the relation between the variables n classes and t time?

5.2 Preliminary Experiment Results

For the preliminary experiments, the four proposed algorithms estimated artificial refactorings according to given constraints, but IO processes and repair functions took a lot of time to execute. ACRA, JfreeChart and CCODEC took between two and four weeks to complete an initial experiment with the same configuration as in formal experiments. The outcome is formatted in JSON.

The Shapiro-Wilk test suggested that the data (fitness evaluations) had no normal distribution in both systems ACRA and CCODEC. Table 5.2 presents the $p-values$ for each system and algorithm. All $p-values$ are less than 0.05, then alternative hypothesis is rejected.

TABLE 5.2. The Shapiro-Wilk Test for Preliminary Analysis

Algorithm/Dataset	CCODEC[2000]	ACRA[60000]
Hill Climbing	0.0049	0.0015
Simulated Annealing	0.0222	0.0157
HaEa	0.0144	0.0340

5.3 Formal Experiment Results

For the formal experiments, this section provides two results that corresponds to the research questions in study design. One result is oriented to provide data about baseline algorithms' performance, whereas the other is oriented to provide data about time complexity of baseline search techniques (HC and SA).

5.3.1 Algorithm's Performance

Presented in this subsection are the results of the performance of the four proposed algorithms. This document introduces three major results of comparative evaluations.

This section is summarized in a blue box the general performance in the last evaluation for each algorithm. This performance is formatted as $a+/-b[c]$ where a indicates maximum average with standard deviation of b reached by each algorithm using c fitness evaluations in 30 independent runs. This document compared the fitness values for HC and SA in three curves: best, median and worst.

ACRA Performance Results

- Hill Climbing: 0.97493 +/- 0.00865 [10000]
- Hill Climbing: 0.96815 +/- 0.00563 [60000]
- Simulated: 0.97167 +/- 0.00354 [10000]
- Simulated: 0.96819 +/- 0.00174 [60000]

CCODEC Performance Results

- Hill Climbing: 1.00902 +/- 0.00339 [2000]
- Simulated: 1.00621 +/- 0.00355 [2000]

5.3.1.1 Comparative Evaluations CCODEC[2000]

One thousand evaluations after initiation of the experiment, the three curves decrease with a stabilized dispersion. However, neither of the two search techniques converged. By two thousand evaluations, the curves still diverged (Figure 5.2), but the fitness value for the Simulated Annealing technique was approximately 1.003 times lower in average than Hill Climbing's. For both algorithms, values were often greater than 1 and the worst curve was stable.

5.3.1.2 Comparative Evaluations ACRA[10000]

Figure 5.3 shows fitness values for more than 2000 evaluations in a different system. Simulated Annealing was more likely to obtain lower values, the three curves are decreasing, whereas the evaluations are getting higher. Five thousand evaluations after initiation of the experiment, the Hill Climbing was becoming stable with constant dispersion. By ten thousand evaluations, Simulated Annealing technique was approximately 1.003 times

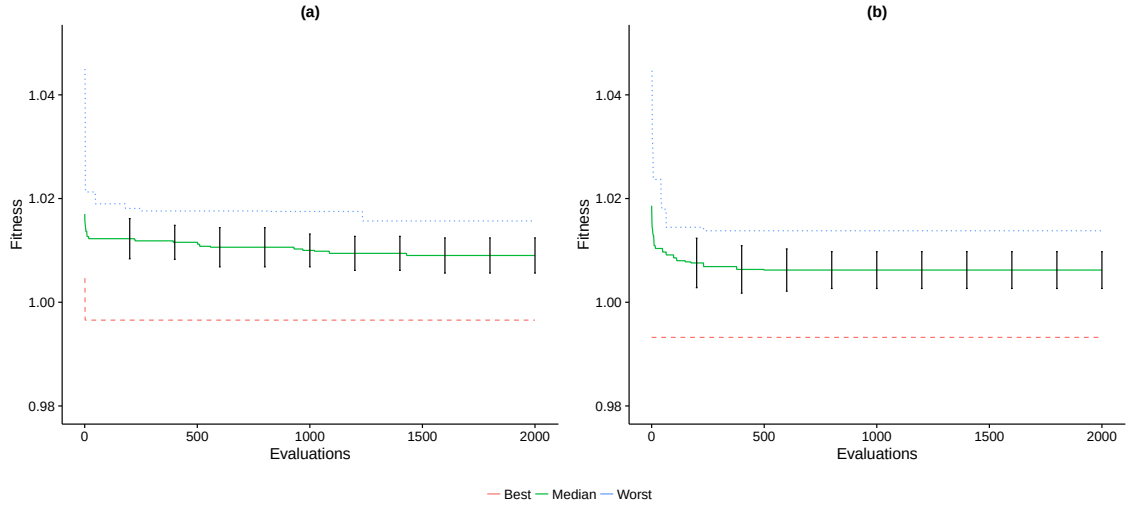


FIGURE 5.2. Two thousand evaluations in 30 independent runs for CCODEC: (a) Hill Climbing and (b) Simulated Annealing.

lower in average than Hill Climbing's. For both algorithms, the reached fitness values were less than 1.

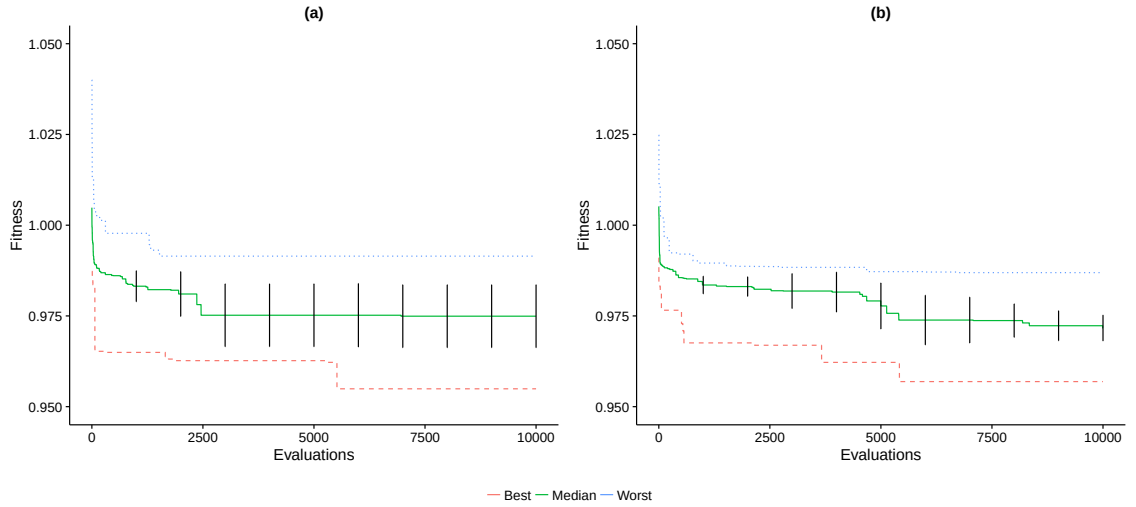


FIGURE 5.3. Ten thousand evaluations in 30 independent runs for ACRA: (a) Hill Climbing and (b) Simulated Annealing.

5.3.1.3 Comparative Large Evaluation ACRA[60000]

After 30000 fitness evaluations after initiation of the experiment, Simulated Annealing and Hill Climbing did not converge yet (Figure 5.4). Even though, the Hill Climbing was more likely to have higher dispersion and stable curves in best and worst values. By 60000 evaluations, however, the Simulated Annealing technique was converging in a mean value of 0.96819 with an standard deviation of 0.00174. A contradictory result showed

that the Hill Climbing was approximately 1.00004 times lower in average than Simulated Annealing's.

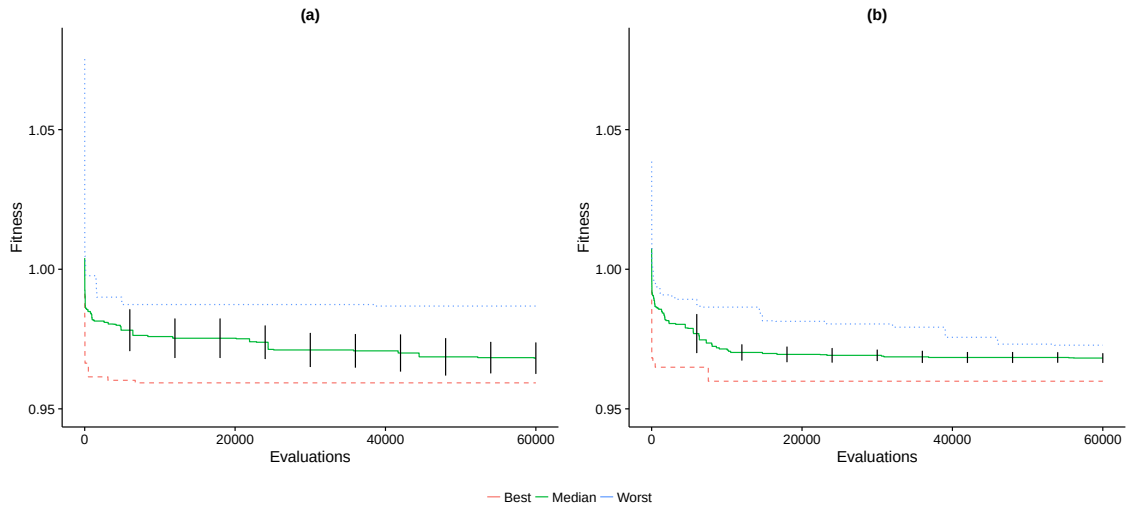


FIGURE 5.4. Sixty thousand evaluations in 30 independent runs for ACRA: (a) Hill Climbing and (b) Simulated Annealing.

5.3.2 Time Complexity

This subsection presents the time complexity results for the Hill Climbing technique in JFreeChart datasets. Table B.1 shows the number of classes, the mean and standard deviation for each set. The higher the numer of classes, the higher average time, in seconds, it took the Hill Climbing and Simulated Annealing to complete 5,000 evaluations.

For Hill Climbing, the percent change increased approximately 287.75%, 251.47%, 654.23%, 421.47% and 504.41% from DAT01 to 22, 43, 48, 60 and 71 number of classes respectively. For Simulated Annealing, the percent change increased approximately 251.37%, 209.47%, 729.96%, 431.35% and 515.92% from DAT01 to the remaining datasets. However, according to Table B.1, the number of classes did not affect the standard deviation values.

Figure 5.5 depicts the mean time complexity for Hill Climbing. The data trended toward an exponential pattern with the related equation $y = 648.623e(0.025x)$. The average time complexity for HC was exponential; by 22, 34, 48, 60 and 71 number of classes, the mean time was approximately 3.88, 3.51, 7.54, 5.22 and 6.04 respectively times higher than the mean time for 11 number of classes.

Figure 5.6 depicts the mean time complexity for Simulated Annealing. The data trended toward an exponential pattern with the related equation $y = 625.171e(0.026x)$. The average time complexity for SA was exponential; by 22, 34, 48, 60 and 71 number of classes, the mean time was approximately 3.51, 3.09, 8.30, 5.31 and 6.16 respectively times higher than the mean time for 11 number of classes.

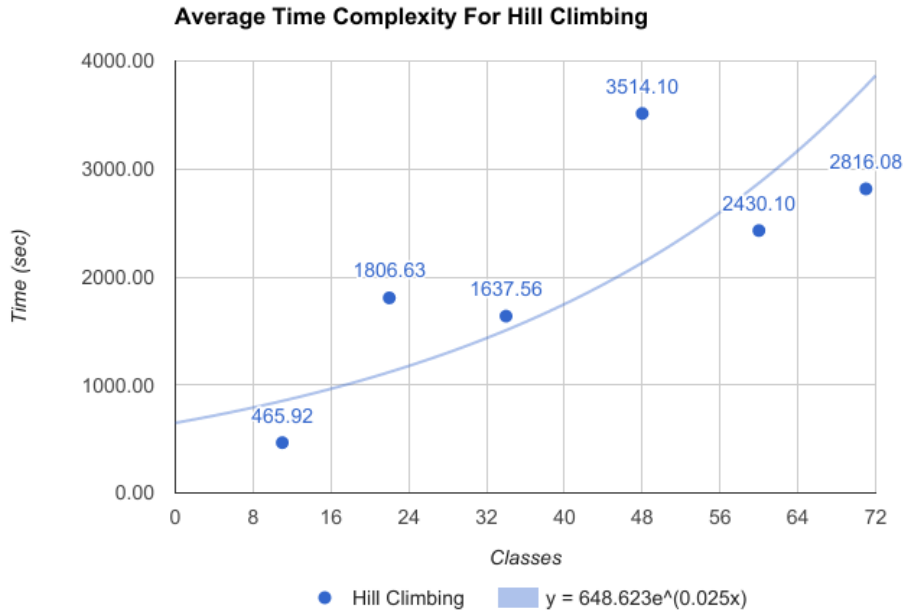


FIGURE 5.5. Average Time Complexity Hill Climbing. In blue, the hill climbing data points. In light blue, HC trendline with the exponential model. Y-axis is the time in seconds and X-axis is the number of classes for JFreeChart System.

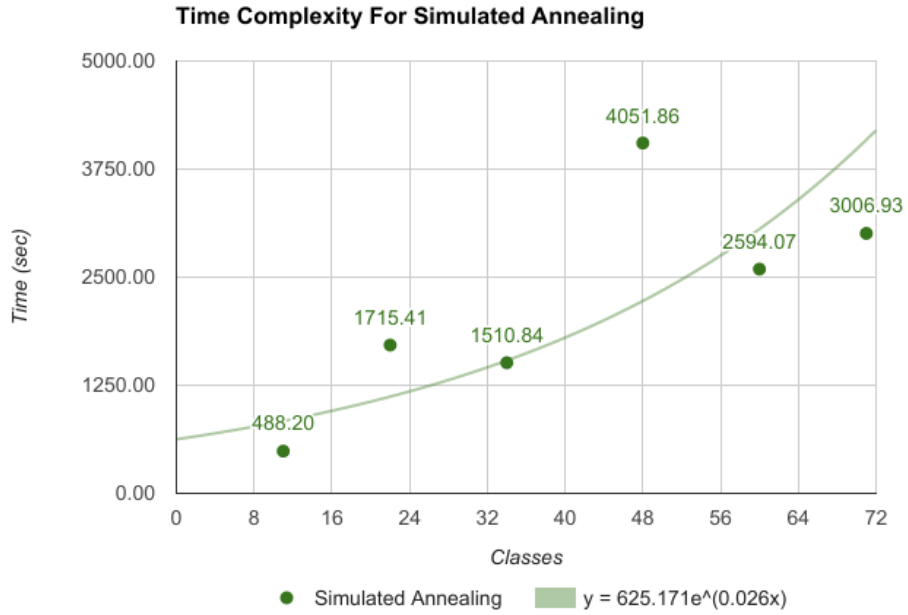


FIGURE 5.6. Average Time Complexity Simulated Annealing. In green, the simulated annealing data points. In light green, SA trendline with the exponential model. Y-axis is the time in seconds and X-axis is the number of classes for JFreeChart System.

CHAPTER 6

On the Use of a Hybrid Adaptive Technique

This document introduces an Artificial Refactoring Hybrid Adaptive Technique (ArHaT) to estimate artificial refactoring operations. In this chapter, Evolutionary Algorithms (EA) are an alternative solution for a proposed combinatorial optimization model. In this case, the author used the Hybrid Adaptive Evolutionary Algorithm (HaEa) that evolves the operator rates and the solutions at the same time [12]. Two versions of HaEa were considered: invariable and variable chromosome. An invariable chromosome indicates that the individual representation of the refactorings do not change because of the genetic operators, this document refers to that algorithm as HaEa. Whereas, a variable chromosome indicates that the individual allows modification in its size. The chromosome can vary between a minimum and maximum range of recommendations, this document refers to that algorithm as HaEa Var. The best individual was keeping it after each iteration; therefore, both algorithms showed a steady-state behavior.

6.1 A Hybrid Adaptive Evolutionary Algorithm for the Refactoring Problem

The software refactoring problem was represented as a combinatorial problem. Thus, it could be modeled with Evolutionary Algorithms (EAs). This section is an extension of the proposed refactoring model in previous chapters. The research community has used EAs successfully to solve several optimization problems; the performance of these techniques depends on the selection of initial parameters. Picking up the initial parameters is a time consuming task, that is the reason why this project used HaEa.

In HaEa, each individual is evolved independently from other individuals in the population. In each generation, one genetic operator (such as crossover or mutation) is selected (for each individual) according to dynamically learned operator rates that are encoded into the individual. Whether the selected operator requires another individual for mating, then this second individual is selected from the population with replacement. For instance, the second individual can be a mate more than once. The offspring is compared against the first parent according to the fitness value. HaEa replaces the parent, and the operator rate is rewarded for improving the individual whether the fitness of the offspring is better

than the parent. Whereas, the operator rate is penalized whether the individual does not improve. The source code of this approach is available online [1].

The proposed Hybrid Adaptive approach used the defined repairing functions and six different genetic operators (Figure 6.1) that were specially created for exploring in spaces where the algorithm preserves the objected-oriented structure. The approach does not need for encoding phenotype to genotype; the chromosome represented the refactoring recommendation directly.

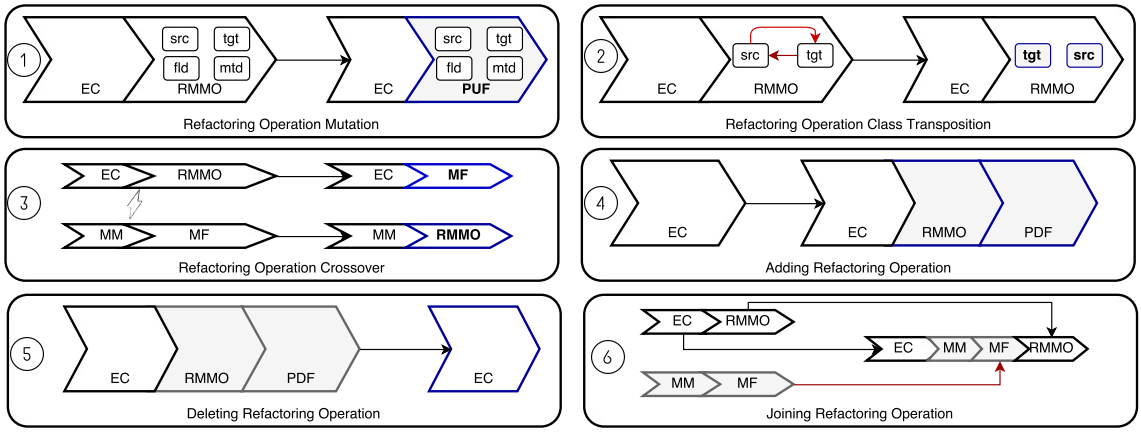


FIGURE 6.1. Proposed Genetic Operator for HaEa: from 1 to 3 are invariable chromosome operators and from 4 to 6 are variable chromosome operators

Figure 6.1 depicts six different operators. For further understanding of the operators, please review refactoring individual configuration in chapter 4. The representation of the chromosome is the same representation of the individual described before along with the search space.

On the one side, HaEa with invariable (fixed) chromosome uses: **(1) Refactoring Operation Mutation** that changes a complete refactoring operation label preserving its internal parameters (src, tgt, fld and mtd); for instance, the operator changes into PUF by following a probability distribution (Gaussian) whether the original gen is a RMMO; **(2) Refactoring Operation Class Transposition** that interchanges the src with tgt parameters inside a specific gen (the gen is selected by following a Gaussian distribution); for example, in a RMMO gen the tgt is replaced with src, and the src is replaced with the tgt; and **(3) Refactoring Operation Crossover** that crosses two refactoring operations in a random position (Gaussian distribution) of the chromosomes, the result after the operator is two chromosomes (offspring) with blended genetic information from two original chromosomes (parents), the crossing includes all the internal parameters -indicates that the genes are preserved.

On the other side, HaEa with variable chromosome uses: **(4) Adding Refactoring Operation** that adds a specific number of refactorings to the individual; for instance, the operator could add a tuned number of genes, in this case RMMO and PDF, when the parent is composed of EC; **(5) Deleting Refactoring Operation** that deletes a specific number of refactorings to the individual; for example, the operator could dispose of a tuned number of genes, in this case RMMO and PDF, when the parent is composed of three genes; and **(6)Joining Refactoring** that mixes two individuals into one; for

[1] https://github.com/BIORIMP

instance, when the parent are two chromosomes, one of the parent is broken into two parts (the broken position is selected by following a Gaussian distribution) and afterwards the other parent is introduced between, the joining includes all the internal parameters -indicates that the genes are preserved.

6.2 Study Design

Hill Climbing and Simulated Annealing were compared against HaEa with an appropriate statistical test that was concluded during the preliminary evaluation. The statistical test consisted of applying the Wilcoxon test that compares the median of two samples thusly this research defined a null (H_0) and an alternative (H_1) hypothesis. This Wilcoxon was applied in intervals for sub-experiment I and sub-experiment II (see Table 6.1).

Table 6.1. Formal Experiment Settings

Sub experiment	**System**	**Opt. Algorithm**	**Genetic Operators**	**Iterations**	**Pop Size**	**Ref Size**
Sub-experiment I: This sub-experiment is oriented to convergence of the algorithms performance with fewer evaluations.	Commons Codec	SA	Mutation	1000	1	7
				2000	1	7
		HC	Mutation	1000	1	7
				2000	1	7
		HaEa	Mutation, crossover and transposition.	50	20	7
				50	20	7
		HaEa-VAR	Join, delete and add genes.	50	20	[5-12]
Sub-experiment II: This sub-experiment is oriented to convergence of the algorithms performance with larger evaluations.	Acra	SA	Mutation	10000	1	7
				60000	1	7
		HC	Mutation	10000	1	7
				60000	1	7
		HaEa	Mutation, crossover and transposition.	420	20	7
				1000	50	7

The null hypothesis H_0 states that the median of Hill Climbing fitness [2] is the same as HaEa's. On the other hand, the alternative hypothesis (H_1) constitutes that the median of HaEa fitness is greater than the baseline algorithms'. The α value followed 0.05 level of significance (95% of confidence interval). The Wilcoxon test is applied on alternative hypothesis whether the null hypothesis is rejected. The $p - value$ calculated is compared to the alpha value. (H_1) is rejected whether alpha is greater than $p - value$; therefore, the median of HaEa is not greater than the baseline algorithms'. However, the statistical evidence for rejecting (H_1) is insufficient whether the p-value is greater than alpha. The objective comprises the median of HaEa fitness in all the samples that are less than the baselines'.

One could establish the objective function based on the Refactoring Impact Prediction (RIPE) [5] because RIPE provides refactorings without implementing the actual changes. Therefore, the model estimated the refactoring recommendations from the impacted (predicted) metric values.

[2]The fitness function used is the same objective function introduced in chapter 3 (definition 10).

The preliminary experiments consolidated the perceived behavior of the proposed model and also the time consumed per operation. Besides, those tests sufficiently informed for redeploying the system.

Eventually, the proposed experiments were important for verifying and validating the performance of the evolutionary algorithm and the candidate solution. Both experiments aimed to demonstrate combinatorial optimization techniques.

6.3 Results and Discussion

6.3.1 Performance and Convergence Analysis

In exploratory analysis (Table 6.2) , the median values for Commons Codec for 1000 evaluations were lower than 2000 evaluations. Likewise, the median deviation in hill climbing for Commons Codec for 1000 evaluations $\pm 0.00323[1000]$ were higher than 2000 evaluations $\pm 0.00339[2000]$. HaEa performance in 800 evaluations was $1.00549 \pm 0.00586[800]$ and HaEa with variable chromosome was $1.01588 \pm 0.0027[800]$.

TABLE 6.2. Performance Exploratory Data Analysis

System	Algorithm	Median	Deviation	Evaluation
Acra	HC	0.97493	0.00865	10000
		0.96815	0.00563	60000
	SA	0.97167	0.00354	10000
		0.96819	0.00174	60000
	HaEa	0.96478	0.00483	10000
		0.95493	0.00398	60000
Ccodec	HC	1.0063	0.00323	1000
		1.00902	0.00339	2000
	SA	0.98121	0.02482	1000
	HeEa	1.00318	0.00665	1000
		1.01015	0.0027	2000
	HaEa Var	1.01588	0.0027	800

After one thousand evaluations (Figure 6.2) in Commons Codec System, HaEa approach demonstrated significantly lower fitness performance than Hill Climbing ($p - value = 0.1013$) and Simulated Annealing (p-value $= 0.9998$). Then, there was not enough statistical evidence for rejecting the alternative hypothesis. Baseline algorithms (a) and (b) have better performance than evolutionary approach (c). Additionally, HaEa with variable chromosome (d) has similar behavior as normal HaEa in 800 evaluations with lower fitness performance than HC ($p - value = 1.0000$) and SA (p-value $= 1$). In (a) the behavior of median and worst values was homogeneous, but the best values had a substantial decrease after 800 evaluations. In (b) the more evaluations were performed, the higher median deviation. A constantly decreasing tendency was observed in best values for (c) (0.98855[100], 0.97619[500], 0.97337[800], 0.96126[1000]), besides the median deviation was slowly incremented ($\pm 0.00378[100]$, $\pm 0.00573[500]$, $\pm 0.00586[800]$, $\pm 0.00665[1000]$). In (d) the worst, median, and best were constant.

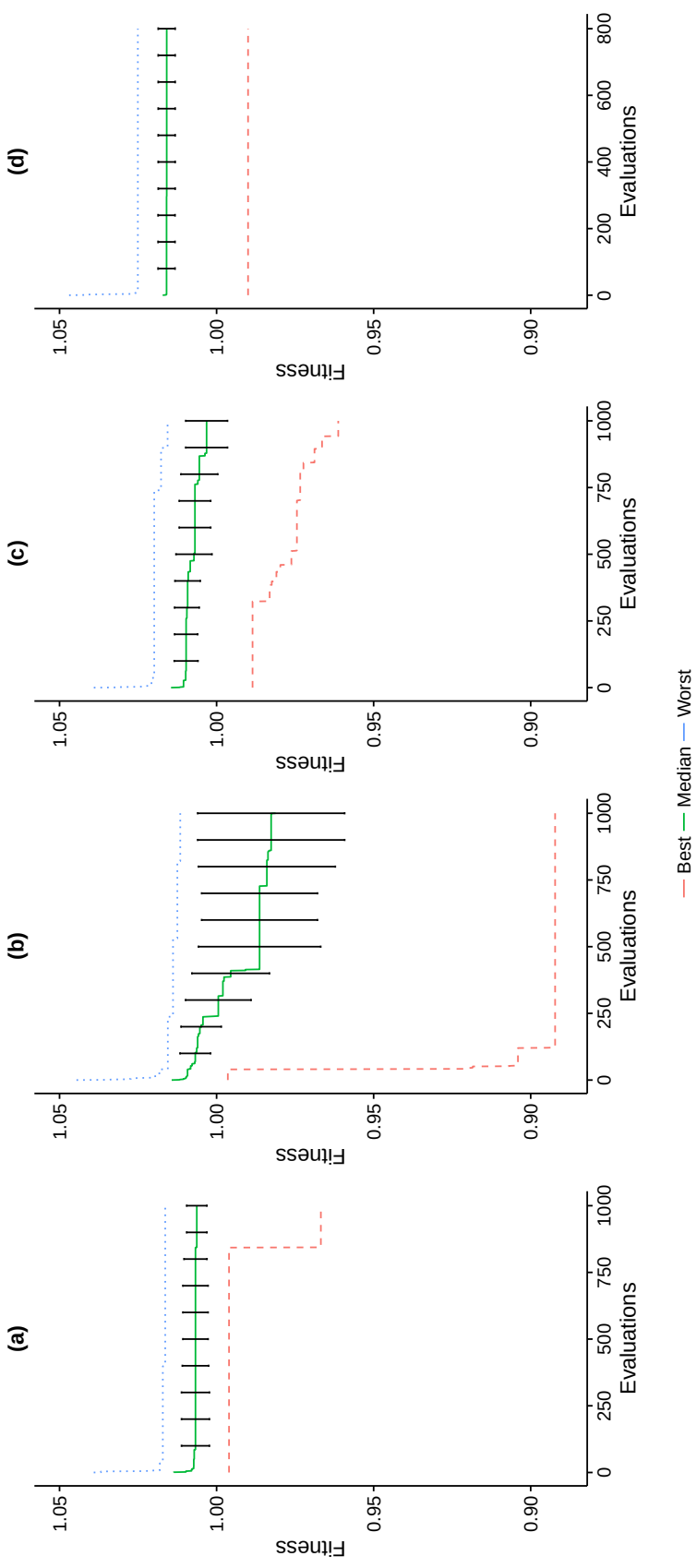


FIGURE 6.2. Commons Codec System Fitness Performance up to 1000 Evaluations for four Combinatorial Algorithms: (a)Hill Climbing, (b)Simulated Annealing, (c)HaEa, (d)HaEa with variable chromosome.

Figure 6.4 shows for three algorithms the largest evaluations attempted during the experiment. Best fitness rates in (a) and (b) are relatively homogeneous during evaluations. In (a) the median and worst values became stable after 3,000 evaluations, however, in (b) median values are variable though all evaluations (0.97775 ± 0.00633[5000], 0.97386 ± 0.00631[7000], 0.97167 ± 0.00354[10000]). The author observed that HaEa (c) obtained best values starting from 6,000 evaluations for HC ($p - value = 0.0350$) and 3,000 evaluations for SA (p-value = 0.0412). By 10,000 evaluations, HaEa values were significantly lower than HC ($p - value = 0.0002$) and SA ($p - value = 0.0001$). Although, HaEa behavior (c) in large evaluations was better in all sample points (Table 6.3 and Figure 6.5) with a remarkable p-value = 0.00023.

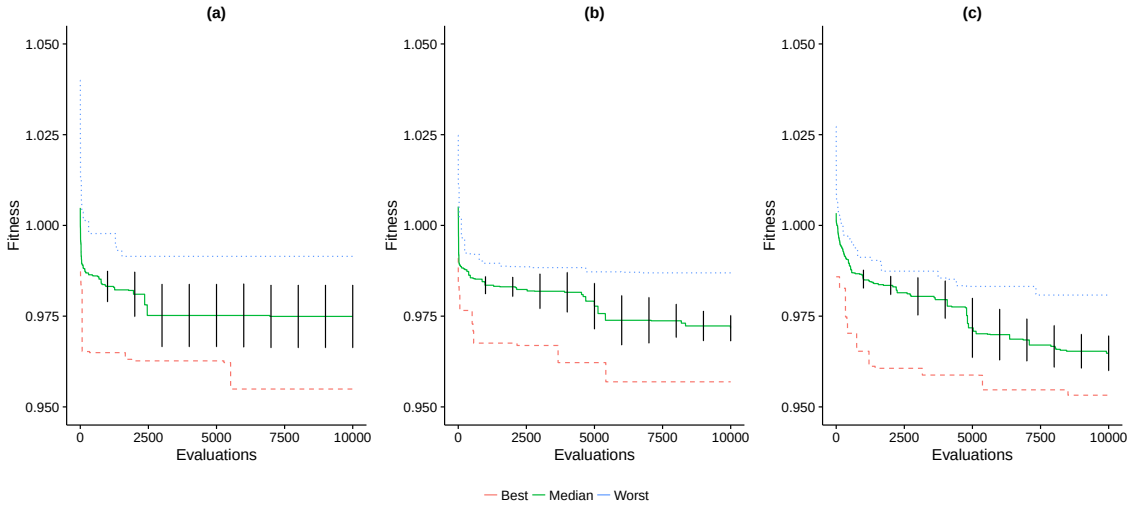


FIGURE 6.3. Algorithm Performance ACRA[10000]: (a) Hill Climbing, (b) Simulated Annealing and (c) HaEa

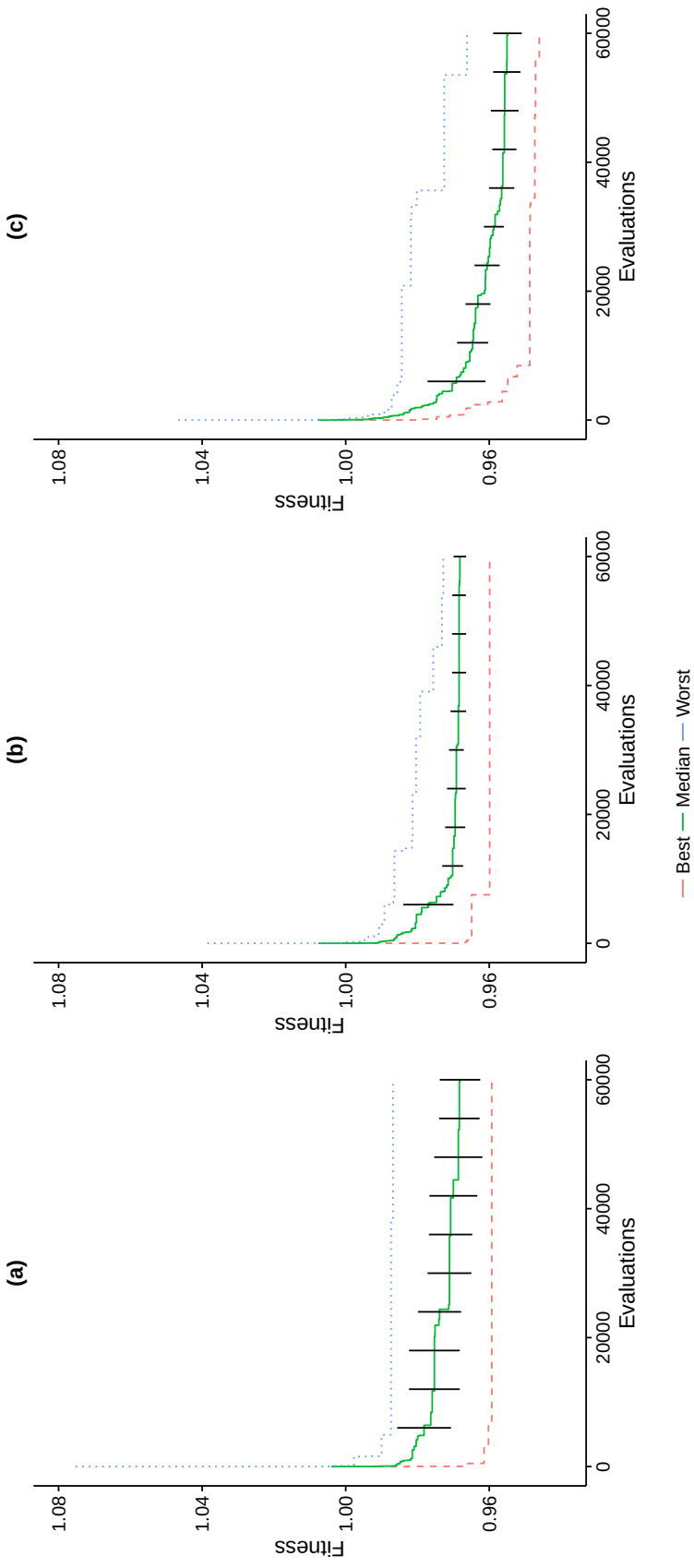


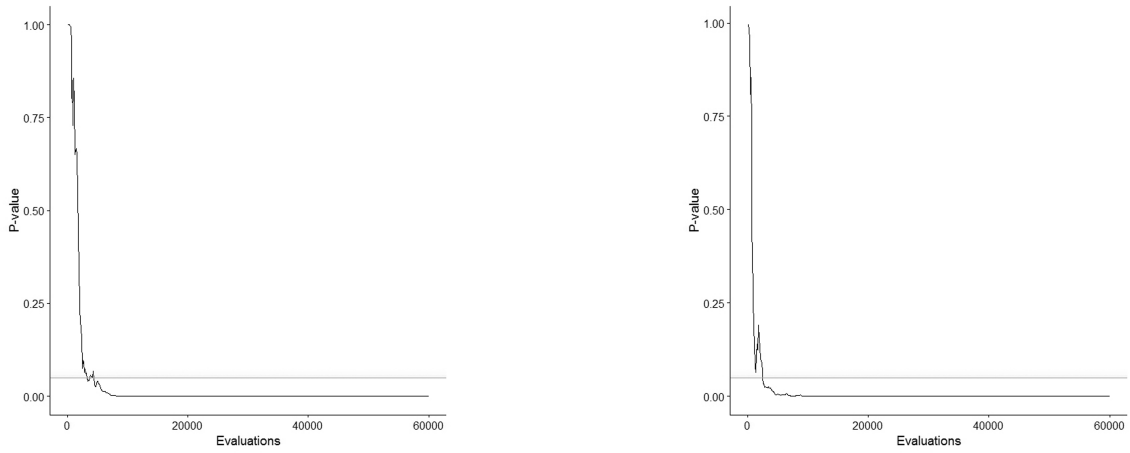
FIGURE 6.4. Acra System Fitness Performance up to 60000 Evaluations for four Combinatorial Algorithms: (a)Hill Climbing, (b)Simulated Annealing, (c)HaEa.

Our results exhibit that HaEa presents good performance in large evaluations. HC and SA experience relative better results in few steps than HaEA, even though evolutionary approach reaches the best values after certain evaluations.

TABLE 6.3. Performance Rates over Number of Evaluations for HC, SA and HaEa in Acra System

	Performance (median fitness)		
Evaluation	Hill Climbing	Simulated Annealing	HaEa
10000	0.97589	0.97145	0.96543
20000	0.97529	0.9695	0.96133
40000	0.97077	0.9684	0.95623
50000	0.96862	0.9684	0.9557
60000	0.96815	0.96819	0.95493

p - value <= 0.000226249503 vs HeEa group.

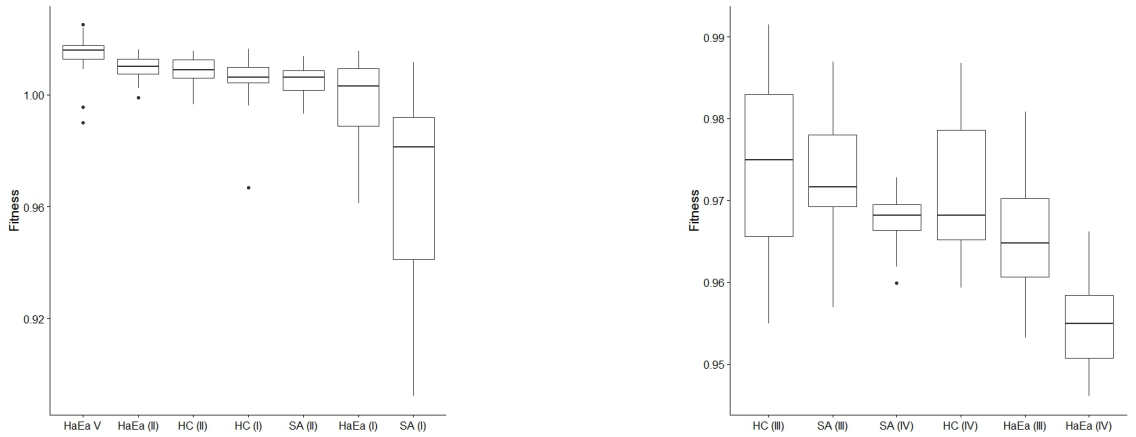


(A) HaEa vs. Hill Climbing (B) HaEa vs. Simulated Annealing

FIGURE 6.5. P- Value Acra System Fitness Performance up to 60000 Evaluations.

The following plots Figure 6.6a and Figure 6.6b aim to compare all four algorithms. In system ccodec, the best behavior corresponded to Simulated Annealing algorithm in 1000 evaluations, yet it presents highest variability. On the other hand, in system Acra, HaEa had the best performance in 10000 and 60000 evaluations, whereas HC and SA did not show the best results.

HaEa presents good performance in large evaluations is supported by box-plot analysis. Ten thousand evaluations were executed on acra system, only HaEa experienced the best median value. Likewise, after 6,000 evaluations, HaEa was still searching for fewer values. Our second finding that Hill Climbing and Simulated Annealing induced better results in few evaluations as in ccodec experiments with 1000 and 2000 evaluations respectively. However, the tendency can not be fully analyzed with that range of steps. Graphics point that median values are homogeneous, especially in HaEa with a variable chromosome.



(A) Commons Codec System (B) Acra System

FIGURE 6.6. Comparative Box Plot: (I)ccodec 1000, (II)ccodec 2000, (III)acra 10000, (IV)acra 60000.

Previous studies have discovered some combinatorial representations of refactorings [10, 14, 35, 50], but this document established a complete benchmark problem contrasting baseline algorithms (HC and SA) versus HaEa with an extensive performance study.

In brief, our research exhibits that evolutionary algorithm experiences better results in time than baseline approaches. Since ccodec system consumed merely up to 2000 evaluations, the author processed an extensive test in Acra for validating the minimization of the median values. Yet, our mathematical model might be extended for including penalization or complete configuration of hyper-parameters.

6.3.2 Time Complexity Analysis

Figure 6.7 depicts a comparison of the mean time complexity between baseline algorithms (HC and SA) and HaEa search techniques for 30 independent experiments. The algorithms data tended to show exponential patterns (HaEa with an equation model $y = 470.537e(0.024x)$).

Five thousand fitness evaluations after initiation of the experiment, the average time complexity for the Hill Climbing (Table 6.4) technique was 1.64 and 1.41 higher than HaEa's with 71 and 11 number of classes respectively. Whereas, the average time complexity for the Simulated Annealing technique (Table B.2) was 1.75 and 1.48 higher than HaEa's with 71 and 11 number of classes respectively. Therefore, HaEa spent less time than both HC (1.45 times in average) and SA (1.50 time in averge) in performing refactoring recommendations. The baseline algorithms trend lines were higher than HaEa trend line (blue line for HC and green for SA in Figure 6.7).

The percentage increase in HC and SA techniques from 11 classes to 71 was 504.41% and 515.92% compared to 419.63% in HaEa's. The result shows that each time the experiment increased the number of the classes, baseline algorithms took more time in processing classes for refactoring than HaEa. HaEa seems to be a better approach to estimate refactoring recommendations in less time.

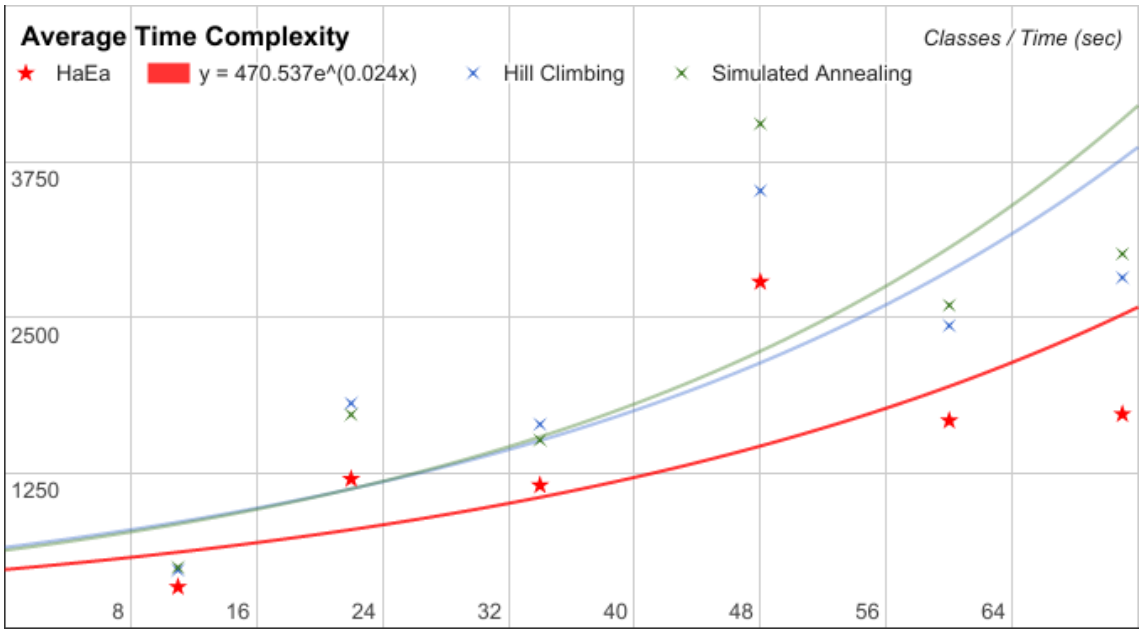


FIGURE 6.7. Average Time Complexity Hill Climbing vs Haea. In blue and green, the hill climbing and simulated annealing trendline. In red, HaEa trendline with the exponential model. Y-axis is the time in seconds and X-axis is the number of classes for JFreeChart System.

TABLE 6.4. Time Complexity Exploratory Analysis for HC vs HaEa

JFreeChart Dataset	Hill climbing			HaEa	
	#Classes	mean(sec)	SD(sec)	mean(sec)	SD(sec)
DAT01	11	465.92	414.04	330.95	226.34
DAT02	22	1806.63	1322.17	1198.45	605.53
DAT03	34	1637.56	1579.98	1147.93	694.25
DAT04	48	3514.10	938.54	2781.87	980.02
DAT05	60	2430.10	1215.08	1667.78	595.10
DAT06	71	2816.08	1121.70	1719.71	775.82

The author found that Hybrid Evolutionary Approach has lower value results in large iterations. Then, evolutionary algorithms seem to work better than baselines. Besides, the main inconvenience of execution was latency. The author noticed significant differences between IO processing from files to databases. Thus, it was decided to redeploy and test with new parameters.

CHAPTER 7

Discussion

In this study, the author poses whether a systematic formal approach based on a formalism and a combinatorial optimization model constitutes an appropriate solution for the software refactoring problem. In this case, the software refactoring problem is defined as the generation of artificial refactoring sequences without actual implementation. The results show that a mathematical representation of an object-oriented system (a) appears to define software refactoring concepts; a combinatorial optimization approach (b) models the software refactoring problem; the author (c) designed the computational techniques such as Hill Climbing, Simulated Annealing and HaEa to solve the software refactoring problem by means of the proposed formalism and combinatorial model and (d) validated those algorithms in terms of performance and time complexity.

The first finding that a formalism appears to define software refactoring concepts is supported by the definitions introduced in chapter 3. The author found that by using a mathematical approximation to the refactoring problem, the research community could model and report a combinatorial optimization approach in a standardized manner. With the proposed formalism, any researcher is able to reproduce the experiments and to extend the combinatorial model.

The second finding that a combinatorial optimization approach models the software refactoring problem is supported by chapter 4 where the author included a characterization of the search space, a description of the objective (fitness) function, the refactoring constraints which support the behavior preservation through repair and generating functions and computational complexity classification of the problem.

Furthermore, due to an unified mathematical approximation, this research supports to improve combinatorial optimization models to solve the software refactoring problem. For instance, Chaparro et al. [5] present RIPE (Refactoring Impact PrEdiction): an approach for estimating the impact of refactoring operation on code quality metrics. The author's technique uses RIPE for creating the optimization model (the formulation of the fitness function). All the prediction functions in RIPE are considered in the author's model when measuring the performance of each individual represented as a set of refactoring operations.

The third finding that designed computational techniques such as HC, SA and HaEa solve the software refactoring problem is supported by algorithm adaptations introduced

in chapter 2 and chapter 6 like refactoring data structures, deployment of the fitness function that embodies Java and Scala technologies, and definition of specific genetic operators. Moreover, the following previous studies reported different forms of adapting search-techniques into the refactoring generation:

Seng, et al. [50] propose an evolutionary algorithm for optimizing class structure in a open source system in order to conserve class level refactoring. The expected outcome is a refactoring produced by the value of several quality metrics and the number of violations of object oriented design guidelines. Their study design considers only one refactoring (move method) and restricted number of classes; thus, the search space is significantly reduced. Whereas, ARGen explores all the possible search space for the given classes with twelve different refactoring operations.

Harman, et al. [10] introduces a search-based refactoring approach based on pareto optimality extending the initial idea of Seng, et al. They use one refactoring (move method) and two metrics (coupling between objects and standard deviation of methods per class) for determining a *sequence of refactoring*. The procedure for applying the sequence to the system is unclear, as well as the importance of choosing those metrics. Our study is different, the author, first, established the foundations to work with refactoring operations in mathematical domain and, second, to evaluate a hybrid adaptive technique by focusing on the assembly of the *refactoring sets*. The research fitness function covers eleven different quality metrics grouped in a ratio.

Ouni, et al. [43] suggest using development history to assemble refactoring solutions in similar context. They used NSGA-II for minimizing the number of bad smells and maximizing the semantic coherence and *the development history use*. Their approach finds refactoring solutions composed of a sequence of refactoring operations where the order of the operations is taken into account. Authors uses a vector representation for the candidate solution for preserving pre and post conditions. Crossover and mutation are the only genetic operators implemented for NSGA-II approximation. Furthermore, authors tested a chemical reaction optimization approach to find solutions of refactorings that maximize the number of fixed riskiest code smells [38] and used genetic programming for generating rules that detects defects when minimizing software effort. Unlike Ouni, et al., the hybrid adaptive technique implemented six different genetic operators and a solution set with independent refactoring operations. These operators can add or delete refactoring operations dynamically. Moreover, the hybrid adaptive algorithm allowed an adaptive tuning that reduces the error-fault in the experiments.

Mksover, et al. [24], oriented their research to the high dimensional problem [27] in refactoring determination. They have done extensive research in multi-objective optimization which includes features such as quality improvements (number of fixed code smells), severity of code smells and importance of the classes that contain bad smells (leading to lack of robustness) [26]. Moreover, they proposed a tool that receives developers feedback and adapts for suggesting refactoring operations [25]. However, any computational execution of refactoring needs not only exhaustive code validations, but also a deep knowledge of the system analyzed; thus, augmenting computational time complexity. This research focused on estimating the quality metrics in favor of *avoiding the refactoring operation execution*. The empirical experimentation is not clear about how large are the sequences of refactorings, yet, HaEa uses both delete and add refactoring operators.

The last finding that the combinatorial techniques are validated in terms of performance and time complexity is supported by the empirical experiments in chapter 5 and

chapter 6. Answering **RQ1**, for dataset CCODEC, three defined curves (best, median and worst) did not converge for HC and SA after 2,000 evaluations; nonetheless, for ACRA, both algorithms converged in a mean value of 0.96819 with a standard deviation of 0.00174 after 60,000 fitness evaluations. Conversely, answering **RQ2**, the relation between a number of classes and time is exponential for both HC and SA. The evolutionary approach was significantly better ($p - value < 0.00023$ vs HaEa) than the baseline techniques when converging to lowest fitness values.

On the other hand, the three techniques (HC, SA and HaEa) exhibited an exponential trend for the time complexity. The Hybrid Adaptive Evolutionary Algorithm outperformed baseline search techniques for n number of classes to be analyzed. The average time complexity for HC and SA when $n = 71$ was 1.64 and 1.75 higher than HaEa's. Therefore, HaEa spent less time in performing refactoring recommendations.

Figure 7.1 summarizes the limitations and key assumptions. I summarize the limitations of this research as below : our research did not develop a refactoring tool to implement recommendations; ARGen does not assist developers in *what software properties need to change* (e.g., reduce cyclomatic complexity by 10% and increase cohesion by 0.2 in class C), but *how to change the software* (e.g., a Replace Method with Method Object is a suggested refactoring operation for this source "src" and target "tgt" classes according to some fixed weights in the objective function); the estimated sets of refactorings did not exclusively represent actionable recommendations -actionable behavior depends upon developers' criteria; nonetheless, the sets do represent feasible individuals that fulfill fitness parameters and object oriented guidelines-; and, the empirical evaluations concentrated on algorithm performance and refactoring structure coherence.

Bearing in mind previous findings, I conclude that regardless of reported empirical studies [38, 26, 42, 23, 40, 24, 43], which exhibits precision and recall measures, the refactoring process highly depends on *human factor* that comprises specific domain knowledge and expertise when designing reconstructions on the code. We, indeed, achieved to estimate massive artificial refactorings, yet I cannot guarantee that none of those refactorings were actionable. Actionability implies not only developers' interest in quality metric but also developers' criteria (how to design a SUA) that should be extracted from their minds. Consequently, search techniques or combinatorial analysis are insufficient approximations to tackle the SRP because those techniques cannot acknowledge how the developer's mind is simulated, used, or mapped to produce the exact context (human design process) for recommending refactorings.

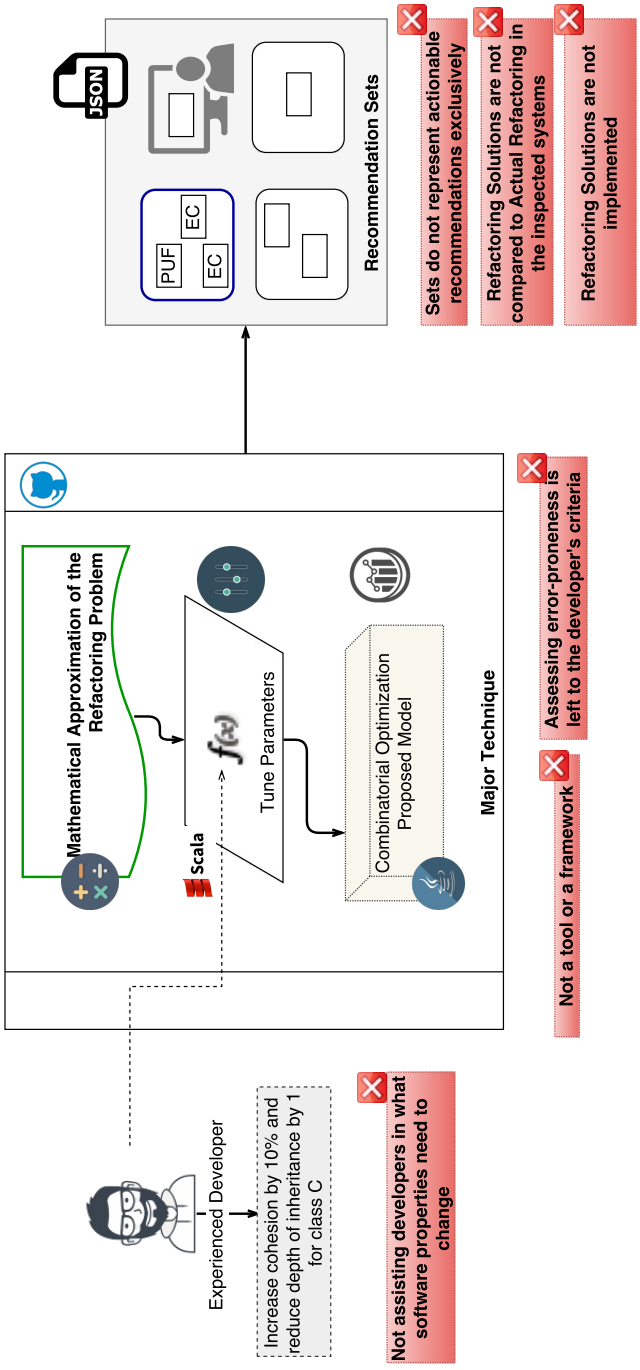


FIGURE 7.1. Limitations

APPENDIX A

Extended Methodology

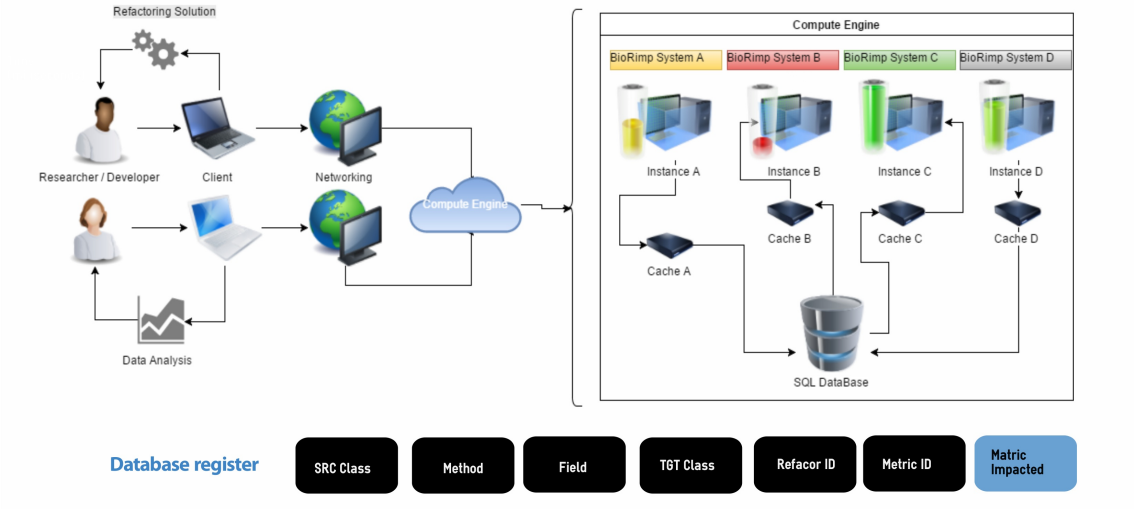


FIGURE A.1. Deployment

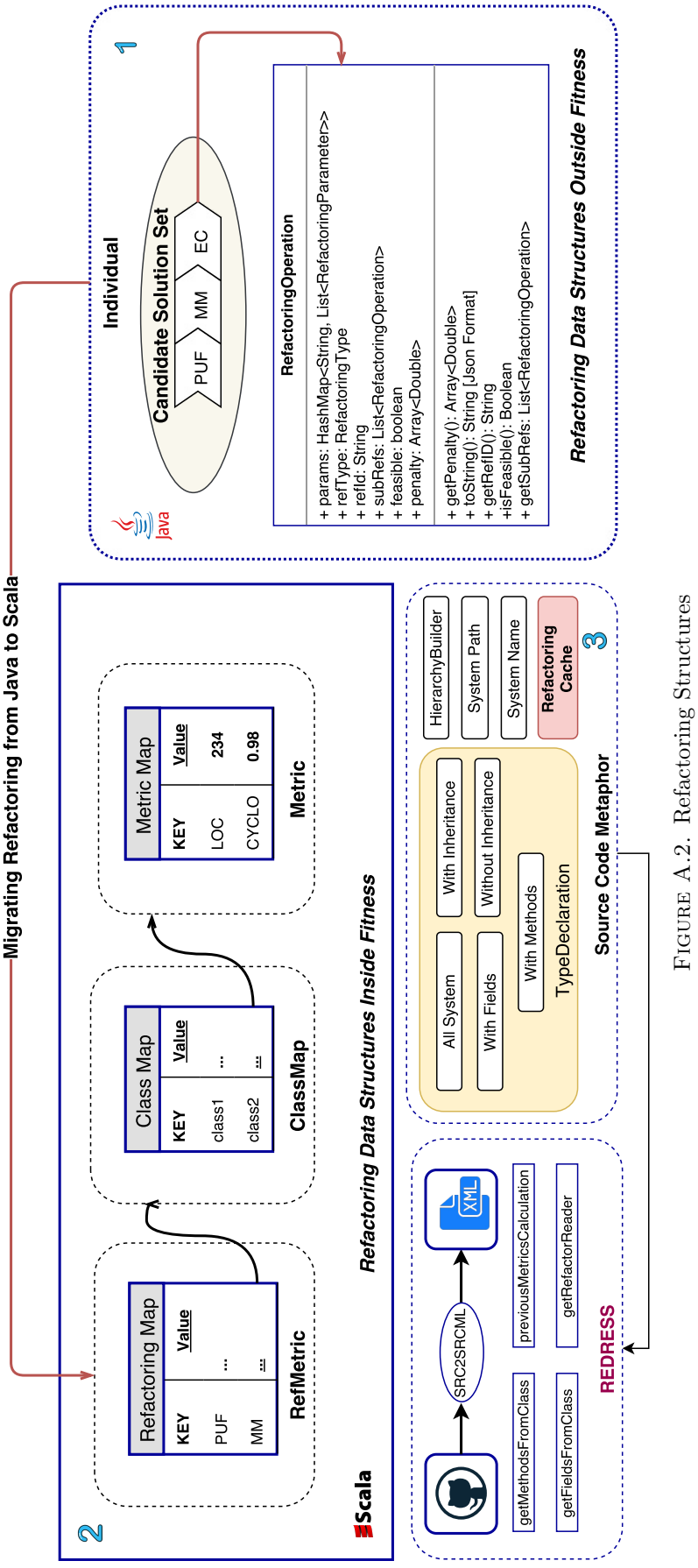


FIGURE A.2. Refactoring Structures

APPENDIX B

Extended Results

TABLE B.1. Time Complexity Exploratory Analysis for Baseline Algorithms in Five Thousand Evaluations

JFreeChart Dataset	Hill climbing			Simulated Annealing	
	#Classes	mean(sec)	SD(sec)	mean(sec)	SD(sec)
DAT01	11	465.92	414.04	488.20	599.69
DAT02	22	1806.63	1322.17	1715.41	1405.44
DAT03	34	1637.56	1579.98	1510.84	1092.08
DAT04	48	3514.10	938.54	4051.86	1703.34
DAT05	60	2430.10	1215.08	2594.07	977.54
DAT06	71	2816.08	1121.70	3006.93	955.58

TABLE B.2. Time Complexity Exploratory Analysis for SA vs HaEa

JFreeChart Dataset	Simulated Annealing			HaEa	
	#Classes	mean(sec)	SD(sec)	mean(sec)	SD(sec)
DAT01	11	488.20	599.69	330.95	226.34
DAT02	22	1715.41	1405.44	1198.45	605.53
DAT03	34	1510.84	1092.08	1147.93	694.25
DAT04	48	4051.86	1703.34	2781.87	980.02
DAT05	60	2594.07	977.54	1667.78	595.10
DAT06	71	3006.93	955.58	1719.71	775.82

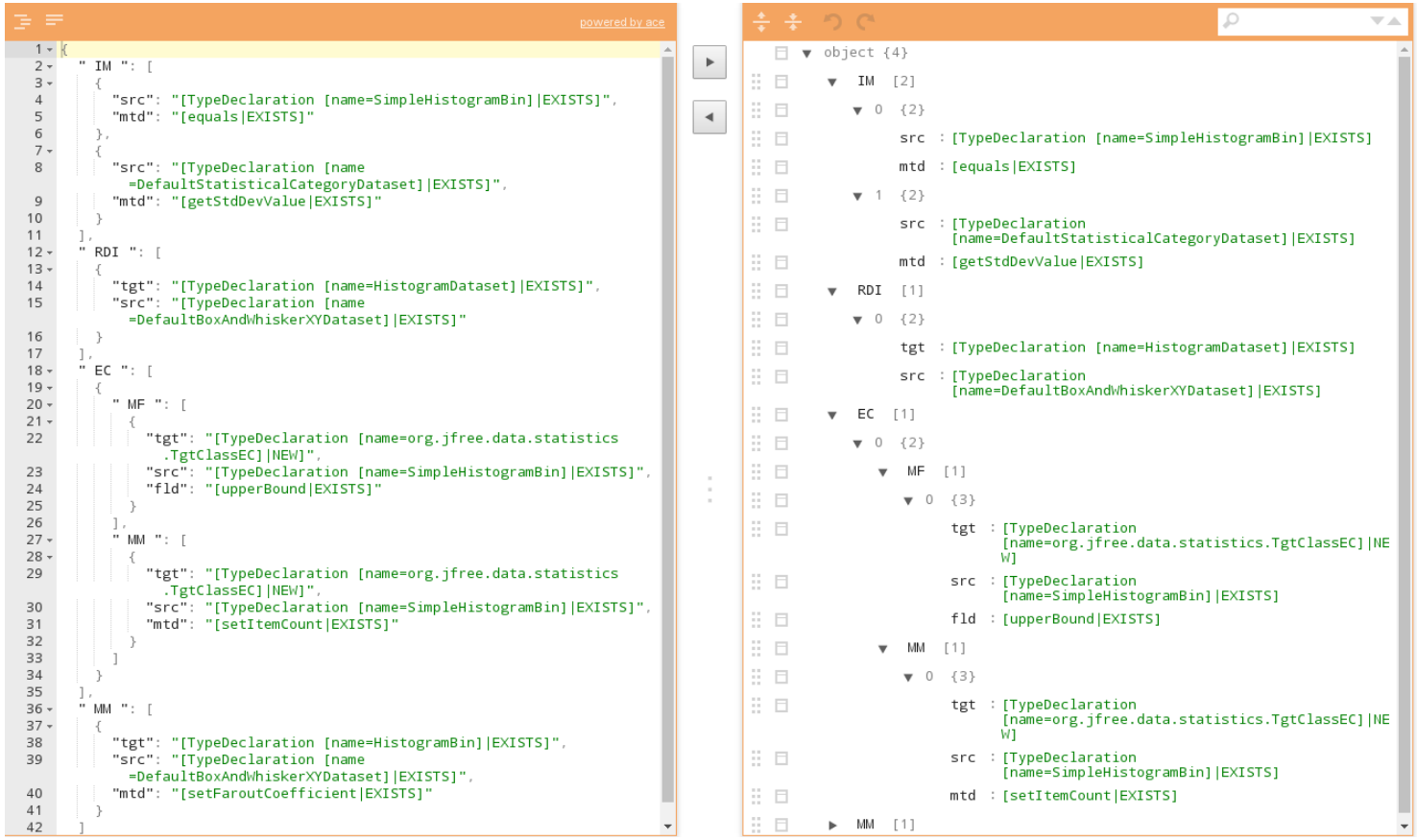


FIGURE B.1. JfreeChart System Output Json File for Time Complexity Experimentation

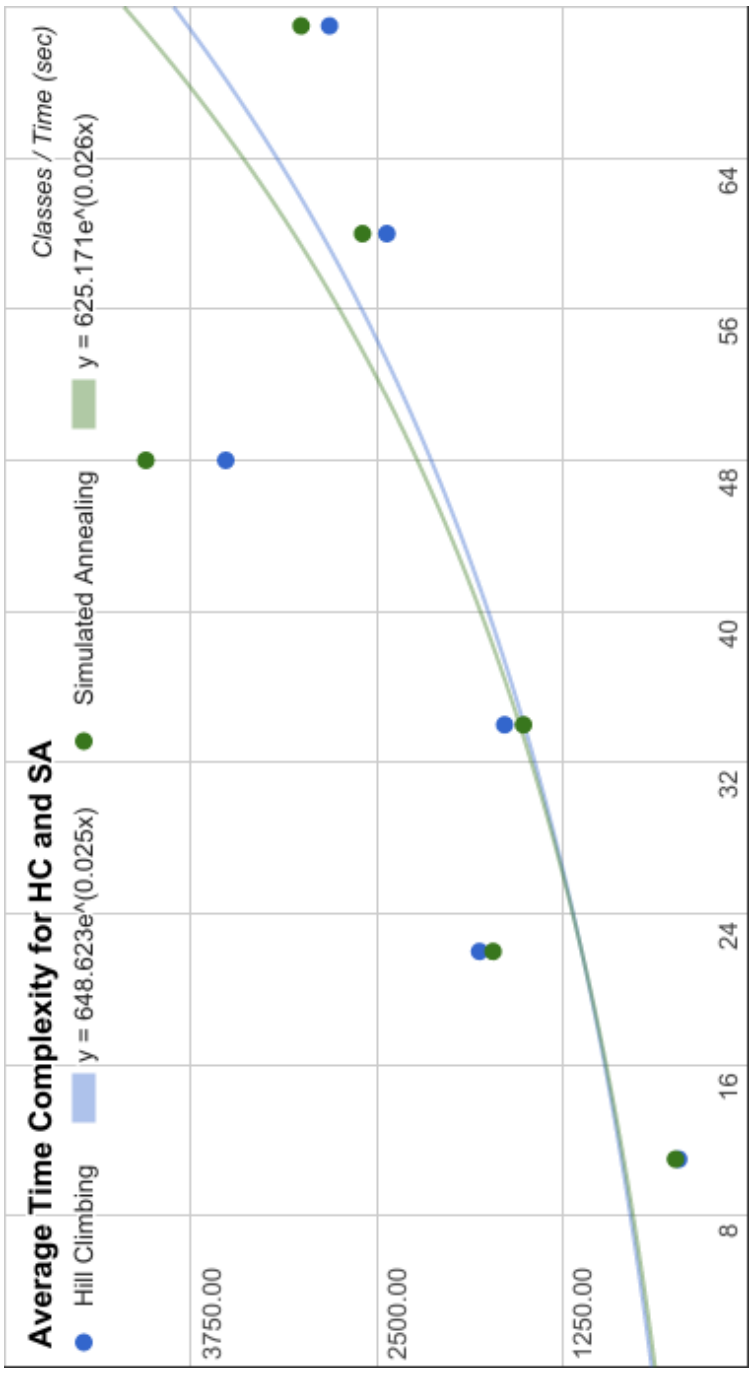


FIGURE B.2. Average Time Complexity Hill Climbing and Simulated Annealing. In blue, the data points. In light blue, the trendline with the exponential model. Y-axis is the time in seconds and X-axis is the number of classes for JFreeChart System.

Conclusions

This document proposed a systematic formal approach to estimate artificial refactoring recommendations based on quality metrics, baseline search techniques and evolutionary optimization. Fitness values are a general representation of the quality impact of refactorings operations sets. In this research, the author did not consider bad-smells detection or any kind of design information for generating the refactoring recommendations: no semantic preservation or maintenance history of the software is included in the study.

The major contributions to the state of the art from this study are: a mathematical approximation to the refactoring problem, this allows the research community to maintain a formalism when dealing with refactoring automation; the generation of the refactorings as a combinatorial problem, this allows software researchers to reproduce any optimization algorithm; and, a design and validation of search techniques through a modeling and empirical study that allow to explore the search space in an efficient manner. The results of our research not only confirm good performance of evolutionary computational approaches but also encloses an effective formalization of the refactoring problem.

Threats to validity include an experimentation for applying the refactorings manually. The author did not measure the error-proneness impact on the code when applying the estimated refactorings and the actionability of those refactorings. In addition, the proposed approach estimated a set of refactorings (not a sequence); thus, the refactoring operations do not configure any sequence that allows an order in the refactorings implementation. The estimated refactoring sets exhibit structural consistency, although error-proneness measures are not taking into account for feasibility constraints.

Lastly, this research fulfills four major aspects:

1. The author defined a unified mathematical approximation for several software refactorings concepts by means of theory of computation and set theory.

2. By using the proposed formalization, the author achieved to model the problem of software refactoring generation as a combinatorial approach.

3. The research presented a design for computational techniques that includes baseline (Hill Climbing & Simulated Annealing) and evolutionary (HaEa) algorithms to solve the software refactoring problem as a combinatorial model.

4. The research introduced an empirical study to validate the performance and time complexity of the estimated artificial refactoring sets with previous designed search-techniques.

Take-home message

It is proposed a systematic formal approach to estimate artificial refactoring recommendations based on quality metrics and baseline search techniques. Our computational solution not only confirms a convergence of baseline algorithms and an exponential time complexity but also encloses an effective formalization of the SRP. The thesis contributed to reducing the gap between the software defect prediction and the software refactoring by establishing the fundamentals of the automation for the refactoring process in a unified mathematical theory toward a proposed combinatorial optimization model, which is able to receive error-proneness information. Our document propounded a definition, an implementation, and an evaluation of a systematic approach that recommends a set of artificial refactoring operations.

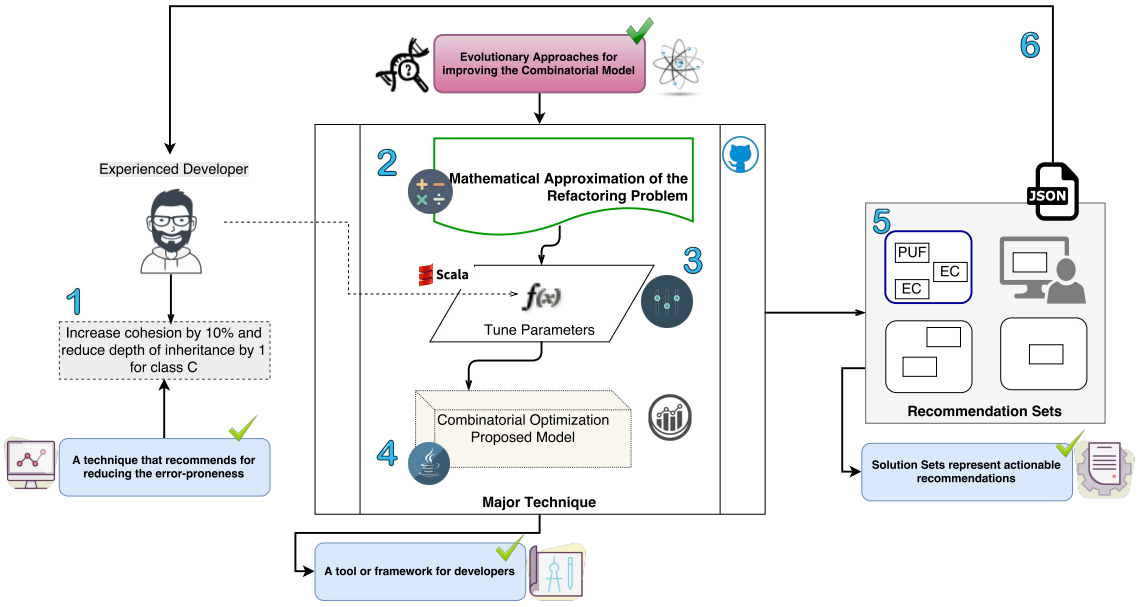
Future Work

.1 Overview

Figure .3 depicts the future work of this research. Four major topics are important in order to extend the proposed approach: (1) a technique that recommends for reducing the error-proneness; (2) a tool or framework for developers; (3) solution sets that represent actionable recommendations; and, evolutionary approaches or heuristics that improve the results of the combinatorial model.

The author suggest to research on how the approach automatically adjusts the fitness function weights in order to reduce the error-proneness (for a given SUA or code fragment). That approach of error-proneness reduction might take into account the design of the system, bad code smells or any other approach that has been already applied directly to the construction of refactoring sequences.

FIGURE .3. Future Work



Four major topics are important in order to extend ARGen: 1) a technique that recommends for reducing the error-proneness; 2) a tool or framework that developers can easily manipulate; 3) other refactoring data structures that simplify the search (e.g., graph

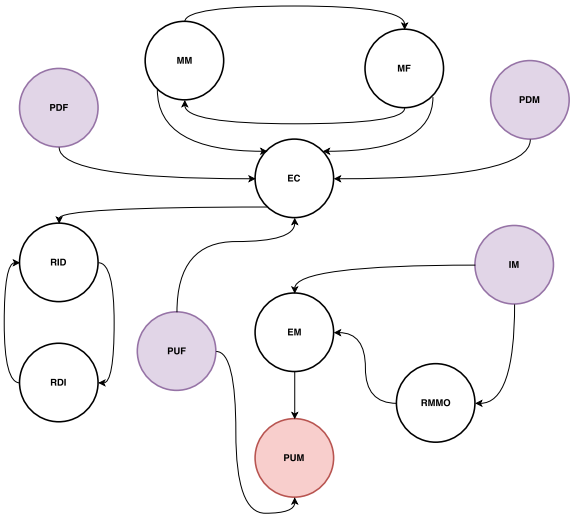


FIGURE .4. Archipelago Organization for Twelve Refactoring Operations

representation); and, 4) evolutionary approaches or heuristics that improve the results of the proposed combinatorial model. I plan to research on how our approach automatically adjusts the objective function weights to reduce the error-proneness (for a given SUA or code fragment). Based on our reproducibility analysis, formalizations, and results; to generate refactoring recommendations relies on human factor; therefore, since the Software Refactoring Problem exhibits a human design process, I expect my model to be extended using concepts from *cognitive sciences* and *artificial life*.

.2 What about the "Refactoring Consistency" ?

The refactoring operations are highly dependent of the context and developers' experience. Therefore, for the generation of refactorings, nobody can establish a general rule or absolute recommendation. Comparing refactoring approaches results depend on specific settings and several design and quality objectives. That is the reason why accuracy does not make sense between proposed refactorings and actual refactorings of an evaluated system. Then, How does one evaluate the consistency of the reported approach?. Research community needs an alternative form of measuring consistency of refactorings. The author introduces some insights in this chapter.

The Refactoring Consistency is based on archipelago [55] study. The relations of each refactorings proposed were converted into a directed graph (Figure .4) which shows the hierarchy of implementation of each refactoring.

The author proposed a refactoring consistency metric. The Refactoring Consistency Metric (RCM) is defined as:

$$RCM = \frac{\pi}{\Pi + \frac{\lambda}{\Lambda} + \Theta}$$

Where is the π number of nodes that actually could be manually implemented. Π is the total number of vertexes in the solution graph. Λ is the number of design patterns detected and λ is the number of design violations of those patterns. Θ is the number

of possible structural violations against OOP style. The idea is to calculate the metric according to archipelago graph order.

Refactoring Consistency Research Question

To what extent the refactoring consistency of proposed refactorings is preserved?

Bibliography

[1] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba, *An experimental investigation on the innate relationship between quality and refactoring*, Journal of Systems and Software (2015).

[2] Andrew P. Black, *Object-oriented programming: Some history, and challenges for the next fifty years*, Information and Computation **231** (2013), 3–20.

[3] Barry Boehm and V.R. Basili, *Top 10 list [software development]*, Computer (2001).

[4] Murphy Brendan, *The difficulties of building generic reliability models for software*, Empirical Software Engineering (2012).

[5] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta, *On the Impact of Refactoring Operations on Code Quality Metrics*, 2014 IEEE International Conference on Software Maintenance and Evolution, 2014.

[6] C Conceição, G Carneiro, and F Abreu, *Streamlining Code Smells: Using Collective Intelligence and Visualization Carlos*, International Conference on the Quality of Information and Communications Technology Streamlining, 2014.

[7] Bogdan Dit, Evan Moritz, Mario Linares-Vasquez, and Denys Poshyvanyk, *Supporting and Accelerating Reproducible Research in Software Maintenance Using TraceLab Component Library*, 2013 IEEE International Conference on Software Maintenance (2013), 330–339.

[8] Martin Fowler and Kent Beck, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[9] Xi Ge, Quinton L Dubose, and Emerson Murphy-Hill, *Reconciling Manual and Automatic Refactoring*, 2012 34th International Conference on Software Engineering (ICSE) (Zurich), IEEE, 2012, pp. 211 – 221.

[10] M Harman and L Tratt, *Pareto Optimal Search Based Refactoring at the Design Level*, Proceeding of the Genetic and Evolutionary Computation Conference GECCO 2007, 2007.

[11] Adam C Jensen and Betty H C Cheng, *On the use of genetic programming for automated refactoring and the introduction of design patterns*, Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10) (2010), 1341–1348.

[12] Jonatan Gomez, *Self adaptation in evolutionary algorithms*, GECCO 2004, LNCS, Universidad Nacional de Colombia, 2004, p. 12.

[13] Mark O Keeffe and Mel Ó Cinnéide, *Search-Based Refactoring : an empirical study*, Journal of Software Maintenance and Evolution: Research and Practice (2007), no. August 2007, 1–7.

[14] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni, *Design defects detection and correction by example*, IEEE International Conference on Program Comprehension, 2011.

[15] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni, *A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection*, IEEE Transactions on Software Engineering (2014).

[16] Hurevren Kilic, Ekin Koc, and Ibrahim Cereci, *Search-Based Parallel Refactoring Using Population-Based Direct Approaches*, Search Based Software Engineering **6956** (2011), 271–272.

[17] Thainá Mariani and Silvia Regina Vergilio, *A systematic review on search-based refactoring*, Information and Software Technology **83** (2017), 14–34.

[18] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp, *Refactoring: Current research and future trends*, Electronic Notes in Theoretical Computer Science **82** (2003), no. 3, 483–499.

[19] Tom Mens and Tom Tourwé, *A survey of software refactoring*, IEEE Transactions on Software Engineering **30** (2004), no. 2, 126–139.

[20] Tim Menzies, Jeremy Greenwald, and Art Frank, *Data Mining Static Code Attributes to Learn Defect Predictors*, IEEE Transactions on Software Engineering **33** (2007), no. 1, 2–14.

[21] Tim Menzies and Thomas Zimmermann, *Goldfish bowl panel: Software development analytics*, Proceedings - International Conference on Software Engineering, 2012.

[22] Mohamed Wiem Mkaouer and Marouane Kessentini, *On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach*, 2015, pp. 1–43.

[23] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide, *A robust multi-objective approach for software refactoring under uncertainty*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2014.

[24] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide, *Recommendation system for software refactoring using innovization and interactive dynamic optimization*, Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14, 2014.

[25] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb, *On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach*, 2015, pp. 1–43.

[26] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb, *A robust multi-objective approach to balance severity and importance of refactoring opportunities*, Empirical Software Engineering (2016), 1–34.

[27] Wiem Mkaouer, Marouane Kessentini, Kalyanmoy Deb, and Mel Ó Cinnéide, *High Dimensional Search-based Software Engineering : Finding Tradeoffs A mong 15 Objectives for Automating Software Refactoring Using NSGA-III*, Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 1263–1270.

[28] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni, *Many-Objective Software Remodularization Using NSGA-III*, ACM Transactions on Software Engineering and Methodology **24** (2015), no. 3.

[29] Iman Hemati Moghadam and Mel Ó Cinnéide, *Automated refactoring using design differencing*, Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR (2012), 43–52.

[30] Brendan Murphy, *The difficulties of building generic reliability models for software*, Empirical Software Engineering (2012).

[31] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam, *Experimental assessment of software metrics using automated refactoring*, Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12 (2012), 49.

[32] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam, *Experimental assessment of software metrics using automated refactoring*, Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12 (2012), 49.

[33] Martin Odersky and Tiark Rompf, *Scala unifies traditionally disparate programming-language philosophies to develop new components and component systems.*, COMMUNICATIONS OF THE ACM **57** (2014), 76–86.

[34] Mark O'Keeffe and Mel Ó Cinnéide, *A Stochastic Approach to Automated Design Improvement*, Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ '03) (2003), 59–62.

[35] Mark O'Keeffe and Mel Ó Cinnéide, *Search-based refactoring for software maintenance*, Journal of Systems and Software **81** (2008), no. November 2006, 502–516.

[36] Ratapong W Ongpiang, *Selecting Sequence of Refactoring Techniques Usage for Code Changing Using Greedy Algorithm*, Electronics Information and Emergency Communication (ICEIEC), 2013 IEEE 4th International Conference on, IEEE, 2013, p. 5.

[37] William F. Opdyke, *Refactoring Object-Oriented Frameworks*, 1992.

[38] Ali Ouni, Marouane Kessentini, Slim Bechikh, and Houari Sahraoui, *Prioritizing code-smells correction tasks using chemical reaction optimization*, Software Quality Journal (2015).

[39] Ali Ouni, Marouane Kessentini, and Houari Sahraoui, *Search-based refactoring using recorded code changes*, Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR (2013), 221–230.

[40] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum, *Maintainability defects detection and correction: A multi-objective approach*, Automated Software Engineering **20** (2013), 47–79.

[41] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Hamdi, *The use of development history in software refactoring using a multi-objective evolutionary algorithm*, Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13 (2013), 1461.

[42] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi, *Search-based refactoring: Towards semantics preservation*, IEEE International Conference on Software Maintenance, ICSM (2012), 347–356.

[43] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi, *Improving multi-objective code-smells correction using development history*, Journal of Systems and Software **105** (2015), 18–39.

[44] Eduardo Gurgel Pinho and Francisco Heron De Carvalho, *An object-oriented parallel programming language for distributed-memory parallel computing platforms*, Science of Computer Programming **80** (2014), no. PART A, 65–90.

[45] Fawad Qayum and Reiko Heckel, *Local search-based refactoring as graph transformation*, Proceedings - 1st International Symposium on Search Based Software Engineering, SSBSE 2009 (2009), 43–46.

[46] Danijel Radjenovi, Marjan Heričko, Richard Torkar, and Aleš Ž Ivkovič, *Software fault prediction metrics: A systematic literature review*, (2013).

[47] Santosh Singh Rathore and Atul Gupta, *Validating the effectiveness of object-oriented metrics over multiple releases for predicting fault proneness*, Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 2012.

[48] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente, *Recommending Move Method refactorings using dependency sets*, Proceedings - Working Conference on Reverse Engineering, WCRE, 2013, pp. 232–241.

[49] Baldoino Fonseca Dos Santos Neto, Márcio Ribeiro, Viviane Torres Da Silva, Christiano Braga, Carlos José Pereira De Lucena, and Evandro De Barros Costa, *AutoRefactoring: A platform to build refactoring agents*, Expert Systems with Applications (2015).

[50] Olaf Seng, Johannes Stammel, and David Burkhart, *Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems Categories and Subject Descriptors*, GECCO 2006 - Genetic and Evolutionary Computation Conference (2006), 1909–1916.

[51] Danilo Silva, Minas Gerais, Nikolaos Tsantalis, and Marco Tulio Valente, *Why We Refactor ? Confessions of GitHub Contributors*, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, 2016, pp. 858–870.

[52] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou, *JDeodorant: Identification and removal of type-checking bad smells*, Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2008, pp. 329–331.

[53] A M Turing, *Computing Machinery and Intelligence*, Mind **49** (1950), 433–460.

[54] Kwaku Yeboah-Antwi and Benoit Baudry, *Online Genetic Improvement on the java virtual machine with ECSELR*, Genetic Programming and Evolvable Machines (2016).

[55] A.V. Zarras, T. Vartziotis, and P. Vassiliadis, *Navigating through the archipelago of refactorings*, 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings, 2015.