Empirical Software Engineering (2022) 27: 110
https://doi.org/10.1007/s10664-022-10151-4

Generation of refactoring algorithms by grammatical evolution

Thainá Mariani[1] · Marouane Kessentini[2] · Silvia Regina Vergilio[1]

Accepted: 18 March 2022 / Published online: 30 May 2022

Abstract

Recent machine learning studies present accurate results generating prediction models to identify refactoring operations for a program. However, such works are limited to prediction, i.e., they learn refactoring operations strictly as applied by developers, but there are possibilities that they might not consider. On the other hand, the *Search-Based Software Refactoring (SBR)* field applies search algorithms to find refactoring operations in a vast space of possibilities to improve diverse quality attributes. Nevertheless, existing SBR approaches do not generate a model as machine learning studies, and then, they need to be reapplied individually for each program needing refactoring. To mitigate this limitation, this work introduces a novel SBR learning approach that generates refactoring algorithms capable of providing refactoring operations to several programs. These algorithms are composed of procedures that use rules to determine the refactoring operations. To create the algorithms, 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 applied to generate the algorithms based on a grammar encompassing details of the extracted patterns. GE works to generate an algorithm that provides refactoring operations similar to those applied in practice while improving quality attributes, such as modularity. The approach is evaluated using refactoring data from 40 Java programs of GitHub repositories. The algorithms are tested against different programs, obtaining an overall average of 60% of modularity improvement and 50% of similarity with actual refactoring operations.

Communicated by: Aldeida Aleti, Annibale Panichella and Shin Yoo

This article belongs to the Topical Collection: *Advances in Search-Based Software Engineering (SSBSE)*

✉ Thainá Mariani
marianithaina@gmail.com

Marouane Kessentini
marouane@umich.edu

Silvia Regina Vergilio
silvia@inf.ufpr.br

[1] Federal University of Parana, Curitiba, Brazil

[2] University of Michigan, Dearborn, Michigan, USA

Keywords Search-based software refactoring · Grammatical evolution · Machine learning

1 Introduction

Software refactoring is an expensive and error-prone activity. Mainly, software maintenance consumes up to 70% of the total cost of a typical software project (Sjøberg et al. 2013). Software refactoring is usually performed by experienced developers, which contributes to increasing the total cost (AlOmar et al. 2020). Researchers also point out that developers spend at least 10% of their monthly hours with refactoring tasks (Kim et al. 2014; Murphy-Hill et al. 2012). Moreover, identifying the best refactoring type for each situation is very time-consuming. Several refactoring types and hundreds of code elements that might need refactoring can be used. In this context, the refactoring problem is highly studied in the literature (Abid et al. 2020; Baqais and Alshayeb 2020). It consists of finding a good set of refactoring operations for a software program.

Some tools can help in this task (AutoRefactor 2021; Spartan Refactoring 2021; Moore 1996; Tsantalis et al. 2008). They are usually integrated into most 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 (Fowler and Beck 2018) or duplicate code. However, they do not consider other important aspects, such as the improvement of quality attributes. Despite the existence of tools, some works point out around 80% of refactoring tasks are manually performed. A survey conducted with 328 software engineers of Microsoft shows that design defects are not the main reason developers apply refactoring operations (Murphy-Hill et al. 2012). They most see benefit by improving quality attributes, such as readability, maintainability, and modularity.

In this context, some refactoring approaches have been proposed in the *Search-Based Software Refactoring (SBR)* field (Mariani and Vergilio 2016). SBR works apply search-based techniques to help the software refactoring activity. Approaches in this field have gained visibility because they can identify refactoring operations for a program by optimizing several quality attributes. Such approaches have been pointed out as the most beneficial for the software refactoring activity (Baqais and Alshayeb 2020). However, existing SBR approaches usually require the execution and configuration of a search-based algorithm for every new version or program, which can be costly and demand effort from the developer, who may not be familiar with a search algorithm.

In this work, we observe that we can learn refactoring patterns from various software programs and provide a more generic 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 study using deep learning shows only between 21% and 36% of code changes can be automatically learned (Tufano et al. 2019). Refactoring was classified as one of the learned code changes. Moreover, learning with multiple software programs increases accuracy by around 10%. This shows evidence that refactoring patterns can be learned from different programs.

Recently, some approaches have explored the use of *Machine Learning (ML)* techniques to predict refactoring operations (Aniche et al. 2020; Dallal 2017; Kumar et al. 2019; Xu et al. 2017). They obtain good results in terms of prediction, but 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 criterion. We want to explore the search space to find other solutions which might also be good regarding quality aspects.

Concerning the presented context, the main goal of this work is to generate algorithms capable of suggesting, for a program, refactoring operations able to improve quality and increase similarity with real refactoring operations. In this sense, we propose GORGEOUS (*Generation of Refactoring Algorithms by Grammatical Evolution)*, which is an SBR machine-learning-based approach to the refactoring problem. Our approach uses a *Grammatical Evolution (GE)* (Ryan et al. 1998) technique. GE is a type of *Genetic Programming (GP)* (Koza 1992) since it is similarly used to evolve programs (Ryan et al. 1998). However, in addition, GE receives a grammar file, which is very flexible and supports mapping several aspects of a program. This work uses the concept of refactoring algorithm, as well as a grammar that formalizes a set of rules identifying where and how refactorings should be applied. In this way, GE supports the generation of more complex algorithms compared with solutions generated by traditional machine learning techniques.

GORGEOUS encompasses the learning of refactoring patterns applied in different programs. To this end, it executes a clustering algorithm to group classes and methods that were refactored in similar ways. Each generated cluster represents a refactoring pattern. Then, in a posterior step and using the obtained patterns, GORGEOUS generates a refactoring algorithm that generalizes the characteristics of each cluster.

We evaluated the approach with 40 open-source Java programs, extracted from *GitbHub* and considering the following refactorings at class and method levels: *Extract Class*, *Move Method*, *Pull-up Method*, *Push-down Method*, *Extract Method*, and *Inline Method*. The results provide evidence to support the claim that our proposal can improve the modularity of the programs and similarity with real refactoring operations. In summary, our work has the following main contributions.

1. It introduces an SBR approach that incorporates a learning step and generates refactoring algorithms, which, when executed for a given program, identifies refactoring solutions for it.
2. It introduces two grammars to be used to generate a refactoring algorithm at class and method levels.
3. It presents the main advantages compared with related work:
   - Regarding SBR approaches: the generated algorithms include a set of rules that works as a prediction model of refactoring opportunities;
   - Regarding ML approaches: to generate the algorithms, a search in a huge space of alternatives is performed, optimizing quality attributes such as modularity while considering similarity with refactoring patterns applied in different programs.
4. We make available a repository with the data used in this work, which allows replication and can be used in future research (Mariani et al. 2021).

The paper is organized as follows. Section 2 reviews related work. Section 3 introduces our approach. Section 4 describes how GORGEOUS was evaluated. The obtained results are presented in Section 5. Section 5.6 analyses the threats to validity. Finally, Section 6 concludes the paper and shows future research directions.

2 Related Work

Related work are from three main groups: work applying Machine Learning (ML) techniques, work from the Search-Based Software Refactoring (SBR) field, and work combining both ML and SBR.

Search-based techniques have been successfully applied to software refactoring and related activities. These techniques are the subject of different surveys and reviews in the literature (Kaur and Dhiman 2019; Mariani and Vergilio 2016; Mohan and Greer 2018). Among several SBR studies reported in these reviews, the most similar to ours are the ones that use external information to guide the search. For instance, some SBR papers use the concept of examples to guide the optimization process (Mariani and Vergilio 2016). Examples are usually refactoring operations applied in previous versions of a program. This kind of approach commonly has as output a refactoring solution that was optimized aiming at increasing the similarity with refactorings applied in the past (Ouni et al. 2013). This metric is usually used in the fitness function in combination with other quality metrics, such as number of modifications (Mkaouer et al. 2015; Ouni et al. 2014; Ouni et al. 2016), semantic coherence (Mkaouer et al. 2015; Ouni et al. 2013; Ouni et al. 2013), and number of bad smells (Kessentini et al. 2012; Ouni et al. 2015). We can find one work with a different output, which is a set of rules used to detect code smells and correct them Mahouachi et al. (2012). We also find interactive approaches (Alizadeh et al. 2020; Mkaouer et al. 2014a), which are usually based on the user preference and can be costly, mainly if they consider the user participating in the loop optimization process.

Although these SBR approaches present good results, they optimize a set of possible refactoring solutions individually for a specific program, leading to a lack of generality. Furthermore, a certain level of expertise is required to configure and execute a search algorithm, which is not trivial for a software engineer, especially if this has to be done several times for different programs. Our work encompasses a learning process to overcome these limitations.

Most ML works use classification techniques to generate a model to predict refactorings. The approach of Imazato et al. (2017) automatically obtains a model to generate a list of methods for the application of the *Extract Method* refactoring. The approach of Kosker et al. (2009) predicts refactoring actors by using the Weighted Naïve Bayes classifier. The main goal is to predict which classes require refactoring to decrease complexity, maintenance cost, and bad smells. 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 (2007) 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.

The works of Al Dallal (2012) and Dallal (2017) use a logistic regression algorithm to predict classes in need of the *Extract Subclass* and *Move Method* refactoring opportunities. 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. Results show a rate of 83.4% to 95.8% of prediction.

Xu et al. (2017) propose an ML approach for extract method refactoring recommendation. The approach generates a probabilistic model built based on structural features related to complexity and function features related to cohesion and coupling. The model learns from a set of positive and negative method extraction examples in the learning process.

The experiments were performed using five different Java projects, and the obtained results outperformed results from popular refactoring tools in terms of different metrics, such as precision and recall.

Kumar et al. (2019) investigate 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 three different sampling methods to deal with class imbalance. Results show accuracy around 98% for AdaBoost and ANN+GD, classifiers that presented the best results.

Aniche et al. (2020) applied six ML algorithms to a dataset comprising over two million refactorings from 11,149 real-world projects from the Apache, F-Droid, and GitHub ecosystems. The study includes a comprehensive set of 20 different refactorings at class, method, and variable-levels. The best classifier found was Random Forest, with accuracy often higher than 90%. The best predictors were process and ownership metrics. The work of Alenezi et al. (2020) evaluates the performance of the deep Gated Recurrent Unit (GRU) algorithm for refactoring prediction at class-level on seven open systems. The performance of the algorithms improves when balanced datasets are used.

The main limitation of the mentioned works (Imazato et al. 2017; Jindal and Khurana 2013; Kosker et al. 2009; Phongpaibul and Boehm 2007) 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 cannot generalize the results in other programs. Regarding the other studies, most of them generate a model based on only one refactoring type. This is mainly because most of them use binary classifiers, which can classify an element into two different categories. Generally, the approach determines whether a code element should receive a predefined refactoring, e.g. *Move Method* in Dallal (2017), or either that a module or element is a candidate to be refactored, such as presented by Kumar et al. (2019). Another limitation of some works (Al Dallal 2012; Dallal 2017) is to use artificial data instead of actual software programs. Moreover, works as Tufano et al. (2019) and Xu et al. (2017) use real software programs, but they do not generate models to predict specifically refactorings. Finally, the work of Aniche et al. (2020) introduces complex models based on existing programs, but it is limited to prediction and does not consider the improvement of quality attributes.

The use of ML and SBR techniques combined is also explored in the literature. Some works use ML models to improve the fitness evaluation function of an SBR approach (Amal et al. 2014; Wang et al. 2015). The approach of Amal et al. (2014) uses a neural network to approximate the fitness function for the evaluation of software refactoring solutions. This approach has a different goal than ours. The idea is to reduce the effort of a preference-based approach, including the user in the loop. Wang et al. (2015) applies the NSGA-II to generate refactoring solutions that maximize the correction of essential quality attributes and minimize the effort. For these two fitness functions, time series forecasting is used to estimate the impact of the generated refactoring solutions on future subsequent releases of the system in order to manage technical debt, with a focus on smells.

This approach differs from ours because it does not consider past refactorings as input for the learning phase. However, the fitness functions used and the adaptations using time series could be adopted by the GE algorithm used in our approach.

We can see that our approach has different goals from existing approaches that combine SBR and ML. The algorithms generated by our approach take into account different kinds of refactorings and a set of different programs, exploring the advantages of both SBR and

ML approaches. The algorithms work as a prediction model and are generated to optimize quality attributes and similarity with past refactoring operations.

3 Gorgeous

This section presents our approach, namely Gorgeous (*Generation of Refactoring Algorithms by Grammatical Evolution)*. Given information of refactoring previously applied in a set of programs, Gorgeous addresses the refactoring problem by generating refactoring algorithms, when executed produce a set of refactoring operations for a given program. A refactoring operation specifies the type of refactoring and elements involved (actors). In this way, it is possible to apply the refactoring type to the actors specified.

To this end, Gorgeous encompasses three main steps as depicted in Figure 1. The first step, *Deriving Instances*, transforms the input information, associated with programs that received refactorings in the past, into programs instances by using a representation schema (Section 3.2). The input information (Section 3.1) concerns, for each program, metric values of its elements (classes or methods), dependencies between these elements, and a list of refactoring operations applied. These programs instances are used in the next two steps.

After this, the step *Learning Patterns* uses a clustering algorithm to group the elements of the program based on similar refactorings they received in the past. As a result of this step, each cluster generated represents a refactoring pattern. In the next step, *Generating Algorithms*, refactorings algorithms are generated for each learned pattern by using Grammatical Evolution (GE). The GE algorithm searches for the best algorithms to maximize some quality attributes regarding the refactoring applied. Later, these algorithms can be executed to find refactoring operations for a given software program. In the following subsections, we describe these steps in detail, as well as the required input and output produced by the approach.

3.1 Programs Information

The information provided as input includes for each program: a metrics matrix, a dependencies matrix, and a refactoring list. The metrics matrix, ME_{n_e,n_m}, represents the values of n_m metrics for each element e in the program. In this work, the elements can be either

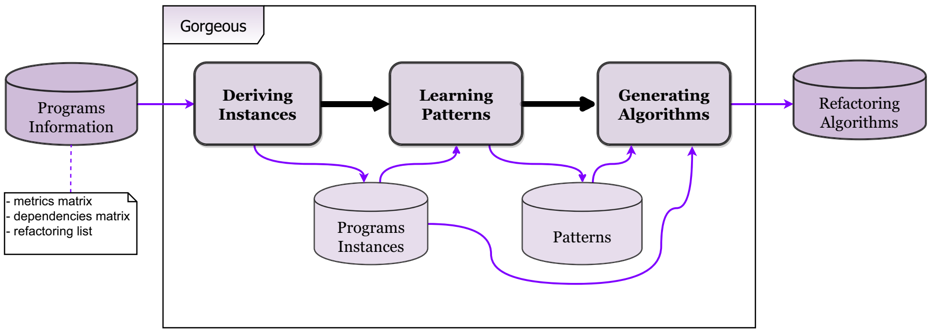


Fig. 1 Gorgeous overview

classes or methods. The classes are associated with six metrics and the methods with two, as described in Table 1. The characteristics of the elements are represented by basic structural object-oriented metrics related to cohesion, coupling, and size at class and method levels. They were chosen because they are related to design principles and models for quality assessment of different attributes of OO oriented design such as QMOOD (Bansiya and Davis 2002), largely adopted in search-based refactoring approaches (Koc et al. 2011; Koc et al. 2012; Mansoor et al. 2015; Mariani and Vergilio 2016; Mkaouer et al. 2015; Mkaouer et al. 2014b).

The dependencies matrix $D_{n_p,3}$, where n_p is the number of pairs of classes of the program, contains three columns representing for each pair of classes, respectively, the number of dependencies between the pair, the number of dependencies from the first class in the pair to the second one, and the number of dependencies from the second class to the first one;

The refactoring list L is composed of the refactoring operations applied in the program. They are listed in a descriptive format for each refactoring type, including the refactoring type and actors, elements that participate in the operation. The format defined for each type and some examples are provided in Table 2. More details about the structure of the input files are found in our replication package (Mariani et al. 2021). The metrics and dependencies matrices are provided for each program in CSV files. The list L of refactoring operations applied to a program is provided in a file whose each line corresponds to an operation.

3.2 Deriving Instances

In the first step, *Deriving Instances*, program instances are derived from the information provided as input. To represent a program instance, we use the schema depicted in Figure 2. In this schema, a program is represented by its name and version and is composed of different elements that received a refactoring operation in the past. An element can be a class or a method and plays an actor in one or more refactoring operations. Moreover, classes may have dependencies with other classes. A set of metrics values characterizes each element. The instances created by using this schema are used to derive the inputs for the clustering algorithms and the GE in the following steps.

This step can be performed with the help of some automated tools. In this work, we use *RefactoringMiner*[1] to generate the list of applied refactorings, as well as the tool *Understand* tool[2] to extract metrics and information about the structure of the programs that received the refactoring operators.

3.3 Learning Patterns

The second step, *Learning Patterns*, is in charge of learning patterns from the instances of the program derived in the previous step. To this end, a clustering algorithm is used to group elements (classes or methods) that were refactored similarly. The input of the clustering algorithm is generated by extracting the refactoring operations from the instances of the

[1] https://github.com/tsantalis/RefactoringMiner
[2] https://scitools.com/features/

Table 1 Metrics calculated to an element (Bansiya and Davis 2002)

Element	Metric	Description
class	m_1	Number of immediate base classes
	m_2	Number of classes coupled
	m_3	Number of classes derived
	m_4	Number of methods
	m_5	Number of all methods
	m_6	Max Inheritance Tree
class / method	m_7	Number of lines of code
	m_8	Number of commented lines of code

Table 2 Refactoring Types and Representation Format

ExtractClass(class)
ExtractClass(org.neo4j.kernel.api.constraints.UniquenessConstraint)
ExtractMethod(method)
ExtractMethod(org.neo4j.kernel.impl.store.TestNeoStore.deleteNode2())
InlineMethod(source_class.method,target_class.method)
InlineMethod(org.neo.ComLockClient.releaseAllExclusive(),org.neo.ComLockClient.releaseAll())
MoveMethod(*source_class.method*,*target_class*)
MoveMethod(org.neo4j.kernel.util.IoPrimitive.arrayAsCollection(),org.neo4j.graphdb.Neo4)
PushDownMethod(source_class.method,target_class)
PushDownMethod(org.grad.GradleRunner.setTasks(),org.grad.DefaultGradleRunner)
PullUpMethod(source_class.method,target_class)
PullUpMethod(org.volt.planner.TestPlansAx.findAllPlanNodes(),org.volt.planner.PlannerTestCase)

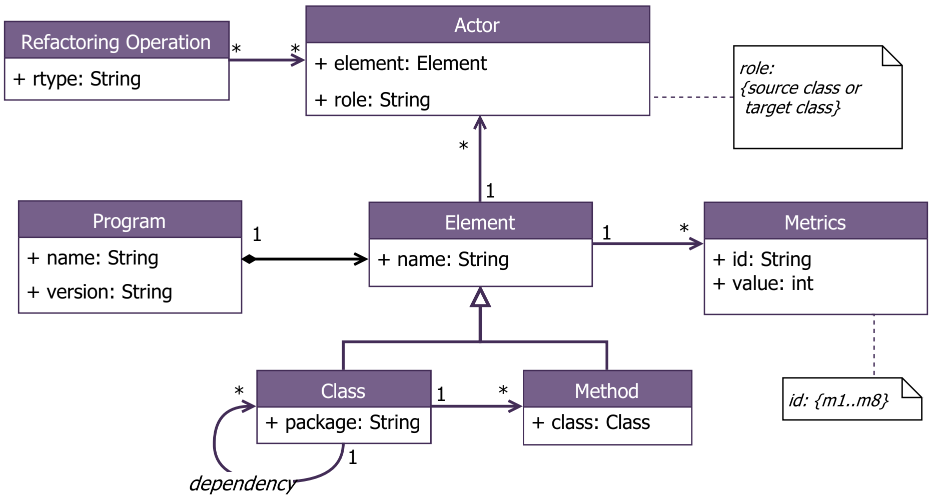


Fig. 2 Program Representation Schema

program, as depicted in Table 3. Each row i represents an element e_i, and the columns represent the number of refactoring types applied in e_i. Two matrices are generated, one containing classes and the other containing methods.

The clustering is performed by considering the number of times each refactoring type was applied in 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 refactored many times from elements receiving usual refactoring operations, 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 that such an element can benefit from a refactoring operation, but it does not necessarily mean it needs to be precisely the same refactoring type. We are willing to find this kind of pattern 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. As the output of each execution, a set of clusters C is generated. Each cluster c, composed of a set of elements E, is interpreted as a refactoring pattern and is used in the next phase to create a refactoring algorithm. In the database of programs instances, each element is associated with a set of metrics values, and with refactoring operations, it was involved. When an element belongs to a cluster, all this information is part of the cluster itself, helping to characterize a refactoring pattern.

3.4 Generating Algorithms

The refactoring problem consists of finding pieces of code that would benefit from applying certain types of refactorings, for instance, a method that should be moved. The last step, *Generating Algorithms*, is responsible for producing the GORGEOUS output, which is a set of refactoring algorithms to solve the refactoring problem. For this end, a GE algorithm is executed for each pattern/cluster obtained in the *Learning patterns* step. In each GE execution, a refactoring algorithm is produced. Each refactoring algorithm, when executed, suggests a set of refactoring operations for a given program to be refactored.

A Grammatical Evolution (GE) algorithm is a type of GP since it is similarly used to evolve programs (Ryan et al. 1998). However, while a conventional GP algorithm typically uses a tree as representation for an individual and applies search operators to those trees (Koza 1992), a GE algorithm uses an array of integers or bits and evolves the solutions similar to a conventional evolutionary algorithm (Ryan et al. 1998). Moreover, in addition to the usual parameters of an evolutionary algorithm, 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 (GPM)*. The evolution is applied to the array (genotype level), but to calculate the fitness function value, the program (phenotype level) needs to be executed (Barros et al. 2013).

Our GE uses an array of integers, and we defined two grammars, one for mapping the solutions to a refactoring algorithm at the class level and the other for the method level. In the following subsections, we describe the format of the refactoring algorithms (phenotype) and after, the grammars defined, and how the GE algorithm uses them.

3.4.1 Refactoring Algorithms

Our definition of a refactoring algorithm is based on the formalization given by Cormen et al. (2009), 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 1 shows the structure of the algorithms produced. An algorithm A is executed for a program P and returns a set of refactoring solutions S for P. A requires as input the metrics and dependencies matrices in the same format provided for Step 1 (*Deriving Instances*), which will be used to derive a representation for P following the schema of Figure 2). Information about past refactoring (operators and actors) is not required at this stage. A refactoring solution s_i represents a suggestion of refactoring operation for P. It is composed of three parts: type of refactoring, actors, and roles, as exemplified in Table 4. In this table, the solution defines the refactoring type Move Method. This refactoring has three actors: the *getSalary()* method (the method to be moved) from the *Person* class (source class) to the *PaymentOptions* class (target class).

Algorithm 1 Pseudocode of Refactoring Algorithm.

```
1  Input: metrics and dependencies matrices for P
2  Output: set of refactoring solutions $S = (s_1, s_2..s_n)$
3  begin
4      Derive an instance to represent $P$ based on its metrics and dependencies;
5      $s_1$=procedure$_1$ (type$_1$, RU$_1$);
6      $s_2$=procedure$_2$ (type$_2$, RU$_2$);
7      ...
8      $s_n$=procedure$_n$ (type$_n$, RU$_n$);
9      procedure$_i$(type$_i$,RU$_i$)
10         repeat
11             actors.insert(search_actor(RU$_i$));
12         until all actors are selected;
13         Instantiate a solution $s_i$ based on type$_i$ and $actors$;
14         return $s_i$;
15     search_actor(RU$_i$)
16         foreach $e \in E$ do
17             if satisfies (e, RU$_i$) is 1 then
18                 select element $e$ as actor;
19                 return $e$;
20             end
21         end
22 end
```

A refactoring algorithm A is composed of n procedures. Each procedure is in charge of instantiating a solution s_i. A procedure is associated with a $type$ of refactoring and a set of rules RU used to search actors from elements of P. The way these rules are generated is described in Section 3.4.3. First, when executed, the refactoring algorithm search for actors that satisfy the set of rules RU. After, a refactoring solution is instantiated based on the type of refactoring and elements (actors) selected. We describe a procedure based on its refactoring type. If we mention a "move method procedure", it means a procedure responsible for instantiating a move method refactoring solution.

The search for an actor is associated with a set (RU) of rules that, based on an interval of values for the metrics m_1 to m_8 (Table 1), describes the characteristics of the actor we are

Table 3 Input of the clustering algorithm

Class	EC	MM	PU	PD	
Class-level					
$Class_1$	1	0	0	1	
$Class_2$	0	0	0	0	
$Class_3$	1	2	1	0	
$Class_5$	0	3	0	4	
Method	MM	PU	PD	EM	IM
Method-level					
$Method_1$	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

looking for. In this way, RU comprises six rules to search for classes and two rules to search for methods. A rule $(ru_j) \in RU$ defines an interval of values $[a, b]$ for a metric m_j. An element (e) from P is selected as an actor if all the rules in RU are satisfied. An individual rule ru_j is satisfied when the value of m_j for (e) belongs to the specified interval.

For example, suppose the procedure searches for an actor (source method) with $ru_7 = 10 \leq m_7 \leq 200$ and $ru_8 = 10 \leq m_8 \leq 20$, an element e (a method) with values of m_7 and m_8 fitting these ranges must be 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, an empty solution is returned, and the algorithm executes the next procedure until all procedures are executed. As a result, a set of refactoring solutions for P is returned.

3.4.2 Fitness Evaluation

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 1.

$$F(A) = (SIM(A) + MQ(A)) * 0.5 \tag{1}$$

The first function $SIM(A)$ measures the similarity between A and the set E of elements grouped in c. The second fitness function MQ (Paixao et al. 2018) measures the quality of a

Table 4 Example of refactoring solution for Move Method

Types	Actors	Roles
Move Method	Person	source class
	getSalary()	method
	PaymentOptions	target class

program after simulating the application of the refactoring solutions given by A. Next, each function is described.

The similarity function $SIM(A)$ (Equation 2) 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 $rsim(pr)$, by summing up the result for each procedure pr, where n is the number of procedures. Equation 3 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 presents $rsim(pr)$, which calculates the similarity by checking if each an element $e \in E$ satisfies the set of rules RU of a procedure pr.

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

$$tsim(pr, c) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 \; if \; pr_t = c_{t_i} \\ 0 \; if \; pr_t \neq c_{t_i} \end{cases} : \quad c_t \in c_T \tag{3}$$

$$rsim(pr, c) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 \; if \; satisfies(e_i, pr_{RU}) \\ 0 \; otherwise \end{cases} \tag{4}$$

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 5) 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.

$$MQ = \sum_{i=1}^{|Pack|} MF(pack_i) \tag{5}$$

where $MF(pack_i)$ (Equation 6) measures the cohesion and coupling of a package $pack_i$.

$$MF(pack_i) = \begin{cases} 0, & \text{if } coh(pack_i) = 0 \\ \frac{coh(pack_i)}{coh(pack_i) + \frac{cop(pack_i)}{2}}, & coh(pack_i) > 0 \end{cases} \tag{6}$$

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

3.4.3 Grammatical Evolution

The GE algorithm is executed at least once for each cluster generated in step *Learning Patterns*, considering both levels: class and methods. As mentioned before, the solution is represented by an array of integers (genotype). Then the population for the GE algorithm is a set of arrays. To calculate the fitness of each solution, the array is mapped to a refactoring algorithm (phenotype) with the help of a grammar. We defined two grammars, respectively, for the class and method levels. The grammars encompass all the elements necessary to derive the refactoring algorithm: the procedures, refactoring types, rules, and intervals. Figure 3 presents the grammar for the class-level. The method-level grammar is very similar, and the main differences are in the refactoring types applied.

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.

```
⟨procedure⟩ ::= ⟨type⟩ ⟨procedure⟩ | ⟨type⟩
      ⟨type⟩ ::= ⟨extractClass⟩ | ⟨moveMethod⟩ | ⟨pullUpMethod⟩ | ⟨pushDownMethod⟩
      ⟨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
```

Fig. 3 Grammar to create refactoring algorithms for the class-level.

For instance, the node ⟨*ru*1⟩ can take either the values `IntervalA` to `IntervalN` when mapped to a program. On the other hand, ⟨*procedure*⟩ can take the value of a single ⟨*type*⟩ or composition of ⟨*type*⟩ ⟨*procedure*⟩. If an alternative for a rule is not wanted by the developer, it can be removed. Similarly, if other types of refactoring are required, they can be added to the grammar.

As mentioned before, each refactoring algorithm is associated with procedures, and each procedure is associated with rules, which are associated with several intervals of metrics. The procedure representation is presented in Figure 4. The intervals are created at run-time

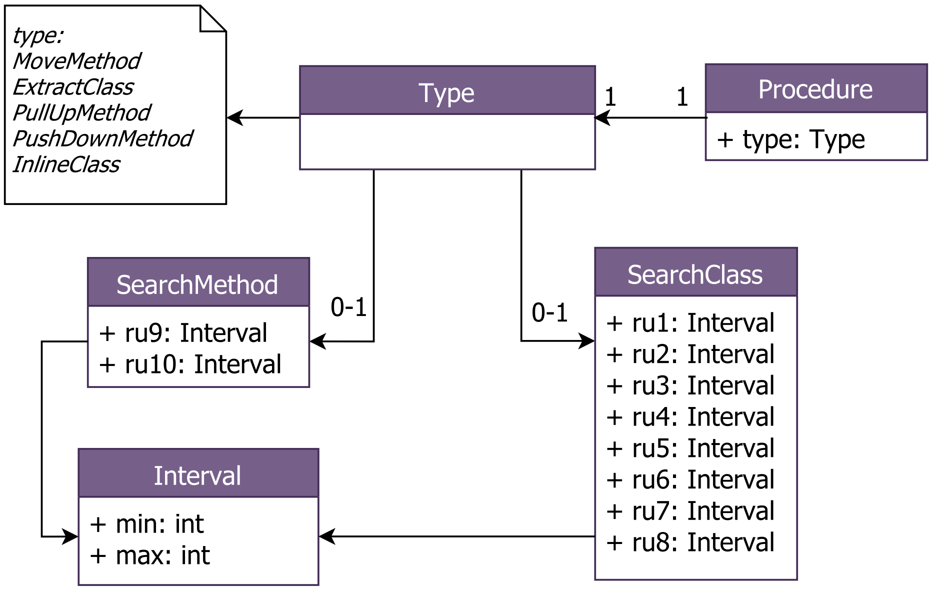


Fig. 4 Procedure Representation

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_j, an interval of size s is generated respecting the minimum and maximum values of m_j, 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 maximum metric 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.

GE searches for the best combination of rules and refactoring types, aiming at improving the fitness value. Each individual, represented by an array of integers, is mapped to a program to calculate the fitness. The values in the array are used to decide which grammar values are assigned to each node. For instance, consider the following individual {5, 9, 79, ...}. The value of the first gene in the chromosome (5) and the first rule of the grammar $\langle procedure \rangle$ is selected. Thus, the number of options in $\langle procedure \rangle$ is counted. In that case, there are two options: $\langle type \rangle$ $\langle procedure \rangle$ or $\langle type \rangle$. Then, the modulo operation 5%2 is performed, considering the gene value and the number of options. The result of this operation represents the gene position. Since it is 1, the second rule $\langle type \rangle$ is selected. In the next step, the rule $\langle type \rangle$ and the second gene of the chromosome (9) are selected. There are four options for $\langle type \rangle$. Then, the modulo operation 9%4 returns 1, and the rule $\langle extractClass \rangle$ is selected. This process continues until reaching the terminal nodes.

The employed array has a variable size. Because of that, in addition to crossover and mutation operators applied in the same way as an evolutionary algorithm, the GE algorithm employs two distinguishable search operators: i) gene duplication operator; and ii) gene pruning operator. They help the algorithm 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. On the other hand, the prune operator 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 (Ryan et al. 1998).

To summarize, the GE algorithm works very similarly to a traditional evolutionary algorithm, replacing the evaluation of the population by GPM and adding duplication and prune operators.

In the end, 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 then be executed over the program to be refactored to find refactoring operations. The use of grammar and GE ensures the generation of only valid algorithms, making the resulting solutions always syntactically correct.

4 Empirical Evaluation

The evaluation goal is to analyze if our approach generates refactoring algorithms that provide good refactoring solutions in terms of quality and similarity with refactoring operations from existing software programs. Moreover, we also validate GORGEOUS phases by assessing the importance of the learning patterns phase and the use of GE. The following sections present the elaborated *Research questions (RQs)* and the experiment setting.

4.1 Research Questions

RQ1: To what extent the grammatical evolution impacts the generation of refactoring algorithms? To answer this RQ, we compare, in terms of fitness values, the results obtained by GORGEOUS using GE with a configuration of GORGEOUS in which a Random Search replaced GE. In addition, we applied the Mann and Whitney (1947) non-parametric test to give more confidence to the comparison.

RQ2: To what extent the refactoring algorithms are able to find refactoring solutions capable of improving the quality of programs? To measure the quality of the programs, 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 5. Then, we assessed how much was the quality improvement compared with the original program. Equation 7 presents how QI is calculated, where P_o is the original program and P is the program after the refactoring simulation.

$$QI(P) = \frac{MQ(P) - MQ(P_o)}{MQ(P_o)} * 100 \tag{7}$$

We also analyzed the quality improvement of refactoring operations applied in practice by developers. In this case, we computed QI based on two program versions, considering before and after the refactoring operations, and compared with the ones obtained by GORGEOUS.

RQ3: To what extent the generated refactoring algorithms are able to find solutions similar to refactoring operations applied in practice? To measure similarity in comparison with these operations, we defined as O, the operations applied in 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 (Powers 2011). However, 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 8 presents $ARate$ definition, such that, Pr is the set of procedures from A, n_o is the size of O, pr_{RU} is the set of rules of pr, and $pr \in Pr$.

$$ARate(Pr, O) = \frac{1}{n_o} \sum_{i=1}^{n_o} \begin{cases} 1 \ if \ \ \exists pr \in Pr : satisfies(o_i, pr_{RU}) = 1 \\ 0 \qquad\qquad\qquad otherwise \end{cases} \tag{8}$$

RQ4: To what extent does the extraction of patterns impact the generation of refactoring algorithms? The refactoring patterns are represented by the clusters, 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 *Learning Patterns* is not performed.

4.2 Experimental Setting

To evaluate our approach, we use 40 popular Java programs of several sizes and domains extracted from software repositories of *GitHub*. These programs belong to the dataset produced by Silva et al. (2016)[3]. This work provides lists of refactoring operations extracted from programs using the *RefactoringMiner*. For each program, we collected the required information to execute GORGEOUS following our program representation schema. First, we collected existing data about the refactoring operations associated with each program. Then, we have downloaded from *GitHub* 80 versions of the programs, 40 before the refactoring operations, and 40 after the refactoring operations. The versions after the refactoring operations were used for 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.

We use the tool *Understand* to extract information about the programs, such as signatures and packages they belong to, as well as metrics and dependencies among elements, which are available in our replication package (Mariani et al. 2021). The information from all programs is given as input to GORGEOUS, which manipulates the programs based on the representation schema presented in Figure 2.

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

The program versions and other details are presented in Table 5. They correspond to the version right before the refactoring applications. For each program, it is presented the number of classes (NC), the number of lines of code (LOC), the original values of Modularization Quality (MQ), and the number of refactoring operations (NO).

Concerning the clustering algorithm, we adopted the Expectation Maximization (EM) (Dempster et al. 1977) algorithm. This algorithm uses a probabilistic model to give a probability of each object to belong to each cluster (Tan et al. 2005). EM generates a mixture distribution for the whole population of objects, composed of several different individual distributions. The parameters, such as mean and variance, are estimated using Maximum Likelihood Estimate (MLE). In this way, each cluster is represented by an individual distribution, and its parameter values represent the patterns for a cluster (Dempster et al. 1977). Initially, the parameters are unknown, then EM estimates the parameters and tries to guess objects that are more likely to fit the distribution. After, the algorithm recalculates the individual distributions' parameters to maximize the expected probabilities. Both steps continue until convergence (parameters are not changing) or until a prefixed number of interactions is achieved (Dempster et al. 1977).

We use the framework *Weka* (Witten and Frank 1999) that provides an EM implementation in which no data distribution needs to be assumed. Also, a 10-fold cross-validation is automatically performed by *Weka* to select a configuration with the best number of clusters. The cross-validation performed by *Weka* to determine the number of clusters is done in the following steps. First, the number of clusters is set to 1, and the training set is split randomly into 10 folds. Then EM is performed 10 times using the 10 folds, and the *loglikelihood* is averaged over all 10 results. If the *loglikelihood* has increased, the number of clusters is increased by 1, and the program continues by splitting the training set again.

[3]http://aserg-ufmg.github.io/why-we-refactor/#/projects.

Table 5 Program details

Fold	Program	NC	LOC	MQ	NO
	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
1	Fabric8 2.1.11	974	47,563	0.1865	6
	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
2	Graylog 1.2.0	1,947	83,566	0.1380	7
	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
3	Voltdb 5.2.3	5,466	461,750	0.1457	16
	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
4	Quasar 0.7.0	1,346	55,641	0.0227	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
5	TextSecure 2.19	850	44,182	0.1826	5
	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
6	Tomahawk A. 0.83	494	26,715	0.1918	5
	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
7	Tachyon 0.6.4	1,092	72,404	0.1445	7
	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
8	WordPress A. 4.0	1,410	67,364	0.0332	6
	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
9	Jersey 2.19	6,822	216,828	0.2092	11
	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
10	Openhab 1.7.0	4,041	264,756	0.0040	5

We integrated *Weka* in our approach using its Java API. 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 before, the clustering is executed separately for classes and methods. *Weka* (Witten and Frank 1999) automatically found the best number of clusters. In 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.

We performed 30 runs of GE for each cluster to generate the refactoring algorithms. GORGEOUS implementation is based on Java, and the framework jMetal (Durillo and Nebro 2011) is employed to support the implementation and execution of GE. EM and GE parameters were selected from the literature, as well as the number of runs (Mariani et al. 2016). We also set a maximum of 20 genes by chromosome, which means 20 procedures for a refactoring algorithm. It was based on the numbers of refactoring operations analyzed in the programs. Table 6 shows the configuration of the GE algorithm.

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

To answer the research questions of this study, we defined two more configurations named RANDOM (RQ1) and NOCLUSTER (RQ4). RANDOM performs a Random Search instead of GE, but we kept the other approach details in the same way, such as the grammars and fitness function. Also, we set for these configurations the same possibility in terms of iterations. In this sense, RANDOM iterates 10,000 times and executes 30 times for each cluster as well. NOCLUSTER also has the same parameters configuration, but it does not perform the *Learning Patterns* step. In this case, no cluster is provided, and GE receives the whole set of elements as input. Results obtained by GORGEOUS and these configurations are presented next.

5 Results

This section presents and discusses the results obtained, answering the posed RQs.

Table 6 GE Algorithm Configuration

Parameter	Value
Initialization	RANDOM
Population Size	100
Number of GE Fitness Evaluations	10,000
Crossover Operator	Single Point
Crossover Probability	90%
Mutation Operator	Integer Mutation
Mutation, Pruning and Duplication Probabilities	1%
Selection Operator	Binary Tournament
Maximum Gene Size	20

5.1 RQ1: GE x RANDOM

To answer RQ1, we compared results obtained by GORGEOUS and RANDOM configuration analyzing their fitness function values. By considering all folds, GORGEOUS generated 60 clusters, being 26 for the class-level and 34 for the method-level. Based on each cluster, GE and RANDOM were executed 30 times each. Then, we computed, for each combination of fold and cluster, the average of the 30 obtained fitness values, along with their standard deviation. In this respect, we compared the 60 averages obtained by GORGEOUS against the ones obtained by RANDOM. As another source of comparison, we executed the Cohen's effect size (Cohen 2013). The difference is classified as *large*, *medium* or *small*, where *large* means bigger than 0.5, *medium* means between 0.1 and 0.3, and *small* means smaller than 0.1.

These values are presented by fold and algorithm, respectively, at the class and method level, in Tables 7 and 8. Each value represents the average of 30 fitness values based on the 30 runs and the standard deviation between parenthesis. Each algorithm (Alg.), represented

Table 7 Fitness values of GORGEOUS and RANDOM Search for the class level

Fold	Alg.	NE	NS	GORGEOUS	RANDOM	p-value	ES
	0	7	10	**0.6937** (0.0311)	0.5859 (0.0240)	3.88E-11	Large
	1	21	65	**0.4794** (0.0205)	0.4380 (0.0196)	1.93E-08	Large
1	2	12	23	**0.5053** (0.0338)	0.4740 (0.0185)	2.84E-04	Large
	0	5	10	**0.6813** (0.0209)	0.5977 (0.0175)	1.19E-07	Large
	1	10	49	**0.7563** (0.0265)	0.7206 (0.0209)	2.85E-11.	Large
2	2	19	28	**0.4760** (0.0293)	0.4292 (0.0250)	2.87E-11	Large
	0	15	61	**0.7786** (0.0250)	0.7270 (0.0407)	3.45E-06	Large
	1	5	10	**0.6813** (0.0347)	0.5692 (0.0213)	4.28E-11	Large
3	2	20	27	**0.5411** (0.0379)	0.4842 (0.0358)	3.45E-06	Large
	0	28	46	**0.5826** (0.0228)	0.5202 (0.0407)	4.88E-08	Large
4	1	15	14	**0.8395** (0.0181)	0.7333 (0.0573)	3.31E-10	Large
	0	19	74	**0.5397** (0.0474)	0.4920 (0.0433)	2.39E-04	Large
	1	17	21	**0.5181** (0.0339)	0.4602 (0.0295)	9.44E-08	Large
5	2	4	7	**0.6934** (0.0165)	0.6127 (0.0193)	3.88E-11	Large
	0	20	68	**0.4826** (0.0357)	0.4390 (0.0292)	2.08E-06	Large
6	1	10	19	**0.4645** (0.0291)	0.3972 (0.0285)	3.06E-09	Large
	0	13	18	**0.7115** (0.0228)	0.6696 (0.0247)	1.30E-07	Large
	1	4	7	**0.6832** (0.0184)	0.6014 (0.0128)	3.51E-11	Large
7	2	24	79	**0.4734** (0.0220)	0.4260 (0.0187)	1.94E-09	Large
	0	26	74	**0.6021** (0.0249)	0.5502 (0.0237)	5.81E-10	Large
8	1	12	27	**0.5797** (0.0293)	0.5350 (0.0275)	2.39E-06	Large
	0	23	36	**0.5106** (0.0278)	0.4746 (0.0272)	2.77E-05	Large
9	1	14	13	**0.7650** (0.0302)	0.7070 (0.0328)	1.93E-08	Large
	0	4	12	**0.6730** (0.0202)	0.5968 (0.0131)	2.87E-11	Large
	1	25	87	**0.5256** (0.0265)	0.4773 (0.0196)	2.79E-09	Large
10	2	15	14	**0.7930** (0.0309)	0.7542 (0.0281)	2.51E-05	Large

Table 8 Fitness values of GE and Random Search for the method level

Fold	Alg	NE	NS	GORGEOUS	RANDOM	p-value	ES
	0	185	189	**0.6860** (0.0128)	0.5257 (0.0260)	2.66E-11	Large
	1	64	66	**0.6673** (0.0121)	0.5739 (0.0250)	2.87E-11	Large
	2	51	54	**0.6682** (0.0117)	0.4852 (0.0293)	2.86E-11	Large
1	3	7	7	0.5782 (0.0000)	**0.6226** (0.0250)	7.11E-10	Small
	0	184	188	**0.6272** (0.0179)	0.4440 (0.0216)	2.87E-11	Large
2	1	123	128	**0.6663** (0.0145)	0.4675 (0.0200)	2.85E-11	Large
	0	13	13	**0.6138** (0.0206)	0.5597 (0.0202)	1.47E-09	Large
	1	167	170	**0.6375** (0.0158)	0.4729 (0.0330)	2.86E-11	Large
	2	65	69	**0.6422** (0.0215)	0.4861 (0.0352)	2.87E-11	Large
3	3	58	60	**0.6171** (0.0124)	0.5606 (0.0180)	1.27E-10	Large
	0	65	66	**0.7335** (0.0079)	0.7239 (0.0173)	1.17E-03	Medium
4	1	54	57	**0.7107** (0.0021)	0.7029 (0.0155)	1.37E-03	Medium
	0	56	59	**0.6435** (0.0160)	0.6146 (0.0132)	1.87E-08	Large
	1	61	63	**0.6885** (0.0229)	0.6316 (0.0352)	1.66E-07	Large
	2	169	172	**0.6801** (0.0246)	0.6239 (0.0258)	6.21E-09	Large
5	3	14	14	**0.6275** (0.0087)	0.5997 (0.0149)	7.43E-09	Large
	0	62	67	**0.7222** (0.0078)	0.6698 (0.0232)	1.02E-09	Large
	1	180	182	**0.7202** (0.0095)	0.6738 (0.0247)	6.23E-09	Large
6	2	57	58	**0.7021** (0.007)	0.6611 (0.0196)	7.09E-09	Large
	0	64	67	**0.6711** (0.0163)	0.6339 (0.0283)	1.02E-06	Large
	1	65	66	**0.6484** (0.0144)	0.6100 (0.0249)	2.26E-07	Large
7	2	164	168	**0.6757** (0.0166)	0.6160 (0.0256)	1.54E-09	Large
	0	60	63	**0.5523** (0.0215)	0.4658 (0.0294)	5.23E-11	Large
	1	164	168	**0.5654** (0.0193)	0.5057 (0.0385)	5.62E-09	Large
	2	61	62	**0.7049** (0.0345)	0.5938 (0.0209)	3.88E-11	Large
8	3	14	14	**0.7002** (0.0295)	0.6339 (0.0303)	2.29E-08	Large
	0	190	194	**0.6747** (0.0138)	0.6124 (0.0193)	4.68E-11	Large
	1	47	52	**0.6669** (0.0196)	0.6204 (0.0212)	8.10E-09	Large
	2	14	14	**0.6587** (0.0112)	0.6202 (0.0153)	1.04E-09	Large
9	3	30	30	**0.6522** (0.0213)	0.5982 (0.0183)	1.01E-09	Large
	0	167	170	**0.7333** (0.0031)	0.7206 (0.0191)	3.28E-04	Large
	1	64	65	**0.7292** (0.0027)	0.7089 (0.0191)	9.40E-09	Large
	2	64	66	**0.7054** (0.0041)	0.6972 (0.0165)	0.08707	Medium
10	3	11	12	**0.7146** (0.0030)	0.7080 (0.0139)	0.0003873	Medium

by 0, 1, or 2, was generated based on a specific cluster. NE represents the number of elements and NS the number of solutions (refactoring operations) found by this algorithm. Bold values represent that, comparing the averages of GORGEOUS and RANDOM, the higher average is statistically significant considering 99% of confidence based on the *Mann–Whitney* non-parametric test (Mann and Whitney 1947). The effect size is represented in these tables as ES; their corresponding values can be found in our repository (Mariani et al. 2021).

We observe that at the class-level (Table 7) GORGEOUS is better than RANDOM with statistical difference and large effect size for all cases. We observe small standard deviation values for both approaches. At the method-level (Table 8) GORGEOUS is better than RANDOM with statistical difference in 29 cases (out of 30). In 25 of these cases, the effect size is large, in 4 cases is medium, and in only one small. The standard deviations values for RANDOM tend to be larger.

Based on that, GORGEOUS is better in 59 out of 60 cases. For the unique case where RANDOM outperforms GORGEOUS, the algorithms are generated based on the cluster with the lower number of elements (7) and refactoring operations (7). We assume GE probably converged to a local optimum in this case, where the random aspect allowed a better exploration of the search space. This difference is not found at the class-level where refactoring algorithms are generated based on a few elements. We analyzed this algorithm details and observed that the similarity value impacts its fitness function result. We assume the few elements and the few metrics considered in the learning of methods limited the results. Despite that, the effect size presented a slight difference for this case.

We calculated the execution time for both approaches, which is the time consumed for training and generating the refactoring algorithms. Detailed tables are in our repository (Mariani et al. 2021). Considering the average time for training a fold (both class and method-level), GORGEOUS took 16 minutes while RANDOM took 13 minutes. This time is acceptable since the steps are executed only once, and the generated algorithms can be adopted to other programs.

As expected, GORGEOUS has a more significant execution time than RANDOM, which can be related to the evolutionary steps of the GE algorithm. However, the difference is slight since it is of approximately 3 minutes. We identified particular cases where RANDOM presented a more significant execution time than GORGEOUS, which may be associated with the solution size since it can be different in both algorithms.

Answer to RQ1

We conclude that the use of GE is important for the generation of refactoring algorithms and produces better results than a random approach in 93% of the cases. GORGEOUS has statistically significantly better results when compared with Random search, and it does not have a significant impact on the execution time.

5.2 RQ2: Quality Improvement

To answer RQ2, we automatically 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 9 presents for GORGEOUS, for each validation program, and considering the original value MQ_{p0}, the average of MQ ($\overline{MQ}$) and the average of QI ($\overline{QI}$) 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.

Table 9 Quality and Similarity Results (Original vs Gorgeous)

Fold	Program	Original		GORGEOUS		
		MQ_{P_0}	QI_S	$\overline{MQ}$	$\overline{QI}$	$\overline{ARate}$
	Activity 5.17.0	0.1038	0.26%	0.1070	0.40%	20.00%
	CyanogenMod 11.0	0.2215	0.00%	0.2644	11.62%	10.00%
	Drools 6.3.0	0.2250	0.00%	0.2283	0.61%	**83.75%**
1	Fabric8 2.1.11	0.1865	0.02%	0.1884	0.04%	**55.95%**
	Facebook SDK 4.2.0	0.1241	0.00%	0.1263	0.21%	40.83%
	Geoserver 2 .7.2	0.1875	0.02%	0.1880	0.07%	**84.58%**
	Gradle 2.6	0.1679	0.00%	0.1686	0.16%	**81.11%**
2	Graylog 1.2.0	0.1380	0.00%	0.1439	3.25%	**70.37%**
	Languagetool	0.1643	0.02%	0.1728	1.94%	0.00%
	Mortar 0.18	0.1426	2.60%	0.2361	53.20%	**96.82%**
	Spring Boot 1.2.4	0.0016	0.00%	0.0411	1701.39%	3.33%
3	Voltdb 5.2.3	0.1457	0.10%	0.1489	0.73%	**95.76%**
	Closure 20150609	0.2690	0.00%	0.2697	0.09%	0.00%
	Drill 0.9.0	0.1327	0.00%	0.1517	8.91%	0.00%
	MPS 3.2.2	0.1067	0.00%	0.1076	0.47%	**61.67%**
4	Quasar 0.7.0	0.0227	6.47%	0.1394	276.12%	25%
	Hive 1.2.1	0.1749	0.06%	0.1778	1.15%	**91.67%**
	jOOQ 3.6.2	0.0768	0.00%	0.0990	19.29%	33.33%
	Netty 3.10.3	0.2480	0.06%	0.2489	0.03%	34.76%
5	TextSecure 2.19	0.1826	0.92%	0.1898	1.53%	**91.21%**
	Bitcoinj 0.12.3	0.2521	0.07%	0.2657	0.64%	**62.67%**
	Neo4j 2.3.0	0.1626	0.04%	0.1652	0.45%	**76.67%**
	Presto 0.107	0.2852	0.03%	0.2863	0.25%	**63.00%**
6	Tomahawk A. 0.83	0.1918	0.00%	0.1974	2.00%	42.86%
	Cassandra 2.2.0	0.1988	0.23%	0.2057	0.46%	**50.00%**
	Java Driver 2.1.6	0.3161	0.00%	0.3402	1.75%	**68.10%**
	Spring F. 4.2.0	0.1984	0.00%	0.1991	0.06%	**97.78%**
7	Tachyon 0.6.4	0.1445	36.67%	0.2772	50.32%	0.00%
	Hazelcast 3.5.1	0.1319	0.14%	0.1378	2.95%	**100%**
	Rest Li 2.6.2	0.1389	0.04%	0.1452	3.96%	**78.89%**
	Vert X 3.0.0	0.1411	0.06%	0.1487	1.00%	**61.67%**
8	WordPress A. 4.0	0.0332	61.39%	0.1557	227.81%	**92.50%**
	Android IMSI 0.1.29	0.1492	0.00%	0.1971	16.05%	**99.72%**
	Checkstyle 6.7	0.2527	0.86%	0.2819	9.89%	33.33%
	Graphhopper 0.7.0	0.3103	0.19%	0.3124	0.19%	32.67%
9	Jersey 2.19	0.2092	0.39%	0.2105	0.20%	48.41%
	Crate 0.49.2	0.1714	0.08%	0.1753	0.37%	17.38%
	Deeplearning4j 0.4	0.1505	0.00%	0.1607	4.27%	0.00%
	Infinispan 5.2.13	0.1537	0.00%	0.1546	0.23%	**75.33%**
10	Openhab 1.7.0	0.0040	116.76%	0.0149	148.39%	**73.94%**

We observed many variations in the quality improvement results based on the presented results since the programs have distinct characteristics and domains. For instance, $\overline{QI}$ varies from 0.03% up to 1701.39%. We noted that the amount of improvement is highly dependent on the original MQ value since the algorithms were able to obtain greater values of $\overline{QI}$ for programs with the original MQ_{p0} lower than the average (0.1654).

In fact, 82% of the programs with high $\overline{QI}$, 60% of the programs with medium $\overline{QI}$, and 31% of the programs with low $\overline{QI}$ have the original MQ value lower than the average. In this sense, the lower the MQ value, the more significant the improvement and the greater the benefit of the refactoring operations for the program. In particular, the programs with the lowest values of MQ are the ones with the greatest $\overline{QI}$, which are: SpringBoot1.2.4 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%. Furthermore, such programs belong to distinct folds, which shows consistency in the results obtained by different folds.

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

The following analysis considers the quality improvement of refactoring operations individually. The goal is to investigate if the quality improvement occurs due to the whole set of refactoring operations, which is not related to only one operation. Table 10 presents, for each program, the number of refactoring solutions (NS) found by the algorithms and; the number and % of good, bad, and neutral solutions. We considered a good solution the one with an improvement ($\overline{QI} > 0$). On the other hand, neutral solutions are the ones with no improvement ($\overline{QI} = 0$), and bad solutions are the ones with a negative improvement ($\overline{QI} < 0$). The greatest percentage from these three groups is highlighted in bold.

These results clearly show that most of the found refactoring solutions improve the program quality. Moreover, there is no case where bad solutions are 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, corresponding to 5% of times. Besides, half of the folds present at least one program where 100% of solutions are good.

We also compared results of GORGEOUS with results obtained by refactoring operations performed by developers. In this sense, column QI_S of Table 9 shows the QI value obtained by the developers' applications. These values show that the refactoring operations applied by developers do not obtain good values of QI. While all programs presented some 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 great 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. Also, GORGEOUS was capable of generating refactoring algorithms that provide solutions able to improve in some level 100% of programs, while the developer applications resulted in improvement of only 60% of programs, being 40% with no improvement.

Table 10 Individual Solutions Results

Fold	Program	NS	% Good	% Bad	% Neutral
	Activity 5.17.0	10	**40%**	20%	**40%**
	CyanogenMod 11.0	9	22%	11%	**67%**
	Drools 6.3.0	11	**64%**	36%	0%
1	Fabric8 2.1.11	9	**67%**	33%	0.04%
	Facebook SDK 4.2.0	3	**100%**	0%	0%
	Geoserver 2 .7.2	4	**50%**	25%	25%
	Gradle 2.6	4	**100%**	0%	0%
2	Graylog 1.2.0	3	**100%**	0%	0%
	Languagetool	12	**100%**	0%	0%
	Mortar 0.18	10	**60%**	0%	40%
	Spring Boot 1.2.4	11	45%	0%	**55%**
3	Voltdb 5.2.3	15	40%	7%	**53%**
	Closure 20150609	3	**67%**	0%	33%
	Drill 0.9.0	3	**67%**	0%	33%
	MPS 3.2.2	3	**67%**	0%	33%
4	Quasar 0.7.0	3	**67%**	0%	33%
	Hive 1.2.1	13	**54%**	38%	8%
	jOOQ 3.6.2	11	**73%**	18%	9%
	Netty 3.10.3	12	33%	17%	**50%**
5	TextSecure 2.19	10	**90%**	10%	0%
	Bitcoinj 0.12.3	8	**88%**	0%	13%
	Neo4j 2.3.0	10	**50%**	10%	40%
	Presto 0.107	7	**57%**	0%	43%
6	Tomahawk A. 0.83	7	**100%**	0%	0%
	Cassandra 2.2.0	9	**78%**	11%	11%
	Java Driver 2.1.6	8	38%	13%	**50%**
	Spring F. 4.2.0	13	**54%**	38%	8%
7	Tachyon 0.6.4	7	**86%**	0%	14%
	Hazelcast 3.5.1	14	**64%**	21%	14%
	Rest Li 2.6.2	11	**82%**	0%	18%
	Vert X 3.0.0	9	**100%**	0%	0%
8	WordPress A. 4.0	8	**50%**	0%	**50%**
	Android IMSI 0.1.29	9	**56%**	11%	33%
	Checkstyle 6.7	9	**78%**	11%	11%
	Graphhopper 0.7.0	10	40%	10%	**50%**
9	Jersey 2.19	11	**45%**	27%	27%
	Crate 0.49.2	5	**100%**	0%	0%
	Deeplearning4j 0.4	5	**100%**	0%	0%
	Infinispan 5.2.13	5	**100%**	0%	0%
10	Openhab 1.7.0	5	**100%**	0%	0%

Answer to RQ2

We can conclude that the generated refactoring algorithms could find solutions capable of improving the quality of programs in terms of the metrics used. In some programs, the improvement was not great, but still, at least some improvement occurs in all programs. Furthermore, we have observed that programs with very low modularity can benefit more from the refactoring operations. The quality improvement obtained by GORGEOUS is greater than the ones obtained by refactorings applied by developers.

5.3 RQ3: Similarity with Refactorings Applied by Developers

To answer RQ3, we analyzed if the refactoring algorithms generated by GORGEOUS can find the refactoring operations applied by developers. As mentioned before, this analysis was performed using the previously defined measure $ARate$. In this respect, Table 9 presents $\overline{ARate}$ formatted as percentages. The values greater than 50% are highlighted in bold. We considered the set of refactoring algorithms generated based on a fold to obtain this value. The average is calculated based on 30 runs in this process.

The results of $\overline{ARate}$ presented in Table 9 show several variations. The variations of the results are explained by the different characteristics of the considered programs. It would be challenging to achieve similar metrics results when dealing with the software changes. Briefly, the overall average is 55.58%, which is a good value if we consider balancing both quality and similarity. However, it is important to analyze the results in deep.

Table 9 shows 60% of the programs have $\overline{ARate}$ 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 obtained a value of 100%. On the other hand, the value is 0% for 5 programs. By analyzing these programs, we found some similarities in their refactoring operations. First, most of them have several refactoring solutions involving the same element as the actor. Then, the set of elements used to measure $ARate$ is small. Moreover, 93% of the refactoring operations not found by the refactoring algorithms are composed of method-level refactorings.

In fact, low values of $ARate$ do not mean bad results since some good refactoring solutions might not be 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*[4] was designed taking into account the design pattern called Façade (Gamma et al. 1995). In this sense, such a class is responsible for 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 Façade design pattern, an additional Façade can be created to divide responsibilities and prevent the original Façade from growing and becoming complex. In

[4]http://activemq.apache.org/maven/5.9.0/apidocs

this sense, the suggested solution makes sense since we could extract *BrokerFacadeSupport* into two different Façade for dealing with connectors.

Answer to RQ2

We can conclude that the generated refactoring algorithms could find solutions capable of improving the quality of programs in terms of the metrics used. In some programs, the improvement was not great, but still, at least some improvement occurs in all programs. Furthermore, we have observed that programs with very low modularity can benefit more from the refactoring operations. The quality improvement obtained by GORGEOUS is greater than the ones obtained by refactorings applied by developers.

5.4 RQ4: Use of Patterns Extracted with Clusters Algorithms

To answer RQ4, we analyze values of $\overline{QI}$ and $\overline{ARate}$ obtained by GORGEOUS and NOCLUSTER experiments. Table 11 summarizes these results. In general, GORGEOUS performs better than NOCLUSTER considering both quality and similarity results. Regarding $\overline{QI}$, NOCLUSTER has an average of 26.26% of improvement, while GORGEOUS has 63.81%.

Especially, considering $\overline{QI}$, GORGEOUS presents the best values for all programs. Moreover, concerning NOCLUSTER, most programs have presented a very low improvement. On the other hand, there are five cases where NOCLUSTER performs better when analyzing $\overline{ARate}$ values. The values in which NOCLUSTER is better are highlighted in bold in Table 11. A unique group of elements can lead to a more generic refactoring algorithm, which results in more embracing rules. In this sense, depending on the characteristics of the program, it can be more easily to find solutions, and this will eventually increase $\overline{ARate}$. However, these solutions will not always improve quality, as shown by $\overline{QI}$ values.

Results show, on average, and based on the 40 programs, GORGEOUS obtains 55% against 9% of $\overline{ARate}$ for NOCLUSTER. Furthermore, GORGEOUS obtained a maximum of 100% of $\overline{ARate}$, while NOCLUSTER could get a maximum of only 27%. The approach without the learning of patterns can negatively impact the similarity measure with real refactoring operations and quality aspects. This is evidence that the clusters can be useful to guide the generation of algorithms able to generalize the learned refactoring patterns.

Answer to RQ4

The use of patterns extracted by clustering algorithms is beneficial for GORGEOUS, which has, in most cases, its performance improved regarding the similarity measure, as well as the quality of solutions.

5.5 Discussion

In this section, we discuss the implications of the results for research and practice and some limitations of the approach.

Table 11 Quality and Similarity Results (GORGEOUS vs NOCLUSTER)

Fold	Program	GORGEOUS			NOCLUSTER		
		$\overline{MQ}$	$\overline{QI}$	$\overline{ARate}$	$\overline{MQ}$	$\overline{QI}$	$\overline{ARate}$
	Activity 5.17.0	0.1070	0.40%	20.00%	0.1040	0.15%	4.00%
	CyanogenMod 11.0	0.2644	11.62%	10.00%	0.2348	6.00%	**18.61%**
	Drools 6.3.0	0.2283	0.61%	**83.75%**	0.2270	0.08%	16.00%
1	Fabric8 2.1.11	0.1884	0.04%	**55.95%**	0.1865	0.00%	0.10%
	Facebook SDK 4.2.0	0.1263	0.21%	40.83%	0.1250	0.00%	16.67%
	Geoserver 2 .7.2	0.1880	0.07%	**84.58%**	0.1876	0.05%	0.00%
	Gradle 2.6	0.1686	0.16%	**81.11%**	0.1681	0.10%	3.57%
2	Graylog 1.2.0	0.1439	3.25%	**70.37%**	0.1385	0.35%	2.38%
	Languagetool	0.1728	1.94%	0.00%	0.1645	0.12%	**10%**
	Mortar 0.18	0.2361	53.20%	**96.82%**	0.1500	5.19%	0%
	Spring Boot 1.2.4	0.0411	1701.39%	3.33%	0.0141	794.44%	32.50%
3	Voltdb 5.2.3	0.1489	0.73%	**95.76%**	0.1464	0.44%	7.86%
	Closure 20150609	0.2697	0.09%	0.00%	0.2690	0.01%	**5.83%**
	Drill 0.9.0	0.1517	8.91%	0.00%	0.1344	1.22%	**7.50%**
	MPS 3.2.2	0.1076	0.47%	**61.67%**	0.1067	0.09%	6.67%
4	Quasar 0.7.0	0.1394	276.12%	25%	0.0469	106.55%	3.33%
	Hive 1.2.1	0.1778	1.15%	**91.67%**	0.1754	0.28%	11.85%
	jOOQ 3.6.2	0.0990	19.29%	33.33%	0.0800	4.24%	20.00%
	Netty 3.10.3	0.2489	0.03%	34.76%	0.2482	0.00%	10.00%
5	TextSecure 2.19	0.1898	1.53%	**91.21%**	0.1837	0.59%	8.67%
	Bitcoinj 0.12.3	0.2657	0.64%	**62.67%**	0.2528	0.25%	15.71%
	Neo4j 2.3.0	0.1652	0.45%	**76.67%**	0.1627	0.09%	13.33%
	Presto 0.107	0.2863	0.25%	**63.00%**	0.2853	0.03%	10.67%
6	Tomahawk A. 0.83	0.1974	2.00%	42.86%	0.1934	0.81%	0%
	Cassandra 2.2.0	0.2057	0.46%	**50.00%**	0.1990	0.07%	13.64%
	Java Driver 2.1.6	0.3402	1.75%	**68.10%**	0.3168	0.23%	50.00%
	Spring F. 4.2.0	0.1991	0.06%	**97.78%**	0.1885	0.04%	15.00%
7	Tachyon 0.6.4	0.2772	50.32%	0.00%	0.1773	22.70%	**8.00%**
	Hazelcast 3.5.1	0.1378	2.95%	**100%**	0.1339	1.52%	21.82%
	Rest Li 2.6.2	0.1452	3.96%	**78.89%**	0.1409	1.43%	15.67%
	Vert X 3.0.0	0.1487	1.00%	**61.67%**	0.1436	1.75%	23.33%
8	WordPress A. 4.0	0.1557	227.81%	**92.50%**	0.0650	95.50%	57.66%
	Android IMSI 0.1.29	0.1971	16.05%	**99.72%**	0.1545	3.60%	0.00%
	Checkstyle 6.7	0.2819	9.89%	33.33%	0.2532	0.20%	8.33%
	Graphhopper 0.7.0	0.3124	0.19%	32.67%	0.3107	0.13%	0.00%
9	Jersey 2.19	0.2105	0.20%	48.41%	0.2092	0.02%	27.33%
	Crate 0.49.2	0.1753	0.37%	17.38%	0.1716	0.10%	1.43%
	Deeplearning4j 0.4	0.1607	4.27%	0.00%	0.1523	1.19%	2.78%
	Infinispan 5.2.13	0.1546	0.23%	**75.33%**	0.1537	0.03%	3.03%
10	Openhab 1.7.0	0.0149	148.39%	**73.94%**	0.1523	1.19%	3.03%

The results show Gorgeous is capable of generating refactoring algorithms that can find solutions similar to actual refactoring operations, as well as improving quality attributes. The generated refactoring algorithms suggest solutions that reach on average 55% of improvement in the quality of the programs. Furthermore, the refactoring algorithms could find on average 50% of actual refactoring operations. These are considered good results, especially by considering the balance between similarity and quality. Maybe the weights of both factors can be adjusted depending on the developers' interests and preferences. We also intend to conduct experiments to evaluate the dependence relation between both factors. Other metrics and evaluation functions should be explored in future research, as well as a multi-objective approach.

Despite these results presented by averages, the values obtained for both similarity and quality vary. While some programs obtained 100% of quality improvement, others presented 0%. However, 80% of the refactoring solutions showed improvement to the program. It is expected those variations when working with software programs. As presented in the program details, they differ significantly in size and domain. Moreover, much information that is used may present variation, such as the number of refactorings, number of elements in a cluster, number of refactoring operations, etc. Despite that, we could not find that these characteristics directly influenced quality improvement. On the other hand, the results seem to be influenced by modularity values, where we can assume that programs with lower modularity values can take more advantage of the approach. In this sense, we believe a future study considering other program characteristics might be feasible to find more answers.

We conducted some tests to evaluate the impact on the results regarding the clustering phase. We evaluated two other algorithms: K-means (Hartigan and Wong 1979) and Agglomerative Hierarchical Clustering (Lance and Williams 1967) by using the indicators Silhouete, Calinski-Harabasz, Distortion, and Davies-Bouldin. As a result, no algorithm could be considered the best according to all indicators, and we did not observe significant changes in the Gorgeous performance. However, we have observed different findings regarding the number of clusters at the class and method levels in any case. At the method-level, having fewer methods causes Gorgeous to generate refactoring algorithms with a low similarity score with actual refactoring operations. On the other hand, at the class-level, having fewer classes causes Gorgeous to generate refactoring algorithms with a better similarity score. A possible reason for this is the number of rules to be satisfied. Only two rules must be satisfied at the method-level, and four rules at the class-level. Using a reduced number of clusters means that each cluster will have a significant number of classes. Since the GE is executed per cluster, this implies many elements that need to satisfy the rules and, consequently, a more restricted space, which makes the search for good solutions more difficult.

We observed that 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. Methods are way more in numbers than classes, so we could not make a fair comparison. In this respect, the clustering of elements brings many advantages in generating refactoring algorithms.

This paper evaluated a few refactorings, only the most common and used. However, our approach is flexible, and other refactorings can also be added in the grammars and considered by the clustering algorithms. Besides the quality and similarity results, it is important to highlight that the refactoring algorithms could also be manipulated to improve the results. For example, we can suggest a refactoring operation only if it improves quality or 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 on at the moment. In this sense, the refactoring algorithms could be manipulated regarding how the solutions are suggested. We intend to evaluate such strategies in future works.

A limitation of GORGEOUS in comparison with SBR approaches is to require information from previous projects for training. This is a usual problem of using ML approaches. Then an initial step to create the database of program instances is necessary. This database can be generated by mining existing software repository tools and can be updated if desired. However, the approach assumes that the training set is representative of all projects to be refactored. Also, there is an effort to extract metrics values from the programs. We used the tools *RefactoringMiner* and *Understand* to this goal and automated the generation of the inputs and all steps of the approach. Nevertheless, other tools could also be used.

Another assumption is that the metrics and the intervals that such metrics vary used in the grammar are sufficient to characterize all classes/methods that underwent similar refactoring operations. This may not hold in practice and constrain the search space. This limitation of the approach may impact its applicability. To deal with this, there are other metrics, as well as some particularities of the systems, which could be used in the training phase. Future work should evaluate a broader set of metrics and corresponding intervals in different domains. The results should be used to make the approach configurable according to some specific domain.

5.6 Threats to Results Validity

In this section, we discuss the main threats to the validity of our results and how we mitigate them. We use the taxonomy of Wohlin et al. (2012).

Threats to *Construct Validity* concern the relationship between theory and observation. A possible threat is the coupling with the program representation in computing the fitness function. Indeed, the quality function is computed by simulating the refactoring operations based on such a representation. In this sense, we can not guarantee that the same results will be obtained by using the original program or other representation. In this respect, we built the program representation to encompass all information needed to compute the metric value.

Internal Validity evaluates the relationship between the treatment and the output and concerns factors that might have influenced our results. One of the main threats of our study is the program folds used 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.

To run GORGEOUS, a preprocessing to collect the data needs to be performed. It involves the collection of elements information and refactoring operations. In this sense, the correctness of such information is related to the tool used in this step. To mitigate this threat, we make sure to use a highly recommended tool to extract the elements information and 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.

The application of the generated refactoring algorithms is also a possible threat. 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 find more solutions or even all solutions consequently. It would impact the results of the values of the measures. We did not perform experiments changing this configuration, but we expect to reach better results by exploring this aspect. Our results encourage more rigorous analyses over the capability of refactoring algorithms.

A possible threat is the choice of the metrics used to characterize the elements (classes and methods). For this initial evaluation, we chose standard metrics related to cohesion, coupling, and size, usually adopted by search-based refactoring approaches. Other metrics could be considered related to process, developers, or class evolution (Catolino et al. 2020). There are many automated tools to collect distinct metrics. As mentioned in the last section, a deeper study should be conducted in future work. But the set of metrics used in this paper does not invalidate the obtained results. We think that using a complete set of metrics can make the results even better.

The intervals for these metrics are also a threat. To minimize this problem, as we mention in Section 3.4.1, the intervals used in the algorithm are generated respecting the minimum and maximum values of the metrics m_1 to m_8, found in the elements of the corresponding cluster. In this way, we consider the real values of these metrics found in the training systems.

Finally, we used parameters from the literature to execute the EM and GE algorithms, but a tuning phase may improve the results. Also, the GE technique is non-deterministic, but to mitigate this threat, we performed 30 runs of each technique as recommended and adopted in the SBSE literature (Colanzi et al. 2019).

Conclusion Validity is related to the ability to draw the correct conclusion from the study. Threats in this category are the indicators used to evaluate our results (QI and $ARate$) and statistical tests used. Other indicators could lead to different results. We used tests commonly adopted in software engineering problems for this kind of algorithm to minimize this threat (Colanzi et al. 2019).

External Validity corresponds to the ability to generalize the results beyond the experimental setting. Although we have a significant number of programs, some of them have few refactorings, and only Java programs were used. Also, we use a few metrics to generate method-level operations. Moreover, although the refactoring solutions improved most programs, quality improvement is highly dependent on the program characteristics. Then, it is not possible to guarantee the same amount of improvement for two different programs.

6 Concluding Remarks

In this work, we present GORGEOUS, an SBR learning approach to generate refactoring algorithms. Refactoring algorithms are generated by GE, considering quality improvement of a program and similarity with actual refactoring operations. We introduce the idea and structure of a refactoring algorithm and two grammars to guide the search. In addition to this, each algorithm is generated based on a refactoring pattern discovered in a previous step. In such a step, a clustering algorithm is executed to group code elements from different programs refactored in similar ways in the past. The clusters are generated considering refactoring type and frequency of application. Each group of code elements represents a refactoring pattern.

In this way, GORGEOUS combines the advantages of works from both SBR and ML approaches. When executed for a given program, the generated algorithms work as a prediction model of refactoring opportunities and are generated by searching in a vast space of alternatives to optimize quality attributes and similarity with refactoring patterns applied in different programs.

We reported results of an empirical evaluation conducted using a 10-fold cross-validation with 40 Java programs from *GitHub*. We collected several data from programs, such as elements metrics and refactoring operations. Results show GORGEOUS can 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 actual refactoring operations. 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 step. It contributes to generating algorithms capable of suggesting good solutions regarding similarity with real applications. The average of quality improvement obtained by GORGEOUS is greater than the ones obtained by refactorings applied by developers. This shows that besides the similarity with past refactoring patterns, the generated algorithms found other possibilities that they might not have thought of.

As future work, we intend to improve some aspects of GORGEOUS, as well as to perform other experiments. We are currently working on automating the preprocessing step to make GORGEOUS less dependent on external tools. Other future work is to create an IDE plugin to provide refactoring algorithms during development and allow the automatic application of refactorings. In addition, we intend to study other aspects of a refactoring algorithm, such as qualitatively analyzing patterns of each cluster and how they have influenced the rules of the algorithm. Finally, we are working to include other quality metrics and refactorings in the generation of the algorithms.

Acknowledgements The authors would like to thank to CAPES by supporting Thainá Mariani by the program PDSE associated with the process 88881.135198/2016-01. Silvia R. Vergilio is supported by CNPq (Grant:305968/2018).

Declarations

Conflict of Interests We declare that we have no conflict of interests.

References

AutoRefactor (2021) Available at: http://autorefactor.org/. Accessed on March 28

Spartan Refactoring (2021) Available at: https://marketplace.eclipse.org/content/spartan-refactoring. Accessed on March 28

Abid C, Alizadeh V, Kessentini M, Ferreira TN, Dig D (2020) 30 years of software refactoring research: A systematic literature review. CoRR abs/2007.02194

Al Dallal J (2012) Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. Inf Softw Technol 54(10):1125–1141

Alenezi M, Akour M, Alqasem O (2020) Harnessing deep learning algorithms to predict software refactoring. TELKOMNIKA (Telecommunication Computing Electronics and Control) 18:2977–2982. https://doi.org/10.12928/TELKOMNIKA.v18i6.16743

Alizadeh V, Kessentini M, Mkaouer MW, Ocinneide M, Ouni A, Cai Y (2020) An interactive and dynamic search-based approach to software refactoring recommendations. IEEE Trans Softw Eng 46(9):932–961. https://doi.org/10.1109/TSE.2018.2872711

AlOmar EA, Peruma A, Newman CD, Mkaouer MW, Ouni A (2020) On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSEW’20. Association for Computing Machinery, New York, NY, USA, pp 342–349

Amal B, Kessentini M, Bechikh S, Dea J, Said LB (2014) On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In: Le Goues C, Yoo S (eds) Search-Based Software Engineering. Springer International Publishing, Cham, pp 31–45

Aniche M, Maziero E, Durelli R, Durelli V (2020) The effectiveness of supervised machine learning algorithms in predicting software refactoring. IEEE Trans Softw Eng, pp 1–1. https://doi.org/10.1109/TSE.2020.3021736

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28(1):4–17

Baqais A, Alshayeb M (2020) Automatic software refactoring: a systematic literature review. Softw Qual J 28:459–502

Barros RC, Basgalupp MP, Cerri R, da Silva TS, de Carvalho ACPLF (2013) A Grammatical Evolution Approach for Software Effort Estimation. In: Proceedings of the 5th Genetic and Evolutionary Computation Conference. GECCO

Catolino G, Palomba F, Fontana FA, De Lucia A, Andy Z, Ferrucci F (2020) Improving change prediction models with code smell-related information. Empirical Software Engineer 25:49–95. https://doi.org/10.1007/s10664-019-09739-0

Cohen J (2013) Statistical power analysis for the behavioral sciences. Academic press

Colanzi TE, Assunção WKG, Farah PR, Vergilio SR, Guizzo G (2019) A review of ten years of the symposium on search-based software engineering. In: Nejati S, Gay G (eds) Symposium on Search-Based Software Engineering. Springer, Cham, pp 42–57

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, edn. 3. The MIT Press

Dallal JA (2017) Predicting move method refactoring opportunities in object-oriented code. Inf Softw Technol 92:105–120

Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the em algorithm. J R Stat Soc Ser B 39(1):1–38

Durillo JJ, Nebro AJ (2011) jMetal: A Java framework for multi-objective optimization. Adv Eng Softw 42:760–771

Fowler M, Beck K (2018) Refactoring: Improving the Design of Existing Code, edn. 2. Addison-Wesley

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: Elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Hartigan JA, Wong MA (1979) Algorithm as 136: A k-means clustering algorithm. J R Stat Soc 28(1):100–108

Imazato A, Higo Y, Hotta K, Kusumoto S (2017) Finding extract method refactoring opportunities by analyzing development history. In: Proceedings of the 41st Annual Computer Software and Applications Conference. COMPSAC

Jindal S, Khurana G (2013) The statistical analysis of source-code to determine the refactoring opportunities factor (ROF) using a machine learning algorithm. In: Proceedings of the International Conference on Advances in Recent Technologies in Communication and Computing. ARTCom

Kaur A, Dhiman G (2019) A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In: Yadav N, Yadav A, Bansal JC, Deep K, Kim JH (eds) Harmony Search and Nature Inspired Optimization Algorithms. Springer Singapore, Singapore, pp 909–921

Kessentini M, Mahouachi R, Ghedira K (2012) What you like in design use to correct bad-smells. Softw Qual J 21(4):551–571

Kim M, Zimmermann T, Nagappan N (2014) An empirical study of refactoring challenges and benefits at Microsoft. IEEE Trans Softw Eng 40(7):633–649

Koc E, Ersoy N, Andac A, Camlidere ZS, Cereci I, Kilic H (2011) 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, pp 59–66

Koc E, Ersoy N, Camlidere ZS, Kilic H (2012) A Web-Service for Automated Software Refactoring Using Artificial Bee Colony Optimization. In: Proceedings of the International Conference on Advances in Swarm Intelligence. ICSI, pp 318–325

Kosker Y, Turhan B, Bener A (2009) An expert system for determining candidate software classes for refactoring. Expert Syst Appl 36(6):10000–10003

Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press

Kumar L, Satapathy SM, Murthy LB (2019) 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

Lance GN, Williams WT (1967) A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. The Computer Journal 9(4):373–380
Mahouachi R, Kessentini M, Ghedira K (2012) A new design defects classification: Marrying detection and correction. In: Proceedings of the Fundamental Approaches to Software Engineering. FASE
Mann HB, Whitney DR (1947) 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
Mansoor U, Kessentini M, Wimmer M, Deb K (2015) Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Softw Qual J, pp 1–29
Mariani T, Guizzo G, Vergilio SR, Pozo ATR (2016) Grammatical evolution for the multi-objective integration and test order problem. In: Genetic and Evolutionary Computation Conference. GECCO, pp 1069–1076
Mariani T, Kessentini M, Vergilio SR (2021) Dataset and Suplementary Material. https://doi.org/10.6084/m9.figshare.12275981
Mariani T, Vergilio SR (2016) A systematic review on search-based refactoring. Inf Softw Technol 83:14–34
Mkaouer MW, Kessentini M, Bechikh S, Cinnéide MO, Deb K (2015) On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir Softw Eng, pp 1–43
Mkaouer MW, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M (2014) Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, pp 331–336
Mkaouer MW, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M (2014) Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of the International Conference on Automated Software Engineering. ASE, pp 331–336
Mkaouer W, Kessentini M, Kontchou P, Deb K, Bechikh S, Ouni A (2015) Many-Objective Software Remodularization Using NSGA-III. Transactions on Software Engineering and Methodology 24(3):17:1–17:45
Mohan M, Greer D (2018) A survey of search-based refactoring for software maintenance. Journal of Software Engineering Research and Development 6:3:1 – 3:52
Moore I (1996) Automatic inheritance hierarchy restructuring and method refactoring. In: Proceedings of the 11th Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA
Murphy-Hill E, Parnin C, Black AP (2012) How we refactor, and how we know it. IEEE Trans Softw Eng 38(1):5–18
Ouni A, Kessentini M, Sahraoui H (2013) Search-based refactoring using recorded code changes. In: Proceedings of the European Conference on Software Maintenance and Reengineering. CSMR
Ouni A, Kessentini M, Sahraoui H (2014) Multiobjective optimization for software refactoring and evolution. Adv Comput 94:103–167
Ouni A, Kessentini M, Sahraoui H, Hamdi MS (2013) The use of development history in software refactoring using a multi-objective evolutionary algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO
Ouni A, Kessentini M, Sahraoui H, Inoue K, Deb K (2016) Multi-criteria code refactoring using search-based software engineering: An industrial case study. ACM Trans Softw Eng Methodol 25(3):23:1–23:53
Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS (2015) Improving multi-objective code-smells correction using development history. J Syst Softw 105:18–39
Paixao M, Harman M, Zhang Y, Yu Y (2018) An empirical study of cohesion and coupling: Balancing optimization and disruption. IEEE Trans Evol Comput 22(3):394–414
Phongpaibul M, Boehm B (2007) Mining software evolution to predict refactoring. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement. ESEM
Powers DMW (2011) Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. J Mach Learn Technol 2:37–63
Ryan C, Collins JJ, Neill MO (1998) Grammatical evolution: Evolving programs for an arbitrary language. In: Genetic Programming. Lecture Notes in Computer Science, vol 1391. Springer, Berlin Heidelberg, pp 83–96
Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of github contributors. In: Proceedings of the 24th International Symposium on Foundations of Software Engineering. FSE, pp 858–870
Sjøberg DIK, Yamashita A, Anda BCD, Mockus A, Dybå T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156
Tan P-N, Steinbach M, Kumar V (2005) Introduction to data mining. Addison-Wesley

Tsantalis N, Chaikalis T, Chatzigeorgiou A (2008) JDeodorant: Identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering. CSMR

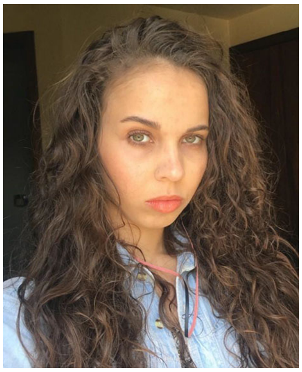
Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D (2019) On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering. ICSE ’19, pp 25–36

Wang H, Kessentini M, Grosky W, Meddeb H (2015) On the use of time series and search based software engineering for refactoring recommendation. In: Proceedings of the 7th International Conference on Management of Computational and Collective IntElligence in Digital EcoSystems. MEDES ’15. Association for Computing Machinery, New York, NY, USA, pp 35–42. https://doi.org/10.1145/2857218.2857224

Witten IH, Frank E (1999) Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media

Xu S, Sivaraman A, Khoo S-C, Xu J (2017) GEMS: An extract method refactoring recommender. In: Proceedings of the 28th International Symposium on Software Reliability Engineering. ISSRE



Thainá Mariani received a Doctoral degree at the Postgraduate Program in Informatics (PPGInf) of Federal University of Paraná (UFPR), Brazil. She also holds the MSc degree from PPGInf of UFPR. Her main research interests are: search-based software engineering, software refactoring, machine learning and evolutionary algorithms.

Marouane Kessentini received the Ph.D. degree from the University of Montreal, Canada, in 2012. He is currently a Tenured Associate Professor and leading a research group on software engineering intelligence. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning, and evolutionary algorithms to address software engineering and services computing problems. He was a recipient of the prestigious 2018 President of Tunisia distinguished Research Award, the University Distinguished Teaching Award, the University Distinguished Digital Education Award, the College of Engineering and Computer Science Distinguished Research Award, four best paper awards. His AI-based software refactoring invention, licensed, and deployed by industrial partners, is selected as one of the top eight inventions at the University of Michigan, in 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office.

Silvia Regina Vergilio received Master and Doctoral degress from University of Campinas (UNICAMP), Brazil. She is currently a professor of Software Engineering in the Computer Science Department of Federal University of Paraná (UFPR), Brasil, where she leads the Research Group on Software Engineering. She has involved in several projects and her research is mainly supported by CNPq (PQ Level 2). Her research interests include software testing, software reliability, Software Product Lines (SPLs) and Search-based Software Engineering (SBSE). She serves as assistant editor of the Journal of Software Engineering: Research and Development, and acts as peer reviewer for diverse international journals. She serves on the Program Committe of the main Brazilian Software Engineering conferences and other international ones, mainly related to Search-Based Software Engineering and software testing. Her publications list includes many papers devoted to the SBSE field, where she is very knonw. She has contributed to consolidate such a field in Brazil.