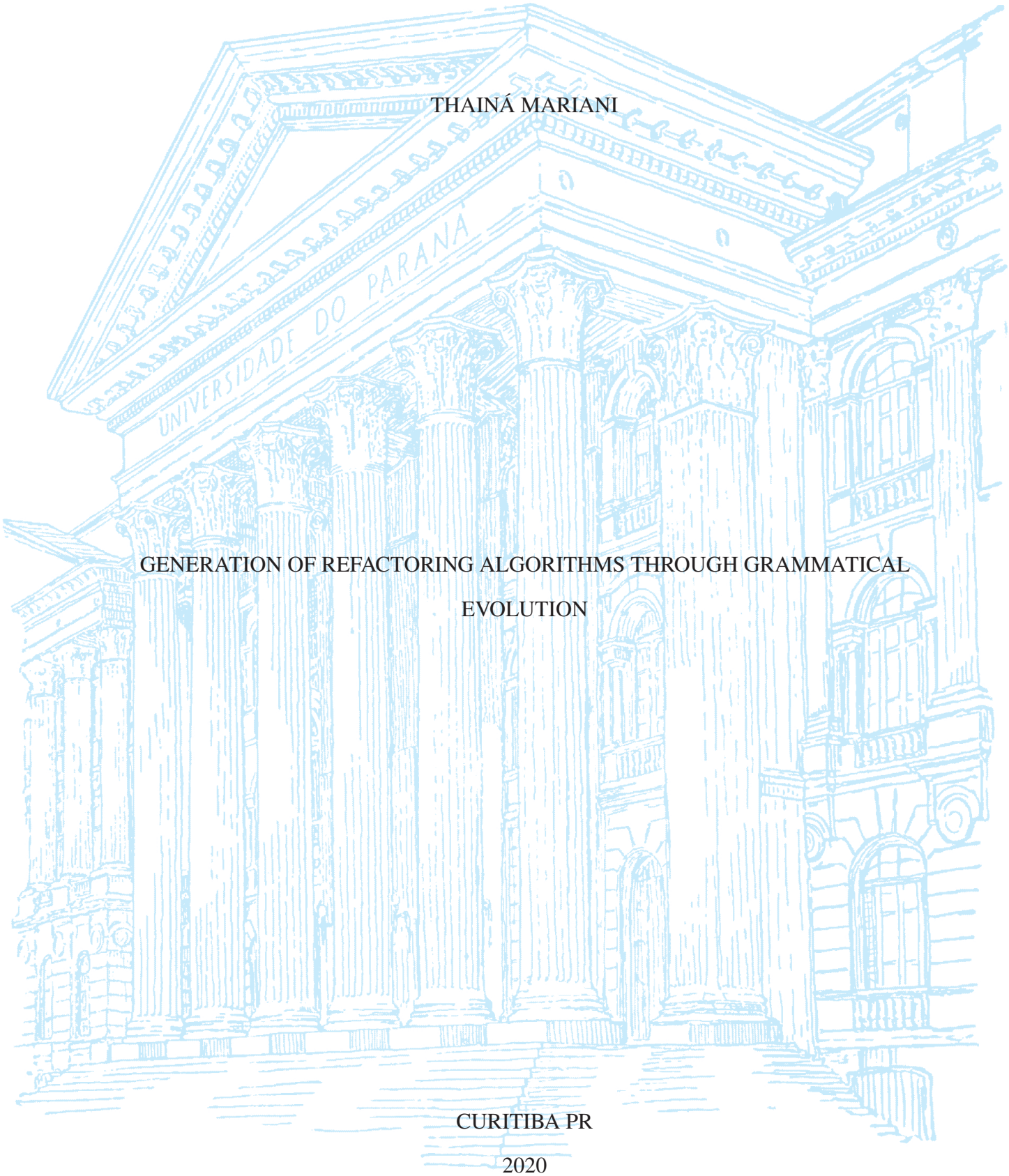
UNIVERSIDADE FEDERAL DO PARANÁ

THAINÁ MARIANI



GENERATION OF REFACTORING ALGORITHMS THROUGH GRAMMATICAL EVOLUTION

CURITIBA PR

2020

THAINÁ MARIANI

GENERATION OF REFACTORING ALGORITHMS THROUGH GRAMMATICAL EVOLUTION

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Silvia Regina Vergilio.

Coorientador: Marouane Kessentini.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

M333g Mariani, Thainá
Generation of refactoring algorithms through grammatical evolution [recurso eletrônico] / Thainá Mariani. – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Silvia Regina Vergilio – Coorientador: Marouane Kessentini - Coorientador

1. Software – Refatoração. 2. Engenharia de Software. 3. Ferramentas de Busca. 4. Análise por agrupamento. I. Universidade Federal do Paraná. II. Vergilio, Silvia Regina. III. Kessentini, Marouane. IV. Título.

CDD: 005.14

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **THAINÁ MARIANI** intitulada: **GENERATION OF REFACTORING ALGORITHMS THROUGH GRAMMATICAL EVOLUTION**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 24 de Abril de 2020.

SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

CELSO GONÇALVES CAMILO JUNIOR
Avaliador Externo (UNIVERSIDADE FEDERAL DE GOIÁS)

AURORA TRINIDAD RAMIREZ POZO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

PLÍNIO DE SÁ LEITÃO JÚNIOR
Avaliador Externo (UNIVERSIDADE FEDERAL DE GOIÁS)

MAROUANE KESSENTINI
Coorientador (UNIVERSITY OF MICHIGAN-DEARBORN)

MAURICIO ANICHE
Avaliador Externo (UNIVERSITY OF TECHNOLOGY - TU DELFT)

Marouane Kessentini
Digitally signed by Marouane Kessentini
DN: cn=Marouane Kessentini, o, ou, email=marouane@umich.edu, c=US
Date: 2020.04.26 19:38:21 -04'00'

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

To my mother...

ACKNOWLEDGMENTS

Firstly, I would like to thank my mother for all love and support, and for always encourage my education. Moreover, I would like to express my gratitude to my brother and sister, who were extremely important, very friendly and helpful, specially in the last year of my PhD.

My sincere thanks to Professor Silvia, who guided me with excellence and never measured effort to help her students. I also would like to thank all professors who had cross my way and taught me different things, specially Professor Aurora who was of great importance during this journey. I am very grateful to Professor Marouane who gave me the opportunity to conduct my internship in University of Michigan, and to Thiago, who have joined me in this international adventure.

My infinite gratitude to my group of sweet friends, called "Bebê Daltônico"(daltonic baby in English, a long story), with Alane, Alissar, Ana, Cristina and Marcela. We met a little bit late but our friendship was essential to help me deal with daily struggles and to give me very special and funny moments.

My thanks to Vitor Hugo and Daiane who have been helping me with mental health issues. I am privileged for having such capable and humanized professionals guiding me in this journey. You both were of primordial importance to the conclusion of this work.

Thanks to all other current and ex-members of cbio-gres lab for all the help (specially with hardware problems), great moments, jokes, lunches, coffees, Hayashidas, long talks and many other things. I will fondly remember my days in the lab. Finally, my gratitude to all other friends and family members who were by my side during all these years. I am grateful for you all.

RESUMO

A atividade de refatoração tem como principal objetivo aplicar um conjunto de transformações em um artefato de software para melhorar sua estrutura sem alterar sua funcionalidade. Alguns estudos recentes, apresentam bons resultados ao gerarem modelos de predição de refatorações. Além disso, os estudos mostram que refatorações similares são aplicadas em diferentes contextos e podem ser aprendidas. Neste sentido, a maioria dos trabalhos existentes utiliza técnicas de aprendizado de máquina para gerar modelos que predizem se um dado trecho de código deve ser refatorado. Entretanto, essas abordagens possuem limitações. Elas buscam por refatorações específicas e exatamente como aplicadas por desenvolvedores, o que limita que outras refatorações sejam encontradas. Dada a natureza subjetiva da atividade de refatoração de software, a exploração por refatorações com base em outros critérios também é vantajosa. Existem trabalhos na área conhecida como Refatoração de Software Baseada em Busca (SBR) (do inglês, *Search Based Software Refactoring*), em que algoritmos de busca são utilizados para encontrar refatorações em um grande espaço de busca e visando a melhorar diversos aspectos. Recentemente, trabalhos em SBR começaram a utilizar exemplos de refatorações já aplicadas por desenvolvedores para incorporar aprendizado na busca. Entretanto, essas abordagens são limitadas em termos de generalização dos resultados, uma vez que não geram um modelo que possa ser utilizado para diferentes programas. Desse modo, abordagens existentes de SBR devem ser configuradas e executadas a cada novo programa. Neste contexto, este trabalho visa a incorporar os benefícios encontrados na área de aprendizado de máquina e na área de SBR, apresentando uma abordagem chamada Gorgeous (do inglês, ***G**eneration **o**f **R**efactoring Algorithms through **G**rammatical **E**volution*). Gorgeous tem como objetivo gerar algoritmos de refatoração compostos por regras, que quando executados, determinam trechos de código que devem ser refatorados e refatorações a serem aplicadas. Os algoritmos são criados de forma que as refatorações sugeridas sejam similares a refatorações aplicadas na prática e que também melhorem a qualidade do software. Os algoritmos são criados utilizando um processo de aprendizado que primeiro extrai padrões de refatoração de programas agrupando elementos que foram refatorados de maneira similar. Após isso, uma evolução gramatical é executada para gerar algoritmos de refatoração com base nos padrões extraídos. Gorgeous é avaliada utilizando dados de refatoração extraídos de 40 programas Java do repositório *GitHub*. Como resultado, os algoritmos gerados foram capazes de obter bons resultados para diferentes programas, melhorando em média 60% a qualidade do programa e obtendo 50% de similaridade com refatorações aplicadas na prática.

Palavras-chave: Refatoração, Engenharia de Software Baseada em Busca, Agrupamento, Evolução Gramatical

ABSTRACT

The refactoring activity addresses the application of a set of transformations in software artifacts to improve their structure while preserving their functionality. Recent studies present promising results generating prediction models for refactoring. Furthermore, they provide evidences that similar refactoring operations are applied in different contexts and they can be learned using Machine Learning (ML). Most works on ML based refactoring generate models to predict if a piece of code should be refactored. Despite the capability of prediction, existing works are limited to learn specific refactoring operations as applied by developers. However, to explore refactoring operations possibilities based on other criteria is also beneficial, mainly by the subjective context of refactoring. In this context, the Search-Based Software Refactoring (SBR) area addresses studies using search algorithms to find refactoring operations in a huge search space, aiming at improving several other aspects. However, existing SBR approaches do not support generalization of results since they do not generate a model as ML studies. In this way, a SBR approach needs to be configured and executed for each program in need of refactoring. In this context, this work introduces a SBR learning approach aiming at taking most advantage of both fields. Gorgeous (**G**eneration **o**f **R**efactoring Algorithms through **G**rammatical **E**volution) generates refactoring algorithms composed by several rules determining pieces of code that should be refactored and the refactoring types to be used. A refactoring algorithm provides as solution a set of refactoring operations to be applied in a program. In this respect, the algorithm is generated with the goal of increasing similarity of the refactoring operations with the ones applied in practice, and also improving program quality. To do this, a learning process first extracts refactoring patterns from programs by grouping their elements that were refactored in similar ways. After that, a Grammatical Evolution (GE) is executed to generate the algorithms based on the extracted patterns. Gorgeous is evaluated using refactoring data from 40 *Java* programs of *GitHub* repository. The refactoring algorithms are capable of obtaining good results to different programs, obtaining around 60% of program quality improvement and 50% of similarity with real refactoring applications.

Keywords: Refactoring, Search-Based Software Engineering, Clustering, Grammatical Evolution

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The software refactoring activity [4] is used to change a software artifact structure without modifying its external behavior. It is performed with the goal of improving the software quality in respect to some quality attributes, such as understandability, modularity, and maintainability. It is mainly used in the software development and maintenance phases to keep and reestablish the quality of code elements, but it is also used in other early software engineering phases, such as software design and re-engineering [5].

The main task of software refactoring is the application of meaning-preserving restructurings, called refactorings [2], which are simple operations performed to change a software, such as to move or to extract a method. A refactoring operation has two important components: a refactoring type, such as *move method* or *extract method*; and a set of actors, which represents the code elements involved in the operation, e.g., a class or a method from a software. They were originally proposed in the context of object-oriented code [4], where some catalogs exist [2, 4]. Since then, they have been extended to different contexts and artifacts, but object-oriented code still remains the main focus of refactoring.

Software refactoring is an expensive and error-prone activity [6, 7]. Specially, software maintenance activities consume up to 70% of the total cost of a typical software project [8]. Researchers also point out developers spend at least 10% of their monthly hours with refactoring tasks [9, 10]. The task of identifying a good refactoring type for each situation is very consuming itself. There are several refactoring types which can be used and specially hundreds of code elements that might be in need of refactoring. In this context, the refactoring problem is highly studied in the literature [11, 12]. It consists of automatically finding a good set of refactoring operations for a software program.

Some tools were proposed in the literature to help in this task [13, 14, 15, 16]. Some of them are integrated into most of existing IDEs such as Eclipse, NetBeans, IntelliJ, and Visual Studio. They are mainly focused on the identification of refactoring operations capable of fixing code smells [2] or duplicate code, but they do not take into account other important aspects, such as the improvement of quality attributes. Despite the existence of tools, most refactoring work is still performed manually. A survey conducted with 328 software engineers of Microsoft [10] pointed out around 80% of the developers refactoring tasks are manual. It also indicated design defects are not the main reason why developers apply refactoring operations. They most see benefit by improving quality attributes, such as readability, maintainability, and modularity.

In this context, some refactoring works have been proposed in the Search-Based Software Refactoring (SBR) field [11]. SBR works apply search-based techniques to help the software refactoring activity. Studies in this field have gained visibility because they are capable to identify refactoring operations for a program by optimizing several quality factors [11]. Actually, SBR have been pointed out as the most beneficial approach for the software refactoring problem [12]. However, existing SBR approaches do not provide reusable solutions, i.e., the approach needs to be configured and executed for every new version or program. It can demand an unnecessary effort from the developer, who also might not be familiar with a search algorithm.

In this work, we start from the observation that we can learn refactoring patterns from various software programs, and provide a more reusable solution for the refactoring problem. We consider a refactoring pattern as a similar refactoring operation found in different places, which can be, i.e., different versions or programs. A recent study [17] using deep learning shows only between 21% and 36% of code changes can be automatically learned. However, refactoring

was classified as one of the learned code changes. Moreover, the learning with multiple software programs increases accuracy around 10%. This shows an evidence that refactoring patterns can be learned from different programs.

Recently, literature has shown some approaches incorporating machine learning techniques to predict refactoring operations from software programs. In this kind of study, the reusable solution is a prediction model generated either using a regression or a classification technique. The approaches are restricted to predict elements in need of refactoring [18, 19] or to predict if an element should receive a predefined refactoring type [20, 18]. In fact, existing machine learning based approaches are able to obtain good results in terms of prediction, but they lack in the number of refactoring types possibilities.

Additionally, existing machine learning approaches are limited to the prediction of refactoring operations. Due to the subjective nature of the refactoring problem, it is not possible to guarantee if a refactoring operation is good, so the labeling of a refactoring operation itself could lead to different interpretations. Due to this fact, our work addresses the learning of refactoring patterns to guide us during the search for solutions, but not as a unique criteria. We want to explore the search space to find other solutions which might be also good in quality aspects. To support this, however, we will have to extend current state-of-the-art machine learning techniques to make them applicable to this context.

In this sense, our approach incorporates a Grammatical Evolution (GE) [21] technique. GE is a type of Genetic Programming (GP) [22], since it is similarly used to evolve programs [21]. However, while a conventional GP algorithm uses a tree as representation for an individual and applies search operators to those trees [22], a GE algorithm manipulates an array of integers and evolves the solutions similarly to a conventional Evolutionary Algorithm (EA) [21] since it performs the same steps and applies conventional search operators . Moreover, in addition to the usual steps of an EA, a GE receives a grammar file, usually in Backus Normal Form (BNF), to map each solution into a program. The grammar is very flexible and supports the mapping of several aspects of a program.

1.1 MOTIVATIONS

With respect to the presented context, this work presents a SBR learning approach to the refactoring problem. It automatically generates algorithms used to find refactoring operations for object-oriented programs. In this sense, the motivations that justify this work are:

1. Refactoring of object-oriented programs is highly used and important to improve and keep the quality of a program;

2. The refactoring problem is a hard problem that demands a lot of effort if performed manually;

3. Existing refactoring tools are not very used and focus most on the correction of bad smells;

4. Existing SBR approaches are capable of optimizing many software quality attributes, but they are not able to provide a reusable solution;

5. Refactoring operations are one of the few code changes proved to be automatically learned;

6. Learning code changes from multiple programs can help to improve accuracy results;

7. ML based refactoring approaches can provide good accuracy results, but they are restricted to prediction and do not explore solutions to improve other quality aspects.

8. GE presents promising results in the literature to generate different kinds of algorithms. In addition, it has a flexible grammar that is easy to compose, change, extend and understand.

1.2 GOAL OF THIS WORK

The main goal of this work is to generate algorithms capable of suggesting for a program refactoring operations able to improve quality, and to increase similarity with real refactoring applications. In this sense, we introduce in this work the concept of refactoring algorithm. A refactoring algorithm receives as input a program and produces for it a set of refactoring operations. This algorithm is composed by several procedures, each of them defining rules used to find the refactoring operations.

Based on that, the specific goals of this work are:

- To provide refactoring algorithms that can be generalized to find refactoring operations to Java programs;
- To provide refactoring algorithms capable of suggesting refactoring operations that optimize quality by improving modularity.
- To provide refactoring algorithms capable of suggesting refactoring operations similar to real ones applied by developers.

To achieve such goals, this work proposes Gorgeous (**G**eneration **o**f **R**efactoring Algorithms through **G**rammatical **E**volution), which is a SBR learning approach to the refactoring problem. Our approach provides a refactoring algorithm as a reusable solution. The refactoring algorithm receives as input a *Java* program information and produce for it a set of refactoring operations. This algorithm is composed by several procedures, each of them defining rules used to create the refactoring operations.

Gorgeous is composed by three steps: 1) Instantiating Programs; 2) Extracting Patterns; and 3) Generating Algorithms. The first step receives information from software programs and instantiates them based on a predefined representation. The second step is in charge of learning refactoring patterns from the set of program instances. It executes a clustering algorithm [23] to group classes and methods that were refactored in similar ways. Each generated cluster represents a refactoring pattern that serves as input to the next step. In this way, the next step generates a refactoring algorithm based on each cluster. Each refactoring algorithm incorporates the characteristics of the corresponding cluster.

1.3 HYPOTHESIS

The hypothesis of this work is: "Gorgeous is capable of generating refactoring algorithms able to find refactoring operations similar to real ones, while bringing improvement in terms of quality".

In order to validate this hypothesis, Gorgeous is implemented and validated in a set of 40 Java programs extracted from real software repositories. Java was chosen since it is one of the most employed programming languages. Furthermore, we selected the Expectation Maximization clustering algorithm to be used, since it is a popular and used clustering algorithm of the literature [23].

1.4 CONTRIBUTIONS

The contributions of this work are summarized as follows:

- This work introduces the concept of refactoring algorithm, as well as a grammar which formalizes a set of rules identifying where and how refactorings should be applied. In this way, GE supports the generation of more complex algorithms when compared with solutions generated by traditional machine learning techniques.
- Gorgeous supports the learning of refactoring data from several programs automatically using GE and clustering algorithm. It allows flexibility to deal with different complex operations.
- This work reports evaluation results obtained from 40 open source Java programs extracted from *GitbHub*.
- The results provide evidence to support the claim that our proposal is able to generate refactoring algorithms that can be used to identify refactoring operations to several programs improving quality and similarity with real refactoring applications.

1.5 ORGANIZATION

This work is organized as follows:

Chapter 2 - Background: This chapter describes the main concepts needed to understand this work. It presents concepts related to evolutionary algorithms, clustering algorithms, software refactoring, and the SBR field.

Chapter 3 - Related Work: This chapter presents related work, which are related to the following subjects: refactoring tools, SBR and Machine Learning based refactoring.

Chapter 4 - Proposed Approach: This chapter describes the structure and functionality of the proposed approach.

Chapter 5 - Empirical Evaluation: This chapter presents details about the conducted experiments and the obtained results.

Chapter 6 - Final Remarks: This chapter concludes this work and present some future directions.

Appendix A: This appendix presents a systematic review we conducted in SBR [11], which was published in the Information and Software Technology journal.

2 BACKGROUND

This chapter reviews the main concepts needed to understand this work. Section 2.1 describes the software refactoring concepts and main tasks. Section 2.2 overviews the structure and functionality of SBR approaches. Section 2.3 presents the main characteristics of evolutionary algorithms, given particular emphasis on Grammatical Evolution (GE), used in this work. Section 2.4 reviews the Machine Learning (ML) fields, including supervised and unsupervised learning, and giving special attention for cluster analysis. Finally, Section 2.5 concludes this chapter by relating the presented concepts with this work.

2.1 SOFTWARE REFACTORING

The term refactoring was introduced in 1990 by Opdyke and Johnson [4]. They proposed a set of meaning-preserving restructurings to be applied in C++ programs. Each of these restructurings was called a refactoring [24]. The term was popularized by Fowler [2] after the publication of his book. Since then, the term has also been used to denote the whole process of changing a software artifact and, gained different meanings and definitions. In this work, we distinguish both meanings according to Fowler's definitions [2], by using the terms refactoring and software refactoring, as below:

> **Refactoring:** "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [2].

> **Software Refactoring:** "The activity of software restructuring by applying a set of refactorings without changing its observable behavior" [2].

The main goal of the software refactoring activity is to improve the quality of software artifacts, since it tends to decay over the time [2]. In this sense, many quality attributes can be improved, such as understandability, reusability, flexibility, and modularity. Indeed, understandability is a key point of software refactoring, since by making a program more understandable many other benefits can be achieved. In summary, by having a better understandability of a software artifact may be easy to find problems, such as bugs and duplicate elements. Furthermore, it can also help to obtain a faster development process, which may speed up the software delivery. In addition to this, to add new functionalities may be easy to the system, since the engineer will have a better understandability of the artifact [2].

The software refactoring activity can be used in several software engineering phases. The most common one is the software maintenance phase, where the lack of software structure is more evident and expensive. In this sense, it is possible to dedicate a time in the maintenance phase only to perform the software refactoring activity. In addition, the software refactoring activity is also used in other phases, such as re-engineering, design and development [25, 5]. For instance, in the software design phase, software refactoring can be used to improve a preliminary design, helping to obtain a final one [26].

In the development phase, some software development methods use software refactoring as part of the process to improve the program, such as Test-Driven Development and Agile Software Development [5]. To support refactorings in the development phase, some popular tools of the literature, such as NetBeans and Eclipse, provide semi-automatic support to apply

refactoring operations. In general, the software refactoring activity includes six main tasks [5] described next.

1. **Identify where the software should be refactored.** The first step of this task is to define the artifact to be refactored. Examples of artifacts are codes, models, and requirements. The second step is the identification of actors, which are the elements that should receive a refactoring application. There are different directions to identify the actors. One of them may be recognized when a meaningful effort is being wasted to maintain and understand an artifact.

2. **Determine which refactorings should be applied to the identified places.** In this task, it is necessary to determine the refactoring types that should be applied to the actors identified in the previous task. Usually, these two tasks are coupled and can be performed at the same time. Refactoring catalogs might be used to guide this task. The most popular and widely used refactoring catalogs are for C++ and Java programs [25, 2]. However, there are catalogs for other type of artifacts, such as models [27, 28], software product lines [29], databases [30], and HTML [31].

3. **Guarantee that the applied refactoring preserves behavior.** Define methods to guarantee behavior preservation of the software artifact. The original definition of behavior preservation states the output values should be the same before and after a refactoring when using the same set of inputs.

4. **Apply the refactoring.** This task consists of applying the defined refactorings in the identified actors using the established methods to guarantee behavior preservation.

5. **Assess the effect of refactoring on quality characteristics of the software.** This task refers to the assessment of the impact of the applied refactorings on software quality attributes. It can be performed by analyzing manually the impact on the attributes from the point of view of the user. Furthermore, it can also be assessed using software metrics to measure different aspects related to quality attributes.

6. **Maintain the consistency between the refactored artifact and other software artifacts.** A software is usually associated with many artifacts, such as code, design models, and tests. In this sense, this task refers to the use of mechanisms to maintain the consistency of the refactored artifact (usually code) and the other software artifacts.

The first two tasks together are responsible for defining the refactoring operations. Each of these is associated with a set of actors and the refactoring type that should be applied on them. Table 2.1 describes some popular refactoring types used in the object-oriented context. Each type is associated with the corresponding actor elements and their roles in a refactoring operation, e.g., in a move method refactoring operation, a method should be moved from a *source class* to a *target class*.

In this sense, there exists the refactoring problem [32]. It consists of finding a good set of refactoring operations for a given software artifact. It is a NP-complete problem since the search space of possible solutions is extensive. Usually, this problem is explored in the context of object-oriented code. The set of refactoring operations can be evaluated based on different points of view and using different metrics as stated by the 5*th* task.

Tabela 2.1: Refactoring types and actors (Adapted from [2]).

Acronym	Type	Description	Actor	Role
MM	Move Method	Move a method from a class to another	*method*	-
			class	source class
			class	target class
PU	Pull Up Method	Move a method from a subclass to its superclass	*method*	-
			class	source class
			class	target class
PD	Push Down Method	Move a method from a superclass to one of their subclasses	*method*	-
			class	source class
			class	target class
EM	Extract Method	Extract a method into another one	*method*	-
			class	-
IM	Inline Method	Merge two methods of a class	$method_1$	-
			$method_2$	-
			class	-
EC	Extract Class	Extract a class into another one	*class*	-

2.2 SEARCH-BASED SOFTWARE REFACTORING

Search-Based Software Engineering (SBSE) [33] addresses the application of search algorithms to solve difficult software engineering problems. Search algorithms can be applied to different areas of software engineering, such as testing [34, 35, 36], requirements [37], design [26] and refactoring. In this way, the area that applies search-based techniques to perform software refactoring is called Search-Based Software Refactoring (SBR).

Existing approaches commonly use search techniques to suggest or apply refactoring operations in artifacts. Figure 2.1 shows a generic overview of them.

This kind of approach has as input the artifact to be improved. It is converted to a representation used by the search technique. This representation can be the artifact itself or a more abstract representation, such as trees for representing code artifacts. The search technique can optionally receive as input refactorings, metrics and any other additional information to guide the search process. As output, the search technique returns a solution (or a set of solutions) for the problem, which is usually the refactoring problem.

A solution representation is also needed by the search technique. An example of it is a vector where each position represents a refactoring and its associated actors. The provided metrics are used in the fitness function to evaluate the quality of the solutions. In relation to the additional information, it may be given to help in the optimization process, for example, an instance of a good refactoring application can help the approach in searching for similar solutions.

2.3 EVOLUTIONARY ALGORITHMS

Some optimization problems are difficult to solve since an optimal solution can not be found in a polynomial time using a deterministic algorithm [1]. In this sense, meta-heuristics are

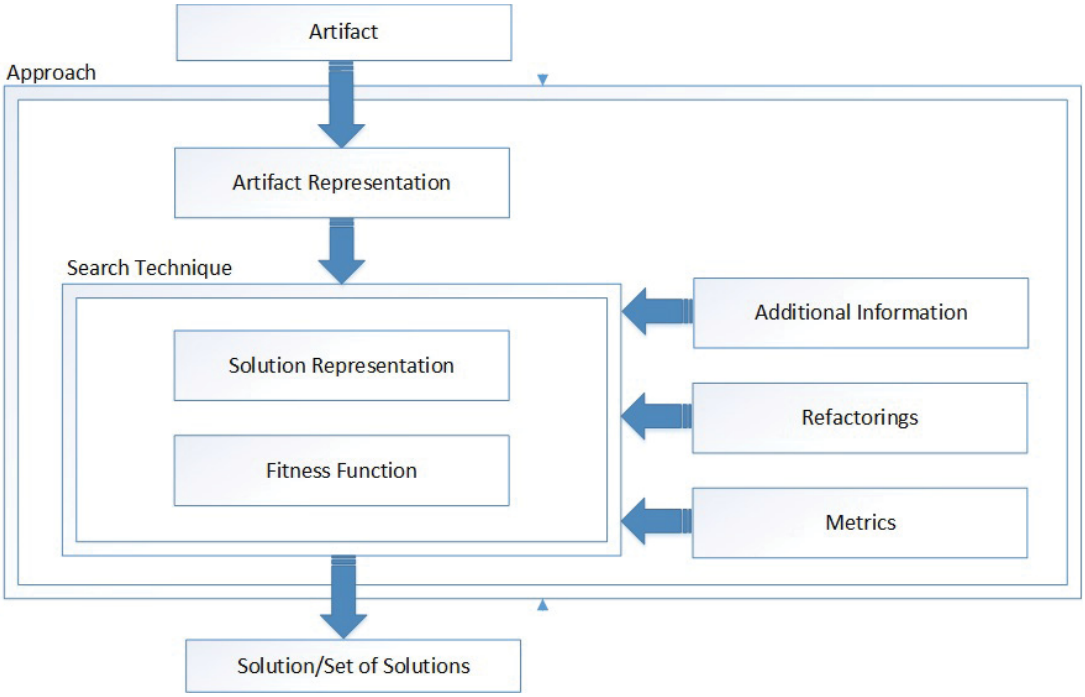


Figura 2.1: Generic overview of SBR approaches

capable of generating suboptimal solutions for a problem. That means they can not guarantee the optimal solution, but they can obtain acceptable solutions in a reasonable time [1]. A type of meta-heuristic are the Evolutionary Algorithms (EA). There is a wide range of EAs in the literature, such as Genetic Algorithms (GA), Evolutionary Programming, Differential Evolution and Genetic Programming (GP).

EAs are based on the evolution of species and use the main concepts of generation and reproduction. A population is composed of multiple individuals manipulated through the search process. Each individual is called a chromosome and encodes a possible solution for the problem. EAs also use the concept of genotype and phenotype. In this sense, the phenotype is the solution, and the genotype is its encoding. Figure 2.2 illustrates how a generation works. At each generation, individuals, called parents, are selected to be reproduced by applying crossover and mutation operators. The crossover operator combines the parents in order to generate off-springs and the mutation operator introduces transformations in the chromosome in order to bring genetic diversity. A fitness function is defined to measure the quality of each generated solution. At the end of the generation, a replacement strategy is used to determine which individuals, being parents or offspring, will compose the next generation. This process is performed until a stopping criterion is achieved [1].

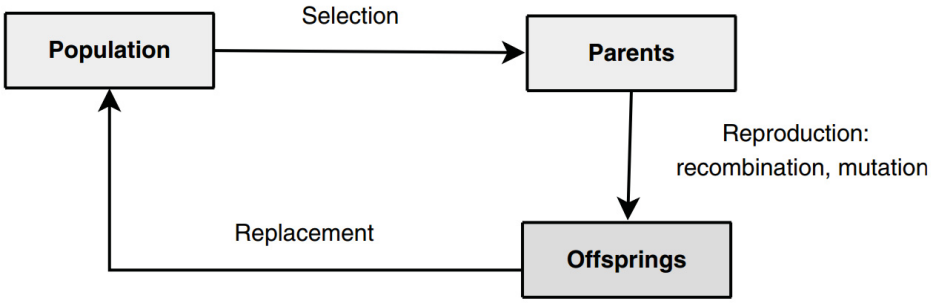


Figura 2.2: A generation of an Evolutionary Algorithm [1]

2.3.1 Genetic Programming

Genetic Programming (GP) [22] is a type of EA used to evolve programs. GP allows the automatic generation of programs that can be further used to solve a given problem. The representation used by GP to encode a program is usually a tree. Figure 2.3 shows a tree-based representation of the expression $(x * 5) + (z + (x * y)) + y$. The variables and constants (x, y, z and 5) are leaves called *terminals*, while the arithmetical operations ($+$ and $*$) are internal nodes called *functions* [1].

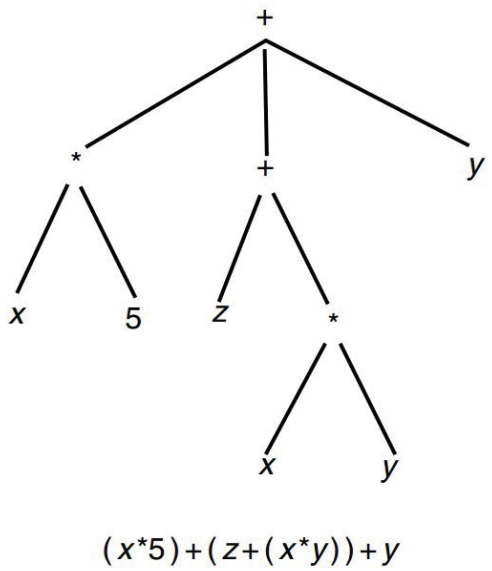


Figura 2.3: Tree-based representation used in GP [1]

GP starts with an initial population of randomly generated programs. After that, two individuals are selected to be reproduced by applying crossover and mutation operators. The most common crossover operator is *subtree crossover*. It works by randomly selecting a *crossover point* (node) in the selected parents. Thus, it generates the off-springs by replacing the subtree rooted at the *crossover point* of the first parent by the subtree rooted at the *crossover point* of the second parent, and vice versa. Similarly, the most common GP mutation operator, called *subtree mutation*, selects a *mutation point* in the tree and replaces the subtree rooted by a subtree randomly generated [38]. The next steps of the evolution are similar to the ones used in conventional EAs. At the end of the search, the best program is returned, according to the defined objective function.

2.3.2 Grammatical Evolution

A Grammatical Evolution (GE) algorithm is a type of GP, since it is similarly used to evolve programs [21]. However, while a conventional GP algorithm typically uses a tree as representation for an individual and applies search operators to those trees [22], a GE algorithm uses an array of integers or bits and evolves the solutions similarly to a conventional EA [21]. Moreover, in addition to the usual parameters of an EA, a GE receives a grammar file, usually in Backus Normal Form (BNF), to map each solution into a program. This mapping is called *genotype-phenotype mapping*. The evolution is applied to the array (genotype level), but to calculate the fitness function value, the program (phenotype level) needs to be executed. (GPM) [39]. Therefore, the GPM procedure is needed by the GE algorithm to transform each array into an executable program [39].

The common solution representation used by the GE algorithms is an array of integers or bits. If an array of bits is used, it is first mapped to an array of integers and, then, to a program. Most of times, an array of integers is used directly. In any case, the GE algorithm reads the grammar file and learns the grammatical rules by assigning each values of the array to the corresponding rule. To illustrate this, Figure 2.4 shows a BNF grammar used to evolve simple mathematical expressions.

$\langle expr \rangle$ ::= $\langle var \rangle$ | $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$

$\langle var \rangle$ ::= x | y

$\langle op \rangle$ ::= $*$ | / | + | −

Figura 2.4: Example of grammar for mathematical expressions

The items between ⟨ ⟩ are non-terminal rules, | represents the logical operator OR, ::= means the rule can take any of the next options, and the remaining items are terminal nodes. For instance, the rule $\langle var \rangle$ can take either the value x or y when mapped to a program. On the other hand, $\langle expr \rangle$ can take the value of a single $\langle var \rangle$ or a composition of $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$. The choice between each option is given by the genes of the chromosome of each individual (array of integers). An example of a choice is presented next.

By taking into consideration the following individual {4, 9, 79}, the value of the first gene in the chromosome (4) and the first rule of the grammar $\langle expr \rangle$ are selected. Thus, the number of options in $\langle expr \rangle$ is counted. In that case, there are two options: $\langle var \rangle$ or $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$. Then, the modulo operation 4%2 is performed taking into consideration the gene value and the number of options. The result of this operation represents the gene position, since it is 0, the first rule $\langle var \rangle$ is selected. In the next step, the rule $\langle var \rangle$ and the second gene of the chromosome (9) are selected. There are two options for $\langle var \rangle$: x or y. Then, the modulo operation 9%2 returns 1. It selects the terminal node y, which is then the expression obtained as final result. The last chromosome (79) is discarded.

The employed integer array commonly has a variable size. Because of that, in addition to crossover and mutation operators applied in the same way as an EA, the GE algorithm employs two distinguishable search operators: i) gene duplication operator; and ii) gene pruning operator. They help the algorithm to eliminate useless genes or reinsert new genes into the chromosomes. Hence, the duplication operator selects a random sub-array of the chromosome and copies it to the end of the chromosome. The prune operator, on the other hand, selects an index to truncate the array. These operators are usually applied with the same probability as the mutation operator and as additional steps in the evolution process [21].

Summarizing, Algorithm 2.1 presents the pseudo-code of a conventional GE algorithm. As the algorithm shows, it is very similar to an traditional EA, replacing the evaluation of the population by GPM, and adding duplication and prune operators.

2.4 MACHINE LEARNING

Machine Learning (ML) is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data [40]. In other words, ML algorithms learn to recognize complex patterns and make intelligent decisions based on them.

Algorithm 2.1: Pseudocode of a GE algorithm

```
1  Input: GF – Grammar File;
2  begin
3      population ← Initialize the population;
4      programs ← Map population to programs using GF;
5      Execute programs;
6      Assign a fitness value to the solutions of population according to the output of their respective
         programs;
7      while stop condition is not achieved do
8          matingPopulation ← Select parents;
9          offspring ← Recombine matingPopulation;
10         Apply gene prune operator to the solutions of offspring;
11         Apply gene duplication operator to the solutions of offspring;
12         Apply mutation operator to the solutions of offspring;
13         programs ← Map offspring to programs using GF;
14         Execute programs;
15         Assign a fitness value to the solutions of offspring according to the output of their
             respective programs;
16         population ← Perform replacement;
17     end
18     return Best program of population;
19 end
```

The process to learn from data usually implies the algorithm must generalize and build a model that will be used in the future to produce a useful output with new data.

Machine Learning methods are classified according to the characteristics of the used data. Supervised learning uses labeled data. Unsupervised learning learns from data without label. The first type of learning is the most used [40]. In Software Engineering, supervised ML methods are used to classify, predict or estimate: software quality, software size, software development cost, software effort, fault proneness, refactoring, etc.

Unsupervised methods are used to dimension reduction by pre-processing data and to cluster analysis [41, 42, 43]. The last is a field devoted to the study of techniques able to group data instances based on their own features. Hence, the objects within a group should be similar to the others of the same group and different from the objects of other groups. Each group is called a cluster, and a collection of them is called a clustering [23].

Clustering algorithms typically operate over two types of structures. First, a matrix n-by-m where n represents the number of data instances and m represents the number of attributes. The second structure is a n-by-n matrix representing the proximities for each pair of instances. Proximities are calculated based on different distance metrics, such as Euclidean and Manhattan distances [23].

There are several clustering algorithms available in the literature, but there is not an algorithm which is better to all cases. In this sense, they are divided in some categories. First, partitional and hierarchical are the most known and employed types of clustering algorithms [23]. The partitional clustering is a simple division of clusters in a way that each object is present in only one cluster. On the other hand, the hierarchical clustering uses the concept of subcluster. Hence, it is organized as a tree where each cluster is the union of its subclusters, and the root of the tree is a cluster containing all the objects.

There is a wide range of clustering algorithms in the literature, each of them exploring different types of clustering and clusters. Next sections describe two popular ones.

2.4.1 K-means

A popular and widely used one is K-means [44]. It is a prototype-based and partitional clustering algorithm that defines a prototype in terms of a centroid (average of an attribute in a cluster) [23]. K-means is usually applied in a continuous n-dimensional space.

The procedure executed by K-means is presented on Algorithm 2.2. Firstly, the algorithm initializes K centroids (Line 3), where K is the number of desired clusters defined by the user. Then, each point is assigned to the closest centroid establishing the clusters (Line 5). Finally, the centroid of each cluster is recomputed (Line 6). These steps are repeated until no centroid is changed.

Algorithm 2.2: Pseudocode of K-means [23]

```
1 Input: K
2 begin
3     Select K points as initial centroids;
4     repeat
5         Form K clusters by assigning each point to its closest centroid;
6         Recompute the centroid of each cluster;
7     until Centroids do not change;
8 end
```

K-means uses a proximity function to measure the distance of each point to the centroids, in order to find the closest one. The most used functions are Euclidean Distance, Manhattan, and Cosine, but there are others available for different kinds of data [23]. In addition, an objective function is used by K-means to mathematically define the goal of the clustering. Usually, the goal is to minimize the Euclidean Distance between an object and the centroid of its cluster.

2.4.2 Expectation Maximization

Another popular algorithm used for clustering is called Expectation Maximization (EM) [45]. EM is originally a parameter estimator statistical method that have been also applied in the clustering field. EM performs similar to K-means, but it can be classified as a fuzzy clustering since it uses a probabilistic model to give a probability of each object to belong to each cluster [23].

EM generates a mixture distribution for the whole population of objects. A mixture distribution is composed by several different individual distributions. The parameters are estimated using Maximum Likelihood Estimate (MLE). It estimates several parameters values, such as mean and variance, with the goal of maximizing the probabilities of the objects to belong to any individual distribution. In this way, each cluster is represented by an individual distribution and its parameter values represent the patterns for a cluster [45]. EM is divided in two steps: E-Step (expectation) and M-Step (maximization). Algorithm 2.3 presents the pseudocode for EM.

Initially, the parameters are unknown, then EM estimates the parameters and try to guess objects that are more likely to fit the distribution. E-step (Line 4) calculates the expectation of the probabilities distribution of the objects. M-step (Line 5) recalculates the parameters of the individual distributions aiming at maximizing the expected probabilities. Both steps continue until convergence (parameters are not changing) or until a prefixed number of interactions is achieved [45].

Algorithm 2.3: Pseudocode of Expectation Maximization Algorithm [23]

```
1 begin
2     Select initial set of parameters;
3     repeat
4         E-step: Compute probabilities P for each object under each distribution;
5         M-step: MLE estimates new parameter values for each distribution considering P.
6     until Parameters do not change;
7 end
```

2.5 CONCLUDING REMARKS

This chapter presented the topics related to this work, including the main concepts related to GE, clustering, and software refactoring. Moreover, it also described the refactoring problem and the structure of SBR approaches usually used to solve it.

This work is addressed in the field of SBR by using GE to automate three tasks of the software refactoring activity: 1) to identify where the software should be refactored; 2) to determine which refactorings should be applied to the identified places; and 5) to assess the effect of the refactoring on quality characteristics of the software. In this respect, refactoring algorithms are generated with the goal of finding a set of solutions for the refactoring problem. A refactoring solution is a refactoring operation found for a given program.

This is performed by incorporating a two-step learning process that first uses a clustering algorithm to learn refactoring patterns by grouping code elements that were refactored similarly. Then, each cluster represents a pattern that is learned by a grammatical evolution algorithm in order to generate the refactoring algorithms. The next chapter describes works that are related to the concepts and fields described in this section.

3 RELATED WORK

This chapter describes related work. Section 3.1 starts by presenting state-of-the-art software refactoring tools. Section 3.2 describes works in the field of SBR based on a systematic review we conducted. Section 3.3 describes the works found in the context of ML based refactoring, considering both unsupervised and supervised learning. Finally, Section 3.4 concludes this chapter by presenting the limitations and main differences among related work and our work.

3.1 SOFTWARE REFACTORING TOOLS

Some popular IDEs of the literature, such as IntelliJ IDEA[1], Eclipse[2], and Netbeans[3], support semi-automatic refactorings by applying some refactorings selected by the user. Furthermore, the first refactoring tools of the literature were proposed with the same goal. The first to be proposed was *Refactoring Browser* [46], a tool for Smalltalk language. Furthermore, other semi-automatic tools were proposed for other languages, such as *JRefactory* [47] for Java, and *XRefactory* [48] for C++ and Java. More related to our work, there are tools able to suggest refactoring solutions for several programs. Based on our definition, such tools can be classified as refactoring algorithms as well.

In this context, there are popular tools in the literature. Guru [13] is a tool to perform refactoring on programs developed using Self programming language. The main goal of Guru is to refactor inheritance hierarchies and methods, aiming at eliminating duplicate code. It works by automatically selecting a collection of classes connected by an inheritance hierarchy and replacing them by a new hierarchy. This is performed in such a way to avoid duplicate methods and preserve the behavior of the refactorings.

Most approaches were proposed with the goal of refactoring Java programs. AutoRefactor [14] is a tool implemented as an Eclipse Plugin to refactor Java programs. To apply the refactorings, one or a set of Java files should be selected and then, two refactoring options can be chosen: i) to automatically apply all the refactorings following its pre and post conditions; or ii) to select the refactorings to be automatically applied. The available refactorings have a set of systematic steps, such as to eliminate dead code, add brackets to control statements, and remove unnecessary expressions. When a refactoring is selected, all code fragments are inspected in order to identify places where the refactoring can be applied. Spartan Refactoring [15] is another tool used to find and correct fragments of Java. It is an Eclipse plugin used to review the code, provide and apply suggestions to make the code cleaner, shorter and more understandable.

The most popular tool is JDeodorant [16]. It is used to identify bad smells and eliminate them with refactoring applications. It is implemented as an Eclipse Plugin and it is applied to Java programs. The tool identifies the following bad smells: *Feature Envy*, *State Checking*, *Long Method* and *God Class*. JDeodorant uses different techniques to identify each bad smell, such as clustering algorithms to identify possible *God Class*. Thus, the following refactorings can be applied to solve these bad smells: *Move Method*, *Replace Conditional with Polymorphism*, *Replace Type code with State/Strategy*, *Extract Method* and *Extract Class*.

[1] https://www.jetbrains.com/idea/

[2] https://eclipse.org/

[3] https://netbeans.org/

Generally, the presented refactoring tools focus on the understandability of the program and on the elimination of bad smells. The main aspects explored are the elimination of duplicate code and dead code. However, such tools do not explore the improvement of other quality attributes, such as modularity.

3.2 SEARCH-BASED SOFTWARE REFACTORING

Search-Based Software Refactoring (SBR) [11] is the field devoted to the application of search-based algorithms to improve the software refactoring activity. In this context, we conducted a more rigorous search for papers as we present in a systematic review published in 2017 [11]. We have collected several papers and analyzed different aspects of the field, such as representations, metrics, search techniques, and evaluation methods. The approaches encompass many features and they present encouraging results, leading us to conclude there is sufficient evidence to make SBR a solid field. This section summarizes the results of the systematic review, which is presented in Appendix A.

Among several SBR studies, we highlight works more related to ours, which focus on the refactoring problem and that are guided by external refactoring information. In fact, existing SBR papers use examples to guide the optimization process [11]. Examples, in this context, are usually refactoring operations applied in previous versions of a program. In this sense, the search can be directed to find refactoring operations similar to the ones already applied in past versions. This kind of approaches commonly has as output a refactoring solution that was optimized aiming at increasing the similarity with examples.

Similarity measures [49] encompasses the number of times the current refactoring type was applied in past versions. Also, it focus on finding similar refactoring operations. A refactoring operation impacts more if it matches the refactoring type and all the actors of the current solution. The impact is smaller, but it is still considered, if a refactoring operation is composed by another type (e.g. to extract a class we have to move a method), and also if the same code fragments are actors. This metric is usually used in the fitness function in combination with other quality metrics, such as number of modifications [50, 51, 52], semantic coherence [50, 53, 49], and number of bad smells [54, 55]. We can found one work with a different output, which is a set of rules used to detect code smells and correct them [56].

Recent papers on SBR explore many-objective algorithms [57], and introduce a measure called recentness of a code element [58, 59]. It prioritizes, based on several previous versions of a program, elements that have been added recently. The idea behind such a metric is that new elements have more chances to have code smells or quality problems, while old elements probably have been revised more. This is also a way of introducing external knowledge.

Although SBR approaches present good results, they optimize a set of possible refactoring solutions individually for a specific program, leading to a lack of generality of the refactoring solutions. Furthermore, a certain level of expertise is required to configure and execute a search algorithm, which may not be trivial for a software engineer, specially if it has to be done several times for different programs. Our work encompasses a learning process to overcome these limitations. Next section presents some of the main contributions that incorporate learning mechanisms in the refactoring context.

3.3 MACHINE LEARNING BASED REFACTORING

In the field of machine learning, we have searched for works using unsupervised and supervised learning in the context of refactoring. Sections 3.3.1 and 3.3.2 reviews our main findings, most focusing on clustering and prediction modeling.

3.3.1 Unsupervised Learning

Many papers are found in the literature of unsupervised learning proposing the use of clustering algorithms in the refactoring context. The main focus of such papers is to restructure a program. Alkhalid et al. [60] present an approach to refactor programs at the method level. The goal is to find ill-structured and low-cohesive methods by calculating the similarity between the elements into a method. As an output, each cluster represents the code statements that should be extracted as method and grouped together. Czibula and Czibula [61, 62, 63, 64, 65, 66] propose an approach that extracts the relationship between different entities, such as classes, methods and attributes. A clustering algorithm is used to restructure the entities with the goal of improving the program. The new structure is compared with the first one to provide a list of refactorings able to improve the project.

Fokaefs et al. [67] present a clustering algorithm to recognize refactoring opportunities for extract class refactoring. Based on the dependency information among class members, it recognizes a set of data and behaviors that would improve the system if extracted into a new class. The candidates fragments to be refactored are ranked by priority based on cohesion and coupling metrics. In addition, we also found works using clustering with SBR techniques, in order to help in the process of reducing the number of feasible refactoring solutions [68].

Differently from the presented works, our approach uses a clustering algorithm to group elements that were refactored similarly. In this work, the main goal of the clustering algorithm is not to restructure a software or to identify refactoring opportunities, but to recognize patterns that can guide the generation of refactoring algorithms.

3.3.2 Supervised Learning

In supervised learning, most papers use classification techniques to generate a model to predict refactoring operations. Imazato et al. [69] propose a machine-learning approach to automatically predict methods to apply the extract method refactoring. The approach is based on the development history of a program, by analyzing the extract method refactorings applied in all project versions. The syntactic information of the methods is collected, which involves the number of appearances of each program element, such as statements, identifiers, symbols, and so on. This information is used in the learning process to build a prediction model. As a result, the prediction model is able to provide, for a given version of the system, a list of methods that are candidates to be refactored. Experiments were conducted using 5 Java projects, and the generated models are able to suggest refactorings with a high value of precision and recall.

Kosker et al. [70] present an approach to predict refactoring actors by using the Weighted Naïve Bayes classifier. The main goal is to predict which classes are in need of refactoring in order to decrease complexity, maintenance cost, and bad smells. The approach uses a set of 26 metrics as dimension of the classifiers, they include different aspects, such as cyclomatic complexity and lines of code. The experiments are conducted on three versions of a program. Based on that, the approach reveals the classes in need of refactorings.

Phongpaibul and Boehm [71] investigate different classification algorithms, such as decision trees and logistic model trees, to create prediction models to detect refactoring operations.

The approach reveals the elements that should be refactored, but it does not reveal the refactorings that should be applied. They collected data from different versions of the projects in order to extract several features, such as: growth measures, relationships between classes, the number of authors working in a code fragment, and so on. Jindal and Khurana [72] introduce the Refactoring Opportunity Factor (ROF), which predict the need of a whole module to be refactored. ROF can be high, medium or low. The metrics are manually collected based on a UML model drawing for each module.

Dallal [73] uses a logistic regression algorithm to predict classes in need of the extract subclass refactoring. Quality metrics related to different aspects are used, such as size, cohesion, and coupling. They developed an automatic tool to mutate a set of classes in different ways to obtain the classes in need of refactoring, for example, by merging a superclass and a class together. In this way, the original version of the program is considered the version after refactoring and the mutated version is considered the version before refactoring. The mentioned metrics are collected for both mutated and original version to create the prediction model. The expectation is that the mutated versions of the systems have classes in need of refactoring, and then, their quality is worse than the original version. The experiments were conducted using 6 Java projects and the results reveal a strong relation between the quality attributes of a class and its need of refactoring. As a result, the model can be used to determine the probability of a class to need the extract subclass refactoring.

Dallal et al. [20] introduce an approach capable of building models that are used to predict refactoring opportunities for the move method refactoring. The prediction models are building using logistic regression. Different metrics regarding cohesion and coupling were calculated for each element and such values were used in order to build the models. Several refactored classes are used in the learning process, such classes were refactored manually by the author. The experiments were performed on 7 Java projects by building several prediction models based on different metrics. The resulted models were capable of predicting correctly between 83.4% and 95.8% of the methods in need of the move method refactoring.

Xu et al. [19] propose a machine learning approach for extract method refactoring recommendation. The approach generates a probabilistic model which was built based on structural features related to complexity, and function features related to cohesion and coupling. In the learning process, the model learns from a set of positive and negative method extraction examples. Positive examples were extracted from open source repositories and by using data augmentation. In contrast, negative examples are randomly generated candidates. The approach works by suggesting, for a given method, all the candidates code fragments to be extracted with exception of the invalid ones. Then, the model gives a probability of this candidate to be a good option for extraction. The experiments were performed using 5 different Java projects, and the obtained results were able to outperform results from popular refactoring tools in terms of different metrics, such as precision and recall. Kumar et al. [18] investigates the use of different classifiers to predict methods in need of refactoring. The features used by the classifiers are 25 different metrics at the method-level. Results of 10 techniques are evaluated over a data set of 5 programs using 3 different sampling methods to deal with class imbalance. Results show accuracy around 98% for the two classifiers able to obtain the best results: AdaBoost and ANN+GD.

A more recent study [17] uses a deep learning approach, called Neural Machine Translation (NMT), to learn and apply code changes in a code. NMT is usually applied to linguistic translation problems. The main goal of the work is to replicate code changes exactly as they were originally. They use dataset with several methods extracted from 3 real software repositories. The work presents an extensive qualitative analysis over the corrected predictions,

in order to understand which kind of code changes NMT is able to learn. Indeed, the two main categories of code changes were: bug fix and refactoring. Although this approach does not generate a prediction model to identify or apply refactorings, the resulted model was capable of predicting several code changes later identified as refactorings. This is an important evidence that a refactoring can be automatically learned from software repositories.

The main limitation of [69, 70, 71, 72] is the lack of generality since they are limited to the learning across versions of a program instead of learning from several programs. In this sense, the models are not able to generalize the results in other programs. In respect to the other studies, they usually generate a model based on only one refactoring type. This is mainly due to the fact that most of them use binary classifiers, which are able to classify an element into one of two different categories. Generally, the approach determines if a code element should receive or not a predefined refactoring, e.g. move method in [20], or either that a module or element is a candidate to be refactored, such as presented by [18]. Other limitation of some works [73, 20] is the data used in the learning, since they use artificial data instead of real software programs. Moreover, works as [19, 17] use real software programs but they do not generate models to predict specifically refactorings.

3.4 CONCLUDING REMARKS

This chapter presented related work regarding existing software refactoring tools, SBR, and ML based refactoring. Several refactoring tools are available for refactoring of object-oriented programs, but they are limited to specific purposes, such as to correct code smells. On the other hand, SBR approaches obtained encouraging results by using several refactorings and by assessing the improvement of different aspects, but they are not capable of providing solutions for different programs. Also, they are not trivial algorithms to be simply configured and executed. In addition, machine learning works have been applied to learn refactoring applications and generate models able to predict refactoring operations. But, given the subjective nature of refactoring, prediction modeling can be restrictive since it is learning operations exactly as they were applied by developers. We believe one of the benefits of SBR approaches is a better exploration of the space of possibilities, which can result in different refactoring solutions that might be good in other aspects.

To overcome the limitations of related work mentioned above, and based on the good results obtained by SBR studies, we present a SBR based ML approach to the refactoring problem. It automatically generates refactoring algorithms by learning patterns from real software programs. Some popular characteristics of SBR works were employed in our approach, such as the use of object-oriented programs, similar metrics and refactoring types. The algorithm is generated aiming at improving similarity with real refactoring operations, as well as improving software quality. A cluster technique is also employed to guide the learning for algorithms. More details about our approach are presented in the next chapter.

4 PROPOSED APPROACH

This chapter presents our approach named *Gorgeous* (**G**eneration **o**f **R**efactoring Algorithms through **G**rammatical **E**volution). It addresses the refactoring problem by generating refactoring algorithms capable of finding a set of refactoring solutions for a program.

In this respect, Section 4.1 overviews Gorgeous by introducing each step of the approach. Sections 4.2 and 4.3 present, respectively, the input and output of Gorgeous. The following sections describe the steps performed to generate the algorithms. Sections 4.4 shows how Gorgeous process the input and represent the programs. Sections 4.5 presents how patterns are learned and Section 4.6 describes how these patterns are used to generate the refactoring algorithms. Section 4.7 shows some implementation aspects, and finally, Section 4.8 concludes this chapter.

4.1 OVERVIEW

Gorgeous generates refactoring algorithms by learning and incorporating refactoring patterns from object-oriented programs. To this end, our approach encompasses three steps: 1) Instantiating Programs; 2) Extracting Patterns and; 3) Generating Algorithms. Figure 4.1 illustrates how these steps interact with each other, considering their inputs and outputs.

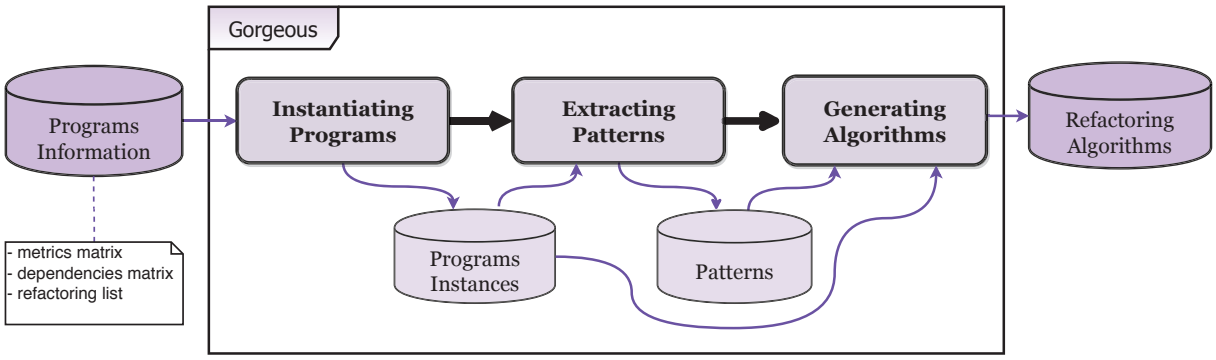


Figura 4.1: Gorgeous overview.

Gorgeous receives as input information collected from several programs and produces refactoring algorithms as output. This information concerns, for each program, metrics of classes and methods, dependencies between classes, and refactoring operations applied in the program. First, an step is performed to transform the information given as input into program instances following a representation schema. These program instances are the input of next steps. Then, the second step learns to find patterns from applied refactoring operations. The main activity of this step is to execute a clustering algorithm to create groups of elements based on similar refactoring applications. Then, each group of elements represents a pattern. In the next step, each pattern is used in the process of generating a refactoring algorithm. The idea is to generate algorithms that learn from the refactoring applications of the cluster elements. Later, these algorithms can be executed to find refactoring operations for other software programs.

4.2 PROGRAMS INFORMATION

The programs information should be provided as input of Gorgeous. This information must follow a specific format that will be later read by Gorgeous to perform the approach steps. In this way, for each program, three components are needed: a metric matrix, a dependency matrix, and a refactoring list.

- **Metric matrix:** It is a $n_e x n_m$ matrix, where n_e is the number of elements (classes and methods) in the program and n_m is the number of metrics computed. We considered a total of eight metrics defined from m_1 to m_8. A class element is associated with all eight metrics, while a method element is associated with two metrics. An element value for a metric m_i is denoted as e_{m_i}, such that $1 \leq i \leq 8$. Table 4.1 describes more details about these metrics.
- **Dependency matrix:** It is a $n_c x_3$ matrix, where n_c is the number of pairs of classes of the program. For each pair of classes, the columns represent, respectively, the number of dependencies between the pair, the number of dependencies from the first class to the second one, and the number of dependencies from the second class to the first one;
- **Refactoring list:** It is composed of refactoring operations applied in the program. They are listed in a descriptive format, such as ExtractMethod(*method*) and MoveMethod(*source_class.method*, *target_class*).

Tabela 4.1: Metrics calculated to an element [3].

Element	Metric	Description
class	m_1	Number of immediate base classes (INIFAN)
	m_2	Number of classes coupled (CBO)
	m_3	Number of classes derived (NOC)
	m_4	Number of methods (WMC)
	m_5	Number of all methods (RFC)
	m_6	Max Inheritance Tree (DIT)
class / method	m_7	Number of lines of code (LOC)
	m_8	Number of commented lines of code (CLOC)

4.3 REFACTORING ALGORITHMS

The refactoring problem consists of finding pieces of code that would benefit from the application of certain types of refactorings, e.g., a method that should be moved. Our algorithm explores this problem by suggesting a set of refactoring operations for a given object-oriented program. In this context, we define a refactoring algorithm based on a formalization given by Cormen et al. [74], which states an algorithm is "*any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output*".

Algorithm 4.1 shows the structure of the algorithms produced. An algorithm A receives as input program information (as described above) that will be used to instantiate a program P. When executed the algorithm returns as output a set of refactoring solutions S for P. A

refactoring solution represents a suggestion of refactoring operation for P. It is composed of three parts: type of refactoring, actors, and roles, as exemplified in Table 2.1.

Algorithm 4.1: Pseudocode of Refactoring Algorithm

```
1  Input: metrics matrix
2  Output: set of refactoring solutions S = (s1, s2..sn)
3  begin
4      Instantiate program P based on its metrics and dependencies;
5      s1=procedure1 (type1, RU1);
6      s2=procedure2 (type2, RU2);
7      ..
8      sn=proceduren (typen, RUn);
9      procedurei(typei,RUi)
10         repeat
11             actors.insert(search_actor(RUi));
12         until all actors are selected;
13         Instantiate a solution si based on typei and actors;
14         return si;
15     search_actor(RUi)
16         foreach e ∈ E do
17             if satisfies (e, RUi) is 1 then
18                 select element e as actor;
19                 return e;
20             end
21         end
22 end
```

For instance, a move method refactoring solution has three actors: a source class, a target class, and a method to be moved. Table 4.2 shows an example of object-oriented elements defined as actors. This refactoring solution means the *getSalary()* method should be moved from the *Person* class to the *PaymentOptions* class.

Tabela 4.2: Example of a move method refactoring solution.

Type	**Actor**	**Role**
Move Method	Person	source class
	getSalary()	method
	PaymentOptions	target class

Figure 4.2 shows a sequence diagram to illustrate how a refactoring algorithm works. First, the user needs to provide the metrics matrix in order to instantiate the program representation. After that, a set of procedures is executed in sequence.

Each procedure is in charge of instantiating a refactoring solution. A procedure defines a refactoring type and rules used to search for actors from available elements. Basically, a procedure has two main steps: 1) search for actors and 2) instantiate a refactoring solution. We describe a procedure based on its refactoring type, then if we mention a "move method procedure", it means a procedure is in charge of instantiating a move method refactoring solution.

A search for actors is performed by each actor that needs to be found. Each search for an actor is associated with a set of rules (RU). It describes the characteristics of the actor we are

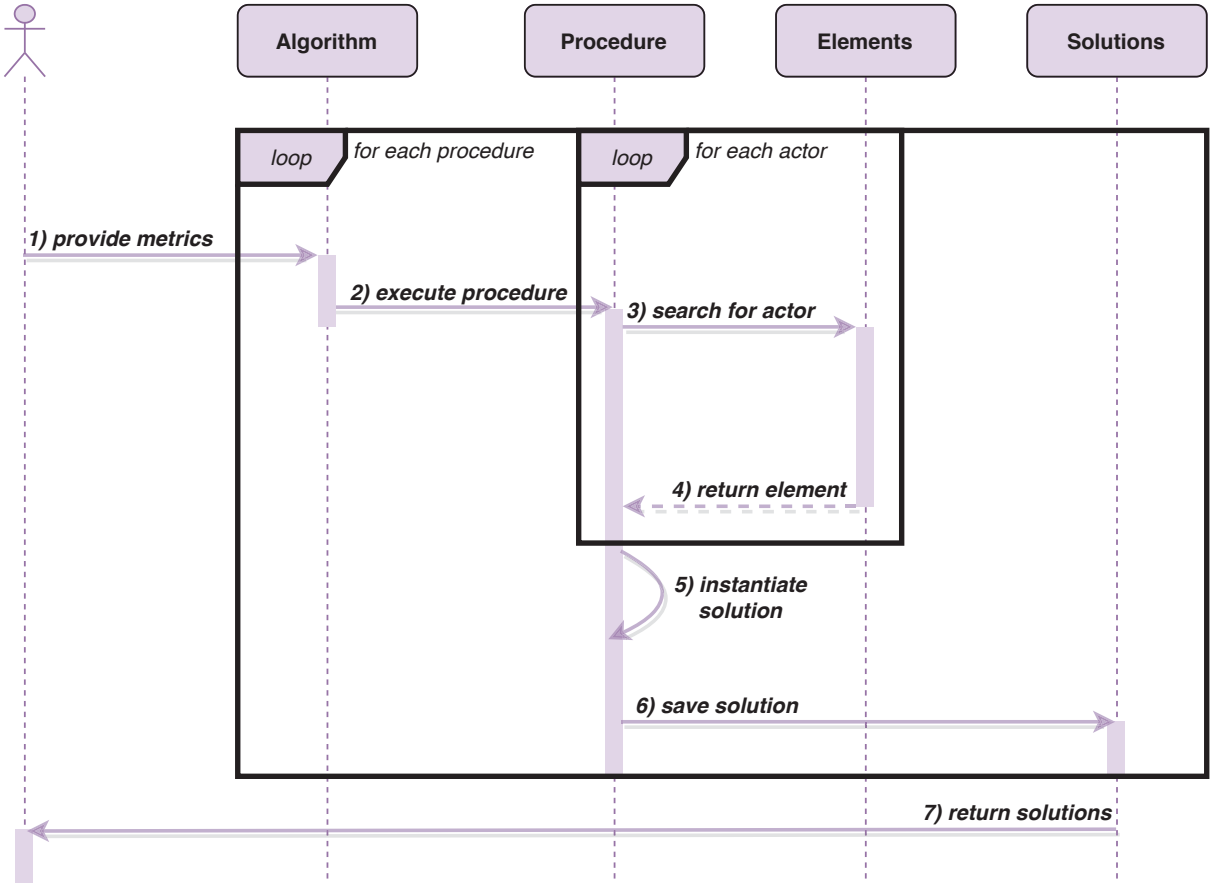


Figura 4.2: Sequence diagram of the refactoring algorithm process.

looking for, based on an interval of values for the metrics. RU is composed by eight rules to search for classes and two rules to search for methods.

A rule (ru_j) defines an interval of values $[a, b]$ for a metric m_i, as described in Equation 4.1. An element (e) is selected as an actor if the whole set RU is satisfied. An individual rule ru_j is satisfied when the element value e_{m_i} fits between the specified interval. Equation 4.2 shows how this measure is computed. For each rule ru_j, where n is the number of rules, the result is 1 if e_{m_i} belongs to the specified interval of ru_j, otherwise it is 0. The final value is the average result of RU, which is a number in the interval [0, 1], where 1 means all rules are satisfied and 0 means no rule is satisfied.

$$ru_j = [a, b] : 1 \leq i \leq 8 \tag{4.1}$$

$$satisfies(e, RU) = \frac{1}{n} \sum_{j=1}^{n} \begin{cases} 1 & if\ e_{m_i} \in ru_j \\ 0 & if\ e_{m_i} \notin ru_j \end{cases} \tag{4.2}$$

For example, if a procedure needs to search for a source method, where $10 \leq m_7 \leq 200$ and $10 \leq m_8 \leq 20$ are, respectively, the interval for ru_7 and ru_8, a method with values of m_7 and m_8 fitting these ranges is selected as actor. Once the actors are found, a solution is instantiated using the predefined refactoring type and the found actors. A procedure has a predefined random number of trials to select an actor. If no actor is found after that, the algorithm skip it and goes to the next procedure, which is executed until no more are found. As a result, a set of refactoring solutions is returned.

4.4 INSTANTIATING PROGRAMS

This first step to generate the refactoring algorithms is the instantiation of programs instances based on the information provided as input. A program instance follows a specific program representation schema, as presented in Figure 4.3. Each diagram class represents a different information.

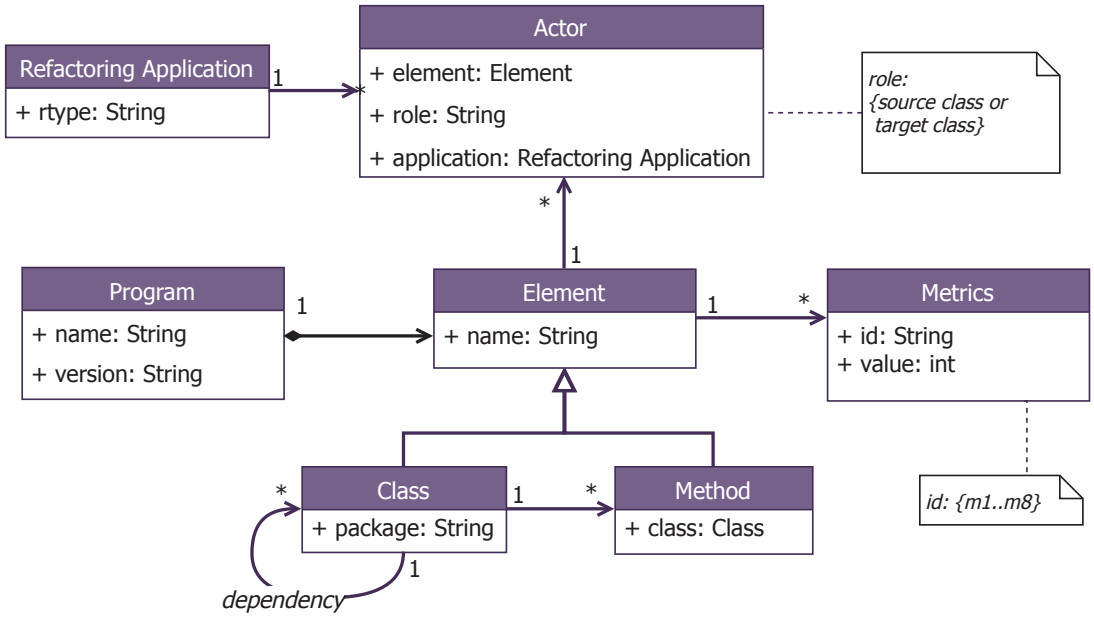


Figura 4.3: Program representation schema.

A program is composed by several elements. An element, that can be a class or a method, can play a role as an actor in a refactoring application. This one represents a refactoring operation that was applied in the program and version under consideration. Moreover, class elements may have dependencies with other class elements. The characteristics of the elements are represented by well-known object-oriented metrics for the class and method levels [3]. This step has as output a set of programs instances that will be used in the next steps.

The same schema is used to instantiate a program during a refactoring algorithm execution. However, in that case, information about dependencies and refactoring applications are not needed.

4.5 EXTRACTING PATTERNS

This step is in charge of learning and extracting patterns from refactoring applications of different programs. In this sense, the input of this step are the programs instances obtained in the last step. A clustering algorithm is executed to generate groups composed of elements (classes or methods) that were refactored in a similar way. The main idea behind this clustering is to find different refactoring patterns. In this way, each cluster of elements represents a refactoring pattern. Based on the program representation, each element is associated with a set of metrics values and refactoring applications it was involved. In this sense, when an element belongs to a cluster, all these information are part of the cluster itself, helping to characterizing a refactoring pattern. Figure 4.4 shows a representation schema illustrating how a pattern/cluster is structured.

The input of the clustering algorithm is generated by extracting the refactoring applications from the programs instances. In this respect, the input of the clustering algorithm is a $n_e x n_t$

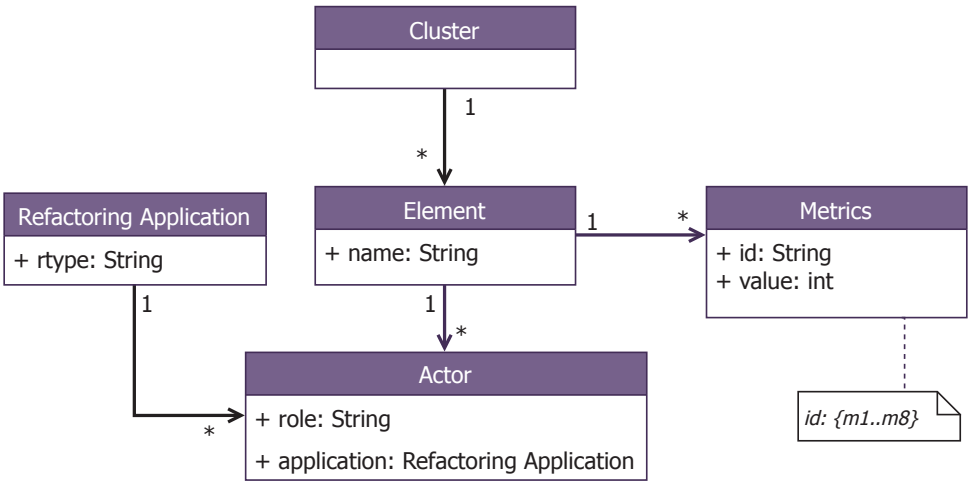


Figura 4.4: Pattern representation schema.

matrix, in which n_e is the number of elements and n_t is the number of refactoring types. Two matrices are generated, one for class-level and another for method-level. Each row represents an element and each column represents the number of refactoring applications by type. Table 4.3 presents an example of input for both class and method levels. A different number of refactoring types is used if comparing class-level and method-level, because some of them only apply to one level, such as extract class and extract method.

Tabela 4.3: Input of the clustering algorithm.

Tabela 4.4: Class-level.

Class	**EC**	**MM**	**PU**	**PD**
$Class_1$	1	0	0	1
$Class_2$	0	0	0	0
$Class_3$	1	2	1	0
$Class_4$	0	0	2	0
$Class_5$	0	3	0	4

Tabela 4.5: Method-level.

Method	**MM**	**PU**	**PD**	**EM**	**IM**
$Method_1$	2	0	0	1	1
$Method_2$	0	0	0	0	0
$Method_3$	1	1	1	0	0
$Method_4$	0	0	2	0	0
$Method_5$	0	1	0	2	0

EC extract class, MM move method, PU pull-up method, PD push down method, EM extract method, IM inline method.

The clustering is performed by considering the number of times each type of refactoring has been applied to an element. In this way, this information represents the dimensions/features of the clustering algorithm. The main idea behind using the number of times is to differentiate the elements that were refactored many times from elements receiving usual refactoring applications, rather than differentiate them only by the refactoring type applied. Hypothetically, a refactoring algorithm could find an actor similar to elements in which refactorings were applied many times. This could indicate such an actor can benefit from a refactoring application, but it does not necessarily mean it needs to be exactly the same refactoring type. This is the kind of pattern we are willing to find by using a clustering algorithm.

In this way, the clustering algorithm is executed twice, first to create the clusters of classes, and after that, to create the clusters of methods. Our idea is to find different patterns by

clustering and learning separately from classes and methods. As output, a set of clusters C is generated for each execution of the clustering algorithm. Then, each cluster c is composed of a set of elements E. Each cluster is interpreted as a refactoring pattern and it is used in the next step to generate a refactoring algorithm.

4.6 GENERATING ALGORITHMS

This step has as goal the generation of algorithms based on the patterns identified in the last step. GE is executed at least n times, where n is the number of patterns/clusters. To do this, we have defined two different grammars, one to generate refactoring algorithms based on the class-level patterns, and another one to the method-level patterns. The defined grammars are presented in Section 4.6.1.

The corresponding grammar is used by GE, which is executed separately using each cluster, to learn its patterns and to generate a corresponding refactoring algorithm. Each GE iteration has a set of solutions represented by arrays of integers. To evaluate each solution, the array is mapped into a refactoring algorithm. To do this, GE reads the grammar file, learns the grammatical aspects and uses the values of the array to decide which grammar values are assigned to each node.

GE searches for the best combination of rules and refactoring types, aiming at improving the fitness value. Section 4.6.2 presents in detail how the fitness function is calculated. The refactoring algorithm with the best fitness value is stored in the repository of refactoring algorithms. The greater the fitness function value, the better the algorithm. The generation continues using the next cluster, until no more clusters are left. The generated refactoring algorithms could be then executed over other object-oriented programs to find refactoring solutions.

4.6.1 Grammars

The grammars encompass the procedures, refactoring types, rules, and intervals. Figures 4.5 and 4.6 present, respectively, the grammar for the class and method-levels.

The items between ⟨ ⟩ are non-terminal nodes, | represents the logical operator OR, ::= means the node can take any of the next options, and the other items are terminal nodes. For instance, the node $\langle ru1 \rangle$ can take either the values `IntervalA` to `IntervalN` when mapped to a program. On the other hand, $\langle procedure \rangle$ can take the value of a single $\langle rtype \rangle$ or a composition of $\langle rtype \rangle$ $\langle procedure \rangle$.

As mentioned before, each refactoring is associated with a set of procedures, and each procedure is associated with rules, which in turn are associated with several intervals of metrics. The intervals are created at runtime and their size assumes one of the following: 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, or 500. As such, for a given ru_i, an interval of size s is generated respecting the minimum and maximum values of m_i, based on the elements of the cluster. Furthermore, we adopted some restrictions to avoid the generation of only one interval encompassing from the minimum to the maximum value. Given a metric maximum value as max, the size s of an interval must assume a maximum value of $max/2$. Then, we reduce by half the maximum size of an interval.

⟨*procedure*⟩ ::= ⟨*rtype*⟩ ⟨*procedure*⟩ | ⟨*rtype*⟩

⟨*rtype*⟩ ::= ⟨*extractClass*⟩ | ⟨*moveMethod*⟩ | ⟨*pullUpMethod*⟩ | ⟨*pushDownMethod*⟩ | ⟨*inlineClass*⟩

⟨*extractClass*⟩ ::= ⟨*searchClass*⟩

⟨*moveMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*pullUpMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*pushDownMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*inlineClass*⟩ ::= ⟨*searchClass*⟩ ⟨*searchClass*⟩

⟨*searchClass*⟩ ::= ⟨*ru1*⟩ ⟨*ru2*⟩ ⟨*ru3*⟩ ⟨*ru4*⟩ ⟨*ru5*⟩ ⟨*ru6*⟩ ⟨*ru7*⟩ ⟨*ru8*⟩

⟨*searchMethod*⟩ ::= ⟨*ru9*⟩ ⟨*ru10*⟩

⟨*ru1*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru2*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru3*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru4*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru5*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru6*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru7*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru8*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru9*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru10*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

Figura 4.5: Grammar to create refactoring algorithms for the class-level

4.6.2 Fitness Function

A refactoring algorithm (A) generated based on a cluster (c) is evaluated using a fitness function composed of two different objective functions, given by Equation 4.3.

$$F(A) = (SIM(A) + MQ(A)) * 0.5 \tag{4.3}$$

The first function $SIM(A)$ measures the similarity between A and the set E of elements grouped in c. The second fitness function MQ [75] measures the quality of a program after simulating the application of the refactoring solutions given by A. The definition of the functions are described in the next paragraphs.

The similarity function $SIM(A)$ (Equation 4.4) takes the set of procedures Pr from A and, to measure its similarity with a cluster, it uses refactoring types applied in E, as well as the characteristics of E. Based on that, we compute an average of two functions, $tsim(pr)$ and

⟨*procedure*⟩ ::= ⟨*rtype*⟩ ⟨*procedure*⟩ | ⟨*rtype*⟩

⟨*rtype*⟩ ::= ⟨*extractMethod*⟩ | ⟨*moveMethod*⟩ | ⟨*pullUpMethod*⟩ | ⟨*pushDownMethod*⟩ | ⟨*inlineMethod*⟩

⟨*extractMethod*⟩ ::= ⟨*searchMethod*⟩

⟨*moveMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*pullUpMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*pushDownMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchClass*⟩

⟨*inlineMethod*⟩ ::= ⟨*searchMethod*⟩ ⟨*searchMethod*⟩

⟨*searchClass*⟩ ::= ⟨*ru1*⟩ ⟨*ru2*⟩ ⟨*ru3*⟩ ⟨*ru4*⟩ ⟨*ru5*⟩ ⟨*ru6*⟩ ⟨*ru7*⟩ ⟨*ru8*⟩

⟨*searchMethod*⟩ ::= ⟨*ru9*⟩ ⟨*ru10*⟩

⟨*ru1*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru2*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru3*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru4*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru5*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru6*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru7*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru8*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru9*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

⟨*ru10*⟩ ::= `IntervalA|IntervalB|IntervalC|...|IntervalN`

Figura 4.6: Grammar to create refactoring algorithms for the method-level

$rsim(pr)$, by summing up the result for each procedure pr, where n is the number of procedures. Equation 4.5 presents $tsim(pr)$, which measures the similarity of the refactoring type pr_t with the refactoring types associated with c, i.e., the ones applied on E. Equation 4.6 presents $rsim(pr)$, which calculates the similarity by checking if each an element e satisfies the set of rules RU of a procedure pr.

$$SIM(A, c) = \frac{1}{2n} \sum_{i=1}^{n} tsim(pr_i) + rsim(pr_i) : \qquad pr \in Pr \tag{4.4}$$

$$tsim(pr, c) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & if\ pr_t = c_{t_i} \\ 0 & if\ pr_t \neq c_{t_i} \end{cases} : \qquad c_t \in c_T \tag{4.5}$$

$$rsim(pr, c) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & if\ satisfies(e_i, pr_{RU}) \\ 0 & otherwise \end{cases} \tag{4.6}$$

To calculate the quality function MQ, our approach searches for actors that satisfy at least 75% of the rules of A. If the actors are found, the application of the corresponding refactoring operation is simulated. This process is repeated for each procedure in A. MQ (Equation 4.7) measures the trade-off between cohesion and coupling of packages, such that $Pack$ is the set of packages involved in the simulations, i.e. the actors packages.

$$MQ = \sum_{i=1}^{|Pack|} MF(pack_i) \tag{4.7}$$

where $MF(pack_i)$ (Equation 4.8) measures the cohesion and coupling of a package $pack_i$.

$$MF(pack_i) = \begin{cases} 0, & \text{if } coh(pack_i) = 0 \\ \frac{coh(pack_i)}{coh(pack_i) + \frac{cop(pack_i)}{2}}, & coh(pack_i) > 0 \end{cases} \tag{4.8}$$

$coh(pack_i)$ measures the cohesion by counting the dependencies between classes within $pack_i$. $cop(pack_i)$ measures the coupling by counting the dependencies of classes within $pack_i$ to classes in other packages.

4.7 IMPLEMENTATION ASPECTS

Gorgeous implementation is based in Java and uses the jMetal framework [76] to support the implementation and execution of GE. In addition, Gorgeous is implemented to generate refactoring solutions for Java programs. The input data should be obtained by external tools, but following the format in which matrices are represented by comma-separated values (CSV) files, and lists are represented as text files (TXT).

In relation to clustering algorithm, we adopted the Expectation Maximization [45] algorithm through the use of *Weka* [77] since it provides an EM implementation in which no data distribution needs to be assumed. Also, a 10-fold cross-validation is automatically performed for EM by *Weka*, to select a configuration with the best number of clusters. We integrated *Weka* in our approach using its Java API.

4.8 CONCLUDING REMARKS

This chapter presented Gorgeous, a SBR learning approach used to find solutions for the refactoring problem. Gorgeous addresses the generation of refactoring algorithms capable of finding refactoring operations for a given object-oriented program. A refactoring algorithm is generated by learning and incorporating refactoring patterns into a grammar used by a GE technique. The approach encompasses three main steps in charge of: a) instantiating programs; b) extracting patterns; and 3) generating algorithms.

After the instantiation of programs, a step is performed to learn refactoring patterns from refactoring applications of existing software programs. It uses a clustering algorithm to group code elements that were refactored similarly. Each generated group represents a refactoring pattern and it is used in the next step to guide the generation of refactoring algorithms. The second step generates a set of refactoring algorithms based on the refactoring patterns. Each refactoring

pattern guides the generation of one algorithm by using a GE technique. Refactoring algorithms are generated aiming at improving quality and similarity with real refactoring applications.

Gorgeous is implemented using the Java programming language, as well as some tools/frameworks to support its activities, such as clustering and evolutionary algorithms application. In this way, we performed an empirical evaluation to analyze results obtained by Gorgeous. The next chapter shows our study and findings.

5 EMPIRICAL EVALUATION

The empirical evaluation was performed to assess different aspects of Gorgeous. The main goal of the evaluation is to analyze if our approach generates refactoring algorithms able to provide good refactoring solutions, in terms of quality and similarity with refactoring applications from existing software programs. Moreover, we also validate the importance of some steps and techniques used in our approach. Section 5.1 presents the research questions that guided us during this evaluation, Section 5.2 describes our experiment design and the process conducted to extract the dataset. Section 5.3 presents the obtained results and the answers for the research questions. Section 5.5 describes the threats to validity of this work, and finally, Section 5.6 concludes this chapter.

5.1 RESEARCH QUESTIONS

Each research question is evaluated using a measure related with an specific attribute to assess some aspect of Gorgeous. Some of the measures we are proposing in this work to assess quality and similarity attributes of a refactoring algorithm. In this respect, the research questions and the evaluation measures are described next.

- **RQ1: To what extend the grammatical evolution impact the generation of refactoring algorithms?** The goal of this question is to validate the use of GE in the third step of Gorgeous by assessing its impact in the generation of refactoring algorithms. In this sense, we performed an experiment comparing Gorgeous with a configuration in which GE was replaced by a Random Search. As well as in RQ1, the comparison was performed based on the fitness values and by using *Mann–Whitney* statistical test.

- **RQ2: To what extend the refactoring algorithms are able to find refactoring solutions capable of improving the quality of programs?** This question aims at evaluating if the refactoring solutions found by the refactoring algorithms are capable of improving the quality of programs. To measure the quality, we defined a metric called Quality Improvement (QI). To compute QI, we assess the quality value of a program by simulating the application of the refactoring operations given as solutions by the refactoring algorithms. The quality is evaluated in terms of modularity and represented by the MQ value, as defined in Equation 4.7. Then, we assessed how much was the improvement of the quality value comparing with the original program. Equation 5.1 presents how QI is calculated, where P_o is the original program and P is the program after the refactoring simulation.

$$QI(P) = \frac{MQ(P) - MQ(P_o)}{MQ(P_o)} \tag{5.1}$$

  A manual validation of the solutions was also performed to check if the refactoring solutions suggested by the refactoring algorithms makes sense semantically. Equation 5.2 presents the definition of Manual Correctness (MC), such that, S is the set of refactoring

solutions found by the algorithms, and S_1 is a subset of solutions from S, manually classified as semantically coherent.

$$MC(S) = \frac{|S_1|}{|S|} \tag{5.2}$$

- **RQ3: To what extend the solutions provided by refactoring algorithms improve program quality when compared with refactoring operations applied in practice?** The goal of this question is to compare, in terms of quality improvement, the refactoring operations given as solutions by the refactoring algorithms, with the operations applied in practice by developers. To answer this question, we computed QI based on two program versions, considering before and after the refactoring applications, and compared with the ones obtained by Gorgeous.

- **RQ4: To what extend the generated refactoring algorithms are able to find refactoring operations similar to the ones applied in practice?** This question aims at evaluating if the refactoring algorithms are able to identify refactoring operations applied by developers in practice. To measure similarity in comparison with these operations, we defined as O, the operations applied on the program version under consideration. Then, we defined a measure called $ARate$, which measures the rate based on the number of operations from O that a refactoring algorithm A is able to find. It might seem similar to the popular accuracy and recall measures [78], but instead of checking the solutions provided by a refactoring algorithm, it measures the capability of our algorithm to find refactoring operations applied in practice. Equation 5.3 presents $ARate$ definition, such that, Pr is the set of procedures from A, n is the size of O, pr_{RU} is the set of rules of pr, and $pr \in PR$.

$$ARate(Pr, O) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & if \ \ \exists pr \in Pr : satisfies(o_i, pr_{RU}) = 1 \\ 0 & otherwise \end{cases} \tag{5.3}$$

- **RQ5: To what extend the extraction of patterns impact the generation of refactoring algorithms?** This question was elaborated to validate the extracting step of Gorgeous. The main goal of this step is to learn and extract different patterns from the refactoring data. The refactoring patterns are represented by the clusters, which are used to generate algorithms. To answer this RQ, we compared, in terms of QI and $ARate$, Gorgeous results with results obtained by a configuration where the step extracting patterns is not performed.

5.2 EXPERIMENTAL SETTING

In this evaluation, we follow the Goal Question Metric (GQM) [79] approach, as described in Table 5.1. Based on that, the main goal of our evaluation is to analyze if the refactoring solutions, obtained by the refactoring algorithms generated by Gorgeous, are good in terms of quality and similarity with real refactoring applications. Moreover, we also validate the impact of some steps and techniques used in our approach. Next sections present the details of the conducted experimental setting.

To evaluate our approach, we use popular Java programs of several sizes and domains extracted from software repositories of *GitHub*. First, we collected information about the refacto-

Tabela 5.1: Evaluation of Gorgeous with GQM

Goal	to analyze the solutions obtained by the refactoring algorithms	
Purpose	to evaluate the capability of the refactoring algorithms to generalize good results	
with respect to	program quality and similarity with real refactoring applications	
from the point of view	of the user	
in the context of	refactoring of Java programs	
Question	**Measure**	**Attribute**
RQ1	Fitness Values Analysis	Modularity/Similarity
	Mann–Whitney Test	Confiability
RQ2	Manual Correctness (MC)	Correctness
RQ2, RQ3, RQ5	Quality Improvement (QI)	Modularity
RQ4, RQ5	ARate	Similarity
H_0	Gorgeous is not capable of generating refactoring algorithms able to find refactoring operations similar to real ones, while bringing improvement in terms of quality.	
H_1	Gorgeous is capable of generating refactoring algorithms able to find refactoring operations similar to real ones, while bringing improvement in terms of quality.	

ring applications associated with each program [1]. We selected 40 programs by excluding the ones with less than 5 refactoring applications. For each program we collected the required information to execute Gorgeous following our program representation. Then, we have downloaded from *GitHub* 80 versions of the programs, 40 before the refactoring applications, and 40 after the refactoring applications. The versions after the refactoring applications were used to evaluation purposes only, while the others were used in the learning process, since the idea is to learn the patterns that can lead to refactoring.

Using the downloaded programs, we extracted information about the programs structure using the *Understand* tool[2]. We extracted basic information of the classes and methods, such as signatures and packages they belong. Also, we collected the metrics and dependencies among elements. The information from all programs is given as input to Gorgeous, which manipulates the programs based on the representation presented in Figure 4.3.

We performed a 10-fold cross-validation dividing the programs in 10 different samples, each one composed of 4 programs with different numbers of refactoring solutions. Thus, each of the 10 folds is composed by 36 programs (9 samples) used for training and 4 programs (1 sample) used for validation and testing.

The program versions and other details are presented in Table 5.2. They correspond to the version right before the refactoring applications. For each program, it is presented the number of classes (NC), number of lines of code (LOC), original values of Modularization Quality (MQ), and the number of refactoring applications (NA). We built the folds trying to balance the number of refactoring applications in each one.

Expectation Maximization (EM) was executed for each fold using 9 samples. In this way, the clusters were generated based on the 36 training programs of the current fold. As mentioned

[1]More information about the programs and refactorings can be found at http://aserg-ufmg.github.io/why-we-refactor/#/projects.

[2]https://scitools.com/features/

Tabela 5.2: Program details

Fold	Program	NC	LOC	MQ	NA
1	Activity 5.17.0	3,333	183,502	0.1038	5
	CyanogenMod A. F. 11.0	10,064	883,564	0.2215	14
	Drools 6.3.0	5,924	531,292	0.2250	7
	Fabric8 2.1.11	974	47,563	0.1865	6
2	Facebook A. SDK 4.2.0	581	38,314	0.1241	5
	Geoserver 2 .7.2	7,521	464,440	0.1875	6
	Gradle 2.6	8,360	222,120	0.1679	14
	Graylog 1.2.0	1,947	83,566	0.1380	7
3	Languagetool 3.3	1,201	70,295	0.1643	5
	Mortar 0.18	175	3,921	0.1426	6
	Spring Boot 1.2.4	2,855	99,002	0.0016	8
	Voltdb 5.2.3	5,466	461,750	0.1457	16
4	Closure C. 20150609	2,083	238,360	0.2690	17
	Drill 0.9.0	3,048	181,479	0.1327	9
	MPS 3.2.2	36,240	1,187,832	0.1067	6
	Quasar 0.7.0	1,346	55,641	0.0227	5
5	Hive 1.2.1	9,414	753,208	0.1749	20
	jOOQ 3.6.2	1,413	110,677	0.0768	9
	Netty 3.10.3	1,222	78,225	0.2480	6
	TextSecure 2.19	850	44,182	0.1826	5
6	Bitcoinj 0.12.3	1,167	93,038	0.2521	10
	Neo4j 2.3.0	10,451	495,818	0.1626	21
	Presto 0.107	3,051	235,964	0.2852	6
	Tomahawk A. 0.83	494	26,715	0.1918	5
7	Cassandra 2.2.0	4,108	271,439	0.1988	23
	Java Driver 2.1.6	876	41,358	0.3161	10
	Spring Framework 4.2.0	12,596	526,146	0.1984	5
	Tachyon 0.6.4	1,092	72,404	0.1445	7
8	Hazelcast 3.5.1	7,401	345,652	0.1319	25
	Rest Li 2.6.2	2,737	202,248	0.1389	10
	Vert X 3.0.0	599	49,791	0.1411	7
	WordPress A. 4.0	1,410	67,364	0.0332	6
9	Android IMSI C. D. 0.1.29	244	13,439	0.1492	7
	Checkstyle 6.7	1,737	60,775	0.2527	6
	Graphhopper 0.7.0	554	46,985	0.3103	35
	Jersey 2.19	6,822	216,828	0.2092	11
10	Crate 0.49.2	2,625	122,281	0.1714	7
	Deeplearning4j 0.4	808	45,103	0.1505	6
	Infinispan 5.2.13	4,320	219,253	0.1537	13
	Openhab 1.7.0	4,041	264,756	0.0040	5

before, the clustering is executed separately for classes and methods. EM was executed using *Weka* [77] that automatically finds the best number of clusters. At the end, at the class-level,

between 2 and 3 clusters were generated by fold and, at the method-level, between 2 and 4 clusters.

Once the clusters were generated, in the next Gorgeous step, we performed 30 runs of GE for each cluster aiming at generating the refactoring algorithms. EM and GE parameters were selected from the literature, as well as the number of runs [80]. We also set a maximum of 20 procedures for a refactoring algorithm. It was based on the numbers of refactoring applications analyzed in the programs. Table 5.3 shows the configuration of the GE algorithm.

Tabela 5.3: GE Algorithm Configuration

Parameter	**Value**
Initialization	Random
Population Size	100
Number of GE Fitness Evaluations	10.000
Crossover Operator	Single Point Crossover
Crossover Probability	90%
Mutation Operator	Integer Mutation
Mutation Probability	1%
Selection Operator	Binary Tournament
Pruning Operator Probability	1%
Duplication Operator Probability	1%
Maximum Procedures	20

The quality function was computed by simulating the application of the refactoring algorithms on the 4 validation/testing programs. We defined a maximum of 5,000 iterations to search for an actor, to limit long executions in situations where an actor is not found quickly. At the end, a set of refactoring algorithms was generated for each fold, each algorithm based in a cluster. We apply these refactoring algorithms in the validation/testing programs to return refactoring solutions for them.

Based on the presented information, we defined three experiments. To answer RQ1, we replaced GE by a Random Search. We performed the other steps and kept other details of Gorgeous in the same way, such as the grammars and fitness function. Also, we set for this experiment the same possibility in terms of iterations. In this sense, Random iterates 10,000 times and executes 30 times for each cluster as well. To answer RQ5, we derived a configuration of the approach without the extracting step and called it NoCluster. In this case, the folds are divided as presented but no cluster is provided. Instead of that, a GE execution receives as input the whole set of elements from the fold. The other settings are the same as presented above. To answer the other RQs, we execute Gorgeous as presented above. Results obtained by these experiments are presented in the next section.

5.3 RESULTS

This section presents a description and discussion of the results obtained in our evaluation. Each subsection presents the answer for a specific research question and the results obtained by the experiment designed to answer it.

5.3.1 Answering RQ1 - To what extend the grammatical evolution impact the generation of refactoring algorithms?

To answer RQ1, we compared results obtained by Gorgeous and Random experiments analyzing their fitness function values. These values are presented by fold and algorithm, respectively, at the class and method levels, in Tables 5.4 and 5.5. Each value represents the average of 30 fitness values based on the 30 runs. Each algorithm, represented by 0, 1 or 2, was generated based on a specific cluster. Bold values represent that, comparing the averages of Gorgeous and Random, the higher average is statistically significant considering 95% of confidence based on the *Mann–Whitney* non-parametric test [81].

Tabela 5.4: Fitness values of Gorgeous and Random Search for the class level.

Fold	Algorithm	N. Elements	N. Solutions	Gorgeous	Random	p-value
1	0	7	10	**0.6937**	0.5859	3.88E-11
	1	21	65	**0.4794**	0.4380	1.93E-08
	2	12	23	**0.5053**	0.4740	2.84E-04
2	0	5	10	**0.6813**	0.5977	1.19E-07
	1	10	49	**0.7563**	0.7206	2.85E-11
	2	19	28	**0.4760**	0.4292	2.87E-11
3	0	15	61	**0.7786**	0.7270	3.45E-06
	1	5	10	**0.6813**	0.5692	4.28E-11
	2	20	27	**0.5411**	0.4842	3.45E-06
4	0	28	46	**0.5826**	0.5202	4.88E-08
	1	15	14	**0.8395**	0.7333	3.31E-10
5	0	19	74	**0.5397**	0.4920	2.39E-04
	1	17	21	**0.5181**	0.4602	9.44E-08
	2	4	7	**0.6934**	0.6127	3.88E-11
6	0	20	68	**0.4826**	0.4390	2.08E-06
	1	10	19	**0.4645**	0.3972	3.06E-09
7	0	13	18	**0.7115**	0.6696	1.30E-07
	1	4	7	**0.6832**	0.6014	3.51E-11
	2	24	79	**0.4734**	0.4260	1.94E-09
8	0	26	74	**0.6021**	0.5502	5.81E-10
	1	12	27	**0.5797**	0.5350	2.39E-06
9	0	23	36	**0.5106**	0.4746	2.77E-05
	1	14	13	**0.7650**	0.7070	1.93E-08
10	0	4	12	**0.6730**	0.5968	2.87E-11
	1	25	87	**0.5256**	0.4773	2.79E-09
	2	15	14	**0.7930**	0.7542	2.51E-05

Tabela 5.5: Fitness values of GE and Random Search for the method level.

Fold	Algorithm	N. Elements	N. Solutions	Gorgeous	Random	p-value
1	0	185	189	**0.6860**	0.5257	2.66E-11
	1	64	66	**0.6673**	0.5739	2.87E-11
	2	51	54	**0.6682**	0.4852	2.86E-11
	3	7	7	0.5782	**0.6226**	7.11E-10
2	0	184	188	**0.6272**	0.4440	2.87E-11
	1	123	128	**0.6663**	0.4675	2.85E-11
3	0	13	13	**0.6138**	0.5597	1.47E-09
	1	167	170	**0.6375**	0.4729	2.86E-11
	2	65	69	**0.6422**	0.4861	2.87E-11
	3	58	60	**0.6171**	0.5606	1.27E-10
4	0	65	66	**0.7335**	0.7239	1.17E-03
	1	54	57	**0.7107**	0.7029	1.37E-03
5	0	56	59	**0.6435**	0.6146	1.87E-08
	1	61	63	**0.6885**	0.6316	1.66E-07
	2	169	172	**0.6801**	0.6239	6.21E-09
	3	14	14	**0.6275**	0.5997	7.43E-09
6	0	62	67	**0.7222**	0.6698	1.02E-09
	1	180	182	**0.7202**	0.6738	6.23E-09
	2	57	58	**0.7021**	0.6611	7.09E-09
7	0	64	67	**0.6711**	0.6339	1.02E-06
	1	65	66	**0.6484**	0.6100	2.26E-07
	2	164	168	**0.6757**	0.6160	1.54E-09
8	0	60	63	**0.5523**	0.4658	5.23E-11
	1	164	168	**0.5654**	0.5057	5.62E-09
	2	61	62	**0.7049**	0.5938	3.88E-11
	3	14	14	**0.7002**	0.6339	2.29E-08
9	0	190	194	**0.6747**	0.6124	4.68E-11
	1	47	52	**0.6669**	0.6204	8.10E-09
	2	14	14	**0.6587**	0.6202	1.04E-09
	3	30	30	**0.6522**	0.5982	1.01E-09
10	0	167	170	**0.7333**	0.7206	3.28E-04
	1	64	65	**0.7292**	0.7089	9.40E-09
	2	64	66	**0.7054**	0.6972	0.08707
	3	11	12	**0.7146**	0.7080	0.0003873

Based on the results described in Tables 5.4 and 5.5, we note that all averages of Gorgeous are statistically better than the ones of Random. Moreover, by analyzing the p-value, we can not only guarantee 95% of confidence but 99% of confidence, since the p-value is lower than 0.01 for all cases. On the other hand, a sightly difference can be found at the method level. Considering Fold 1 and Algorithm 3, Random has statistically better fitness value average. By analyzing other characteristics, we can see these algorithms were generated based on the cluster with the lower number of elements (7) and refactorings (7). We assume GE probably converge to a local optima. In contrast, Random was able to explore more the search space. This kind of difference was not found at the class-level where there are also refactoring algorithms generated based on few elements. We analyzed this algorithm details and see the similarity value impacts the fitness function result. We assume the few elements along with the few metrics considering in the learning of methods limited the results.

Besides of Random comparison, we analyzed the fitness values obtained from different folds. The goal is to analyze the stability of the values obtained by Gorgeous. In this respect, Figure 5.1 presents a set of boxplots based on the fitness values obtained by Gorgeous at the class-level. Each boxplot is the set of fitness values (30 runs) by cluster, and each color represents a fold.

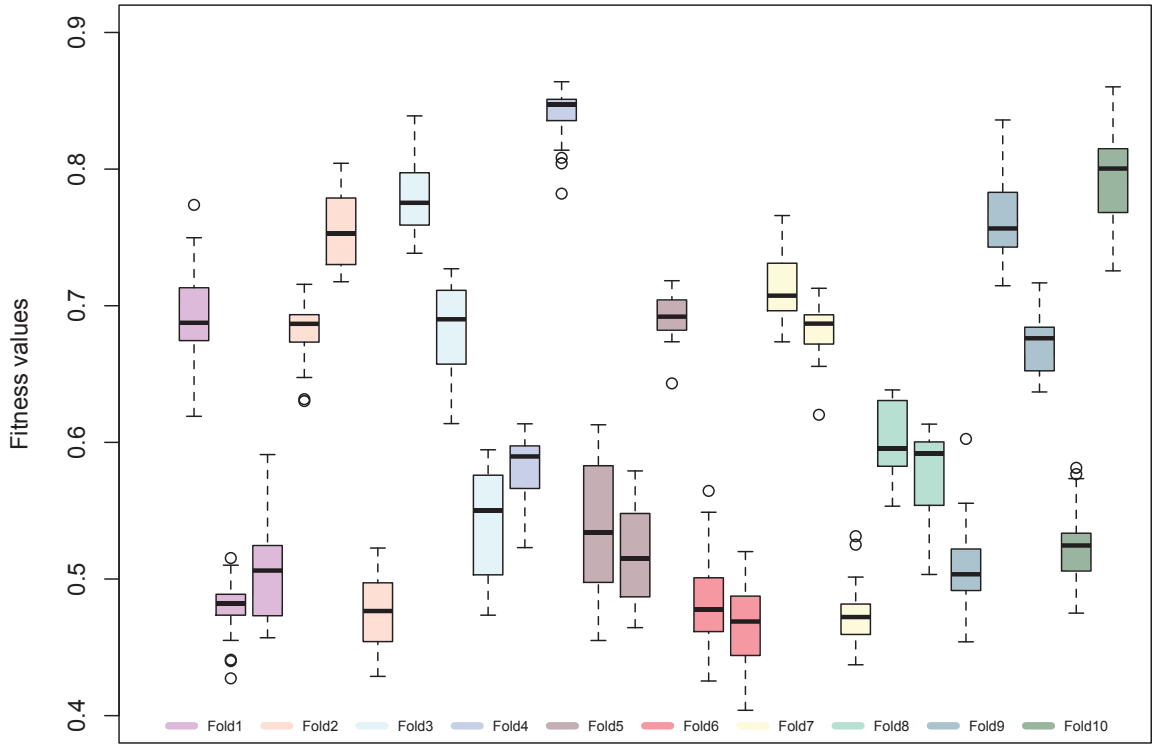


Figura 5.1: GE experiments by cluster

Figure 5.1 shows the results obtained for each fold are stable. If we analyze each fold separately, most of times, at least one of its clusters (boxplots) is in a different space. This is

expected since each cluster has different elements and characteristics. On the other hand, folds must have similar behavior and we can see the clusters from different folds aligned, specially if the fold has the same number of clusters, e.g. fold 2 and 3.

Answer to RQ1
GE is important for the generation of refactoring algorithms. First, because it has statistically significant better results when compared with Random Search. Second, because the results obtained from different folds are consistent, which represents Gorgeous has the capability of getting similar results from different training sets.

5.3.2 Answering RQ2 - To what extend the refactoring algorithms are able to find refactoring solutions capable of improving the quality of programs?

To answer RQ2, we simulated the application of the solutions generated by the refactoring algorithms in order to analyze quality aspects. For each fold, we have simulated the application of the refactoring operations in each validation program and computed against MQ. Table 5.6 presents, for each validation program, the original value of MQ, and the average of MQ ($avg(MQ)$) and the average of QI ($avg(QI)$), based on the algorithms obtained in the 30 runs. Light gray rows indicate programs with less than 1% of improvement, gray rows indicate programs between 1% and 5% of improvement, and dark gray rows indicate programs with more than 5% of improvement.

Based on the presented results, we have observed the quality improvement results have many variations, which is expected since the programs have distinct characteristics and domains. For instance, $avg(QI)$ varies from 0.03% up to 1701.39%. We noted the amount of improvement is highly dependent on the original MQ value, since the algorithms were able to obtain greater values of $avg(QI)$ for programs with the original MQ lower than the average (0.1654). Figure 5.2 illustrates this situation by a trendline (orange dashed line) of quality improvement based on the original MQ values from the programs.

We can see many variations of improvement by analyzing the orange dots. Moreover, most of them, present low improvement. Indeed, by analyzing QI for programs with MQ lower than 0.1654, QI seems to improve more as shown by the chart lines. We excluded from this chart the program with the lower MQ value since its QI was 1701.39%, so it would difficult the visualization.

In fact, 82% of the programs with high QI, 60% of the programs with medium QI, and 31% of the programs with low QI have the original MQ value lower than the average. In this sense, the lower the MQ value the higher tends to be the improvement, and more the program will benefit from the refactoring applications. In particular, the programs with the lowest values of MQ are the ones with the greatest $avg(QI)$, which are: Spring Boot 1.2.4 with 1701.38% of improvement, Openhab 1.7.0 with 148.39% of improvement, Quasar 0.7.0 with 276.12% of improvement, and WordPress Android 4.0 with 227.81% of improvement. Furthermore, such programs belong to different folds, which shows consistency in the results obtained by different folds.

We also observed that larger programs, in terms of number of classes, usually present less than 1% of improvement. Most of times, such programs have the original MQ value greater than the average. This can lead us to the idea that larger programs are usually better modularized than small projects. Probably, for a big project, more effort is devoted to the design and refactoring activities.

Tabela 5.6: Quality and Similarity Results

Fold	Program	MQ	$avg(MQ)$	$avg(QI)$	$avg(ARate)$	$QI(S)$
1	Activity 5.17.0	0.1038	0.1070	0.40%	20.00%	0.26%
	CyanogenMod A. F. 11.0	0.2215	0.2644	11.62%	10.00%	0.00%
	Drools 6.3.0	0.2250	0.2283	0.61%	**83.75%**	0.00%
	Fabric8 2.1.11	0.1865	0.1884	0.04%	**55.95%**	0.02%
2	Facebook A.SDK 4.2.0	0.1241	0.1263	0.21%	40.83%	0.00%
	Geoserver 2 .7.2	0.1875	0.1880	0.07%	**84.58%**	0.02%
	Gradle 2.6	0.1679	0.1686	0.16%	**81.11%**	0.00%
	Graylog 1.2.0	0.1380	0.1439	3.25%	**70.37%**	0.00%
3	Languagetool 3.3	0.1643	0.1728	1.94%	0.00%	0.02%
	Mortar 0.18	0.1426	0.2361	53.20%	**96.82%**	2.60%
	Spring Boot 1.2.4	0.0016	0.0411	1701.39%	3.33%	0.00%
	Voltdb 5.2.3	0.1457	0.1489	0.73%	**95.76%**	0.10%
4	Closure C.20150609	0.2690	0.2697	0.09%	0.00%	0.00%
	Drill 0.9.0	0.1327	0.1517	8.91%	0.00%	0.00%
	MPS 3.2.2	0.1067	0.1076	0.47%	**61.67%**	0.00%
	Quasar 0.7.0	0.0227	0.1394	276.12%	25%	6.47%
5	Hive 1.2.1	0.1749	0.1778	1.15%	**91.67%**	0.06%
	jOOQ 3.6.2	0.0768	0.0990	19.29%	33.33%	0.00%
	Netty 3.10.3	0.2480	0.2489	0.03%	34.76%	0.06%
	TextSecure 2.19	0.1826	0.1898	1.53%	**91.21%**	0.92%
6	Bitcoinj 0.12.3	0.2521	0.2657	0.64%	**62.67%**	0.07%
	Neo4j 2.3.0	0.1626	0.1652	0.45%	**76.67%**	0.04%
	Presto 0.107	0.2852	0.2863	0.25%	**63.00%**	0.03%
	Tomahawk A. 0.83	0.1918	0.1974	2.00%	42.86%	0.00%
7	Cassandra 2.2.0	0.1988	0.2057	0.46%	**50.00%**	0.23%
	Java Driver 2.1.6	0.3161	0.3402	1.75%	**68.10%**	0.00%
	Spring F. 4.2.0	0.1984	0.1991	0.06%	**97.78%**	0.00%
	Tachyon 0.6.4	0.1445	0.2772	50.32%	0.00%	36.67%
8	Hazelcast 3.5.1	0.1319	0.1378	2.95%	**100%**	0.14%
	Rest Li 2.6.2	0.1389	0.1452	3.96%	**78.89%**	0.04%
	Vert X 3.0.0	0.1411	0.1487	1.00%	**61.67%**	0.06%
	WordPress A. 4.0	0.0332	0.1557	227.81%	**92.50%**	61.39%
9	Android IMSI C. D. 0.1.29	0.1492	0.1971	16.05%	**99.72%**	0.00%
	Checkstyle 6.7	0.2527	0.2819	9.89%	33.33%	0.86%
	Graphhopper 0.7.0	0.3103	0.3124	0.19%	32.67%	0.19%
	Jersey 2.19	0.2092	0.2105	0.20%	48.41%	0.39%
10	Crate 0.49.2	0.1714	0.1753	0.37%	17.38%	0.08%
	Deeplearning4j 0.4	0.1505	0.1607	4.27%	0.00%	0.00%
	Infinispan 5.2.13	0.1537	0.1546	0.23%	**75.33%**	0.00%
	Openhab 1.7.0	0.0040	0.0149	148.39%	**73.94%**	116.76%
-	Average	0.1654	0.1857	63.81%	55.58%	5.69%

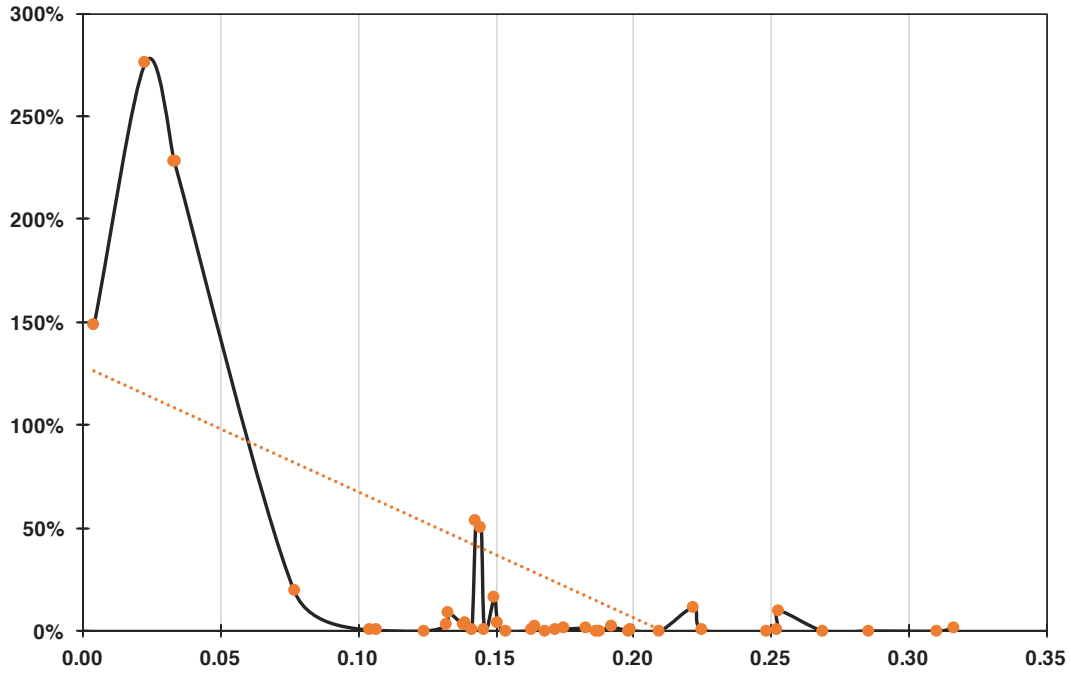


Figura 5.2: Trendline of QI based in the original MQ values.

The next analysis considers the quality improvement of refactoring solutions individually. The goal is to investigate if the quality improvement occurs due to the whole set of refactoring solutions and it is not related to only one solution. Table 5.7 presents, for each program, the number of refactoring solutions found by the algorithms and; the number and % of good, bad, and neutral solutions. We considered a good solution any one with a positive improvement ($QI > 0$). On the other hand, neutral solutions are the ones with no improvement ($QI = 0$), and bad solutions are the ones with a negative improvement ($QI < 0$). The higher percentage from these three groups is highlighted in bold.

The results presented in Table 5.7 clearly show most of found refactoring solutions improve the quality value of the program. Moreover, there is no case where the number of bad solutions was greater than good or neutral ones. The number of good solutions was higher than others in 80% of times. Also, the number of neutral solutions was higher than others in 15% of programs. However, in these cases, good solutions still present at least $\frac{1}{3}$ of solutions. In two other cases, the number of neutral and good solutions was the same, which corresponds to 5% of times. Besides, half of folds presents at least one program where 100% of solutions are good.

In addition, we manually calculated the correctness of the suggested refactorings. The following programs were randomly selected to be analyzed: Fabric8, Geoserver, Netty, and Tachyon. We selected the best solution, based on the fitness value, of the 30 runs of each selected program. We obtained the following MC values for the programs: Fabric8 with 58%, Geoserver with 60%, Netty with 43%, and Tachyon with 40%. The average of MC based on such programs resulted on 50%. Besides that, these are only some of the suggestions that can be found, since the refactoring algorithms can be executed again through the program to find more refactoring solutions.

Next, we describe the refactorings suggested by the refactoring algorithms for the Fabric8 program. The operations are described based on the following notation: **MoveMethod** (*sourceclass.method*) (*targetclass*), **ExtractClass** (*class*), **ExtractMethod** (*class.method*), and **InlineMethod** (*class1.method*) (*class2.method*).

Tabela 5.7: Impact of refactoring solutions on quality.

Fold	Project	Solutions	Good	Bad	Neutral	Good	Bad	Neutral
1	Activity 5.17.0	10	4	2	4	**40%**	20%	**40%**
	CyanogenMod A. F. 11.0	9	2	1	6	22%	11%	**67%**
	Drools 6.3.0	11	7	4	0	**64%**	36%	0%
	Fabric8 2.1.11	9	6	3	0	**67%**	33%	0%
2	Facebook A. SDK 4.2.0	3	3	0	0	**100%**	0%	0%
	Geoserver 2 .7.2	4	2	1	1	**50%**	25%	25%
	Gradle 2.6	4	4	0	0	**100%**	0%	0%
	Graylog 1.2.0	3	3	0	0	**100%**	0%	0%
3	Languagetool	12	12	0	0	**100%**	0%	0%
	Mortar 0.18	10	6	0	4	**60%**	0%	40%
	Spring Boot 1.2.4	11	5	0	6	45%	0%	**55%**
	Voltdb 5.2.3	15	6	1	8	40%	7%	**53%**
4	Closure C. 20150609	3	2	0	1	**67%**	0%	33%
	Drill 0.9.0	3	2	0	1	**67%**	0%	33%
	MPS 3.2.2	3	2	0	1	**67%**	0%	33%
	Quasar 0.7.0	3	2	0	1	**67%**	0%	33%
5	Hive 1.2.1	13	7	5	1	**54%**	38%	8%
	jOOQ 3.6.2	11	8	2	1	**73%**	18%	9%
	Netty 3.10.3	12	4	2	6	33%	17%	**50%**
	TextSecure 2.19	10	9	1	0	**90%**	10%	0%
6	Bitcoinj 0.12.3	8	7	0	1	**88%**	0%	13%
	Neo4j 2.3.0	10	5	1	4	**50%**	10%	40%
	Presto 0.107	7	4	0	3	**57%**	0%	43%
	Tomahawk A. 0.83	7	7	0	0	**100%**	0%	0%
7	Cassandra 2.2.0	9	7	1	1	**78%**	11%	11%
	Java Driver 2.1.6	8	3	1	4	38%	13%	**50%**
	Spring F. 4.2.0	13	7	5	1	**54%**	38%	8%
	Tachyon 0.6.4	7	6	0	1	**86%**	0%	14%
8	Hazelcast 3.5.1	14	9	3	2	**64%**	21%	14%
	Rest Li 2.6.2	11	9	0	2	**82%**	0%	18%
	Vert X 3.0.0	9	9	0	0	**100%**	0%	0%
	WordPress A. 4.0	8	4	0	4	**50%**	0%	**50%**
9	Android IMSI 0.1.29	9	5	1	3	**56%**	11%	33%
	Checkstyle 6.7	9	7	1	1	**78%**	11%	11%
	Graphhopper 0.7.0	10	4	1	5	40%	10%	**50%**
	Jersey 2.19	11	5	3	3	**45%**	27%	27%
10	Crate 0.49.2	5	5	0	0	**100%**	0%	0%
	Deeplearning4j 0.4	5	5	0	0	**100%**	0%	0%
	Infinispan 5.2.13	5	5	0	0	**100%**	0%	0%
	Openhab 1.7.0	5	5	0	0	**100%**	0%	0%

1. **MoveMethod**
   (*DestinationFacade.getBrokerAdmin*)
   (*BrokerFacadeSupport*)

2. **MoveMethod**
   (*BrokerFacadeSupport.getQueueProducers*)
   (*ActiveMQAutoConfiguration*)

3. **MoveMethod**
   (*BrokerFacade.isJobSchedulerStarted*)
   (*XmlModel*)

4. **MoveMethod**
   (*BrokerFacadeCallback.doWithBrokerFacade*)
   (*BrokerFacadeSupport*)

5. **ExtractClass**
   (*DestinationFacade*)

6. **ExtractClass**
   (*FabricFabConfiguration*)

7. **ExtractClass**
   (*BrokerFacadeSupport*)

8. **ExtractMethod**
   (*BrokerFacade.getId*)

9. **ExtractMethod**
   (*BrokerFacade.getBrokerAdmin*)

10. **InlineMethod**
    (*BrokerFacade.getConnections*)
    (*BrokerFacade.getBrokerAdmin*)

11. **InlineMethod**
    (*RouteXml.unmarshal*)
    (*BrokerFacade.getId*)

12. **InlineMethod**
    (*RemoteBrokerFacadeSupport.findBrokers*)
    (*BrokerFacade.getId*)

By analyzing the suggested refactorings, we found 5 suggestions that are not correct semantically, they are: 3, 8, 9, 11, and 12. In the cases presented on 8 and 9 there was no need to the method to be extracted. In the cases presented on 3, 11, and 12, the relation between the source and the target elements was not strong enough to apply the refactorings. Nevertheless, the other refactorings make sense semantically. We found by analyzing this solution and the ones of the other programs that most of the Extract Class refactorings were suggested correctly. We also found that most of the suggestions classified incorrectly involved the Inline Method refactoring, mainly because of the lack of semantic correlation between the source and the target methods.

Answer to RQ2
Refactoring algorithms were able to find refactoring solutions capable of improving the quality of programs in terms of modularity. In some programs, the improvement was not high, but still at least some improvement occur in every program. Also, we have observed that programs with very low modularity can benefit more from the refactoring applications.

5.3.3 Answering RQ3 - To what extend the solutions provided by refactoring algorithms improve program quality when compared with refactoring operations applied in practice?

To answer this question, we present in Table 5.6, in column $QI(S)$, the value of QI obtained by the solutions (S) applied by developers. These values show the refactoring operations applied by developers do not obtain good values of quality improvement. While all programs presented some quality improvement using the refactoring solutions suggested by Gorgeous, refactoring operations applied by developers presented no improvement at all for 40% of the programs. Also, 10% of the other programs show high improvement, 2% medium improvement, 48% low improvement. By comparing the overall average, Gorgeous performs on average 63% of improvement, while the developer solution obtained on average 5% of improvement.

Answer to RQ3
Gorgeous was capable of generating refactoring algorithms that provide solutions able to improve some level of quality in 100% of programs, while the developer applications resulted in improvement of only 60% of programs, being 40% resulted in no improvement.

5.3.4 Answering RQ4 - To what extend the generated refactoring algorithms are able to find refactoring operations similar to the ones applied in practice?

To answer RQ4, we analyze if the refactoring algorithms generated by Gorgeous would be able to find the refactoring operations applied by developers. As mentioned before, this analysis was performed using the previously defined measure $ARate$. In this respect, Table 5.6 presents in Column 6, $avg(ARate)$ formatted as percentages. The values greater than 50% are highlighted in bold. To obtain this value, we considered the set of refactoring algorithms generated based on a fold. The average is calculated based on 30 runs in this process.

The presented results of $avg(ARate)$ show several variations if compared among programs. The variations of the results are explained by the different characteristics of the considered programs. It would be very difficult to achieve very similar metrics results when dealing with the software changes. Briefly, the overall average is 55.58%, which is a good value if we consider we are trying balance both quality and similarity. However, it is important to analyze the results in deep.

The results show 60% of the programs have $avg(ARate)$ value greater than 50%, which means the refactored algorithms are capable of finding a set of refactoring solutions performed by developers. In particular, for one of the programs the algorithms were able to obtain a value of 100%. On the other hand, 5 programs presented 0% as result. By analyzing these programs, we found some similarities in their refactoring applications. First, most of them have several refactoring solutions involving the same element as actor. Then, the set of elements used to measure $ARate$ is small. Moreover, 93% of the refactoring applications not found by the refactoring algorithms are composed of method-level refactorings.

In fact, low values of $avg(ARate)$ do not exactly means bad results since some good refactoring solutions might be not identified by developers. In this sense, Gorgeous may be useful to identify other refactoring solutions rather than only similar ones. We present a more specific analysis using a solution found by a generated refactoring algorithm for the *Fabric8* program. It suggests to extract a class called *BrokerFacadeSupport*. The class under analysis was not part of a true solution and was not identified as a bad design. In the context of the program, a broker routes messages, handles transactions, and maintains subscriptions and connections. *BrokerFacadeSupport*[3] was designed taking into account the design pattern called Facade. In this sense, such class has the responsibility of providing a simplified interface to a complex system of classes. *BrokerFacadeSupport* provides several operations to deal with two aspects of a broker, which are connectors and topics. As presented by the Facade design pattern[4], an additional Facade can be created to divide responsibilities and to prevent the original Facade of growing and become complex. In this sense, the suggested solution makes sense since we could extract *BrokerFacadeSupport* into two different Facade for dealing with connectors.

Answer to RQ4
Refactoring algorithms found around 55% of real refactoring applications on average. Moreover, at least 50% of applications were found for half of programs. However, the results have many variations because of program particularities. We also find out that method-level operations are the main cause of low rates.

5.3.5 Answering RQ5 - To what extend the extraction of patterns impact the generation of refactoring algorithms?

To answer RQ5, we analyze values of $avg(QI)$ and $avg(ARate)$ obtained by Gorgeous and NoCluster experiments. Table 5.8 summarize these results. In general, Gorgeous performs better than NoCluster considering both quality and similarity results.

Specially, considering $avg(QI)$, Gorgeous presents the best values for all programs, except one, Vert X 3.0.0, with a small difference. Moreover, in relation to NoCluster, most programs have presented a very low improvement. On the other hand, there are six cases where NoCluster performs better when analyzing $avg(ARate)$ values. The values in which NoCluster is better are highlighted in bold in its column.

Results show, on average, and based on the 40 programs, Gorgeous obtained around 55% against 9% of $(ARate)$ for NoCluster. Furthermore, Gorgeous obtained a maximum of 100% of $(ARate)$ for a program, while NoCluster could get a maximum of only 32%.

Answer to RQ5
The approach without the extracting and learning of patterns impacts negatively not only on the similarity measure with real refactoring applications but also on quality aspects. This is an evidence the clusters are useful to guide the generation of algorithms able to incorporate the learned refactoring patterns.

[3]http://activemq.apache.org/maven/5.9.0/apidocs
[4]https://refactoring.guru/design-patterns/facade

Tabela 5.8: Gorgeous and NoCluster Comparison

Fold	Program	Gorgeous		NoCluster	
		$avg(QI)$	$avg(ARate)$	$avg(QI)$	$avg(ARate)$
1	Activity 5.17.0	0.40%	**20.00%**	0.15%	4.00%
	Cyanogen. 11.0	11.62%	**10.00%**	6.00%	**18.61%**
	Drools 6.3.0	0.61%	**83.75%**	0.08%	16.00%
	Fabric8 2.1.11	0.04%	**55.95%**	0.00%	0.10%
2	Faceb. SDK 4.2.0	0.21%	**40.83%**	0.00%	16.67%
	Geoserver 2 .7.2	0.07%	**84.58%**	0.05%	0.00%
	Gradle 2.6	0.16%	**81.11%**	0.10%	3.57%
	Graylog 1.2.0	3.25%	**70.37%**	0.35%	2.38%
3	Languagetool	1.94%	0.00%	0.12%	10.00%
	Mortar 0.18	53.20%	**96.82%**	5.19%	0.00%
	Spring Boot 1.2.4	1701.39%	3.33%	794.44%	**32.50%**
	Voltdb 5.2.3	0.73%	**95.76%**	0.44%	7.86%
4	Closure 20150609	0.09%	0.00%	0.01%	**5.83%**
	Drill 0.9.0	8.91%	0.00%	1.22%	**7.50%**
	MPS 3.2.2	0.47%	**61.67%**	0.09%	6.67%
	Quasar 0.7.0	276.12%	**25.00%**	106.55%	3.33%
5	Hive 1.2.1	1.15%	**91.67%**	0.28%	11.85%
	jOOQ 3.6.2	19.29%	**33.33%**	4.24%	20.00%
	Netty 3.10.3	0.03%	**34.76%**	0.00%	10.00%
	TextSecure 2.19	1.53%	**91.21%**	0.59%	8.67%
6	Bitcoinj 0.12.3	0.64%	**62.67%**	0.25%	15.71%
	Neo4j 2.3.0	0.45%	**76.67%**	0.09%	13.33%
	Presto 0.107	0.25%	**63.00%**	0.03%	10.67%
	Tomahawk A. 0.83	2.00%	**42.86%**	0.81%	0.00%
7	Cassandra 2.2.0	0.46%	**50.00%**	0.07%	13.64%
	Java Driver 2.1.6	1.75%	**68.10%**	0.23%	50.00%
	Spring F. 4.2.0	0.06%	**97.78%**	0.04%	15.00%
	Tachyon 0.6.4	50.32%	0.00%	22.70%	**8.00%**
8	Hazelcast 3.5.1	2.95%	**100%**	1.52%	21.82%
	Rest Li 2.6.2	3.96%	**78.89%**	1.43%	15.67%
	Vert X 3.0.0	1.00%	**61.67%**	1.75%	23.33%
	WordPress A. 4.0	227.81%	**92.50%**	95.50%	57.66%
9	And. IMSI 0.1.29	16.05%	**99.72%**	3.60%	0.00%
	Checkstyle 6.7	9.89%	**33.33%**	0.20%	8.33%
	Graphhopper 0.7.0	0.19%	**32.67%**	0.13%	0.00%
	Jersey 2.19	0.20%	**48.41%**	0.02%	27.33%
10	Crate 0.49.2	0.37%	**17.38%**	0.10%	1.43%
	Deeplear.4j 0.4	4.27%	0.00%	1.19%	**2.78%**
	Infinispan 5.2.13	0.23%	**75.33%**	0.03%	3.03%
	Openhab 1.7.0	148.39%	**73.94%**	1.19%	3.03%

5.4 DISCUSSION

Based on the results presented in this section, we can reject H_0 and accept H_1 stating that Gorgeous is capable of generating refactoring algorithms able to find solutions similar to real refactoring applications while also improving quality. In general, Gorgeous was able to generate refactoring algorithms that suggest solutions capable of improving the programs 55% on average. Furthermore, the refactoring algorithms could find on average 50% of real refactoring applications. These are good results considering the balance between similarity and quality.

Despite these results presented by averages, the values obtained for both similarity and quality have many variations. While some programs obtained 100% of quality improvement, others presented 0%. However, 80% of found solutions presented some kind of improvement to the program. It is expected those variations when working with software programs. As presented in the program details, they differ a lot from each other in terms of size and domain. Moreover, many information that are used may present variation, such as number of refactorings, number of elements in a cluster, number of refactoring applications, etc. All of these can impact positively or negatively in the results.

Another finding concerns the quality aspects of the solutions from developers. They were much lower in value compared with Gorgeous or even the NoCluster configuration. In fact, we do not have information about the intention of developers when refactoring these programs. In this way, we probably are not being able to capture with our metric the improvement expected by the developer. In this sense, a focused study may be useful to establish the relationship among refactorings and software quality metrics.

We have observed different findings for classes and methods. In fact, clusters with few methods impact negatively the similarity function, while clusters with few classes impact positively. This depends on the number of rules used, since the higher the number of rules, the more the space is restrict. If we increase the number of elements more rules needs to be satisfied and it is more difficult to find good solutions. However, if we reduce the number of elements we may restrict too much the search space. At the method-level, the first problem is hard to occur since only two rules are used. On the other hand, a cluster with many classes will restrict the space since we will have more rules to be satisfied. We observed the higher the number of classes the lower the similarity result. However, the opposite does not occur, since the experiment without clustering presented worse results at the method-level. In fact, methods are way more in numbers than classes, then we could not make a fair comparison. In this respect, the clustering of elements brings many advantages in the generation of refactoring algorithms.

Besides the quality and similarity results, it is important to highlight the refactoring algorithms could also be manipulated to improve the results. For example, we can opt by suggesting a refactoring operation only if it improves quality, or even if it improves quality more than a specific percentage. Also, the algorithm could suggest solutions only by considering some part of the program, e.g., a package the developer is working at the moment. In this sense, the refactoring algorithms could be manipulated regarding the way the solutions are suggested.

5.5 THREATS TO VALIDITY

The main threat of our study are the program folds using for validation and testing. We use the same set of programs in these phases. It might influence the results of our approach, since we do not perform a test with a set of programs not used during Gorgeous execution. However, our approach uses individual characteristics of elements in the learning, so we believe general

aspects of a program would not have a significant impact. Also, the similarity function was measured in a separate set of programs.

Another threat to the validity of our study is the coupling with the program representation in computing the fitness function. Indeed, the quality function is computed by simulating the refactoring applications based on such a representation. In this sense, we can not guarantee the same results will be obtained by using the original program or other representation. In this respect, we built the program representation making sure to encompass all information needed to compute the metric value.

To run Gorgeous, a preprocessing to collect the data needs to be performed. It involves the collection of elements information and refactoring applications. In this sense, the correctness of such an information is related to the tool used in this step. To instigate this threat we make sure to use a popular tool to extract the elements information, and to use refactoring applications extracted by studies following a methodology with recommended tools. Moreover, this can also impact the reproducibility of the results, since one might use other tools to extract the data.

Another threat is related to the application of the generated refactoring algorithms. In this evaluation, we defined one execution for each procedure, for example, an algorithm with 6 procedures generates a maximum of 6 solutions. This could be changed by increasing the running times of a procedure to consequently find more solutions or even all solutions. It would impact the results of the measures values. We did not perform experiments changing this configuration, but we expect we would reach better results by exploring this aspect. Our results encourage more rigorous analyses over the capability of refactoring algorithms.

Other aspects may limit the generalization of the refactoring algorithms results. First, although we have a great number of programs, some of them have few refactorings. Also, we use few metrics to generate method-level operations. Moreover, although most programs were improved by the refactoring solutions, the amount of quality improvement is highly dependent with the program characteristics. Then, it is not possible to guarantee the same amount of improvement for two different programs.

Finally, to execute the EM and GE algorithms, we used parameters from the literature, but a tunning phase may improve the results. Also, the GE technique is non-deterministic, but to mitigate this threat, we performed 30 runs of each technique.

5.6 CONCLUDING REMARKS

This chapter described the methodology followed to evaluate Gorgeous and discussed the results. An evaluation was conducted using 40 *Java* programs extracted from *GitHub*. A preprocessing was performed to extract the information needed by Gorgeous, such as elements information, metrics and refactoring applications. The experiments were designed aiming at answering five research questions regarding Gorgeous steps and following training and testing phases, in which 36 programs (90%) were used for training and 4 programs (10%) for testing.

Gorgeous was able to generate refactoring algorithms that could find different solutions for the testing programs. The refactoring algorithms obtained good results in terms of quality improvement compared with the original program, as well as similarity to refactoring applications performed by developers. However, the programs have different particularities which might affect the amount of quality improvement.

Results show the clustering is very useful to generate algorithms, specially by improving similarity, since it is capable of finding solutions similar to real refactoring applications. Furthermore, GE benefit the generation of algorithms and was also capable of obtaining stable results across different folds.

6 FINAL REMARKS

In this work, we presented Gorgeous, a SBR learning approach to generate refactoring algorithms. In this sense, we introduced the definition of a refactoring algorithm, which produces refactoring operations for a program given as input. Each algorithm is composed by a set of procedures, and each procedure has a set of rules defining characteristics of code elements that should receive a refactoring application. A refactoring algorithm is generated by considering quality improvement of a program and similarity with real refactoring applications.

Gorgeous includes three steps. The first step is in charge of instantiating program instances to be used in the next steps. The second one refers to the extraction of patterns. In this step, a clustering algorithm is executed to group code elements from different programs that were refactored in similar ways, considering the refactoring type and frequency of application. Each group of code elements represents a refactoring pattern. In the third step, the generation of algorithms is performed. In this way, a GE technique is executed to generate one refactoring algorithm based on the characteristics of each pattern discovered. We have defined two grammars used by GE in the generation of algorithms, for class and method-levels.

Gorgeous was implemented focusing on the refactoring of Java programs, as well as using it as programming language. Moreover, it uses *jMetal* and *Weka* frameworks, to provide, respectively, search and clustering techniques. We reported results of an empirical evaluation conducted using a 10-fold cross-validation with 40 Java programs extracted from *GitHub*. We collected several data from programs, such as elements metrics and refactoring applications. The experiments were performed aiming at answering five research questions concerning the impact of each Gorgeous step, as well as the solutions of the refactoring algorithms.

The approach was able to generate refactoring algorithms capable of identifying refactoring operations for different programs. Encouraging results were obtained in terms of quality improvement of the original program, and similarity in comparison with real refactoring applications. Although good results were obtained they have many variations in terms of values, mainly because of the particularities of each program. Our evaluation also showed the importance of the clustering in generating algorithms capable of suggesting good solutions regarding similarity with real applications. The next section presents the main contributions of this work.

6.1 CONTRIBUTIONS

This work has the main following contributions:

- The concept of refactoring algorithm, which receives a program as input and produces a set of refactoring operations for it. One of its advantages is the possible application to several programs;
- A SBR learning approach to learn patterns and automatically generate refactoring algorithms taking into account quality improvement and similarity with real refactoring applications;
- Grammars formalizing a set of rules to identify where and how refactorings should be applied;

- Learning of similar refactoring applications (patterns) by applying a clustering technique to existing elements;
- The generation of refactoring algorithms using GE;
- It reports an evaluation based on 40 open source Java programs extracted from *GitbHub*.
- It provides evidence to support our proposal is capable of generating refactoring algorithms able to identify refactoring operations to several programs, while improving modularity and similarity with real refactoring applications.

6.2 FUTURE WORK

As future work, we intend to improve some aspects of Gorgeous, as well as to perform other experiments. We are currently working to automate the preprocessing step to make Gorgeous less dependent on external tools. In this sense, an Abstract Syntax Tree (AST) is being used as representation. It is widely used to represent programs and it can map many aspects of a program. Based on this, we are able to extract all the information we need and consequently to calculate the metrics without other tools. In this way, Gorgeous and the refactoring algorithms can receive as input the program itself.

ASTs are also used by several IDEs to provide automatic support for refactoring application. In this sense, other future work, is to create an IDE plugin to provide the use of refactoring algorithms during development and to allow the automatic application of refactoring operations. By allowing this, we could apply the refactoring algorithm directly to a part of the program, e.g., a package or a class the developer is working on.

In addition, we intend to study other aspects of a refactoring algorithm. For example, to change the number of solutions a procedure can find. Also, we want to apply them to other programs not used in this work. Moreover, we intend to perform a qualitative study of the generated refactoring algorithms and clusters, in order to understand better the patterns of each cluster and how they have influenced the algorithms rules.

Finally, we are working to add other quality metrics in the fitness function, such as understandability and extensibility. Furthermore, we intend to test different weights for the quality and similarity metrics. Hence, we could evaluate better how these metrics impact each other. Regarding the experiments, we intend to run Gorgeous over a large dataset with more refactoring applications. In fact, we are currently working with a dataset of 1,932 *GitHub* repositories. Moreover, we want to conduct more experiments changing GE and EM parameters, as well as to evaluate different clustering algorithms.

6.3 AWARDS AND PUBLICATIONS

This section lists the awards and publications received.

- **Awards:**
  - Best paper award. WESB, 2018.
  - ACM-W scholarship award. GECCO, 2016.
- **Publications:**

- Thainá Mariani, Marouane Kessentini, and Silvia R. Vergilio. Uma proposta de geração automática de algoritmos de refatoração. *Workshop de Engenharia de Software Baseada em Busca (WESB)*, 2018 [82].
- Thainá Mariani and Silvia R. Vergilio. A systematic review on search-based refactoring. *Information and Software Technology*, 2017 [11].
- Thainá Mariani, Giovani Guizzo, Aurora T. R. Pozo, and Silvia R. Vergilio. Automatic Design of Algorithms Applied to the Multi-Objective TSP Problem. *Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)*, 2016 [83].
- Thainá Mariani, Giovani Guizzo, Silvia R. Vergilio, and Aurora T. R. Pozo. Grammatical Evolution for the Multi-Objective Integration and Test Order Problem. *Genetic and Evolutionary Computation Conference (GECCO)*, 2016 [80].
- Thiago N. Ferreira, Thainá Mariani, and Silvia R. Vergilio. Reviewing Six Years of Brazilian Workshop on Search-Based Software Engineering. *Workshop de Engenharia de Software Baseada em Busca (WESB)*, 2016 [84].

REFERENCES

[1] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[2] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2 edition, 2018.

[3] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[4] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*, SOOPPA, pages 274–282, 1990.

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

[6] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO, 2006.

[7] V. Alizadeh and M. Kessentini. Reducing interactive refactoring effort via clustering-based multi-objective search. pages 464–474, 2018.

[8] Dag I. K. Sjøberg, Aiko Yamashita, Bente C. D. Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[9] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[10] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[11] Thainá Mariani and Silvia Regina Vergilio. A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34, 2016.

[12] Abdulrahman Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 2019.

[13] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the 11th Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, 1996.

[14] AutoRefactor. Available at: http://autorefactor.org/. Accessed on April 03, 2020.

[15] Spartan Refactoring. Available at: https://marketplace.eclipse.org/content/spartan-refactoring. Accessed on April 03, 2020.

[16] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR, 2008.

[17] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. *International Conference on Software Engineering (ICSE)*, 2019.

[18] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference*, ISEC, 2019.

[19] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. GEMS: An extract method refactoring recommender. In *Proceedings of the 28th International Symposium on Software Reliability Engineering*, ISSRE, 2017.

[20] Jehad Al Dallal. Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, 92:105 –120, 2017.

[21] Conor Ryan, J. J. Collins, and Michael O.d Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer Berlin Heidelberg, 1998.

[22] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[23] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[24] William G. Griswold and William F. Opdyke. The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research. *IEEE Software*, 32(6):30–38, 2015.

[25] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, 1992.

[26] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[27] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In *Proceedings of the International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML, 2001.

[28] Barbara Weber, Manfred Reichert, Jan Mendling, and Hajo A. Reijers. Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486, 2011.

[29] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE, 2006.

[30] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006.

[31] Elliotte Harold. *Refactoring HTML: Improving the Design of Existing Web Applications*. Addison-Wesley, 2008.

[32] Mohamed W. 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. *Empirical Software Engineering*, pages 1–43, 2015.

[33] Mark Harman and Bryan F. Jones. Search-Based Software Engineering. *Information and Software Technology*, 43:833–839, 2001.

[34] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST, 2015.

[35] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information Software and Technology*, 51(6):957–976, 2009.

[36] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[37] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *Proceedings of the International Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ, 2008.

[38] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[39] Rodrigo C. Barros, Márcio P. Basgalupp, Ricardo Cerri, Tiago S. da Silva, and André C. P. L. F. de Carvalho. A Grammatical Evolution Approach for Software Effort Estimation. In *Proceedings of the 5th Genetic and Evolutionary Computation Conference*, GECCO, 2013.

[40] Michael I. Jordan and Tom M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[41] Harold E. Driver and Alfred L. Kroeber. *Quantitative Expression of Cultural Relationships*. University of California, 1932.

[42] Joseph Zubin. A technique for measuring like-mindedness. *The Journal of Abnormal and Social Psychology*, 33(4):508–516, 1938.

[43] Robert C. Tryon. *Cluster Analysis: Correlation Profile and Orthometric (factor) Analysis for the Isolation of Unities in Mind and Personality*. Edwards brother, Incorporated, lithoprinters and publishers, 1939.

[44] J. A. Hartigan and Mae C. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society*, 28(1):100–108, 1979.

[45] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[46] Refactoring Browser. Available at: http://wiki.c2.com/?RefactoringBrowser. Accessed on April 03, 2020.

[47] Mariele Cortés, Marcus Fountoura, and Carlos Lucena. Framework evolution tool. *Journal of Object Technology*, 5(8):101–124, 2006.

[48] XRefactory. Available at: http://www.xref.sk/xrefactory/main.html. Accessed on April 03, 2020.

[49] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO, 2013.

[50] Wiem Mkaouer, Marouane Kessentini, Patrice Kontchou, Kalyanmoy Deb, Slim Bechikh, and Ali Ouni. Many-Objective Software Remodularization Using NSGA-III. *Transactions on Software Engineering and Methodology*, 24(3):17:1–17:45, 2015.

[51] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. Multiobjective optimization for software refactoring and evolution. *Advances in Computers*, 94:103–167, 2014.

[52] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 25(3):23:1–23:53, 2016.

[53] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. Search-based refactoring using recorded code changes. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR, 2013.

[54] Marouane Kessentini, Rim Mahouachi, and Khaled Ghedira. What you like in design use to correct bad-smells. *Software Quality Journal*, 21(4):551–571, 2012.

[55] 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*, 105:18–39, 2015.

[56] Rim Mahouachi, Marouane Kessentini, and Khaled Ghedira. A new design defects classification: Marrying detection and correction. In *Proceedings of the Fundamental Approaches to Software Engineering*, FASE, 2012.

[57] Micheal Mohan and Des Greer. Using a many-objective approach to investigate automated refactoring. *Information and Software Technology*, 112:83–101, 2019.

[58] Michael Mohan and Des Greer. Automated refactoring of software using version history and a code element recentness measure. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018.

[59] Michael Mohan and Des Greer. Automated multi-objective refactoring based on quality and code element recentness. In *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering*, 2019.

[60] Abdulaziz Alkhalid, Mohammad Alshayeb, and Sabri Mahmoud. Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm. *Advances in Engineering Software*, 41(10-11):1160–1178, 2010.

[61] Istvan-Gergely Czibula and Gabriela Serban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security*, pages 40–49, 2006.

[62] Istvan-Gergely Czibula and Gabriela Czibula. Clustering based automatic refactorings identification. In *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC, 2008.

[63] Istvan-Gergely Czibula and Gabriela Czibula. Hierarchical clustering for adaptive refactorings identification. In *Proceedings of the International Conference on Automation, Quality and Testing*, AQTR, 2010.

[64] Gabriela Czibula and Gabriela Serban. A programming interface for determining refactorings of object-oriented software systems using clustering. In *Proceedings of the International Conference on Intelligent Computer Communication and Processing*, 2007.

[65] Istvan-Gergely Czibula and Gabriela Czibula. Hierarchical clustering based automatic refactorings detection. *WSEAS Transactions on Electronics*, 5(7):291–302, 2008.

[66] Gabriela Serban and Istvan-Gergely Czibula. Restructuring software systems using clustering. In *Proceedings of the International International Symposium on Computer and Information Sciences*, 2007.

[67] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[68] Vahid Alizadeh, Houcem Fehri, and Marouane Kessentini. Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction. SCAM, 2019.

[69] Ayaka Imazato, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Finding extract method refactoring opportunities by analyzing development history. In *Proceedings of the 41st Annual Computer Software and Applications Conference*, COMPSAC, 2017.

[70] Yasemin Kosker, Burak Turhan, and Ayse Bener. An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003, 2009.

[71] Monvorath Phongpaibul and Barry Boehm. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM, 2007.

[72] Sonika Jindal and Gauri Khurana. In *Proceedings of the International Conference on Advances in Recent Technologies in Communication and Computing*, ARTCom.

[73] Jehad Al Dallal. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 54(10):1125–1141, 2012.

[74] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[75] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[76] Juan J. Durillo and Antonio J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[77] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[78] David M. W. Powers. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2:37–63, 01 2011.

[79] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[80] Thainá Mariani, Giovani Guizzo, Silvia R. Vergilio, and Aurora T. R. Pozo. Grammatical evolution for the multi-objective integration and test order problem. In *Genetic and Evolutionary Computation Conference*, GECCO, 2016.

[81] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[82] Thainá Mariani and Silvia R. Vergilio. Uma proposta de geração automática de algoritmos de refatoração. In *Workshop de Engenharia de Software Baseada em Busca*, WESB, 2018.

[83] Thainá Mariani, Giovani Guizzo, Silvia R. Vergilio, and Aurora T. R. Pozo. Automatic design of algorithms applied to the multi-objective tsp problem. In *Encontro Nacional de Inteligência Artificial e Computacional*, ENIAC, 2016.

[84] Thiago N. Ferreira, Thainá Mariani, and Silvia R. Vergilio. Reviewing six years of brazilian workshop on search-based software engineering. In *Workshop de Engenharia de Software Baseada em Busca*, WESB, 2016.

APPENDIX A – A SYSTEMATIC REVIEW ON SEARCH-BASED REFACTORING

This appendix presents the paper: a systematic review on search-based refactoring.

Information and Software Technology 83 (2017) 14–34

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A systematic review on search-based refactoring

Thainá Mariani*, Silvia Regina Vergilio

Computer Science Department, Federal University of Paraná (UFPR), CP 19:081, CEP: 81531-970 Curitiba, Brazil

ARTICLE INFO

Article history:
Received 6 May 2016
Revised 28 October 2016
Accepted 28 November 2016
Available online 29 November 2016

Keywords:
Search-based software engineering
Refactoring
Evolutionary algorithms

ABSTRACT

Context: To find the best sequence of refactorings to be applied in a software artifact is an optimization problem that can be solved using search techniques, in the field called Search-Based Refactoring (SBR). Over the last years, the field has gained importance, and many SBR approaches have appeared, arousing research interest.

Objective: The objective of this paper is to provide an overview of existing SBR approaches, by presenting their common characteristics, and to identify trends and research opportunities.

Method: A systematic review was conducted following a plan that includes the definition of research questions, selection criteria, a search string, and selection of search engines. 71 primary studies were selected, published in the last sixteen years. They were classified considering dimensions related to the main SBR elements, such as addressed artifacts, encoding, search technique, used metrics, available tools, and conducted evaluation.

Results: Some results show that code is the most addressed artifact, and evolutionary algorithms are the most employed search technique. Furthermore, most times, the generated solution is a sequence of refactorings. In this respect, the refactorings considered are usually the ones of the Fowler's Catalog. Some trends and opportunities for future research include the use of models as artifacts, the use of many objectives, the study of the bad smells effect, and the use of hyper-heuristics.

Conclusions: We have found many SBR approaches, most of them published recently. The approaches are presented, analyzed, and grouped following a classification scheme. The paper contributes to the SBR field as we identify a range of possibilities that serve as a basis to motivate future researches.

1. Introduction

A software product is frequently evolving to address different functionalities. These evolutions may make the software design more complex and different from the original one, decreasing the software quality. In this sense, a meaningful effort is devoted to the software maintenance phase, where the software refactoring activity can be used. This activity is used to improve the software quality, by improving some quality attributes, such as understandability, maintainability, extensibility and performance [1]. Moreover, such an activity can be also used in the early software engineering phases, such as software development, design and re-engineering.

Software refactoring is performed by changing the software structure without modifying its external behavior. It applies a set of meaning-preserving restructurings, called refactorings [2], which are very simple operations performed to change a software artifact, such as move a method, move a field and extract a class. Refactorings were originally proposed in the context of object-oriented software [3], where some catalogs of refactorings exist [2,3]. Since then, software refactoring has been applied in different contexts, such as aspect-oriented software, software product line, and in distinct artifacts such as code, models, documentations, requirements and so on.

Finding a good sequence of refactorings to be applied in a software artifact is considered a hard task [P66], since there is a wide range of refactorings and the ideal sequence is correlated to different quality attributes to be improved. In fact, this is an optimization problem that can be solved by search techniques in the field known as Search-Based Software Engineering (SBSE) [4]. Search algorithms allow the addition of several metrics to compute the solution quality. This is one of the factors that makes the use of search techniques for software refactoring very attractive. Furthermore, these algorithms are capable to automatically find, in a huge space, solutions that a software engineer might not have been able to think of [4].

We found successful SBSE approaches that show the applicability of search techniques in a wide variety of problems from diverse software engineering areas in many ways, using different

* Corresponding author.
E-mail addresses: tmariani@inf.ufpr.br, marianithaina@gmail.com (T. Mariani), silvia@inf.ufpr.br (S.R. Vergilio).

http://dx.doi.org/10.1016/j.infsof.2016.11.009
0950-5849/

search algorithms, such as Genetic Algorithms and other ones from the operation research area [5]. In this way, the area that applies search-based techniques to perform software refactoring is called *Search-Based Refactoring (SBR)*. This area has been growing over recent years.

A growing number of SBSE works is reported by reviews found in the literature. We can find surveys addressing specific SBSE areas such as software test [6–8], software design [9], and requirements [10]. None of them addresses specifically SBR. General SBSE surveys [4,5,11,12] provide an overview of the SBSE field, by discussing research directions on SBSE, presenting the software engineering activities and search-based algorithms. Such works include software refactoring but do not explore specific SBR characteristics in depth, like the addressed artifacts, encoding, available tools, used metrics and evaluations conducted. Existing surveys [1] and systematic reviews [13] on software refactoring do not consider search-based approaches.

To contribute to the SBR area, this paper presents results from a systematic review, aiming at finding specific details about the existing SBR approaches. A systematic review is a study to identify, evaluate and interpret available researches related to a particular research question or topic area [14]. It is a good technique to extract information about the papers and identify research trends, since it presents a systematic methodology to be followed.

To conduct this systematic review we followed Kitchenham's guidelines [14]. We planned the review by defining the research questions, the search string, the sources for searching and the selection criteria. We searched for primary sources and, after the final selection, the data was extracted in order to answer the research questions. Detailed results about the extracted data are presented, such as the most used artifacts, metrics and refactorings. In this way, this paper adds to the contributions of existing SBSE surveys focusing specifically on software refactoring and: (i) offering a more complete and updated list of works obtained systematically and covering the last sixteen years; (ii) providing a classification schema and grouping works, considering specific SBR characteristics that are not addressed in related work; (iii) identifying in the found works, the main contributions and best practices that can point out trends in the area, as well as, gaps and limitations that can suggest need of further study and research opportunities.

This paper is organized as follows. Section 2 reviews background on software refactoring and search-based techniques. Section 3 discusses related work. Section 4 describes how the review was conducted. Section 5 presents and analyses the obtained results. Section 6 describes trends and research opportunities. Section 7 contains the threats to validity of our results. Finally, Section 8 concludes the paper.

2. Background

This section introduces the fields explored in our research: software refactoring, search algorithms and Search-Based Refactoring (SBR).

2.1. Software refactoring

The term refactoring was introduced in 1990 by Opdyke and Johnson [3]. They proposed a set of meaning-preserving restructurings to be applied in C++ programs. Each of these restructurings was called a refactoring [15]. The term was popularized by Fowler [2] after the publication of his book. Since then, the term has also been used to denote the whole process of changing a software artifact and, gained different meanings and definitions. In this paper, we distinguish both meanings according to Fowler definitions [2], by using here the terms refactoring and software refactoring, as below:

> Refactoring: "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"
>
> Software refactoring: "The activity of software restructuring by applying a set of refactorings without changing its observable behavior"

Refactorings were initially proposed for C++ and Java programs. In such context, some well-known refactoring catalogs can be found in the literature [2,16]. However, over time, refactorings have been created for other program languages, such as AspectJ [17], Smalltalk [18] and PHP [19]. Furthermore, they have been also proposed and applied for other artifacts and contexts, such as models [20,21], software product lines [22], databases [23] and HTML [24].

According to [2], software refactoring can help to improve the design of software. Consequently, many quality attributes can be improved, such as reusability, flexibility, and understandability. It can bring many benefits, for example, by making a program more understandable, may be easy to find bugs and to obtain a faster development process [2].

The software refactoring activity is commonly used in software evolution, and in the maintenance phase, where the lack of software structure is more evident and expensive. However, it is also used in other phases, such as re-engineering, design and development [1,16]. For instance, in the software design phase, software refactoring can be used to improve a preliminary design, helping to obtain a final one [9]. In addition, some software development methods use software refactoring as part of the process, such as Test-Driven Development and Agile Software Development [1].

According to [1], the software refactoring activity includes the following tasks:

1. *Identify where the software should be refactored:*Firstly, it is necessary to determine the artifact to be refactored. Examples of artifacts are codes, models, and requirements. After that, it is necessary to identify the elements that should receive a refactoring. In this sense, there are different directions to identify refactoring opportunities. One of them is the existence of bad smells, which are metaphors to describe software patterns that may be associated with a bad design and a bad programming [2]. Another direction may be indicated when a meaningful effort is being wasted to maintain and understand a software. Furthermore, this identification task can be highly dependent of the application domain.
2. *Determine which refactorings should be applied to the identified places:* In this task, it is necessary to determine the refactorings that should be applied to the elements identified in the previous task. Usually, these two tasks are coupled and can be performed at the same time.
3. *Guarantee that the applied refactoring preserves behavior:* This task refers to the use of methods to guarantee behavior preservation of the software after the refactoring application. The original definition of behavior preservation states that the output values should be the same before and after a refactoring, when using the same set of inputs. Moreover, depending on the domain, other aspects can be used to ensure behavior preservation, such as execution time, memory constraints and power consumption.
4. *Apply the refactoring:* This task consists of applying the defined refactorings in the identified elements using the established methods to guarantee behavior preservation.
5. *Assess the effect of the refactoring on quality characteristics of the software:* This task refers to the assessment of the impact of the applied refactorings on software quality attributes. It can be performed, for example, by analyzing manually the impact on the software understandability from the point of view of

the user. Furthermore, it can also be assessed using some techniques, such as metrics to measure different aspects related to the quality attributes.

6. *Maintain the consistency between the refactored artifact and other software artifact:* This task addresses the use of mechanisms to maintain the consistency of the refactored artifact and the other software artifacts. For instance, after the refactoring of a program, it is necessary to update the related artifacts, such as documentation, model and tests.

2.2. Search algorithms

The optimization field usually deals with hard problems that can be efficiently solved by search algorithms. Such algorithms can be divided into two groups. The first one includes classical techniques from the operation research field, such as branch and bound algorithm, and linear programming. The classical algorithms are deterministic, i.e., they generally determine only one solution. The other group includes meta-heuristics, the most preferred in the SBSE field [11]. A reason for this is the nature of the software engineering problems: they are real world problems, and generally are related to objectives that cannot be characterized by a set of linear equations, and they are not tractable by deterministic methods.

There are meta-heuristics, such as greedy algorithms, that work by creating a solution, selecting at each step the local optimal choice from a set of candidates. Other ones generate a set of candidate solutions, called neighborhood, obtained by applying transformations in the current solution. The quality of the solutions is evaluated, and based on that, a candidate solution can be selected to be the current one. When the stopping criteria is reached, the current solution is returned. The most used search-techniques of this type are hill climbing and simulated annealing. They differ in the procedures for the generation of candidate solutions and in the selection of the current one [25].

There are other techniques based on the concept of population that manipulate a set of solutions. At each iteration of the algorithm, a population is generated and merged to the current one based on a selection procedure. The most common search techniques are the evolutionary algorithms, being genetic algorithms (GAs) and genetic programming (GP) the most used in SBSE [11]. They are based on the evolution of species, where the best individuals (solutions) are selected to be reproduced by applying crossover and mutation operators. At each generation (iteration), the algorithm selects the best individuals (the ones containing the best quality, being parents or off-spring) to survive for the next generation. This is performed until a stopping criterion is achieved, at which point the best solution of the current population is returned [25]. We can also mention other bio-inspired algorithms such as Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and so on.

According to Harman [5], two essential keys ingredients are necessary to apply a search technique to a software engineering problem. The first ingredient is the representation (encoding) of a solution, which is required to allow symbolic manipulation. A common representation is given by a vector of binary values, a vector of real values or a vector of a permutation sequence [25]. For example, if a solution represents the methods of a class to which a refactoring should be applied, a vector of binary values can be used, where each method is represented by a vector element whose value must be 0 or 1. If it is 1, the refactoring should be applied, otherwise, it should not. The second ingredient is the objective function, defined in terms of the representation, to evaluate the quality of the solutions. Hence, the problems usually are associated with a wide range of software metrics that can be used.

Multi-objective problems are very common problems associated with two or more objectives. This kind of problem can be solved using a single-objective algorithm with a weighted sum. Hence, each objective to be optimized is related to a defined weight. Furthermore, this problem can also be solved with multi-objective algorithms. The main difference between multi -objective and single-objective optimization is the number of final solutions. In multi-objective optimization, different good solutions exist, because usually, the objectives are in conflict and to optimize one, the others are compromised. Hence, a set containing all found solutions compose the called Pareto front. These solutions are called non-dominated, because none of them is considered better than the others [26] considering all the objectives. Some widely used multi-objective algorithms in SBSE are the evolutionary ones: Non-dominated Sorting Genetic Algorithm II (NSGA-II) [27] and Strength Pareto Evolutionary Algorithm 2 (SPEA2) [28].

Search-based algorithms can be applied to different areas of software engineering with relative ease, and due to this, the number of works is on the increase. We can find a great number of SBSE surveys [5,12]. There are some specific areas that have their own surveys: software test [6–8], software design [9] and requirements [10]. Our paper focuses on the area of software refactoring, subject of the next subsection.

2.3. Search-based refactoring

The field of SBR is devoted to apply search algorithms in the software refactoring activity. Existing approaches commonly use search techniques to suggest or apply refactorings in artifacts. This kind of approaches usually encompasses most of the software refactoring tasks. Fig. 1 shows a generic overview of such approaches.

An approach has as input the artifact to be improved. It is converted in a representation to be used by the search-techniques. This representation can be the artifact itself or a more abstract representation, such as graphs for representing code artifacts. The search technique can optionally receive as input refactorings, metrics and any other additional information to guide the process. As output, the search technique returns a solution (or a set of solutions) for the problem. In this sense, a solution representation has to be defined for being manipulated by the search technique. An example of a solution is a sequence of refactorings where its representation can be, for instance, a vector containing the refactorings. Refactorings can be mapped in the solution representation and then, applied in the artifact representation. The provided metrics are used in the fitness function to evaluate the solutions quality. Regarding the additional information, it may be given to help in the optimization process, for example, an instance of a well-designed artifact to let the approach learn what such an artifact looks like.

3. Related work

In the literature, we can find works about software refactoring in general. The work of Mens [1] provides a review of the software refactoring tasks, artifacts to be used, formalisms and techniques to be applied, and essential issues to be considered in the development of software refactoring tools. In a most recent paper, Abebe and Yoo [13] describe results from a systematic review conducted to identify the trends, opportunities and challenges of the field. The works are described and gaps identified considering some groups of paper and dimensions like surveys, tools, metrics, design patterns, bad smells, programming language, and so on. Such works provide an overview of many aspects related to software refactoring, however, the goal is not to present a review of existing papers, and both of them do not address search-based approaches.

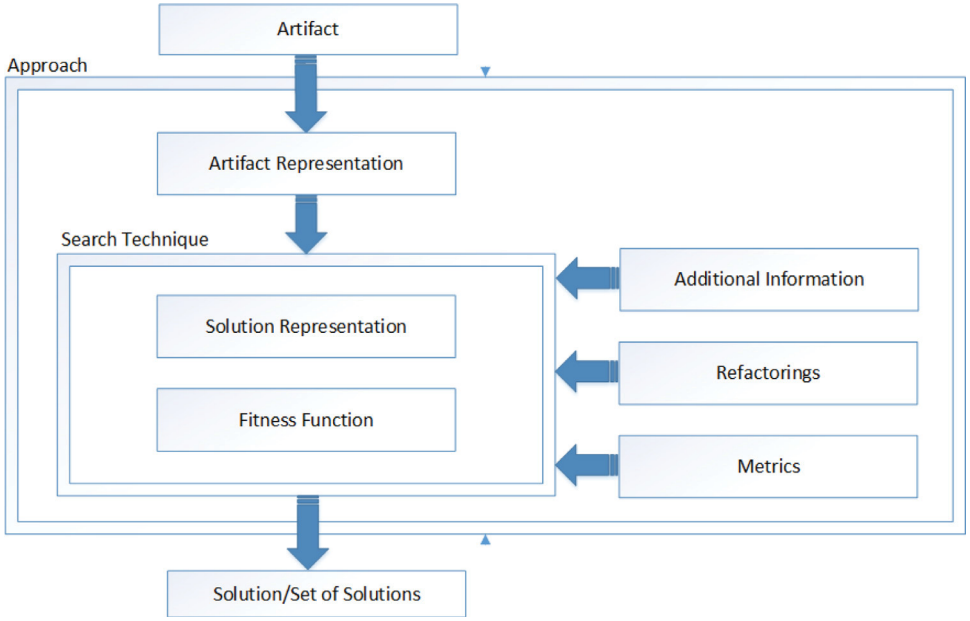


Fig. 1. Generic overview of SBR approaches.

As mentioned before, surveys in the SBSE field [5,11,12] (2007, 2009 and 2012) address different areas of software engineering, and report applications of search-based algorithms on software bug fixing, project management, planning and cost estimation, refactoring, software slicing, software comprehension, service-oriented software engineering, compiler optimization, quality assessment, and so on. They analyze the most used search-based algorithms and point research directions on SBSE. However, those surveys are not dedicated exclusively to software refactoring. Among the existing surveys on specific areas of software engineering, the survey on search-based design [9] is the most related to ours. It focuses on approaches to generate design from requirements and also addresses search-based software clustering and SBR, since they can be applied to improve an existing design. The survey presents more works on SBR than the general SBSE surveys, however, the work is not very new, it was published in 2010. Furthermore, the author do not conduct a systematic review and do not provide detailed information about the papers.

Our study differs from the above studies since it presents results from a systematic review focused on SBR. In this way, we followed a protocol and performed a search in most used databases covering the last sixteen years, which were essential for the development of the software refactoring area. As a consequence, a more complete and updated list of papers is discussed. In addition to this, different aspects not addressed in related work are analyzed with some deep, such as artifacts, encoding, tools and kind of evaluation conducted. Such an analysis allows the identification of successful initiatives as well as, gaps in the SBR, which can point out new research opportunities and trends.

4. Research method

We performed this systematic review following the three phases presented in the guidelines of Kitchenham [14]. The first phase, planning the review, creates the research protocol to be followed. In the second one, conducting the review, the search is performed, the papers are selected, and their data are extracted and synthesized. The third phase, reporting the review, specifies the dissemination mechanisms and formats the main report. In this sense, such a phase resulted in the elaboration of this paper. In this section, we describe how the first two phases were executed.

4.1. Planning the review

As we mentioned in the last section, this systematic review was motivated by the lack of reviews emphasizing SBR. The main goal of the review is to provide an overview of the SBR field, and a deep analysis of the search-based approaches elements, such as artifacts, solution representations, search techniques, metrics, systems and evaluations methods used in the experimentation. We considered here SBR approaches, as the one presented on Fig. 1, where the goal is to suggest or apply refactorings aiming at improving the software quality. In order to reach our goals, some research questions were elaborated. They are presented next.

4.1.1. Research questions

We followed the PICOC (Population, Intervention, Comparison, Outcome, and Context) structure [29] to define our research questions. Such a structure encompasses the attributes to be considered when defining the research questions of a systematic review.

- (P) Population: search-based approaches for software refactoring.
- (I) Intervention (what will be observed): main elements associated (Fig. 1) to software refactoring: artifacts, solution representation, search-techniques applications, and evaluation conducted.
- (C) Comparison: Although comparisons are not applicable to systematic reviews, some were conducted between primary studies and are provided in the results section.
- (O) Outcome: a characterization of the SBR field.
- (C) Context: domain of SBR.

Our review aims at answering the research questions presented in Table 1. They were defined with the goal of covering the main aspects of SBR approaches. RQ1–RQ7 are related to the SBR aspects that are part of the software refactoring tasks. In this sense, we intend to discover the type of artifacts improved and evaluated by the approaches, how their elements are selected, the methods used to guarantee behavior preservation, the considered

Table 1
Research questions.

Identify where the software should be refactored
RQ1: What type of artifact is refactored, and how is the artifact represented?
Determine which refactorings should be applied to the identified places
RQ2: What are the considered refactorings?
Guarantee that the applied refactoring preserves behavior
RQ3: What are the methods employed to preserve behavior?
Apply the refactoring
RQ4: Are the refactorings applied in the artifact?
Assess the effect of the refactoring on quality characteristics of the software
RQ5: What are the most common metrics used to assess the software quality during the search?
Maintain the consistency between the refactored artifact and other software artifact
RQ6: Does the approach take into consideration the consistency with other software artifacts?
Search-based formulation
RQ7: What are the most common obtained solutions and their representations?
RQ8: What are the most common used search-based algorithms?
RQ9: Is any additional information used to guide the optimization process?
Evaluation aspects
RQ10: What are the used evaluation methods?
RQ11: What are the most common used systems?
RQ12: What are the most common used refactoring tools?

refactorings and how they are applied, metrics used in the fitness evaluation, and how the consistency with the other artifacts is treated. RQ8–RQ10 cover aspects related to the search-based formulation, such as the search algorithms, the solutions produced and how they are represented to allow symbolic manipulation by the search algorithm, and if any additional information is used to guide the optimization process, such as the user-preference or an ideal refactoring. Answers for RQ11–RQ13 identify ways to evaluate the approaches and main experimental elements: tools, systems and evaluation methods.

4.1.2. Search string

To build the search string (Fig. 2), we defined the main terms (search-based technique and refactoring) and synonymous identified by analyzing search strings used in related work [1,5]. We defined the start year of the search as 2000, since the term SBSE was coined by Harman and Jones in 2001 [4], and some surveys report a significant number of SBSE papers appeared in 2000 [1].

To verify the accuracy of the keywords chosen to build the search string, we used a control group composed by a set of relevant studies on SBR, found in the SBSE repository.[1] Then, we identified that our search string found all papers of the control group.

4.1.3. Selection criteria

We established a set of inclusion and exclusion criteria used to select the primary sources. They are presented in Table 2. Firstly, the papers were selected according to the inclusion criteria. After that, the exclusion criteria were applied in the selected papers. A paper was included or excluded by reading it in the following order until no doubts left about its selection: title, abstract, introduction, conclusion, and the entire paper.

4.2. Conducting the review

The search started on October 22, 2016, and finished on October 23, 2016. The review was conducted in a set of steps presented in Fig. 3. As presented, the first step was the selection of

[1] http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/. A repository that contains a comprehensive set of SBSE works, updated by the SBSE community.

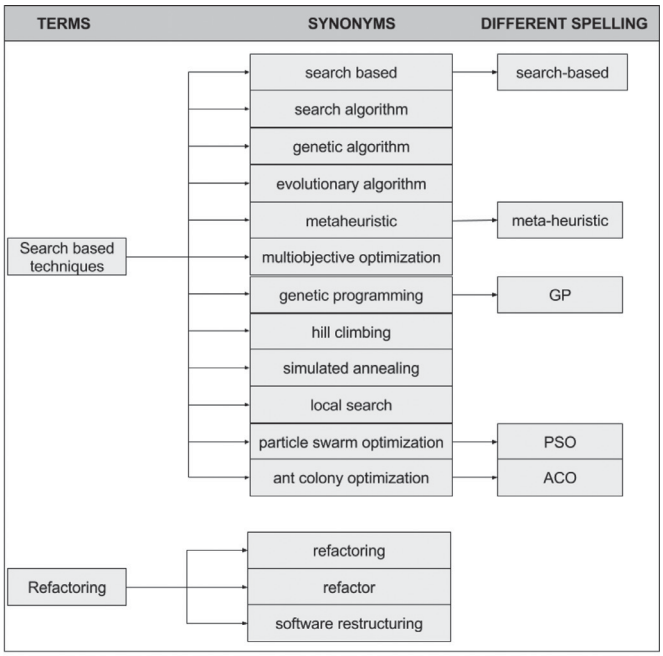


Fig. 2. Search terms and search string.

Table 2
Inclusion and exclusion criteria.

Inclusion criteria
Papers that use search-based techniques to apply or suggest refactorings with the goal of improving the quality of an artifact given as input.
Exclusion criteria
Out of scope;
Not available on-line;
Not in English;
Abstracts, posters, technical reports, thesis, keynotes, doctoral symposiums, books, conference reviews and patents.

Table 3
Number of papers returned by each source.

Source	URL	Number of papers
Scopus	http://www.scopus.com	129
Web of Science	http://www.webofknowledge.com	53
IEEE	http://ieeexplore.ieee.org	28
Springer	http://www.springerlink.com	35
ACM	http://dl.acm.org	25
Science Direct	http://www.sciencedirect.com	10
Wiley Online Library	http://onlinelibrary.wiley.com	3
Total		283

the primary sources by using the search string in the selected engines, considering the title, abstract, and keywords. The engines for performing the search are online repositories chosen due to their popularity and because they provide many leadings software engineering publications and include major well-known SBSE conferences. The selected sources are: (i) ACM Digital Library; (ii) IEEE Xplore; (iii) Springer; (iv) Scopus; (v) Science Direct; (vi) Web of Science; and (vii) Wiley Online Library. Table 3 shows the number of papers returned by each engine. As we can see, 283 papers were

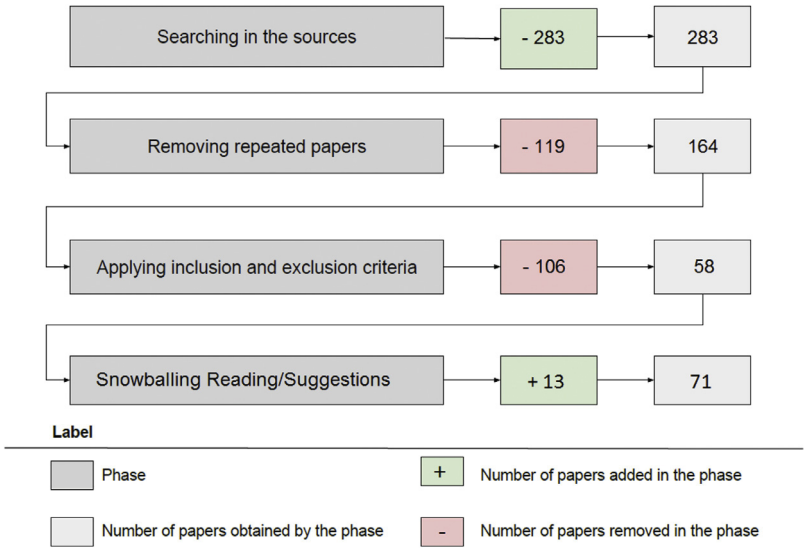


Fig. 3. Number of papers returned by each step.

found. After that, repeated papers were removed and the selection criteria were applied, totalizing 58 papers. Then, we performed a snowballing reading by searching in these 58 papers other ones attending the defined criteria. In this phase, we also included papers suggested by the experts. Finally, 71 papers were selected to the next phase of data extraction. The selected papers are presented in the end of this paper ("Primary Sources" Section).

Regarding the papers removed by applying the inclusion and exclusion criteria, we found works that just mention the terms but are not related to software refactoring. Some works apply refactorings to improve a search-based algorithm or improve a software artifact in other ways that are not considered refactorings. Some excluded papers are related to SBR, but they are out of the scope of this systematic review, since these papers do not use search-based techniques to apply or suggest refactorings. For example, papers using search-based techniques to detect bad smells [30–32]. An element containing a bad smell is a good candidate for receiving a refactoring. Thus, such kind of approach can be useful for the software refactoring activity, but they are not the focus of our review. Other examples are papers analyzing metrics or representations used in SBR. In this context, relationships between cohesion metrics are investigated [33]. Simons et al. [34] investigate the capability of metrics in the assessment of software quality attributes. In such work, a review was done to find the metrics used in the SBR literature. Other papers proposed a graph transformation representation for software architectures [35] and a representation to be used by the Ant Colony Optimization (ACO) algorithm [36].

The data of the selected primary sources was extracted by reading the papers, and stored in a spreadsheet. A data extraction form was used including: paper title, publication year and venue, authors, database where each work was found, and some additional attributes. Such attributes were extracted in order to answer the research questions and to group the found studies according to the classification schema presented in Table 4.

The results were synthesized and analyzed. Answers to our questions were obtained. Our main findings and answers are presented in next section.

5. Results

In this section, we first give an overview of the selected studies with respect to some basic information about the authors, years and publication venues. Then, we answer each research question based on the extracted information.

5.1. Basic information

We first analyzed the main venues where the papers have been published. Fig. 4 shows the main conferences, journals, and workshops, along with the corresponding number of published papers. Only venues containing two or more papers are presented.

The International Symposium on Search-Based Software Engineering (SSBSE) and Genetic Evolutionary Computation Conference (GECCO) are the preferred events, which are dedicated to the SBSE field. Other conferences, not dedicated to SBSE, are related to software maintenance, International Conference on Software Maintenance (ICSM) and European Conference on Software Maintenance and Reengineering (CSMR). We also can see that the papers are published in journals of software engineering areas. 51 papers (70%) were published in conferences and workshops, and 22 (30%) in journals.

Regarding the authors, we identified that 104 authors have been published in 48 different institutions. In addition, we analyzed the authors that published more papers in the SBR field. In this sense, Fig. 5 presents authors that published more than 5 papers. Two of them have more than 15 papers published. They are: Marouane Kessentini and Mel Ó Cinnéide. This crude measure does not take into consideration characteristics of the venues in which the papers were published.

Fig. 6 shows the number of papers published per year. Based on this chart, the number of publications stayed stable between 2006 and 2009, increasing significantly in 2011 and 2014. Despite a little decreasing in 2013, in the last two years, the number has increased again. This shows that the interest in the field keeps growing.

5.2. RQ1 – artifacts and representations

Table 5 shows the artifacts used, the number of papers (NP) using them, and the references. Most papers aim at optimizing code, some of them focus on models, and a few optimize other types of artifacts, like grammar files [P23,P34] and build scripts [P20]. Most of the papers that optimize code are focused on the Java language. Furthermore, some papers also optimize C and C++ code. Regarding the optimized models, class diagrams are the most used.

Table 4
Classification schema.

Dimension	Description
Artifact	The artifact used as input to be improved by the approaches.
Artifact Representation	The representation used by the approach to represent the artifact.
Refactorings	The refactorings used by the approaches to be applied in the elements.
Behavior preservation	The methods used to guarantee behavior preservation.
Refactoring application	The way the refactorings are applied, eg. manually or automatically.
Metrics	The metrics used in the fitness function to evaluate the solutions.
Consistency	The methods used to maintain the consistency with related artifacts.
Solution	The solution type given as output by the search technique.
Solution Representation	The representation used by the search technique to manipulate the solution.
Search Technique	The search techniques used by the approaches.
Additional Information	Additional information used to guide the optimization process.
Refactoring Tools	Tools used in the context of software refactoring.
Evaluation Methods	The evaluation methods used to assess the obtained results.
Systems	The systems used in the experimentation phase.

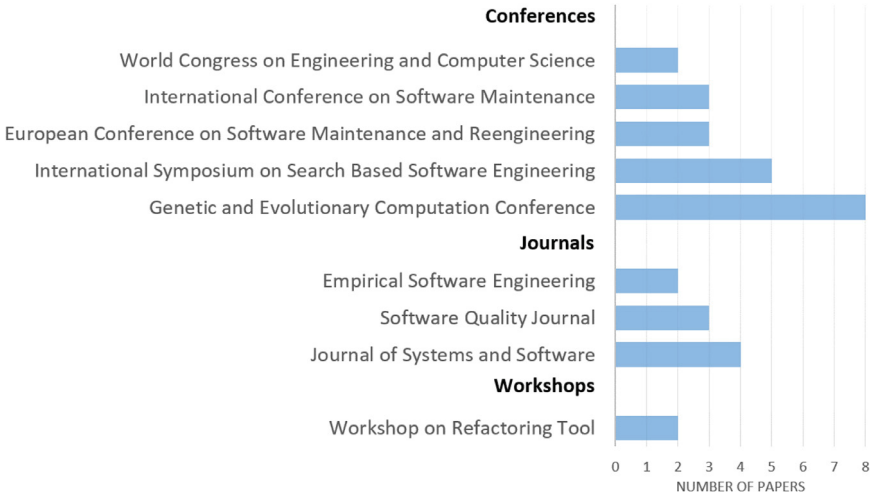


Fig. 4. Main venues where SBR approaches have been published.

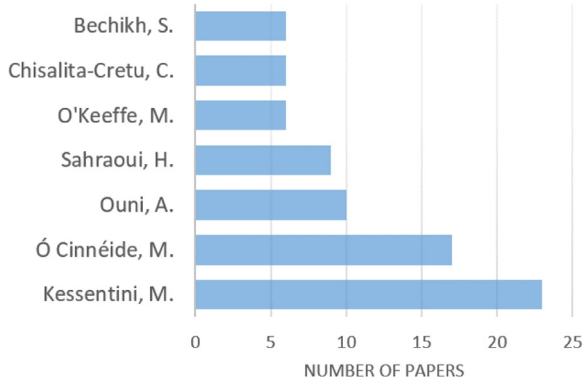


Fig. 5. Authors with the greater number of publications in SBR.

Table 5
Artifacts used by the papers.

Artifact		NP	References
Code	Java	51	[P2,P3,P7–P15,P17,P21,P22,P24,P29–P33,P36,P38,P41–P45,P47–P69,P71]
	C/C++	4	[P4,P6,P16,P35]
	Not mentioned	6	[P25,P26,P28,P37,P40,P70]
Model	Class diagram	6	[P5,P18,P19,P27,P39,P46]
	Activity diagram	1	[P39]
	Other	2	[P1,P25]
Other		3	[P20,P23,P34]

Table 6
Artifact representations used by the papers.

Representation	NP	References
Artifact	49	[P1–P12,P15,P16,P18,P19,P23,P24,P26,P28–P33,P35,P35–P39,P41–P44,P49,P56,P58–P66,P68,P70,P71]
Abstract Syntax Tree (AST)	10	[P14,P17,P47,P48,P51–P55,P67]
Graph	6	[P21,P22,P27,P40,P46,P57]
Other	6	[P13,P16,P25,P50,P64,P69]

One of the works optimizes class diagrams and also activity diagrams [P39]. Few works do not mention the language or model addressed.

The types of representations used to manipulate the artifacts are presented in Table 6. It can be the artifact itself (49 papers) or a more abstract representation. Some works do not mention what is the representation, but it is possible to guess that the artifact is used.

The second most used artifact representation is the Abstract Syntax Tree (AST), which is a representation of the abstract syntactic structure of the program to be refactored. It receives as input Java code and a second level representation model called Java Program Model (JPM). From JPM, metric values are determined and refactoring preconditions are checked. JPM contains information regarding the interdependence of attributes, methods, constructors and structural information, such as the number of methods per class. By analyzing the JPM it is possible to determine which refac-

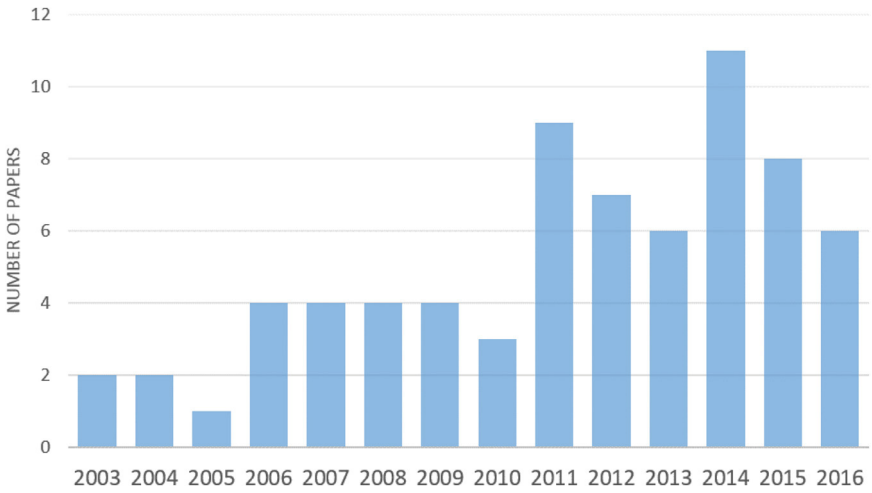


Fig. 6. Number of SBR papers published per year.

Table 7
Type of refactorings used by the papers.

Refactorings	NP	References
Fowler's catalog	51	[P5,P7–P14,P17–P19,P21,P22,P24,P27–P29,P31–P33,P36,P37,P39,P41–P63,P66,P68–P71]
Object-Oriented refactorings	27	[P13,P14,P17–P19,P29,P30,P32,P33,P37,P39,P45–P55,P58,P59,P62,P63,P65]
Design patterns	5	[P3,P27,P32,P33,P67]
Domain dependent	12	[P1,P4,P6,P16,P20,P23,P26,P34,P35,P37,P40,P64]

torings can be legally applied. Although, the refactorings are applied in the AST. At the end, the AST is translated in Java source code [P53].

The third most used artifact representation is a graph. We identified three types of graphs used in the papers. Control Flow Graph (CFG) [P21,P22,P40] and Call Graph [P57] are used to capture the information about refactorings, metrics analysis and code smell measurements in Java programs. Design Graph [P27] is used to represent a class diagram, where the classes are represented by vertices and the relationships between them are the edges.

Other representations include parse trees [P13,P50], bipartite network [P69], and specific representations created based on the problem [P25].

5.3. RQ2 – refactorings

The refactorings were divided into three main categories: refactorings of the Fowler's catalog [2], other kind of object-oriented refactorings, and refactorings associated with design patterns. Table 7 shows the number of papers (NP) and the references of each category.

The most used refactorings are the ones of the Fowler's catalog, they are used by 51 papers. Moreover, 27 different refactorings of such catalog are employed. All of them are applied in code and some are also used in models. Table A.16, in appendix, presents a description of those refactorings, the number of papers (NP) using them and the type of artifact for which they are applied or suggested.

The second category is composed by other types of object-oriented refactorings, they are a total of 18 refactorings used by 27 papers. Table A.17 shows more details about them. Since they do not compose any catalog, their functionality was extracted from the related studies.

Another category is related to studies using design patterns as refactorings. Design patterns [37] are documented solutions commonly used by developers to solve recurrent problems in a specific context. The design patterns Template Method, Decorator, Abstract Factory and Factory Method are used as refactorings in the papers. One of the papers [P27] uses a mini-transformation as a refactoring, which is an operation performed in order to introduce part of a design pattern. 12 papers use other types of refactorings applied for a very specific domain.

5.4. RQ3 – methods to preserve behavior

To answer such a question, we searched for papers that use methods for behavior preservation. The methods found are presented in Table 8, along with the number of papers and their references.

According to the results of Table 8, 25 (35%) papers present no evidence of behavior preservation. Among the papers using some method, most of them use the functions proposed by Opdyke [16]. The author presents a set of preconditions and postconditions that must be checked before and after the refactoring of an object-oriented software. These functions were defined in order to achieve the next seven properties:

1. **Unique superclass:** A class must have at most one direct superclass in order to avoid cycles in an inheritance hierarchy.
2. **Distinct class names:** The classes must have a unique name;
3. **Distinct member names:** All member variables and functions of the same class must have unique names;
4. **Inherited member variables not redefined:** A member variable inherited from a superclass must not be redefined in any of its subclasses;
5. **Compatible signatures in member function redefinition:** Signatures must be the same after a function redefinition;
6. **Type-safe assignments:** The type of an expression assigned to a variable must be an instance of the variable type;
7. **Semantically equivalent references and operations:** If a program is called twice, with the same set of inputs, the output must be the same.

The second most used method was proposed by Cinnéide [38], and it is also for the object-oriented context. This method uses

Table 8
Methods used by the papers to preserve behavior.

Methods	NP	References
No evidence of behavior preservation	25	[P3–P12,P16,P18,P19,P23,P24,P27,P28,P38,P43,P49,P64,P67,P69–P71]
Opdyke's Functions [16]	21	[P1,P15,P17,P21,P22,P29–P31,P39,P41,P42,P44,P56–P63,P68]
Cinnéide's Functions [38]	12	[P2,P14,P45–P48,P50–P55]
Domain specific	10	[P20,P25,P26,P34–P37,P40,P65,P66]
Do not mention the method	3	[P13,P32,P33]

analysis functions to extract information from the artifact being refactored. Then, these functions are used to specify the preconditions and postconditions of the refactorings.

Ten papers present methods to preserve behavior specific to a domain and context. Many of these methods are proposed in the papers. This happens for uncommon domains where refactoring catalogs are not popular, such as C programs [P35], grammar files [P34] and build script [P20]. Three other papers also employ behavior preservation methods but do not mention which one.

In addition to such methods, there are papers using a metric named "Semantic Coherence" [P60] in the search algorithm objective function, in order to improve the semantic property. Section 5.6 presents more details about this metric and papers using it.

5.5. RQ4 – refactoring application

To answer this question we first classify the approaches in two types: direct and indirect [P24]. In a direct approach, the refactorings application is automated, since they are applied directly to the artifact. In that case, it is easy to ensure behavior preservation because the refactorings are legally applied. On the other hand, in an indirect approach, a sequence of refactorings is the solution optimized, and later, this sequence is applied to the artifact. Hence, the artifact is indirectly optimized.

We found that only 40% (28) of the papers [P3,P4,P13,P14, P16,P17,P20–P23,P26,P27,P33–P35,P37,P46–P55,P64,P65] automate the task of applying the refactorings to the artifact by using a direct approach. The other 60% (43) [P1,P2,P5–P12,P15,P18,P19,P24,P25,P28–P32,P36,P38–P45,P56–P63,P66–P71] are indirect approaches that suggest refactorings that can be latter applied by the user as he/she prefers.

One of the reasons leading to the great number of indirect approaches may be the difficulty to ensure behavior preservation. In this sense, we also found that only 30% (21) of the papers [P13,P14,P17,P20–P22,P26,P33–P35,P37,P46–P48,P50–P55,P65] present direct approaches that consider the behavior preservation. Furthermore, 35%(25) of the papers [P1,P2,P15,P25,P29–P32,P36,P39–P42,P44,P45,P56–P63,P66,P68] present indirect approaches that also consider the behavior preservation, since it can increase the chances for a user to legally apply the suggested refactorings.

5.6. RQ5 – metrics

Table 9 shows the used metrics, the number of papers (NP) using them, and the references. The papers commonly use more than one of the presented metrics.

The most used metric is the number of bad smells, employed in 18 papers (24%). The main bad smells considered are *blob, feature envy, long parameter list* and *lazy class* [2]. Number of modifications is also one of the most used metrics. It is used to reduce the software refactoring effort by minimizing the number of modifications needed to apply refactorings [P58]. Fewer modifications are seen as more effective than many ones, since they can reduce the effort and improve the understandability.

Other widely used metrics (in 15 papers, 20%) are the ones from the Quality Model for Object-Oriented Design (QMOOD) [39]. QMOOD is a hierarchical model for the assessment of quality attributes in object-oriented designs, such as reusability, flexibility, understandability, functionality, extensibility and effectiveness. The attribute values are obtained according to the functions presented in Table 10. The functions consider a set of 11 design properties that are assessed by using some design metrics according to Table 11.

Cohesion and coupling [40] are basic design principles used as metrics. Coupling refers to the degree of interdependence between software modules. Cohesion refers to the degree of dependence between elements in the same module. Low coupling and high cohesion are usually related to a good software design. These metrics are commonly used together, and in aggregation with other ones, to evaluate other kind of aspects. As shown, QMOOD contains specific metrics for evaluating cohesion and coupling, they were excluded from this category, since they are part of the set. In this sense, there are other several metrics to evaluate cohesion and coupling, the following two are the most used by the papers: (i) Lack of cohesion of methods [41] to assess the similarity between methods of a class; and (ii) Coupling between objects [41] to count the number of classes coupled with a specific class.

Semantic coherence [P60] is based on the semantic similarity between the elements when refactorings are applied. Two measures are used to determine the semantic coherence: vocabulary-based similarity and dependency-based similarity. Considering the vocabulary-based similarity, two elements can be semantically similar if they use a common vocabulary. The vocabulary of an element includes names of methods, fields, parameters and others. It is usually related to the software domain. For instance, this measure can be used by moving methods between classes, where the refactoring makes sense if the method and the target class have a similar vocabulary. Regarding the dependency-based similarity, it argues that elements strongly connected are semantically related. That way, applying refactorings in this kind of situation is likely to be successful. The dependencies considered between elements are the ones sharing method calls and field access.

Similarity to examples [P31] is based on examples of well-designed systems, obtained by refactorings, to guide the optimization process. The goal is to find a sequence of refactorings to maximize the similarity between the examples and the initial design. The refactorings considered are the same used in the example.

Eight papers use other object-oriented metrics [41,42], [P24]. These metrics assess the number of elements, visibility of elements and specific aspects of classes, inheritance, and polymorphism. They are briefly described in Table 12.

The user feedback is a metric used by three papers. The solutions are evaluated based on a value given by the user. The category Other includes uncommon metrics used by only one paper. Most papers using these metrics use other presented metrics too.

The presented metrics are used for evaluating different types of artifacts, including model and code. Fig. 7 shows the number of papers by artifact using each of the presented metrics.

As shown in Fig. 7, the number of bad smells and the number of modifications are the most used metrics for Java code; user

Table 9
Metrics used by the papers.

Metric	NP	References
Number of bad smells	18	[P15,P22,P25,P28,P30,P38,P43,P44,P56–P63,P65,P68]
Number of modifications	16	[P7–P12,P27,P38,P41,P43,P45,P48,P58,P59,P62,P68]
QMOOD	15	[P27,P32,P33,P36,P37,P39,P41,P43,P47,P51–P55,P66]
Cohesion	12	[P5,P6,P14,P21,P22,P24,P26,P42,P46,P64,P70,P71]
Coupling	12	[P1,P5,P6,P21,P22,P24,P26,P42,P46,P49,P70,P71]
Semantic coherence	10	[P41,P43,P45,P57–P63]
Similarity to examples	9	[P18,P19,P31,P45,P57,P58,P61–P63]
OO metrics	8	[P21,P22,P24,P40,P42,P49,P70,P71]
User feedback	3	[P16,P18,P35]
Other	28	[P1–P6,P13,P15–P17,P20,P22–P24,P26,P27,P29,P32–P34,P39,P42,P44,P45,P50,P64,P67,P69]

Table 10
Quality attributes functions of QMOOD (Adapted from [39]).

Quality attribute	Function
Reusability	−0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation − 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	−0.33 * Abstraction + 0.33 * Encapsulation − 0.33 * Coupling + 0.33 * Cohesion − 0.33 * Polymorphism − 0.33 * Complexity − 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extensibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Table 11
Design metrics for design properties of QMOOD (Adapted from [39]).

Design property	Design metric
Design size	Design size in classes (DSC).
Hierarchies	Number of hierarchies (NOH).
Abstraction	Average number of ancestors (ANA).
Encapsulation	Data access metric (DAM).
Coupling	Direct class coupling (DCC).
Cohesion	Cohesion among methods in class (CAM).
Composition	Measure of aggregation (MOA).
Inheritance	Measure of functional abstraction (MFA).
Polymorphism	Number of polymorphic methods (NOP).
Messaging	Class interface size (CIS).
Complexity	Number of methods (NOM).

feedback is the most used for C code; cohesion and coupling are the most used for class diagrams; and QMOOD is the only metric used for activity diagrams.

In the code level, almost all the metrics are used to evaluate Java code, since it is the most used type of artifact. In the model level, many metrics are used for evaluating class diagrams, but there are some unexplored ones, such as number of bad smells, semantic coherence and other types of object-oriented metrics. Since few papers use activity diagrams and C code, few metrics are used for these artifacts.

5.7. RQ6 – consistency with other artifacts

We found only one approach addressing consistency between artifacts. The other papers do not present any evidence that this task is performed.

The found approach proposed in [P39] suggests a sequence of refactorings to improve both class and activity diagrams in order to maintain the consistency between them. The employed algorithm evaluates the overall quality considering the impact of the refactorings applied to a class diagram on the corresponding activity diagram. The approach also uses the activity diagram to check behavior preservation.

5.8. RQ7 – solutions and representations

We identified three kinds of solutions produced by the approaches: sequence of refactorings, artifacts and set of elements to be refactored. Moreover, different representations are used. Information about each solution and used representation are presented in Table 13, and described in the next sections.

5.8.1. Sequence of refactorings

As shown in Table 13, the solution produced by most works (42 out of 73) is a sequence of refactorings. Furthermore, a vector represents this sequence, where each dimension of the vector corresponds to a refactoring operation to be applied to the artifact

Table 12
Other object-oriented metrics.

Metric	Description
Number of attributes	Number of attributes of a class.
Number of lines of code	Number of lines of code of an object-oriented program.
Number of associations	Number of associations of a class.
Number of classes	Number of classes of a system.
SD of methods per class	Standard deviation of methods per class.
Attribute hiding factor	Percentage of classes from which an attribute is not visible.
Method hiding factor	Percentage of classes from which a method is not visible.
Weighted methods per class	The complexity of a class based on its elements.
Cyclomatic complexity	Complexity of a program.
Depth of inheritance tree	The depth of a class in a inheritance tree.
Number of children	Number of subclasses of a class hierarchy.
Attribute inheritance factor	Number of attributes inherited from the superclass.
Polymorphism factor	Degree of method overriding in a class inheritance tree.
Response for class	Methods that can be executed in response to a message received by an object.

Table 13
Solutions and representations used by the papers.

Solution	Representation	NP	References
Sequence of refactorings	Vector	35	[P1–P3,P15,P20–P22,P24–P26,P28–P33,P39–P45,P56–P63,P66–P68,P71]
	Entity-based	3	[P7,P9,P10]
	Other	4	[P18,P19,P35,P36]
Artifact	Abstract Syntax Tree	9	[P14,P17,P47,P48,P51–P55]
	Trees	5	[P13,P27,P38,P50,P70]
	Artifact	4	[P23,P34,P49,P65]
	Graph	2	[P37,P46]
Set of elements	Vector or Matrix	6	[P4–P6,P16,P64,P69]
	Refactoring-based	3	[P8,P11,P12]

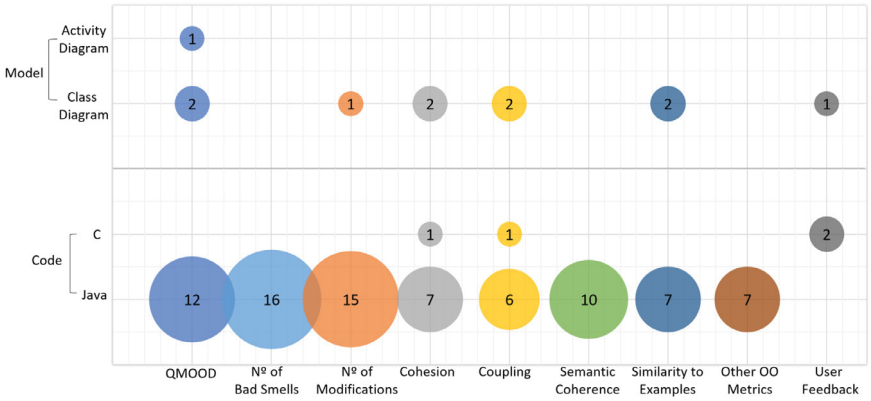


Fig. 7. Metrics used for each artifact.

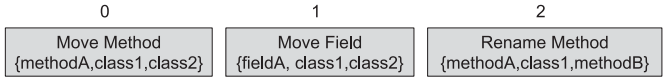


Fig. 8. Vector representation of a sequence of refactorings.

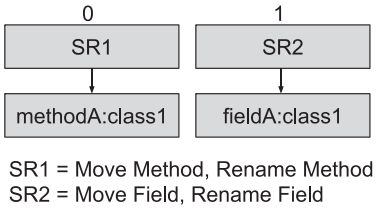


Fig. 9. Entity-based representation of a sequence of refactorings.

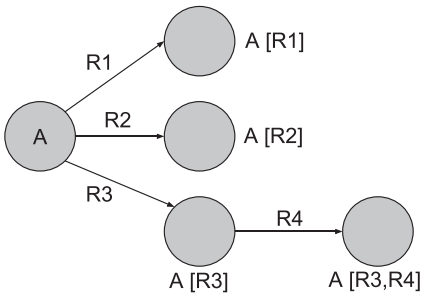


Fig. 10. Path tree representation of an artifact (Adapted from [P70]).

representation. The order they appear in the vector is the order they are applied. A refactoring operation specifies the elements involved, such as actors, code fragments, and roles. This kind of representation is used in the software refactoring of Java and C programs, as well as in models. Fig. 8 shows an example of a vector containing three refactoring operations, each of them specifying the involved actors.

In the example of Fig. 8, firstly, the *move method* refactoring will be applied by moving *methodA* from *class1* to *class2*. The *move field* refactoring will be applied by moving *fieldA* from *class1* to *class2*. Finally, the *rename method* refactoring will be applied by renaming *methodA* of *class1* as *methodB*.

Entity-based [P11] is another used representation for a sequence of refactorings. A vector also represents the solution where each position contains a sequence of refactorings, but in addition, another vector is stored where each dimension represents a software entity, such as a class, a method or a field. That way, the sequence of refactorings in the *i*th position will be applied to the entity of the same position of the other vector. Refactorings are only available for entities they can be applied. Fig. 9 illustrates this representation.

In the example of Fig. 9, two sequences of refactorings are available: SR1, containing the refactorings *move method* and *rename method*, and SR2, containing the refactorings *move field* and *rename field*. According to position 0, SR1 will be applied in *methodA* of *class1*. According to position 1, SR2 will be applied in *fieldA* of *class1*.

5.8.2. Artifact

The second most used solution is the artifact itself. In such cases, the artifact is used as input and produced as output. That way, the solution representation and the artifact representation are the same. The representation most used in such cases is the Abstract Syntax Tree (AST).

Other types of trees are also used to represent the solutions and compose another category. They generally map the refactorings to be applied and the elements to receive them. They include simple tree [P38], parse tree [P13,P50], path tree [P70] and transformation tree [P38]. Fig. 10 shows an example of a path tree used to describe the possibilities of refactorings.

Table 14
Search algorithms used by the papers.

Search technique	NP	References
Evolutionary algorithm	47	[P1–P12,P15,P16,P18,P19,P21,P22,P26–P31,P35,P36,P38–P45,P47,P53,P54,P57–P64,P67,P68]
Hill Climbing	16	[P14,P16,P17,P24,P25,P32,P40,P47–P49,P51,P53–P55,P65,P71]
Simulated Annealing	10	[P13,P17,P32,P47,P49–P51,P53–P55]
Greedy Algorithm	3	[P65,P70,P71]
Other	13	[P20,P23,P27,P32–P34,P37,P38,P46,P56,P66,P69,P71]

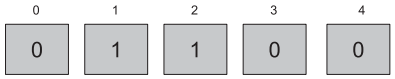


Fig. 11. Binary vector representation for a set of elements.

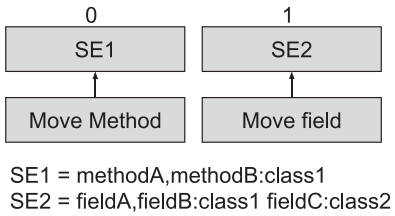


Fig. 12. Refactoring-based representation for a set of elements.

In the example of Fig. 10, the initial node A represents an element of the artifact to be improved, such as a method, a class or a field. Arcs R1 to R4 represent the refactorings that can be applied to the element. The other nodes indicate the element after the application of a refactoring. For example, A[R1] after the application of R1 in A, and A[R3,R4] after the application of R3 and R4 in A.

Some papers also use the artifact as solution representation, which means that the approach manipulates directly the artifact. Other papers [P37,P46] use a graph to represent the solution.

5.8.3. Set of elements

Other kind of solution found is a set of elements to receive refactorings. Papers that predefine a refactoring or a set of them commonly use this representation. Thus, it is only necessary to search the elements to receive them.

Vectors and matrices are used to represent this kind of solution. Fig. 11 shows an example of a binary vector used to define which elements will receive refactorings. It is employed in [P6].

In the example of Fig. 11, each position represents a C function. It will receive a refactoring only if the position value is 1. In this case, the functions represented by the positions 1 and 2 will receive refactorings.

Refactoring-based [P11] is another used representation for a set of elements. It is the opposite of the entity-based representation. Thus, a vector represents the solution where each position contains a set of entities. Another vector is created storing a refactoring in each position. That way, the refactoring of the *i*th position will be applied in the software entities of the same position in the solution representation. Fig. 12 shows an example of this representation.

In the example of Fig. 12, each position of the solution representation contains a set of software entities. SE1 contains two methods of *class1* and SE2 contains two fields of *class1* and one field of *class2*. According to position 0, the refactoring *move method* will be applied in *methodA* and *methodB* of *class1*. According to position 1, the refactoring *move field* will be applied in *fieldA* and *fieldB* of *class1*, and *fieldC* of *class2*.

5.9. RQ8 – search-based algorithms

Table 14 shows the search algorithms used by the papers, the number of papers (NP) using them, and the references.

Most papers use evolutionary algorithms. Among them, we can mention the mono-objective genetic algorithm Steady State [P5,P7–P9,P11,P12] and the simple GA [P2–P4,P6,P10, P16,P18,P19,P21,P22,P27,P28,P30,P31,P35,P36,P47,P53,P54,P67]. In addition, there are also papers that use the multi-objective ones Strength Pareto Evolutionary Algorithm 2 (SPEA2) [P26], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [P1,P15,P29,P39,P43,P44,P57–P64,P68] and Non-dominated Sorting Genetic Algorithm III (NSGA-III) [P41,P42,P45].

Hill Climbing [43] is used in 16 studies, in which the following types are employed: First Ascent, Steepest Ascent, Steepest Descent, Multiple First Descent, Multiple Steepest Descent, Multiple Restart and Multiple Ascent. We can also notice the use of Simulated Annealing. These last algorithms are differentiated by the criteria used to select a solution from the neighborhood. Few papers employed the Greedy Algorithm, these works create a solution by selecting, at each step, the local optimal choice of a set of candidates. The category Other includes Artificial Bee Colony (ABC) [P32,P33] and Chemical Reaction Optimization [P56].

Since the refactoring problem is a multi-objective problem impacted by many factors, most of the mono-objective algorithms use a weighted sum to deal with the objectives.

Fig. 13 shows the artifacts and solutions used by the search algorithms. All the algorithms are used to improve code artifacts. In the model level, the used algorithms are all evolutionary ones, including the mono-objective GA, Steady State and Genetic Programming, and the multi-objective NSGA-II. Nevertheless, the many-objective algorithm NSGA-III has not been explored for model refactoring.

The simple GA is the most used algorithm and consequently, all types of solutions and almost all types of representations are explored for it. Genetic Programming, Hill Climbing, Simulated Annealing and Greedy Algorithm commonly use the “Artifact” as the solution, mainly using AST and other trees as representation. The solution “Set of elements” and its representations are usually explored by evolutionary algorithms in general. The solution “Sequence of Refactorings” is used by all algorithms, except Genetic Programming, because this algorithm representation is commonly a tree. In this sense, such a solution seems to be the more advantageous to be used, because it is compatible with many algorithms and is the most employed.

5.10. RQ9 – additional information in the process

We identified the papers using any additional information to guide the optimization process. Four categories were defined, one for each type, and a category including papers that do not use any additional information. Fig. 14 presents the frequency of papers in each category.

Most papers (72%) use no additional information in the process. Among the ones using some information, 16% use examples [P31] to guide the process. The examples are used in the

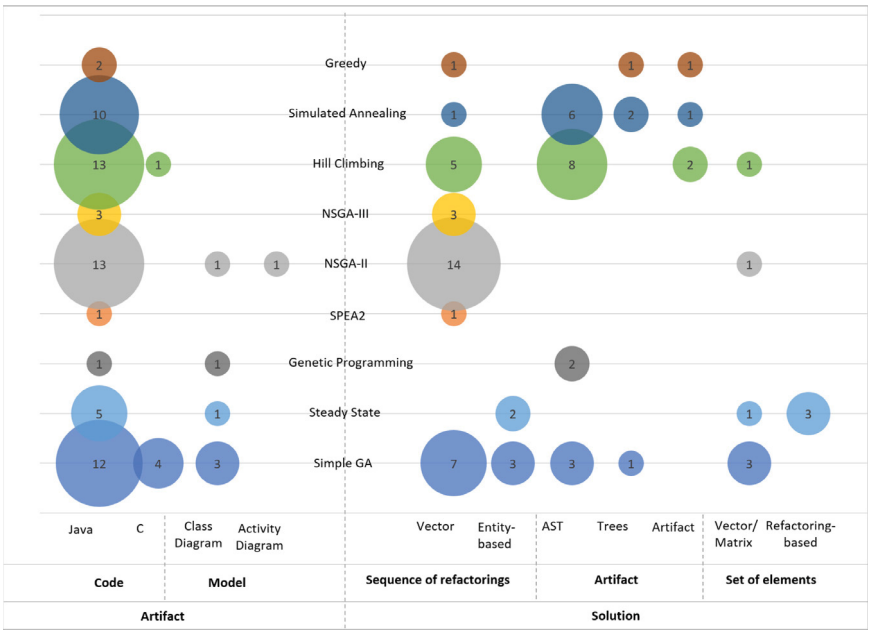


Fig. 13. Artifacts and solutions used by the search algorithms.

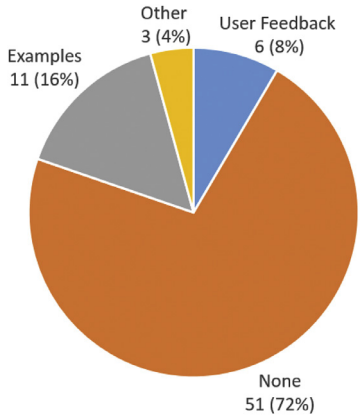


Fig. 14. Type of additional information used by the papers.

code [P31,P38,P45,P52,P57,P58,P61–P63] and model [P18,P19] level. These examples include refactorings successfully applied in a previous version of the system and refactorings successfully applied in systems of a similar domain or similar features. In this sense, the suggested refactorings will probably be similar to the ones given as examples. The user feedback is used in the code [P2,P6,P16,P35,P43] and model [P18] level by 8% of the papers. Papers employing that, commonly use an interactive algorithm when the user has to evaluate the solution quality. Other types of additional information (4%) include an optimization process guided by a desired final design [P46,P48] and predetermined bad smells to be corrected [P56].

5.11. RQ10 – evaluation method

We checked if the proposed approaches were evaluated in some way. Some papers [P5,P13,P15,P35,P47,P50] (6 out 73) do not present any kind of evaluation. Other ones conducted different types of evaluations, such as empirical ones and case studies. They are usually performed to answer research questions or to validate a given hypothesis.

Fig. 15 presents the types of evaluation methods used, the number of papers employing them, and the references. In general, the papers use more than one of these methods.

Two evaluation methods are based on the fitness value of the solutions, which are “Analysis of the fitness value” and “Quality indicator and statistical test”. The first is the most employed [P3,P4,P7–P12,P16,P17,P23,P24,P26,P27,P29,P34,P40, P45,P48,P49,P51–P55,P58,P63–P66,P66,P67,P70] and it is used to evaluate a solution obtained by a mono-objective algorithm. The fitness value of a solution is analyzed in order to compare the solution quality to other ones, or to analyze the evolution of a fitness value during the optimization process. The second method is used to evaluate a set of solutions obtained by a multi-objective algorithm [P1,P29,P41,P42,P49,P63]. Quality indicators are calculated based on the values of each objective in the fitness function, and the statistical test is applied to the quality indicators results. Papers using this evaluation method use at least one of the following: the statistical test Wilcoxon [44] and the quality indicators Inverted Generational Distance (IGD), Hypervolume and Spread [45].

The second most used evaluation method is “Comparison with other approaches”, involving any type of comparisons between approaches [P1–P3,P19,P25,P29,P31,P32,P36–P42,P42–P46,P51,P53–P64,P68,P69,P71]. “Execution time” is the third most used method, evaluating the time that an approach takes to execute [P2,P17,P19,P29–P31,P38,P39,P42,P43,P54,P56–P61,P63].

The category “Analysis of bad smells” encompasses papers analyzing the number of bad smells after the software refactoring. They are generally compared to the number of bad smells in the original artifact or in old versions [P6,P21,P22,P25,P36,P39,P41,P42,P44,P56–P63].

Two evaluation methods are used to analyze the refactorings performed in an artifact. In the “Manual validation of refactorings”, the engineer can manually evaluate if an applied refactoring makes sense [P1,P2,P14,P18,P20,P28,P30,P31,P38,P39,P42,P43, P43,P45,P46,P62,P68,P69]. The evaluation method “Comparing the refactoring to the expected ones” is usually performed when the artifact to be improved is an old version of a software

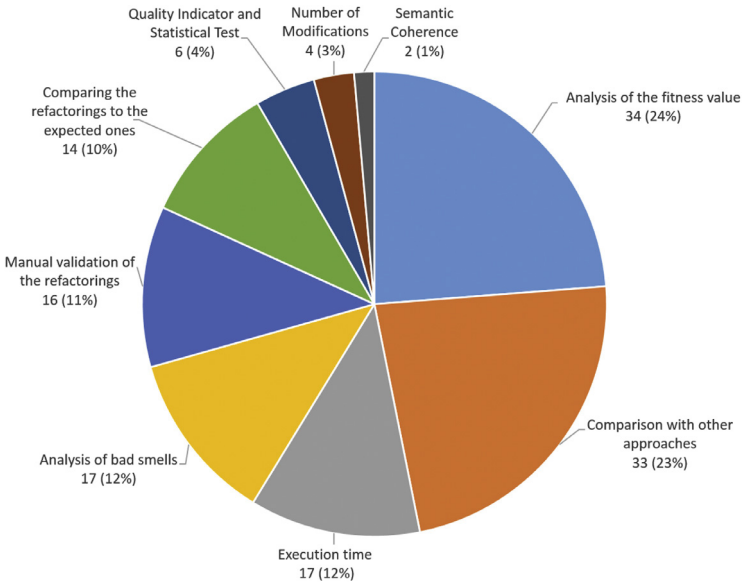


Fig. 15. Evaluation methods used by the papers.

Table 15
Systems used by the papers.

System	NP	References
Xerces-J	20	[P2,P19,P29–P31,P38,P39,P41–P45,P56–P63]
GanttProject	18	[P2,P19,P29–P31,P38,P39,P41–P43,P45,P56–P60,P62,P63]
JHotDraw	19	[P2,P18,P19,P24,P29,P31,P36,P43–P45,P48,P49,P56–P58,P62,P63,P66,P69]
JFreeChart	13	[P2,P29,P36,P39,P43–P45,P56,P58,P61–P63,P68]
AntApache	10	[P2,P18,P19,P29,P41,P42,P44,P49,P58,P62]
LAN	7	[P7–P12,P69]
Beaver	7	[P32,P49,P51–P55]
ArgoUML	5	[P30,P38,P41,P42,P59]
QuickUML	4	[P30,P31,P38,P59]
SpecCheck	4	[P52–P55]
Rhino	5	[P2,P29,P39,P58,P62]
Azureus	4	[P38,P41,P42,P59]
JDI-Ford	4	[P43–P45,P68]
Mango	4	[P49,P52–P54]
LOG4J	3	[P38,P44,P59]
Other	24	[P1,P3,P4,P6,P14,P16–P19,P24,P27,P32,P33,P36,P40,P43,P46,P49,P53,P54,P56,P63,P64,P71]

[P1,P2,P18,P19,P30,P31,P38,P39,P41,P43,P58,P59,P62,P63]. That way, the refactorings applied in such a version are compared to the ones already applied by an engineer in a new version of the software. In this sense, it is possible to evaluate if the refactorings automatically applied are similar to the ones identified by an engineer.

Some evaluation methods are metrics calculated for the refactored artifact, they are “Semantic coherence” [P57,118] and “Number of modifications” [P27,P36,P48,P58]. These evaluations were conducted to find artifacts containing the values of the metrics improved.

5.12. RQ11 – used systems

Many systems are used in the experiments to validate the approaches proposed in the papers. Approaches are evaluated using a different number of systems that can be industrial or academic. They are presented in Table 15.

All the presented systems are open-source Java programs and different versions of them are used in the papers. In addition, some studies that present approaches to improve class diagrams [P18,P19,P39], convert these programs into class diagrams to be used as input artifacts. The systems used to this end are: AntApache, JHotDraw, JFreeChart, GanttProject and Xerces-J. Systems of other languages and artifacts are classified in the category Other, since they are generally used by only one paper.

5.13. RQ12 – refactoring tools

We searched in the papers, widely used tools for any purpose in the software refactoring context. We considered only tools used by more than two papers.

We identified 8 papers [P38,P43,P56,P57,P59–P61,P63] using Eclipse IDE[2] refactoring support. In these papers, the IDE is used in the evaluation phase with the goal of applying the refactorings suggested by the proposed approach.

Other tool is Ref-Finder [46], used by 11 papers [P2,P18,P19,P41,P45,P56–P58,P61–P63]. Ref-Finder identifies refactorings between two Java program versions. It is an Eclipse plugin capable of showing which refactorings were used among 96 refactorings of the Fowler’s catalog. In the studies, this tool is commonly used in the evaluation step to identify the refactorings applied in an artifact during the optimization process.

In the context of SBR, we found Code-Imp [P47], used by 9 papers [P14,P17,P47,P48,P51–P55]. Code-Imp is an automated SBR

[2] https://eclipse.org.

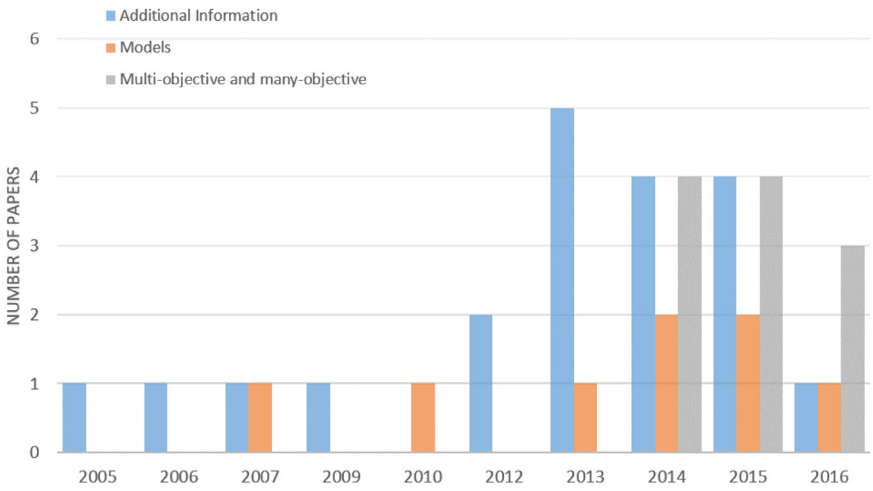


Fig. 16. Number of papers per year using additional information, models and multi/many-objectives algorithms.

tool, which supports a set of refactorings, metrics and search algorithms. The tool works by applying refactorings to an AST representation of Java programs. A set of 14 refactorings are used in the level of methods, fields and classes. The user has to choose the metrics to be used in the fitness function and the search technique to be executed. The available search techniques are Hill Climbing, Simulated Annealing, and Genetic Algorithm. To compose the fitness function, 28 metrics are available and the user can make any combination of them. They can be combined using a weighted sum or Pareto optimality, the first defines a weight to each metric and the second considers an improvement if at least one metric value is increased without decreasing others.

6. Trends and research opportunities

Analyzing the studies found in this review we could identify some research gaps and limitations. Based on them, we discuss in this section main research opportunities related to each investigated software refactoring element. We also present some trends in the SBR by analyzing the frequency of such elements in the studies over the years (see Fig. 16).

6.1. Models as artifacts

According to [13], to use models as artifacts is an open area of study in the software refactoring context. As shown in Section 5.2 (Table 5), few works use models as artifacts to be refactored. This has been changing, since we identified that 75% of the papers in this context have been published recently (Fig. 16). In this sense, refactoring of models seems to be a trend in SBR that may be better investigate in the future. Furthermore, many aspects can be explored in this context. For example, the number of refactorings, metrics, and tools to be used for model refactoring is smaller than the number used in the code level. In addition, the application of refactorings in a specific type of model, such as a class diagram, impacts other ones, such as activity, sequence and object diagrams [P39]. The papers usually address class diagram refactorings. Just one [P39] investigates the impact of refactorings in more than one type of diagram.

6.2. Many objectives in the fitness function

We can observe in Fig. 16 that approaches using multi-objective algorithms have appeared over the last two years [P1,P15,P29,P39,P43,P58,P62,P63], and furthermore, few at the model level (Fig. 13). It can be explained by the multi-objective nature of the software refactoring problem. Hence, the use of multi-objective and many-objective algorithms is appropriate to this kind of problem and seems to be a trend to offer a better treatment in the code and model refactorings. Furthermore, search algorithms to deal with many-objective problems have appeared recent, and can be explored in this context.

6.3. Additional information in the process

As shown in Section 5.10, many papers use some information to guide the optimization process. We identified an increasing number of papers using additional information in the last few years (Fig. 16).

Many recent papers [P19,P45,P58,P63] use examples of applied refactorings. This has been investigated in different ways, such as refactorings applied in different versions of a system, and refactorings applied in different systems of the same domain. Furthermore, some authors [P63] point out, as future work, the investigation of other aspects related to the provided examples, such as to consider bad examples of refactorings (anti-patterns) to avoid bad changes.

Other additional information used by some papers [P2,P43] is the user feedback. It is usually performed by an interactive algorithm, where the user is part of the process by evaluating the generated solutions. This is also presented as future investigation in [P41], and it is a subject explored in recent papers. It is a trend for the SBSE field and also SBR. To allow the user participation in the optimization process can bring many benefits: to improve the quality avoiding inconsistencies and to improve user satisfaction, since he/she is capable of recognizing the produced solution, since it has some expected characteristics.

6.4. Study of bad smells effect

As presented before, many works consider the correction of bad smells in the software refactoring activity by using the number of bad smells as metric and in the evaluation method. Nevertheless, according to [13], using bad smells as a direction to the software refactoring activity is a difficult task. Regardless of the existence of papers questioning the real impact of bad smells on the software refactoring activity [47,48], there are some recent papers in SBR investigating the bad smells effect in detail. Next, some recent investigations are presented.

Firstly, it is hard for the engineer to identify which type of refactoring should be applied to correct a bad smell. There are

some investigations in this context, such as a primary study [P41] presenting as future work the attribution of a set of refactorings to a known type of bad smell. Another difficult task investigated in the software refactoring field is the preference of bad smells to be corrected [13]. Some selected primary studies [P44,P58] incorporate preferences of bad smells based on a previously given importance. Furthermore, they also incorporate preferences for classes to receive refactorings. Another problem is that the correction of some bad smells, by applying refactorings, can lead to the introduction of other ones. In this context, a paper [P68] uses as metric the number of introduced bad smells after applying a sequence of refactorings, in order to decrease the number of bad smells comparing to the old versions. Another paper also presents this idea as future work [P41]. As we mentioned in Section 5.6, we have not found works that use the number of bad smells to evaluate solutions in the model refactoring level. Then, this subject may be another research opportunity.

6.5. Fully automatic approaches

The software refactoring activity addresses six tasks, and the approaches presented in this review encompass most of these tasks. However, none of them provide a fully automatic approach for the whole software refactoring activity by addressing all these tasks.

Based on the results presented here, the most difficult tasks are: (3) Guarantee that the applied refactoring preserves behavior, (4) Apply the refactoring; and (6) Maintain the consistency between the refactored artifact and other software artifacts. Only 40% of the papers present automatic approaches to apply refactorings directly in the artifact. One of the problems of automating this task is the difficulty to preserve the behavior. In fact, our results show that 35% of the papers do not address methods for this goal. Furthermore, only one paper presents an approach that maintains the consistency between artifacts. In this sense, search-based approaches should explore deeply these tasks, and move forward to achieve fully automatic approaches for the software refactoring activity.

6.6. Tools

As presented on Section 5.13, only one popular SBR tool was identified. It is for Java programs and supports the use of many metrics, refactorings and search techniques. Due to the increasing of the SBR field, the development of new tools is considered a research opportunity. Tools should be developed following the new trends, for example, by improving other kind of artifacts, such as models. There are other aspects to be considered for a tool, such as the use of other refactorings and metrics. Moreover, other search techniques can be considered, including the widely used multi-objective algorithms, such as NSGA-II and SPEA2, and the many-objective NSGA-III.

6.7. Systems and evaluation

We identified a list of systems used for evaluation. They are representative and can be used for comparisons in the field considering code refactoring. However, in the evaluation of model refactoring approaches, most works extract models from the source code, resulting in a lack of experiments using models that really represent requirements and design problems. We also observe a lack of studies conducted in industrial scenarios. In this sense, we believe that to consider practical aspects of the SBR approaches and how they can be used in industry is a research opportunity that should be better investigated.

6.8. Dynamic adaptive SBR

Hyper-heuristic is considered a new trend of the SBSE field. It can provide a holistic SBSE and improve the applicability and generality of the search techniques [49]. The goal of hyper-heuristics is to automatically select or generate the best low-level heuristic in a given moment of the search, instead of trying to solve the problem directly. Such low-level heuristics can be, for example, simple heuristics, meta-heuristics or search operators [50].

Some recent SBSE papers investigate the use of hyper-heuristics in the context of software testing [51–54], software clustering [55] and cost estimation [56]. These papers obtained encouraging results, outperforming traditional search-based techniques. Nevertheless, no SBR paper has been found. That way, it seems to be a promising subject for future studies in the field.

7. Threats to validity

In this section, we identify possible threats to the validity of our review results, by using the taxonomy of Wohlin et al. [57].

Construct validity refers to the relation between theory and observation [57]. The main threats in this category are related to the research questions, engines used, and search string. The research questions may not address all the aspects of the SBR field. To minimize such threat, we elaborated research questions covering the main ingredients that compose a search-based solution and the main software refactoring activities. We did not perform a detailed study to select the search engines, we choose them because they are well-known sources and are also used in related work. In addition, we know that these sources return papers published in popular conferences of the field.

A third threat is related to the search string used. It has not too many elements, but to minimize this threat, we take care to create a search string capable of finding coherent results. For the search term *search-based software engineering* many terms were used and most of them were extracted from related work. For the search term *refactoring* we also extracted the terms from related work. To minimize such threat we use a control group composed of main knows papers in the SBR field from a SBSE repository. Our string returned all the papers in the repository.

Internal validity evaluates the relationship between the treatment and the output. In our case, the treatment is the set of papers included and the outcome is the analysis reported. As threats to the internal validity, we can consider the subjective decisions that might have occurred during primary studies selection and data extraction. Some relevant studies may not be selected as primary studies. For minimizing this threat, we follow a rigorous plan, guided by well-defined inclusion and exclusion criteria, applying them by carefully analyzing the papers. Moreover, we also performed a snowballing reading in the primary studies and selected relevant studies. Meetings were made to discuss the data extraction and rereadings were performed to clarify existing doubts.

Threats to conclusion validity are related to issues that affect the ability to draw the correct conclusions from the study. An identified threat is the classification schema and the way we grouped the papers and established relations between them. To avoid bias we follow some procedures. However, other reviews can have other classification schema and ways to group and analyze the papers. Another threat is related to the granularity of the information presented in the reviewed primary studies. If some information was not described in these studies, it may affect our conclusions.

Reliability validity is concerned with issues that affect the ability to draw that the operations of a study can be repeated with the same results. We think that our study can be easy replicated following the steps described and using the search string.

8. Concluding remarks

This paper presented results of a systematic review on SBR, focusing on studies that propose search-based approaches to suggest or apply a sequence of refactorings in an artifact. As far as we know, this is the first review addressing specifically SBR approaches. In this sense, specific aspects of the approaches were revealed and some trends and opportunities were identified to guide future researches in the field. That way, researchers can identify common characteristics of the approaches and directions for future works.

71 papers were returned and classified according to a schema proposed to answer the research questions related to the main elements of the software refactoring activity: artifacts used as input, solutions produced, encoding and algorithms used, tools and evaluation aspects.

Results show that an increase in the number of publications has occurred over the last years. Evolutionary algorithms are the most used search technique. Furthermore, we identify as a trend the use of additional information to help in the optimization process. The most used artifact is Java code, and the refactorings applied are usually the ones of the Fowler's catalog. In this sense, the most appropriate solution is a sequence of refactorings, commonly represented by a vector. The main metrics used to evaluate the solutions are the number of bad smells, the number of modifications, and a set of metrics for object-oriented artifacts. The experiments are usually conducted using open-source programs and evaluated in comparison with other approaches, through the analysis of execution time and fitness value. One widely used SBR tool was identified, encompassing some metrics, search techniques, and refactorings to be used.

Some of the research opportunities observed are: the use of many objectives in the fitness function; the use of models as artifacts; the study of bad smells effect; and the use of hyper-heuristics, which can contribute to an adaptive SBR. Other researches should include the development of supporting tools and experiments especially in the model refactoring context. Experiments in industrial scenarios should be also conducted to make SBR approaches more useful in practice.

Acknowledgments

This work is supported by Brazilian funding agencies CAPES and CNPq [grant numbers 307762/2015-7, 473899/2013-2].

Appendix A. Refactorings

Table A.16
Information about the applied refactorings of Fowler's catalog.

Refactoring	Functionality	NP	Artifact	
			Code	Model
Pull up method	Move a method from subclasses to the superclass.	45	X	X
Move method	Move a method from a class to another.	39	X	X
Push down method	Move a method of a superclass to a subclass.	39	X	X
Pull up field	Move a field from subclasses to the superclass	36	X	X
Push down field	Move a field of a superclass to a subclass	35	X	X
Extract class	Create a new class and move fields and methods from the old class to the new one	27	X	X
Move field	Move a field from a class to another	21	X	X
Inline class	Move all features of a class in another one and remove it	19	X	
Collapse hierarchy	Merge a superclass and a subclass	19	X	X
Extract superclass	Move common features of classes into a new superclass	15	X	
Rename method	Change the name of a method	13	X	X
Add parameter	Add a parameter for an object	13	X	X
Extract interface	Extract methods of a class into an interface	14	X	X
Encapsulate field	Make a public field private and provide accessors	13	X	
Extract method	Extract a code fragment into a method	13	X	
Replace delegation with inheritance	Make the delegating class a subclass of the delegate	11	X	X
Replace inheritance with delegation	Transform a subclass in a delegating class of the superclass	11	X	X
Inline method	Move the body of a method into its callers and remove the method	8	X	
Remove parameter	Remove a parameter	8	X	
Extract subclass	Create a subclass for a set of features	8	X	X
Extract hierarchy	Create a hierarchy of classes where each subclass represents a feature	6	X	
Encapsulate collection	Provide add/remove methods for a class containing a method that returns a collection	3	X	
Encapsulate downcast	Move the downcast of an object into the method that returns it	3	X	
Hide method	Make a method private	3	X	
Remove setting method	Remove a setting method of a field	3	X	
Self encapsulate field	Create getting and setting methods for the field	2	X	
Form template method	Merge similar functionalities of subclasses methods in a new method into the superclass	1	X	

Table A.17
Information about the other applied object-oriented refactorings

Refactoring	Functionality	NP	Artifact	
			Code	Model
Make class abstract	Change a concrete class to abstract	14	X	
Make class concrete	Change an abstract class to concrete	12	X	
Decrease method visibility	Decrease the visibility of a method from private to package, package to protected or protected to public	12	X	
Increase method visibility	Increase the visibility of a method from public to protected, protected to package or package to private	12	X	
Decrease field visibility	Decrease the visibility of a field from private to package, package to protected or protected to public	12	X	
Increase field visibility	Increase the visibility of a field from public to protected, protected to package or package to private	12	X	
Rename field	Rename a field	6	X	X
Move class	Move a class from a package to another	6	X	X
Extract package	Add a package to compose the elements of another package	1	X	
Remove method	Remove a method from a class	4	X	
Rename class	Rename a class	3	X	
Remove class	Remove a class	2	X	X
Remove interface	Remove an interface	2	X	
Merge packages	Merge the elements of a set of packages in one of them	1	X	
Delete generalization	Delete a generalization relationship	1	X	
Add relationship	Add a relationship	1	X	
Change superclass down	Change the superclass of a class, moving the class to the lower point in the inheritance hierarchy	1	X	
Change superclass up	Change the superclass of a class, moving the class to the higher point in the inheritance hierarchy	1	X	

Primary Sources

[P1] B. Alkhazi, T. Ruas, M. Kessentini, M. Wimmer, W. I. Grosky, Automated refactoring of ATL model transformations: A search-based approach, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS, 2016, pp. 295–304.

[P2] B. Amal, M. Kessentini, S. Bechikh, J. Dea, L. B. Said, On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring, in: Proceedings of the International Symposium on Search-Based Software Engineering, SSBSE, 2014, pp. 31–45.

[P3] M. Amoui, S. Mirarab, A genetic algorithm approach to design evolution using design pattern transformation, Int. J. Inf. Technol. Intell. Comput. (2006) 1–10.

[P4] G. Antoniol, M. Di Penta, Library miniaturization using static and dynamic information, in: Proceedings of International Conference on Software Maintenance (2003) 235–244.

[P5] T. Bodhuin, G. Canfora, L. Troiano, SORMASA: a tool for suggesting model refactoring actions by metrics-led genetic algorithm, in: Proceedings of the Workshop on Refactoring Tools, WRT, 2007, pp. 23–24.

[P6] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, A novel approach to optimize clone refactoring activity, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2006, pp. 1885–1892.

[P7] C. Chisalita-Cretu, A multi-objective approach for entity refactoring set selection problem, in: Proceedings of the International Conference on the Applications of Digital Information and Web Technologies, ICADIWT, 2009, pp. 790–795.

[P8] C. Chisalita-Cretu, First results of an evolutionary approach for the entity refactoring set selection problem, in: Proceedings of the International Conference Interdisciplinarity in Engineering (2009) 285–290.

[P9] C. Chisalita-Cretu, The entity refactoring set selection problem - practical experiments for an evolutionary approach, in: Proceedings of the World Congress on Engineering and Computer Science, WCECS, 2009.

[P10] C. Chisalita-Cretu, The optimal refactoring selection problem - a multi-objective evolutionary approach, in: Proceedings of the International Conference on Virtual Learning, ICVL, 2010.

[P11] C. Chisalita-Cretu, The multi-objective refactoring set selection problem - a solution representation analysis, Adv. Comput. Sci. Eng. (2011) 441–462.

[P12] C. Chisalita-Cretu, Evolutionary approach for the strategy-based refactoring selection, in: Proceedings of the World Congress on Engineering and Computer Science, WCECS, 2014, pp. 121–126.

[P13] M. O. Cinnéide, Towards automated design improvements through combinatorial optimization, in: Proceedings of the Workshop on Directions in Software Engineering Environments, WoDiSEE, 2004.

[P14] M. È. Cinnéide, D. Boyle, I. H. Moghadam, Automated refactoring for testability, in: Proceedings of Workshop on Refactoring and Testing, 2011, pp. 437–443.

[P15] T. J. Dea, Improving the performance of many-objective software refactoring technique using dimensionality reduction, in: Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE, 2016, pp. 298–303.

[P16] M. Di Penta, M. Neteler, G. Antoniol, E. Merlo, A language-independent software renovation framework, J. Syst. Softw. 77(3) (2005) 225–240.

[P17] S. Ghaith, M. Ó Cinnéide, Improving software security using search-based refactoring, in: Proceedings of the International Symposium on Search Based Software Engineering, SSBSE, 2012, pp. 121–135.

[P18] A. Ghannem, G. El Boussaidi, M. Kessentini, Model refactoring using interactive genetic algorithm, in: Proceedings of the International Symposium on Search Based Software Engineering, SSBSE, 2013, pp. 96–110.

[P19] A. Ghannem, G. El Boussaidi, M. Kessentini, Model refactoring using examples: a search-based approach, J. Softw.: Evolut. Process 26(7) (2014) 692–713.

[P20] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, B. Livshits, Automated migration of build scripts using dynamic analysis and search-based refactoring, in: Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA, 2014, pp. 599–616.

[P21] I. Griffith, S. Wahl, C. Izurieta, Evolution of legacy system comprehensibility through automated refactoring, in: Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, MALETS, 2011, pp. 35–42.

[P22] I. Griffith, S. Wahl, C. Izurieta, TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility, in: Proceedings of the International Conference on Computer Applications in Industry and Engineering, CAINE, 2011, pp. 316–321.
[P23] I. Halupka, J. Kollár, E. Pietriková, A task-driven grammar refactoring algorithm, Acta Polytechnica 52(5) (2012) 51–57.
[P24] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2007, pp. 1106–1113.
[P25] S. Herold, M. Mair, Recommending refactorings to re-establish architectural consistency, in: Proceedings of the European Conference, ECSA, 2014, pp. 390–397.
[P26] K. Hoste, L. Eeckhout, COLE : compiler optimization level exploration categories and subject descriptors, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO, 2008.
[P27] A. C. Jensen, B. H. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2010, pp. 1341–1348.
[P28] S. Kebir, I. Borne, D. Meslati, Automatic refactoring of component-based software by detecting and eliminating bad smells - a search-based approach, in: Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering, ENASE, 2016, pp. 210–215.
[P29] M. Kessentini, S. Bechikh, M. Ó Cinnéide, A robust multi-objective approach for software refactoring under uncertainty, in: Proceedings of the International Symposium on Search-Based Software Engineering, volume 8636 of SSBSE, 2014.
[P30] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design defects detection and correction by example, in: Proceedings of the International Conference on Program Comprehension, ICPC, 2011, pp. 81–90.
[P31] M. Kessentini, R. Mahouachi, K. Ghedira, What you like in design use to correct bad-smells, Softw. Qual. J. 21(4) (2012) 551–571.
[P32] E. Koc, N. Ersoy, A. Andac, Z. S. Camlidere, I. Cereci, H. Kilic, An empirical study about search-based refactoring using alternative multiple and population-based search techniques, in: Proceedings of the International Symposium on Computer and Information Sciences, ISCIS, 2011, pp. 59–66.
[P33] E. Koc, N. Ersoy, Z. S. Camlidere, H. Kilic, A web-service for automated software refactoring using artificial bee colony optimization, in: Proceedings of the International Conference on Advances in Swarm Intelligence, ICSI, 2012, pp. 318–325.
[P34] J. Kollár, I. Halupka, Role of patterns in automated task-driven grammar refactoring, in: Proceedings of the Symposium on Languages, Applications and Technologies, vol. 29 of SLATE, 2013, pp. 171–186.
[P35] W. B. Langdon, Evo_indent interactive evolution of GNU indent options, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2009, pp. 2081–2084.
[P36] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, Y. R. Kwon, Automated scheduling for clone-based refactoring using a competent GA, Softw.: Pract. Exper. 41(5) (2011) 521–550.
[P37] H. Liu, G. F. Li, Z. Ma, W. Shao, Conflict-aware schedule of software refactorings, IET Softw. 2(5) (2008) 446–460.
[P38] R. Mahouachi, M. Kessentini, K. Ghedira, A new design defects classification: marrying detection and correction, in: Proceedings of the Fundamental Approaches to Software Engineering, FASE, 2012, pp. 455–470.
[P39] U. Mansoor, M. Kessentini, M. Wimmer, K. Deb, Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm, Softw. Qual. J. (2015) 1–29.
[P40] D. F. Mark, M. Harman, R. M. Hierons, Evolving transformation sequences using genetic algorithms, in: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM, 2004, pp. 66–75.
[P41] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. O. Cinnéide, K. Deb, On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach, Empir. Softw. Eng. (2015) 1–43.
[P42] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2014, pp. 1263–1270.
[P43] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the International Conference on Automated Software Engineering, ASE, 2014, pp. 331–336.
[P44] M. W. Mkaouer, M. Kessentini, M. O. Cinnéide, S. Hayash, K. Deb, A robust multi-objective approach to balance severity and importance of refactoring opportunities, Empir. Softw. Eng. (2016) 1–43.
[P45] W. Mkaouer, M. Kessentini, P. Kontchou, K. Deb, S. Bechikh, A. Ouni, Many-Objective Software Remodularization Using NSGA-III, Trans. Softw. Eng. Methodol. 24(3) (2015) 17:1–17:45.
[P46] I. H. Moghadam, M. È. Cinnéide, Resolving conflict and dependency in refactoring to a desired design, e-Informatica Software Engineering Journal 9(1) (2015) 37–56.
[P47] I. H. Moghadam, M. Ó Cinnéide, Code-Imp: a tool for automated search-based refactoring, in: Proceedings of the Workshop on Refactoring Tool, WRT, 2011, p. 41.
[P48] I. H. Moghadam, M. Ó Cinnéide, Automated refactoring using design differencing, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2012, pp. 43–52.
[P49] M. Mohan, D. Greer, P. McMullan, Technical debt reduction using search based automated refactoring, J. Syst. Softw. 120 (2016) 183–194.
[P50] M. O'Keeffe, M. Ó Cinnéide, A stochastic approach to automated design improvement, in: Proceedings of the International Conference on Principles and Practice of Programming in Java, PPPJ, 2003, pp. 59–62.
[P51] M. O'Keeffe, M. Ó Cinnéide, Search-based software maintenance, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2006.
[P52] M. O'Keeffe, M. Ó Cinnéide, Automated design improvement by example, in: Proceedings of the Conference on New Trends in Software Methodologies, Tools and Techniques, 2007, pp. 315–329.
[P53] M. O'Keeffe, M. Ó Cinnéide, Getting the most from search-based refactoring, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2007, pp. 1114–1120.
[P54] M. O'Keeffe, M. Ó Cinnéide, Search-based refactoring: an empirical study, J. Softw. Maint. Evolut.: Res. Pract. 20(5) (2008) 345–364.
[P55] M. O'Keeffe, M. Ó Cinnéide, Search-based refactoring for software maintenance, J. Syst. Softw. 81(4) (2008) 502–516.
[P56] A. Ouni, M. Kessentini, S. Bechikh, H. Sahraoui, Prioritizing code-smells correction tasks using chemical reaction optimization, Softw. Qual. J. 23(2) (2015) 323–361.

[P57] A. Ouni, M. Kessentini, H. Sahraoui, Search-Based Refactoring Using Recorded Code Changes, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2013, pp. 221–230.
[P58] A. Ouni, M. Kessentini, H. Sahraoui, Multiobjective optimization for software refactoring and evolution, Adv. Comput. 94 (2014) 103–167.
[P59] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Autom. Softw. Eng. 20(1) (2012) 47–79.
[P60] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: Proceedings of the International Conference on Software Maintenance, ICSM, 2012, pp. 347–356.
[P61] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, The use of development history in software refactoring using a multi-objective evolutionary algorithm, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2013, pp. 1461–1468.
[P62] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Trans. Softw. Eng. Methodol. 25(3) (2016) 23:1–23:53.
[P63] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, J. Syst. Softw. 105 (2015) 18–39.
[P64] D. Romano, S. Raemaekers, M. Pinzger, Refactoring fat interfaces using a genetic algorithm, in: Proceedings of the International Conference on Software Maintenance and Evolution, ICSME, 2014, pp. 351–360.
[P65] F. Schmidt, S. G. MacDonell, A. M. Connor, An automatic architecture reconstruction and refactoring framework, in: Proceedings of the International Conference on Software Engineering Research, Management and Applications, SERA, 2011, pp. 95–111.
[P66] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2006, pp. 1909–1916.
[P67] T. Shimomura, K. Ikeda, M. Takahashi, An approach to ga-driven automatic refactoring based on design patterns, in: Proceedings of the International Conference on Software Engineering Advances, ICSEA, 2010, pp. 213–218.
[P68] H. Wang, M. Kessentini, W. Grosky, H. Meddeb, On the use of time series and search based software engineering for refactoring recommendation, in: Proceedings of the International Conference on Management of Computational and Collective Intelligence in Digital Ecosystems, MEDES, 2015, pp. 35–42.
[P69] M. Wang, W. Pan, B. Jiang, C. Yuan, CLEAR: class level software refactoring using evolutionary algorithms, J. Intell. Syst. 24(1) (2015) 85–97.
[P70] R. Wongpiang, P. Muenchaisri, Selecting sequence of refactoring techniques usage for code changing using greedy algorithm, in: Proceedings of the International Conference on Electronics Information and Emergency Communication, ICEIEC, 2013, pp. 160–164.
[P71] R. Wongpiang, P. Muenchaisri, Comparing heuristic search methods for selecting sequence of refactoring techniques usage for code changing, in: Proceedings of the International MultiConference of Engineers and Computer Scientists, IMECS, 2014, pp. 590–595.

References

[1] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004) 126–139.
[2] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.
[3] W.F. Opdyke, R.E. Johnson, Refactoring: an aid in designing application frameworks and evolving object-oriented systems, Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, SOOPPA, 1990, pp. 274–282.
[4] M. Harman, B.F. Jones, Search-Based Software Engineering, Inform. Softw. Technol. 43 (2001) 833–839.
[5] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Comput. Surv. 45 (1) (2012) 11:1–11:61.
[6] M. Harman, Y. Jia, Y. Zhang, Achievements, open problems and challenges for search based software testing, in: Proceedings of the International Conference on Software Testing, Verification and Validation, ICST, 2015, pp. 1–12.
[7] W. Afzal, R. Torkar, R. Feldt, A systematic review of search-based testing for non-functional system properties, Inf. Softw. Technol. 51 (6) (2009) 957–976.
[8] P. McMinn, Search-based software test data generation: a survey: research articles, Softw. Test. Verif. Reliab. 14 (2) (2004) 105–156.
[9] O. Räihä, A survey on search-based software design, Comput. Sci. Rev. 4 (4) (2010) 203–249.
[10] Y. Zhang, A. Finkelstein, M. Harman, Search based requirements optimisation: Existing work and challenges, in: Proceedings of the International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ, 2008, pp. 88–94.
[11] M. Harman, The current state and future of search based software engineering, in: Proceedings of the Future of Software Engineering, FOSE, 2007, pp. 342–357.
[12] M. Harman, S.A. Mansouri, Y. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Technical Report, Department of Computer Science, King's College London, 2009.
[13] M. Abebe, C.-J. Yoo, Trends, opportunities and challenges of software refactoring: A systematic literature review, Int. J. Softw. Eng. Appl. 8 (6) (2014) 299–318.
[14] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Technical Report, School of Computer Science and Mathematics, Keele University, 2007.
[15] W. Griswold, W. Opdyke, The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research, IEEE Softw. 32 (6) (2015) 30–38.
[16] W.F. Opdyke, Refactoring object-oriented frameworks, Champaign, IL, USA, 1992 Ph.D. thesis.
[17] S. Hanenberg, C. Oberschulte, R. Unland, Refactoring of aspect-oriented software, in: Proceedings of the International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World, 2003, pp. 19–35.
[18] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theory Prac. Object Syst. 3 (4) (1997) 253–263.
[19] F. Trucchia, J. Romei, Pro PHP Refactoring, Apress, Berkely, CA, USA, 2010.
[20] G. Sunyé, D. Pollet, Y.L. Traon, J.-M. Jézéquel, Refactoring UML Models, in: Proceedings of the International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, UML, 2001, pp. 134–148.
[21] B. Weber, M. Reichert, J. Mendling, H.A. Reijers, Refactoring large process model repositories, Comput. Ind. 62 (5) (2011) 467–486.
[22] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena, Refactoring product lines, in: Proceedings of the International Conference on Generative Programming and Component Engineering, GPCE, 2006, pp. 201–210.
[23] S.W. Ambler, P.J. Sadalage, Refactoring Databases: Evolutionary Database Design, Addison-Wesley, 2006.
[24] E. Harold, Refactoring HTML: Improving the Design of Existing Web Applications, Addison-Wesley, 2008.
[25] E.-G. Talbi, Metaheuristics: From Design to Implementation, Wiley Publishing, 2009.
[26] C.A.C. Coello, G.B. Lamont, D.A.V. Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[27] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evolut. Comput. 6 (2) (2002) 182–197.
[28] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the Strength Pareto Evolutionary Algorithm, Technical Report, Department of Electrical Engineering, Swiss Federal Institute of Technology, 2001.
[29] M. Petticrew, H. Roberts, Systematic Reviews in the Social Sciences: A practical guide, Blackwell Publishing, Oxford, 2006.
[30] A. Ghannem, M. Kessentini, G. El Boussaidi, Detecting model refactoring opportunities using heuristic search, in: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, CASCON, 2011, pp. 175–187.
[31] A. Ghannem, G. El Boussaidi, M. Kessentini, On the use of design defect examples to detect model refactoring opportunities, Softw. Qual. J. (2015) 1–19.
[32] W. Kessentini, M. Kessentini, H.A. Sahraoui, S. Bechikh, A. Ouni, A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, IEEE Trans. Softw. Eng. 40 (9) (2014) 841–861.
[33] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Hemati Moghadam, Experimental assessment of software metrics using automated refactoring, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM, 2012, pp. 49–58.

[34] C. Simons, J. Singer, D.R. White, Search-based refactoring: Metrics are not enough, in: Proceedings of the International Symposium on Search Based Software Engineering, in: SSBSE, 2015, pp. 47–61.
[35] F. Qayum, R. Heckel, Analysing refactoring dependencies using unfolding of graph transformation systems, in: Proceedings of the International Conference on Frontiers of Information Technology, FIT, 2009, pp. 15:1–15:5.
[36] F. Qayum, R. Heckel, Local search-based refactoring as graph transformation, in: Proceedings of the International Symposium on Search Based Software Engineering, SSBSE, 2010, pp. 43–46.
[37] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley, Boston, MA, USA, 1995.
[38] M. Ó Cinnéide, Automated Application of Design Patterns: A Refactoring Approach Ph.D. thesis, Trinity College, University of Dublin, 2000.
[39] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Trans. Softw. Eng. 28 (1) (2002) 4–17.
[40] E. Yourdon, L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1st ed., Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
[41] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Tran. Softw. Eng. 20 (6) (1994) 476–493.
[42] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, 2nd, PWS Publishing Co., Boston, MA, USA, 1998.
[43] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 22nd, Pearson Education, 2003.
[44] S. Siegel, N. Castellan, Nonparametric statistics for the behavioral sciences, 2nd, McGraw–Hill, Inc., 1988.
[45] E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, V. da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evolut. Comput. 7 (2) (2003) 117–132.
[46] M. Kim, M. Gee, A. Loh, N. Rachatasumrit, Ref-finder: a refactoring reconstruction tool based on logic query templates, in: Proceedings of the International Symposium on Foundations of Software Engineering, in: FSE, 2009, pp. 371–372.
[47] S. Counsell, R.M. Hierons, H. Hamza, S. Black, M. Durrand, Exploring the Eradication of Code Smells: An Empirical and Theoretical Perspective, Advances in Software Engineering 2010 (2010) 12.
[48] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of code smells in object-oriented systems, Innov. Syst. Softw. Eng. 10 (1) (2014) 3–18".
[49] M. Harman, E. Burke, J. Clark, X. Yao, Dynamic adaptive search based software engineering, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, in: ESEM, 2012, pp. 1–8.
[50] E.K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, R. Qu, Hyper-heuristics: A survey of the state of the art, J. Oper. Res. Soc. 64 (12) (2013) 1695–1724.
[51] Y. Jia, M. Cohen, M. Harman, J. Petke, Learning combinatorial interaction test generation strategies using hyperheuristic search, in: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), 2015.
[52] Y. Jia, Hyperheuristic Search for SBST, in: Proceedings of the Eighth International Workshop on Search-Based Software Testing, IEEE Press, 2015, pp. 15–16.
[53] G. Guizzo, G.M. Fritsche, S.R. Vergilio, A.T.R. Pozo, A hyper-heuristic for the multi-objective integration and test order problem, in: Proceedings of the Genetic and Evolutionary Computation Conference, in: GECCO, ACM, 2015.
[54] T. Mariani, G. Guizzo, S.R. Vergilio, A.T.R. Pozo, A grammatical evolution hyper--heuristic for the integration and test order problem, in: Proceedings of the Genetic and Evolutionary Computation Conference, in: GECCO, ACM, 2016. To appear
[55] A.C. Kumari, K. Srinivas, Software module clustering using a fast multi-objective hyper-heuristic evolutionary algorithm, Int. J. Appl. Inf. Syst. 5 (6) (2013) 12–18.
[56] M.P. Basgalupp, R.C. Barros, T.S. da Silva, A.C.P.L.F. Carvalho, Software effort prediction: a hyper-heuristic decision-tree based approach, in: Proceedings of the 28th SAC, ACM, 2013, pp. 1109–1116.
[57] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.