

Better Similarity Coefficients to Identify Refactoring Opportunities

Arthur F. Pinto · Ricardo Terra

Received: date / Accepted: date

Abstract Similarity coefficients are used by several techniques to identify refactoring opportunities. As an example, it is expected that a method is located in a class that is structurally similar to it. However, the existing coefficients in Literature have not been designed for the structural analysis of software systems, which may not guarantee satisfactory accuracy. This article, therefore, proposes new coefficients—based on genetic algorithms over a training set of 10 systems—to improve the accuracy of the identification of Move Class, Move Method, and Extract Method refactoring opportunities. We conducted an empirical study comparing these proposed coefficients with other 18 coefficients in other 101 systems. The results indicate, in relation to the best analyzed coefficient, a statistical improvement from 5.23% to 6.81% for the identification of Move Method refactoring opportunities, 12.33% to 14.79% for Move Class, and 0.25% to 0.40% for Extract Method. Moreover, we implemented a tool that relies on the proposed coefficients to recommend refactoring opportunities.

Keywords Software Architecture · Structural Similarity · Code Refactoring · Move Class · Move Method · Extract Method

1 Introduction

During software development, the software architecture is subjected to several problems. Code Smells (also called Bad Smells) are defined as any symptoms

Arthur F. Pinto
Computer Science Department, Universidade Federal de Lavras (UFLA), Brasil
E-mail: fparthur@posgrad.ufla.br

Ricardo Terra
Computer Science Department, Universidade Federal de Lavras (UFLA), Brasil
E-mail: terra@dcc.ufla.br

in the code that might indicate one of these problems (Fowler et al 1999). Frequently, such Code Smells imply in the establishment of unnecessary dependencies. However, considering that ensuring architectural design is extremely important for maintainability, reusability, scalability, and portability of software systems (Passos et al 2010), it is expected that the code elements of a software project be located in structurally similar entities.

In order to ensure the structural similarity among code entities and to avoid the occurrence of Code Smells, it must be verified the similarity of dependencies among its elements (whether in the level of package, class, method or block), besides performing refactorings when necessary. This implies moving methods and classes, and in the extraction of a code block from a method (generating a new method), through refactorings like *Move Class*, *Move Method* and *Extract Method* (Fowler et al 1999).

Figure 1 shows an example of a system that follows the MVC architecture, consisting of three layers: *Model*, *View* and *Controller*. It is possible to note that C_2 is badly located in the *Controller* layer since it depends on graphical elements while other classes of the layer have dependencies to manipulation elements of request and answers (e.g., `HttpRequest` e `HttpResponse`). Therefore, it is possible to suggest moving such class (*Move Class*) to a structurally similar layer that, in this scenario, would be the *View* layer. Such structural similarity occurs because C_2 and classes from the *View* layer (V_1, \dots, V_n) have dependencies on common graphical elements (e.g., `JPanel` and `JLabel`). Thus, refactoring would not only guarantee an architecture with properly located entities, but would also respect the MVC architecture.

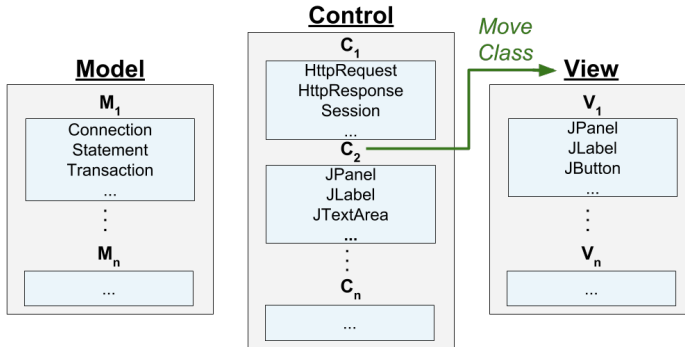


Fig. 1: Move Class Refactoring Example

Several coefficients were proposed for the calculation of similarity among code entities. However, the use of such coefficients may not guarantee satisfactory accuracy. Moreover, the main current coefficients in the literature were not designed for the structural analysis of a software system. For instance, the *Jaccard* coefficient, one of the most used in Software Engineering, was initially designed to compare the similarity among flower species in different districts (Jaccard 1912).

Therefore, this article aims primarily to propose new similarity coefficients for a more precise identification of *Move Class*, *Move Method* and, *Extract Method* refactoring opportunities. This makes it possible to (i) locate more accurately entities improperly positioned on a system architecture and (ii) leverage the accuracy of tools for identification of refactoring opportunities based on structural similarity. It is important to emphasize, however, that the new coefficients are specific to these three refactorings, since they are the most widely used by developers (Silva et al 2016).

Firstly, we investigated the accuracy of 18 similarity coefficients in 10 systems (training set) of *Qualitas.class Corpus* (Tempero et al 2010). Secondly, we adapted the *Simple Matching* coefficient through genetic algorithms in order to generate three new coefficients (*PTMC*, *PTMM* e *PTEM*) with greater precision in the identification of *Move Class*, *Move Method*, and *Extract Method* refactoring opportunities, respectively. Thirdly, we compared the proposed coefficients with those existent in other 101 systems. The results indicate, in relation to the best analyzed coefficient, a statistically significant improvement from 5.23% to 6.81% for identification of *Move Method* opportunities, 12.33% to 14.79% for *Move Class*, and 0.25% to 0.40% for *Extract Method*. Finally, we implemented a tool that identifies refactoring opportunities based on the proposed coefficients.

This article is an extended version of our first work on our first work on proposing such coefficients (Pinto and Terra 2017) where we highlight the following improvements: (i) an adaptation in the proposal of the coefficients, where exponents weights and more concise numbers are applied in their formulas; (ii) the introduction of a new experiment in the methodology to ensure better results of the adapted coefficients; (iii) an improved analysis of the results, addressing the problem that some data do not follow a normal distribution; (iv) new results, with higher precision of the proposed coefficients, thus more efficient; and (v) a more complete and detailed tool section.

The remainder of this article is organized as described below. Section 2 introduces fundamental concepts to the study. Section 3 describes the used methodology. Section 4 presents an analysis of existing coefficients, with the objective of selecting the coefficient to be adapted. Section 5 proposes three new coefficients for identifying refactoring opportunities. Section 6 shows an evaluation of the proposed coefficients in 101 systems. Section 7 describes the implementation of a tool to identify refactoring opportunities based on the proposed coefficients. Section 8 discusses related studies. Finally, Section 9 presents the final considerations and future research.

2 Background

In order to provide the necessary knowledge for the conception and understanding of this article, we introduce the fundamental concepts. Section 2.1 deals with the refactoring process, which involves methods for source code restructuring. Section 2.2 deals with structural similarity, as also presents the

current main coefficients in the literature. Section 2.3 introduces optimization algorithms, focusing on genetic algorithms, besides presenting an illustrative example of optimization.

2.1 Refactoring

Code refactoring is the process of moving a software system in a way that preserves the external code behavior while improves its internal structure (Fowler et al 1999). Through the refactoring, it becomes possible to treat the occurrence of different Code Smells. Although some refactoring processes consider other aspects (e.g., semantics), the present study focuses exclusively on Code Smells structurally manifested in the source code.

Several techniques can be used for code refactoring. Considering that this article focuses its analysis exclusively on classes, methods, and blocks of a system, the *Move Class*, *Move Method*, and *Extract Method* refactorings will be used in order to address the occurrence of Code Smells. Whereas the application of *Move Class* and *Move Method* consists, respectively, in the simple movement from a class to another package and in the movement from a method to another class, *Extract Method* is accomplished through the extraction from a code block to a new method that will be generated by replacing the extracted fragment with a call to the new referred method.

Although the application of *Move Class* for repositioning a class in its due package does not address any specific Code Smell, it is fundamental to avoid the occurrence of any Code Smell since the undue location of a class will contribute to its inadequate internal structure, i.e., it will result in inappropriately positioned methods and blocks.

However, the *Move Method* makes possible treating the following Code Smells: *Large Class*, in which the repositioning of undue methods will reduce the size of the referred class; *Divergent Change* and *Shotgun Surgery*, in which it will be possible to position methods that need changes in a specific class that does not cause the need for changes; *Feature Envy*, in which methods that overuse internal properties from another class can be moved to the other referred class; and *Refused Bequest*, where superclass methods, which do not have properties in common with all subclasses, can be moved to the subclasses that use them.

The *Extract Method* refactoring, in turn, provides means of dealing with *Long Method*, in which extracting code blocks into new methods will reduce the size of the referred method. Furthermore, although *Extract Method* deals only directly with *Long Method*, it can indirectly be used to treat other Code Smells (e.g., *Large Class*, *Divergent Change*, *Shotgun Surgery*, *Feature Envy*, and *Refused Bequest*) since after extracting the code block in a new method, this one can be moved to another class through *Move Method*. In this way, the referred Code Smells can be handled in situations where they occur in only certain code blocks.

2.2 Structural Similarity

In order to identify the occurrence of Code Smells and hence code refactoring opportunities, it is primordial to analyze the structural similarity of the code entities present in a software design. Similarity, in the context of software architecture, deals with the relationship between shared properties between two or more entities present in the code structure.

For the calculation of similarity indices, Table 1 reports the main similarity coefficients proposed in the literature (Terra et al 2013).

Table 1: Similarity Coefficients

Coefficient	Definition	Range
Baroni-Urbani and Buser	$[a + (ad)^{\frac{1}{2}}] / [a + b + c + (ad)^{\frac{1}{2}}]$	[0-1*]
Dot-product	$a / (b + c + 2a)$	[0-1*]
Hamann	$[(a + d) - (b + c)] / [(a + d) + (b + c)]$	[-1-1*]
Jaccard	$a / (a + b + c)$	[0-1*]
Kulczynski	$\frac{1}{2} [a / (a + b) + a / (a + c)]$	[0-1*]
Ochiai	$a / [(a + b)(a + c)]^{\frac{1}{2}}$	[0-1*]
Phi	$(ad - bc) / [(a + b)(a + c)(b + d)(c + d)]^{\frac{1}{2}}$	[-1-1*]
PSC	$a^2 / [(b + a)(c + a)]$	[0-1*]
Relative Matching	$[a + (ad)^{\frac{1}{2}}] / [a + b + c + d + (ad)^{\frac{1}{2}}]$	[0-1*]
Rogers and Tanimoto	$(a + d) / [a + 2(b + c) + d]$	[0-1*]
Russell and Rao	$a / (a + b + c + d)$	[0-1*]
Simple Matching	$(a + d) / (a + b + c + d)$	[0-1*]
Sokal and Sneath	$2(a + d) / [2(a + d) + b + c]$	[0-1*]
Sokal and Sneath 2	$a / [a + 2(b + c)]$	[0-1*]
Sokal and Sneath 4	$\frac{1}{4} [a / (a + b) + a / (a + c) + d / (b + d) + d / (c + d)]$	[0-1*]
Sokal binary distance	$[(b + c) / (a + b + c + d)]^{\frac{1}{2}}$	[0*-1]
Sorenson	$2a / (2a + b + c)$	[0-1*]
Yule	$(ad - bc) / (ad + bc)$	[0-1*]

The * symbol indicates the maximum similarity

For the understanding of each similarity coefficient, consider two code entities A and B . Taking into consideration that this article analyzes structural dependencies among code entities, it has the following variables:

- a = number of dependencies in both entities,
- b = number of exclusive dependencies of entity A ,
- c = number of exclusive dependencies of entity B , and
- d = number of the remainder from the total dependencies considered.

In order to illustrate the calculation of structural similarity between two code entities, we applied the *Jaccard* and *Simple Matching* coefficients in two methods of the system MyAppointments, a system of control and management of commitments (Passos et al 2010). In this way, we selected methods `loadAppointments`, responsible for loading appointments of the system, and `getAppointmentRowAsDate`, responsible for returning the appointment date of a specific row in the data set. Both methods are present in the same control class. Code 1 shows the implementation of both methods.

Based on the analysis of both methods and the structural dependencies present in their structures, it has:

- $a = 2$, since both methods access methods from `AgendaView` and `DateUtils` (highlighted by red color);
- $b = 4$, since method `loadAppointments` has the following exclusive dependencies: the throw of an exception of type `Exception`, the declaration of `List` and `Appointment`, and the access to methods from `AgendaDAO` (highlighted by blue color);

```

1 public class AgendaController
2     ...
3
4     public void loadAppointments() throws Exception {
5         List<Appointment> apps = agendaDAO.getAppointments
6             (DateUtils.getCurrentDay(),
7              DateUtils.getCurrentMonth(),
8              DateUtils.getCurrentYear());
9
10        int i = 0 ;
11        for(Appointment app : apps) {
12            agendaView.insertAppointRow
13                (i, DateUtils.toString(
14                    app.getDate(),
15                    DateUtils.HOUR_FMT),
16                    app.getTitle());
17            i++ ;
18        }
19    }
20
21    private Date getAppointmentRowAsDate(int row) {
22        String[] appHour =
23            agendaView.getAppointmentRow(row)[0].split(":");
24        return DateUtils.newDate
25            (DateUtils.getCurrentDay(),
26             DateUtils.getCurrentMonth(),
27             DateUtils.getCurrentYear(),
28             Integer.parseInt(appHour[0]),
29             Integer.parseInt(appHour[1]));
30    }
31 }

```

Code 1: **AgendaController** Fragment

- $c = 1$, since the exclusive dependency of the method `getAppointmentRowAsDate` is the declaration of type `Date` as return (highlighted by green color);
- $d = 32$, since when considering the entire system, another 32 types are used by other system entities.

It should be noted that dependencies to primitive types (e.g., `int`, `char`, `byte`, etc.), to wrappers classes (e.g., `Integer`, `Long`, `Character`, etc.) and `String` type are disregarded during the analysis since almost all classes establish dependencies with these types, not contributing to the similarity calculation.

Therefore, when applying, as example, the *Jaccard* and *Simple Matching* coefficient, the following similarity indexes are found:

$$Jaccard = \frac{a}{a + b + c} = \frac{2}{2 + 4 + 1} = 0.28 \quad (1)$$

$$Simple\ Matching = \frac{a + d}{a + b + c + d} = \frac{2 + 32}{2 + 4 + 1 + 32} = 0.87 \quad (2)$$

Thus, it is important to mention that only the gross value does not indicate the similarity level with accuracy since it is necessary to observe and compare the other similarity values of the system altogether. For instance, 0.28 (*Jaccard*) can be considered a high similarity value if the mean similarity of the system is 0.12. Similarly, 0.87 (*Simple Matching*) can be considered as a low value, depending from the other values.

2.3 Optimization Algorithms

Optimization refers to the process of finding and comparing solutions to a given problem, maximizing and/or minimizing the values of the variables that compose the objective function until the best possible solution is found (Chis 2010). However, in many cases, a problem does not have an optimal solution, which results in the search for solutions that meet the desired objective satisfactorily. This concept, in turn, can be applied to adapt and improve the similarity coefficients discussed in the previous section.

When searching solution of optimization problems, several approaches can be applied, involving different types of algorithms. Among the several approaches, the application of a simple genetic algorithm has shown to be able to find satisfactory solutions effectively. Therefore, it has become the chosen approach for application in this article.

2.3.1 Genetic Algorithms

A genetic algorithm defines a set of candidates for the proposed solution and, at each iteration (each generation of the genetic algorithm), selects and combines the most suitable candidates, which may undergo slight changes. Thus, at the end of all execution, the solution set will consist of candidates with relative great potential to optimize the objective function (Sivanandam and Deepa 2007).

Considering the problem of optimizing similarity coefficients, the algorithm defines weights and exponents for each variable of the chosen coefficient formula and each iteration seeks to adapt each weight and exponent in order to optimize the result of the objective function, finally selecting the set of best results. For instance, considering the *Jaccard* coefficient (presented in Table 1), and the weights $P_{a'}$, $P_{a''}$, P_b and P_c , as also the exponents $E_{a'}$, $E_{a''}$, E_b , and E_c , corresponding, respectively, to variables a of the numerator and a , b and c of the denominator. Thus, it has as a result:

$$\text{Resulting Coefficient} = \frac{P_{a'} * a'^{E_{a'}}}{P_{a''} * a''^{E_{a''}} + P_b * b^{E_b} + P_c * c^{E_c}} \quad (3)$$

During the execution of a genetic algorithm, the parameters are defined, such as objective function (or fitness function), initial population, number of generations, selection operator, crossover operator, and mutation operator. The objective function represents the data or function to be optimized. The initial population stipulates the initial number of possible candidates for the desired weights and exponents in order to optimize the objective function. Number of generations refers to the number of times the genetic algorithm will be repeated, i.e., iterate by adapting the desired values. The selection operator will choose candidates that are more likely to cross-check between them, thus generating one or more candidates who may show more suitability to the solution set. Finally, the generated candidate(s) may undergo a slight mutation, i.e., a slight change in its value, which prevents its stagnation, besides allowing it arriving at any point of the search space.

Among the different operators of selection, crossing and mutation, this article uses, respectively, *Binary Tournament*, *Simulated Binary Crossover*, and *Polynomial Mutation*. The *Binary Tournament* method selects, among all the possible generated candidates until then, two individuals that present higher results in relation to the objective function so that they are crossed. The *Simulated Binary Crossover* occurs through the crossing of two individuals, combining their binary representations, in order to generate two new individuals. Such a combination considers a defined probability, analyzing whether each binary index should be combined or not. Finally, the *Polynomial Mutation* also considers a defined probability in order to change one or more binary indexes of the individuals resulting from the crossover. In both crossover and mutation operators, a distribution index must be established in order to evaluate the diversity of the selected solutions in search space, which guarantees the selection of more heterogeneous individuals.

2.3.2 Illustrative Example

For a better understanding of the approach of optimization algorithms and concepts of genetic algorithms, this section presents the application of a genetic algorithm to optimize the *Jaccard* similarity coefficient in a little example of *Move Class* refactoring.

For instance, take as assumption a system with two packages *pkg1* and *pkg2* with a set of classes and their dependencies, respectively, as $pkg1 = \{A=\{X,Y,W,Z\}, B=\{X,Y,Z,L\}, C=\{X,Y,W,K\}\}$ and $pkg2 = \{D=\{X,W,R,T\}, E=\{R,T,Z\}, F=\{X,Z,R,T,M\}\}$, according to Figure 2.

Assume it is known that the current system architecture is the ideal one and we intend to use a similarity coefficient in which does not result in suggestions

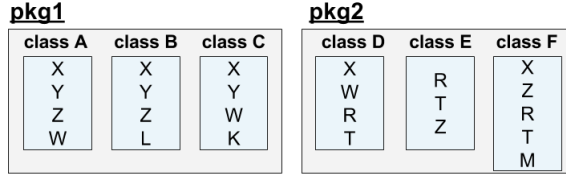


Fig. 2: Illustrative Example

of *Move Class* refactoring, i.e., maintain the current architecture and guarantee good similarity indexes among classes from the same package. To this end, it is intended to adapt a coefficient in order to maximize the similarity of $\text{sim}(A, B)$, $\text{sim}(A, C)$, $\text{sim}(B, C)$, $\text{sim}(D, E)$, $\text{sim}(D, F)$, and $\text{sim}(E, F)$. Simultaneously, it is expected to minimize the similarity $\text{sim}(A, D)$, $\text{sim}(A, E)$, $\text{sim}(A, F)$, $\text{sim}(B, D)$, $\text{sim}(B, E)$, $\text{sim}(B, F)$, $\text{sim}(C, D)$, $\text{sim}(C, E)$, and $\text{sim}(C, F)$. However, only the gross value does not indicate the similarity level with accuracy since it is necessary to observe and compare the other similarity values of the system altogether. In this way, it is intended obtaining the largest possible difference between the arithmetic mean of similarities aimed to maximize and to minimize.

Considering the *Jaccard* coefficient as an example, it is decided to apply the genetic algorithm defining weights and exponents for each coefficient variable, as previously described in Equation 3. In this way, we relied on the configurations presented in Table 2, which was obtained after a series of attempts aimed to improve the coefficients, considering the computational resource disposition and execution time.

Table 2: Genetic Algorithms Configuration

Objective function:	Number of optimized similarities	Crossover operator:	Simulated Binary Crossover
Population size:	1200	Probability of mutation:	0.6
Representation of the population:	$\{x \in \mathbb{R} \mid -2 \leq x \leq 2\}$	Probability of crossover:	0.9
Number of generations:	150	Mutation distribution index:	20.0
Selection operator:	Binary Tournament	Crossover distribution index:	20.0
Mutation operator:	Polynomial Mutation		

In this scenario, we found as a possible solution set, the weights 1.34 ($P_{a'}$), 0.33 ($P_{a''}$), 1.0 (P_b), and 0.34 (P_c), as also 0.5 ($E_{a'}$), 1.0 ($E_{a''}$), 2.0 (E_b), and 2.0 (E_c), respectively to variables a of the numerator and a , b , and c of the denominator. The Comparison of similarity between *Jaccard* and the resulting coefficient is shown in Table 3.

It is possible to observe that the coefficient resulting from the optimization showed to be much more efficient than *Jaccard* since it obtained a difference between the mean of similarities that should be maximized and the mean of

Table 3: Comparison between *Jaccard* and the resulting coefficient

Maximize				
Objective	<i>Jaccard</i>	Similarity	Resulting Coefficient	Similarity
$\text{sim}(\text{A},\text{B}) = \frac{3}{3+1+1} =$		0.6	$\frac{1.34\sqrt{3}}{0.33\sqrt{3^1} + 1.0\sqrt{1^2} + 0.34\sqrt{1^2}} =$	0.9961
$\text{sim}(\text{A},\text{C}) = \frac{3}{3+1+1} =$		0.6	$\frac{1.34\sqrt{3}}{0.33\sqrt{3^1} + 1.0\sqrt{1^2} + 0.34\sqrt{1^2}} =$	0.9961
$\text{sim}(\text{B},\text{C}) = \frac{2}{2+2+2} =$		0.3333	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{2^2}} =$	0.3148
$\text{sim}(\text{D},\text{E}) = \frac{2}{2+2+1} =$		0.4	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{1^2}} =$	0.3790
$\text{sim}(\text{D},\text{F}) = \frac{3}{3+1+2} =$		0.5	$\frac{1.34\sqrt{3}}{0.33\sqrt{3^1} + 1.0\sqrt{1^2} + 0.34\sqrt{2^2}} =$	0.6928
$\text{sim}(\text{E},\text{F}) = \frac{3}{3+0+2} =$		0.6	$\frac{1.34\sqrt{3}}{0.33\sqrt{3^1} + 1.0\sqrt{0^2} + 0.34\sqrt{2^2}} =$	0.9876
Mean		0.5055	Mean	0.7277

Minimize				
Objective	<i>Jaccard</i>	Similarity	Resulting Coefficient	Similarity
$\text{sim}(\text{A},\text{D}) = \frac{2}{2+2+2} =$		0.3333	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{2^2}} =$	0.3148
$\text{sim}(\text{A},\text{E}) = \frac{1}{1+3+2} =$		0.1667	$\frac{1.34\sqrt{1}}{0.33\sqrt{1^1} + 1.0\sqrt{3^2} + 0.34\sqrt{2^2}} =$	0.1253
$\text{sim}(\text{A},\text{F}) = \frac{2}{2+2+3} =$		0.2857	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{3^2}} =$	0.2455
$\text{sim}(\text{B},\text{D}) = \frac{2}{2+2+2} =$		0.3333	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{2^2}} =$	0.3148
$\text{sim}(\text{B},\text{E}) = \frac{0}{0+4+3} =$		0.0	$\frac{1.34\sqrt{0}}{0.33\sqrt{0^1} + 1.0\sqrt{4^2} + 0.34\sqrt{3^2}} =$	0.0
$\text{sim}(\text{B},\text{F}) = \frac{1}{1+3+4} =$		0.125	$\frac{1.34\sqrt{1}}{0.33\sqrt{1^1} + 1.0\sqrt{3^2} + 0.34\sqrt{4^2}} =$	0.0907
$\text{sim}(\text{C},\text{D}) = \frac{1}{1+3+3} =$		0.1429	$\frac{1.34\sqrt{1}}{0.33\sqrt{1^1} + 1.0\sqrt{3^2} + 0.34\sqrt{3^2}} =$	0.1081
$\text{sim}(\text{C},\text{E}) = \frac{1}{1+3+2} =$		0.1667	$\frac{1.34\sqrt{1}}{0.33\sqrt{1^1} + 1.0\sqrt{3^2} + 0.34\sqrt{2^2}} =$	0.1253
$\text{sim}(\text{C},\text{F}) = \frac{2}{2+2+3} =$		0.2857	$\frac{1.34\sqrt{2}}{0.33\sqrt{2^1} + 1.0\sqrt{2^2} + 0.34\sqrt{3^2}} =$	0.2455
Mean		0.2044	Mean	0.1744

the similarities that should be minimized of 0.55, in contrast to *Jaccard* which obtained a difference of only 0.30. Therefore, the resulting coefficient proved to be the more adequate and accurate for the similarity analysis in the presented example.

3 Methodology

In order to create three new similarity coefficients that are more efficient in identifying, respectively *Move Class*, *Move Method* and *Extract Method* refactoring opportunities, this article adopts a methodology based on the selection of a coefficient to be adapted through the analysis of the main existing similarity coefficients (Section 4), in the proposal of the three new coefficients after adaptation of the respective selected coefficient (Section 5) and in the evaluation of the same coefficients (Section 6). Finally, we develop a tool that identifies those refactoring opportunities through our proposed coefficient.

In this respect, the similarity calculation, used in each methodology step, is performed by comparing a given class to a given package, besides comparing a certain method to a particular class, as well as analyzing the resulting similarity of a given class after the extraction of a particular block for generation of a new method. Such process is detailed below.

3.1 Similarity Calculation

Class-Package: The similarity of a class with a package is given through the arithmetic mean of the similarity between the respective class and the other classes in the package. This decision is based on the fact that measuring similarity of a simple class and the entire package itself could result in a high similarity even though the classes were not similar. Consequently, it is possible to suggest *Move Class* refactorings for packages that actually have similar classes.

Method-Class: The same approach used to identify *Move Class* refactoring opportunities is applied to the similarity analysis between methods and classes, in which it is considered the similarity mean between one method and the other methods in the class.

Block-Method: Initially, the similarity of each method of a class is calculated with the other methods of the same class, thus obtaining the internal similarities of the class. Then, for each method, all the extraction possibilities of its blocks will be analyzed in order to generate a new method (*Extract Method*), i.e., for each extracted block, the similarities of the class after extraction will be recalculated, including the new generated method. Finally, the arithmetic means of both sets of values are compared in order to verify whether or not the *Extract Method* refactoring should be suggested.

In this way, the coefficients are analyzed and evaluated using the systems from *Qualitas.class Corpus* (Tempero et al 2010), database with 111 object-oriented systems, more than 18 million LOC (*Lines Of Code*), 200 thousand compiled classes, and 1.5 million compiled methods. Based on the fact that the *Qualitas.class Corpus* is composed of more mature and stable systems, we assume that the current structure of systems has a reasonable similarity index in relation to possible code refactorings. To summarize, it is intended that most of the structure is maintained and few refactoring is performed. Thus, it is possible to evaluate the accuracy of each similarity coefficient in view of each analyzed system.

Experimental Rules: To avoid the occurrence of false positives in the resulting refactoring recommendations, after a series of experimental tests and executions, we defined nine experimental rules for the implementation of the proposed solution:

1. *We disregard the entity under analysis when the system searches for refactoring opportunities.* When the similarity between a class A and its respective package Pkg is calculated, the system considers Pkg to be $Pkg - \{A\}$. In this case, individually located entities are totally discarded;
2. *We did not consider packages and test classes,* since most systems organize their test classes into a single package. Therefore, this package contains classes related to different parts of the system, i.e., they are not structurally related. For this purpose, an approach that disregards packages

and classes containing the text “`test`” (uppercase or lowercase) anywhere in its name is used. Although neither all test package or class containing the text “`test`” and not every package or class that contains it is in fact a test package or class, the gain in precision is greater than the loss when test packages or classes are erroneously detected;

3. *We ignored trivial dependencies.* Dependencies—such as those established with primitive types and wrappers (e.g., `int` and `java.lang.Integer`), dependencies of `java.lang.String` and `java.lang.Object`—are filtered. Since the vast majority of code elements establish dependencies with these types, they do not actually contribute to the similarity calculation;
4. *We did not analyze class, method, or block entities that establish dependencies with fewer than three types.* Although there may be entities with few dependencies that should be refactored, these entities contain little information for calculating similarity or for making any inference based on their structural dependencies. However, it should be noted that although these entities are not analyzed whether they should be refactored, they are still considered as a possible refactoring destination, whether they have at least some dependency;
5. *We did not analyze entities that are not co-located with at least two other considered entities.* For instance, a package with only two classes does not provide enough structural information to recommend whether or not to move its classes. Again, this criterion disregards only the analysis of such entities, but can still be considered as a refactoring destination;
6. *We did not extract the first block of a method.* The extraction of the first block of a method will not change a similarity. Since the extraction of a block will result in the extraction of its internal blocks, this operation will only recreate the respective method. In this situation, the ideal would be to move the method;
7. *We assigned any dependency present on an internal entity class, method or block to its external entities.* Given that an internal code entity (e.g., nested classes, internal methods and blocks, etc.) might be within the scope of the external entity(ies), any established dependency must be assigned to external entities;
8. *We disregard methods and blocks belonging to an `Interface` type.* Due to the very nature of interfaces being composed of abstract members, they should not be moved or extracted since this would lead to a series of conflicts in the project structure; and
9. *We disregard constructor methods and their respective blocks.* Considering the requirement of constructors in a class, the movement of this method type is disregarded, although it is intended to evaluate the possible extraction of its internal blocks.

4 Analysis and Comparison of Coefficients

This section analyzes and compares the existing similarity coefficients in order to find the most adequate coefficient to be adapted. Thus, we propose new similarity coefficients in order to obtain higher precision rates. Thus, we investigate 18 similarity coefficients (Table 1) in 10 completely randomized systems (training set) of *Qualitas.class Corpus*, as can be observed in Table 4.

Table 4: Target-Systems

System	Version	System	Version
Ant	1.8.2	JFreeChart	1.0.13
ArgoUML	0.34	JHotDraw	7.5.1
Collections	3.2.1	JMeter	2.5.1
Hibernate	4.2.0	JUnit	4.1
JEdit	4.3.2	Tomcat	7.0.2

We applied all analyzed similarity coefficients to each entity of the selected target systems, analyzing whether an entity has the highest similarity with the entity it actually is. For instance, for a class *A*, each coefficient is applied by comparing *A* with each package in the system, seeking to verify whether the highest found similarity rate (*Top #1*) corresponds to the package in which *A* is actually positioned. This evaluation also analyzes the second (*Top #2*) and the third (*Top #3*) higher rate in order to verify whether the applied coefficient has at least results close to the desired one. Finally, the arithmetic mean is calculated on each system of training set considering *Top #1*, *#2*, and *#3*. Considering that some data do not follow a normal distribution, we performed the analysis over the median of these sets of means since median represents a better estimate of central tendency than the overall mean for a system.

We evaluated each coefficient was evaluated separately in relation to the types of code refactoring *Move Class*, *Move Method*, and *Extract Method*. Table 5 presents the similarity accuracy of the addressed coefficients in relation to the identification of the *Move Class*, *Move Method*, and *Extract Method* refactoring opportunities. *Extract Method* refactorings consider only a single success rate since for *Extract Method* refactorings it is only analyzed whether the method should be extracted or not, therefore, the possibility of success in the second or third attempt is discarded (*Top #2* or *#3*). For space constraints, the detailed table with the results of each system is publicly available at our comparison website¹.

For a complementary analysis between the data, Figure 3 presents a violin plot regarding *Top #1* of each refactoring, where it is illustrated the density of each coefficient (i.e., probability of a variable assume a certain value), as

¹ https://github.com/rterrabh/2018_emse

Table 5: Similarity precision of the 18 similarity coefficients (training set)

MOVE CLASS			
Coefficient	Median		
	Top1	Top2	Top3
Baroni-Urbani and Buser	26.91%	40.39%	47.10%
Dot-product	44.73%	55.69%	62.34%
Hamann	17.14%	22.94%	25.15%
Jaccard	46.30%	59.24%	64.25%
Kulczynski	41.03%	55.41%	63.02%
Ochiai	43.89%	56.72%	63.55%
Phi	43.80%	56.84%	63.00%
PSC	48.49%	62.33%	71.32%
Relative Matching	33.33%	45.95%	52.75%
Rogers and Tanimoto	17.23%	23.17%	25.35%
Russell and Rao	39.52%	53.14%	61.12%
Simple matching	15.90%	22.94%	25.20%
Sokal and Sneath	14.96%	22.61%	24.80%
Sokal and Sneath 2	47.50%	60.10%	66.08%
Sokal and Sneath 4	41.23%	55.52%	63.12%
Sokal binary distance	18.40%	24.57%	27.06%
Sorenson	44.73%	55.69%	62.34%
Yule	18.51%	29.42%	34.62%

MOVE METHOD			
Coefficient	Median		
	Top1	Top2	Top3
Baroni-Urbani and Buser	14.34%	19.03%	23.66%
Dot-product	27.61%	37.43%	44.09%
Hamann	17.47%	23.78%	28.48%
Jaccard	31.25%	41.54%	48.59%
Kulczynski	20.54%	28.38%	35.27%
Ochiai	25.11%	33.88%	40.41%
Phi	25.03%	33.88%	40.41%
PSC	32.05%	43.75%	51.06%
Relative Matching	19.82%	27.90%	33.41%
Rogers and Tanimoto	17.52%	23.95%	28.84%
Russell and Rao	26.43%	36.13%	43.52%
Simple matching	17.45%	23.78%	28.32%
Sokal and Sneath	16.65%	23.06%	27.58%
Sokal and Sneath 2	34.17%	45.26%	52.27%
Sokal and Sneath 4	20.46%	28.33%	35.27%
Sokal binary distance	19.46%	26.73%	33.12%
Sorenson	27.61%	37.43%	44.09%
Yule	7.72%	12.81%	17.28%

EXTRACT METHOD	
Coefficient	Median Hits
Baroni-Urbani and Buser	54.80%
Dot-product	59.23%
Hamann	10.37%
Jaccard	62.92%
Kulczynski	69.65%
Ochiai	63.71%
Phi	63.69%
PSC	69.42%
Relative Matching	72.27%
Rogers and Tanimoto	10.29%
Russell and Rao	77.55%
Simple matching	10.31%
Sokal and Sneath	10.34%
Sokal and Sneath 2	68.85%
Sokal and Sneath 4	69.48%
Sokal binary distance	18.24%
Sorenson	59.23%
Yule	56.45%

well as its quartiles, including its median. The dashed line indicate the highest coefficient median.

After analyzing and comparing the main existing coefficients, we selected *Simple Matching* to be adapted in order to generate a new coefficient. The *Simple Matching* coefficient has an easy structure to be adapted, defining weights for the variables, thus contributing to obtain good results in the proposal of new coefficients. In contrast, a large part of the analyzed coefficients had already defined weights. Although it was not the best-performing coefficient, the possible weight definition of *Simple Matching* allows simulating coefficients as *Sokal and Sneath 2*, which was the best coefficient in *Move Method* and second best in *Move Class*², as well as *Russell and Rao*, which obtained greater accuracy in *Extract Method*. Additionally, the coefficient considers the universe of dependencies, in contrast to coefficient *PSC*, for instance.

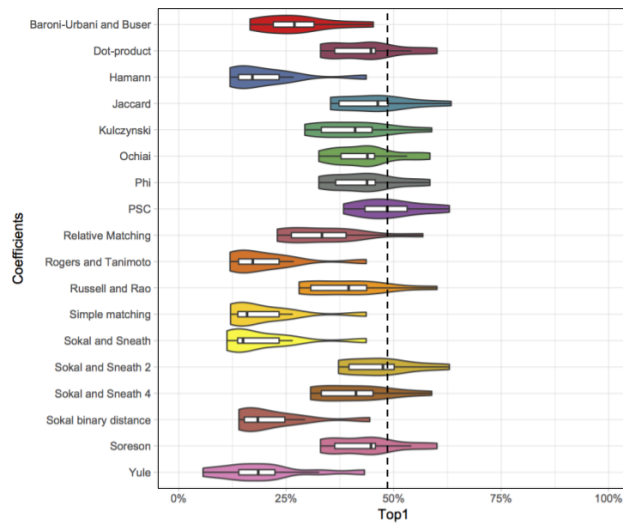
5 Proposal of New Coefficients

Given the fact that genetic algorithms have nondeterministic execution and results, it becomes a problem to know if their results are satisfactory or even adequate for the expected purpose. In order to address this problem, it was designed an experiment for the proposal of the new coefficients. The experiment consists in a treatment combination with replication, i.e., the repetition of multiple executions in different groups of factor levels. This design is composed by two factors, the initial population size and the number of generations for the genetic algorithm.

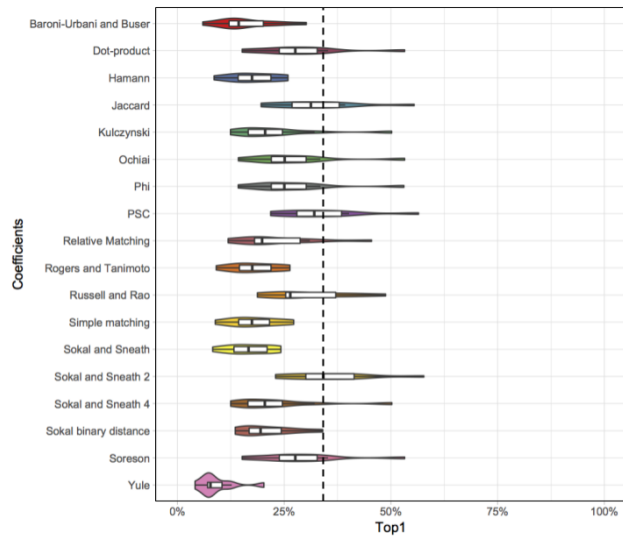
After selecting the *Simple Matching* coefficient to be adapted, several executions of a genetic algorithm is applied to the referred coefficient, having 10 systems of the *Qualitas.class Corpus* as training set.

The conducted experiment aim to find which candidates resulted in a better value for the objective function. For this purpose, the genetic algorithm relies on the configuration used in the illustrative example of Section 2.3.2 (see Table 2). However, in order to find better results and lower the degree of uncertainty, it was defined five configuration groups, with the combination of different factor levels, and replicated five times each, totaling 25 executions. More precisely, each group maintains the original genetic algorithm configuration, changing only the initial population size and the number of generations (the experiment factors), as shown in Table 6.

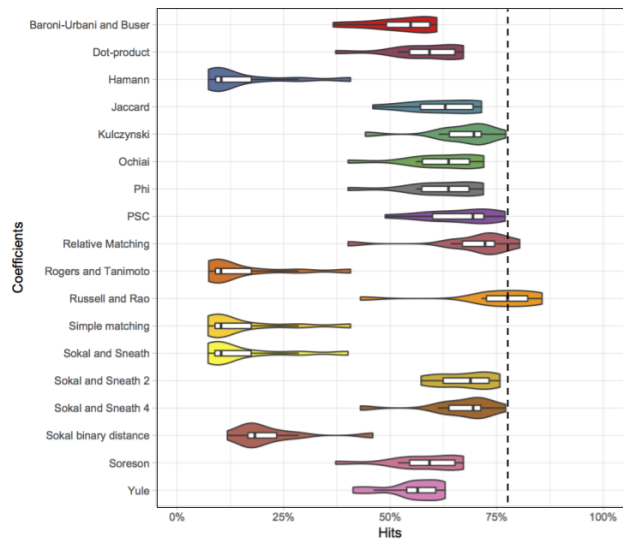
² *PSC* was the best coefficient for *Move Class*, however, it was not selected because it englobes elementary arithmetic operations among the variables *a*, *b*, and *c*. Such mathematical relation is out of the scope of this study.



(a) Move Class



(b) Move Method



(c) Extract Method

Fig. 3: Refactorings Plots

Table 6: Experiment Groups of Factor Level

	Population Size	No of Generations
Group 1	2000	1000
Group 2	1500	1500
Group 3	500	2000
Group 4	300	2500
Group 5	100	3000

It is important to clarify that each new proposed coefficient is resulting from a different set of executions of the genetic algorithm (25 executions for each coefficient). Since a single iteration of the genetic algorithm performs thousands of comparisons between code entities to improve results, we argue that this number of executions is satisfactory.

After generating the set of solutions, in most cases are presented several possibilities that result in the same highest accuracy rate. Therefore, a single solution is selected that shows a greater distance between the mean of similarity indexes that it seeks to maximize with the mean of indexes that it seeks to minimize since a greater distance indicates that the coefficient will tend to maximize and to minimize the similarities correctly in other systems.

Thus, considering the possibility of defining weights to the *Simple Matching* coefficient variables, it was initially simulated as genetic algorithm candidates the weights of *Sokal and Sneath 2* coefficient for *Move Class* and *Move Method* refactorings, as well as *Russell and Rao* weights for *Extract Method* refactorings, which facilitates the selection of new candidates since they were the best results in their respective refactorings. Subsequently, we performed the executions of the genetic algorithm on the 10 systems (training set) in order to obtain more adequate weights for each coefficient variable and to create the new coefficients to be evaluated in the other 101 systems (test set).

Finally, each execution of the experiment was carried out in order to find a higher value for the objective function. Table 7 shows the values of each execution.

In this way, we proposed the following coefficients, where each one corresponds to a type of refactoring code: *PTMC* for *Move Class* operations, *PTMM* for *Move Method* operations and *PTEM* for *Extract Method* operations. The implementation of the genetic algorithm on *Simple Matching* in the defined training set resulted in the first version of the three sought coefficients. The resulting weights $P_{a'}$, $P_{d'}$, $P_{a''}$, P_b , P_c and $P_{d''}$, as also the resulting exponents $E_{a'}$, $E_{d'}$, $E_{a''}$, E_b , E_c and $E_{d''}$, corresponding to the variables a and d of the numerator and a , b , c and d of the denominator, are reported in Equations 4, 5, and 6. Thereby, when assigning each weight to the corresponding variable of *Simple Matching*, we have the following proposed coefficients:

Table 7: Experiment’s Objective Function Results

MOVE CLASS					
	Execution #1	Execution #2	Execution #3	Execution #4	Execution #5
Group 1	3268	3176	3282	3112	3198
Group 2	3288	3061	3241	3281	3186
Group 3	3273	3044	3273	3171	3098
Group 4	3211	2989	3065	3163	3177
Group 5	3138	3250	3151	3382	3281

MOVE METHOD					
	Execution #1	Execution #2	Execution #3	Execution #4	Execution #5
Group 1	9794	9736	10142	9256	9915
Group 2	9376	10058	9794	9736	10129
Group 3	9060	9126	9051	10189	10075
Group 4	9944	9370	9995	9611	9922
Group 5	9890	9446	9164	9892	10091

EXTRACT METHOD					
	Execution #1	Execution #2	Execution #3	Execution #4	Execution #5
Group 1	21128	21165	21217	21192	21110
Group 2	21173	21235	21141	21229	21261
Group 3	21291	21207	21135	21112	21175
Group 4	21285	21156	21262	21287	21262
Group 5	21145	21234	21235	21290	21270

$$PTMC = \frac{2a^3 + 0.1\sqrt{d}}{1.71a^2 + 1.98b^2 + 1.78c^2 + 0.1d} \quad (4)$$

$$PTMM = \frac{2a^3 + 0.85d}{1.64a + 1.95\sqrt{b} + 0.1c + 0.9d} \quad (5)$$

$$PTM = \frac{0.48\sqrt{a} + 1.56d^2}{1.82a + 1.89b + 1.87c^2 + 0.47d^2} \quad (6)$$

6 Evaluation of the Proposed Coefficients

In order to evaluate the efficiency of the proposed coefficients, we compared the accuracy of the respective coefficients with other 18 coefficients in the literature (Table 1), involving the other 101 systems of the *Qualitas.class Corpus* (test set). We analyzed and compared each proposed coefficient according to its respective code refactoring, i.e., this section presents a different comparison for *Move Class*, *Move Method*, and *Extract Method*.

For the evaluation of the results, we adopted the same analysis approach described in Section 4. Thus, Table 8 reports the results of the accuracy rates of coefficients analyzed for *Move Class* (including *PTMC*), *Move Method* (including *PTMM*), and *Extract Method* (including *PTM*) refactorings. Again, the analysis considers the median of the sets of means, since some data do not follow a normal distribution. For space constraints, the detailed table comprehending the results of each of the 101 systems is publicly available at our comparison website³.

³ https://github.com/rterrabh/2018_emse

Move Class: The proposed coefficient reached higher values compared to the others, having an approximate median of 62.20%, 71.52%, and 79.45% in relation to the similarity rates *Top #1*, *#2*, and *#3*. *PSC* presented the second best median among the coefficients analyzed. For *Top #1*, *#2*, and *#3*, *PSC* reached, respectively, the accuracy values of 55.98%, 71.26% e 77.31%.

More importantly, when comparing the *Top #1* of the coefficients, *PTMC* presented a statistically significant improvement from 5.23% to 6.81%, according to the Wilcoxon Test with a 95% confidence ($p\text{-value} = 2.2^{-16}$). Similarly as illustrated in Section 4, Figure 4 presents a violin plot regarding *Top #1* of each *Move Class* refactoring, in order to provide a complementary analysis between the data.

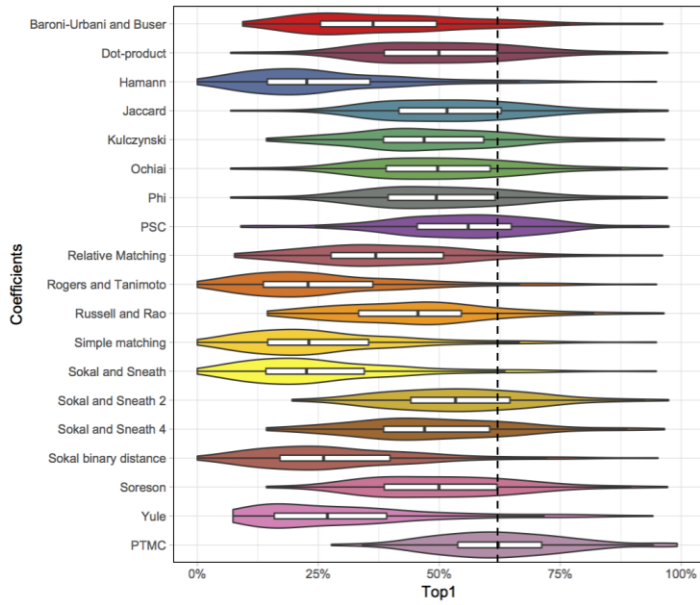


Fig. 4: Move Class Plot

Move Method: *PTMM* surpassed the other coefficients data in the three cases of similarity rate (*Top #1*, *#2*, and *#3*), reaching, respectively, a median of approximately 52.89%, 61.09% and 64.66%. *Sokal and Sneath 2* had the second highest rate where, considering the three cases of similarity rate (*Top #1*, *#2* and *#3*), the coefficient reached, respectively, the median values of 38.02%, 49.35% and 56.04%. Moreover, *PTMM* presented a statistically significant improvement from 12.33% to 14.74%, when comparing *Top #1* of the coefficients, according to the Wilcoxon Test with a 95% confidence ($p\text{-value} = 2.2^{-16}$). Again, Figure 5 presents a violin plot regarding *Top #1* of each *Move Method* refactoring.

Table 8: Similarity precision of the 19 similarity coefficients (test set)

MOVE CLASS			
Coefficient	Top1	Top2 Median	Top3
Baroni-Urbani and Buser	36.31%	48.65%	57.17%
Dot-product	49.93%	65.22%	72.87%
Hamann	22.62%	32.18%	38.77%
Jaccard	51.60%	67.52%	75.48%
Kulczynski	46.86%	64.15%	72.61%
Ochiai	49.66%	64.94%	72.47%
Phi	49.39%	65.22%	72.35%
PSC	55.98%	71.26%	77.31%
Relative Matching	36.86%	54.35%	63.71%
Rogers and Tanimoto	22.88%	32.18%	38.88%
Russell and Rao	45.59%	61.90%	71.31%
Simple matching	23.05%	33.18%	39.23%
Sokal and Sneath	22.56%	32.25%	38.64%
Sokal and Sneath 2	53.35%	69.05%	76.59%
Sokal and Sneath 4	46.95%	63.99%	72.54%
Sokal binary distance	26.09%	35.69%	44.36%
Sorenson	49.93%	65.22%	72.87%
Yule	26.88%	38.64%	47.79%
PTMC	62.20%	71.52%	79.45%

MOVE METHOD			
Coefficient	Top1	Top2 Median	Top3
Baroni-Urbani and Buser	17.09%	25.08%	29.97%
Dot-product	31.25%	41.45%	49.23%
Hamann	20.21%	27.18%	31.61%
Jaccard	35.43%	46.60%	53.09%
Kulczynski	25.33%	36.09%	42.79%
Ochiai	29.30%	41.45%	47.49%
Phi	29.46%	41.29%	47.41%
PSC	36.71%	48.45%	55.98%
Relative Matching	23.23%	34.41%	40.39%
Rogers and Tanimoto	20.36%	27.47%	31.81%
Russell and Rao	30.70%	43.51%	49.86%
Simple matching	20.25%	27.04%	31.59%
Sokal and Sneath	19.56%	26.37%	30.46%
Sokal and Sneath 2	38.02%	49.35%	56.04%
Sokal and Sneath 4	24.35%	35.54%	42.94%
Sokal binary distance	23.75%	30.91%	35.49%
Sorenson	31.25%	41.45%	49.23%
Yule	9.45%	17.38%	22.25%
PTMM	52.89%	61.09%	64.66%

EXTRACT METHOD	
Coefficient	Hits Median
Baroni-Urbani and Buser	61.94%
Dot-product	69.62%
Hamann	10.48%
Jaccard	71.73%
Kulczynski	78.36%
Ochiai	72.95%
Phi	72.68%
PSC	76.11%
Relative Matching	82.46%
Rogers and Tanimoto	10.52%
Russell and Rao	87.93%
Simple matching	10.48%
Sokal and Sneath	10.32%
Sokal and Sneath 2	74.67%
Sokal and Sneath 4	77.27%
Sokal binary distance	16.52%
Sorenson	69.62%
Yule	65.90%
PTM	88.32%

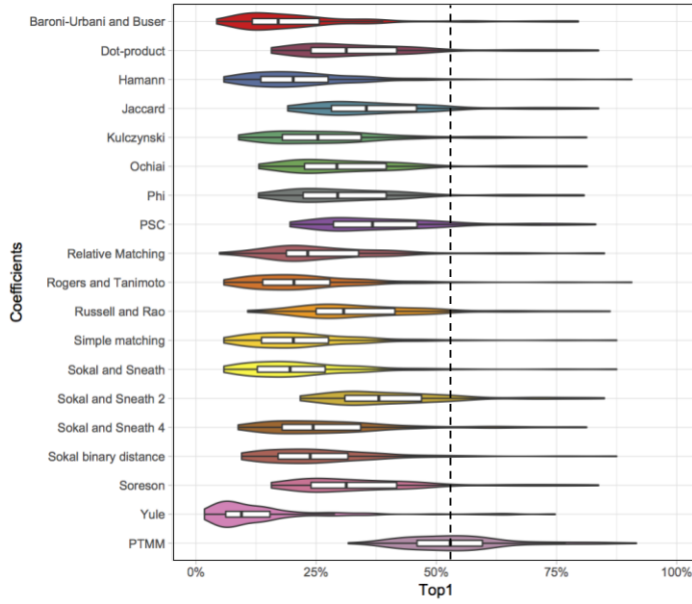


Fig. 5: Move Method Plot

Extract Method: *PTM* presented a median of approximately 88.32%, surpassing the second best coefficient (*Russell and Rao*), which had a median of 87.93%. In addition, *PTM* presented a statistically significant improvement from 0.24% to 0.40% over *Russell and Rao*, according to the Wilcoxon Test with a 95% confidence ($p\text{-value} = 2.2^{-16}$). Finally, Figure ?? presents a violin plot regarding each *Extract Method* refactoring.

Threats to validity: Since the experimental rules play a important role on the results, one could question their underlying design. However, each rule is well well founded and justified at Section 3.1.

Another important thread is that the experiment described in Section 5 is based in a nondeterministic algorithm and, even considering the replications applied, can lead to a set of different results. Also, changing the factors' values can have different results as well since there are infinite possibilites for them.

Moreover, as it is common in empirical studies in Software Engineering, the results cannot be extrapolated (external validity). Although the test set has a reasonable number of 101 systems, it is important to evaluate the new coefficients in real development scenarios.

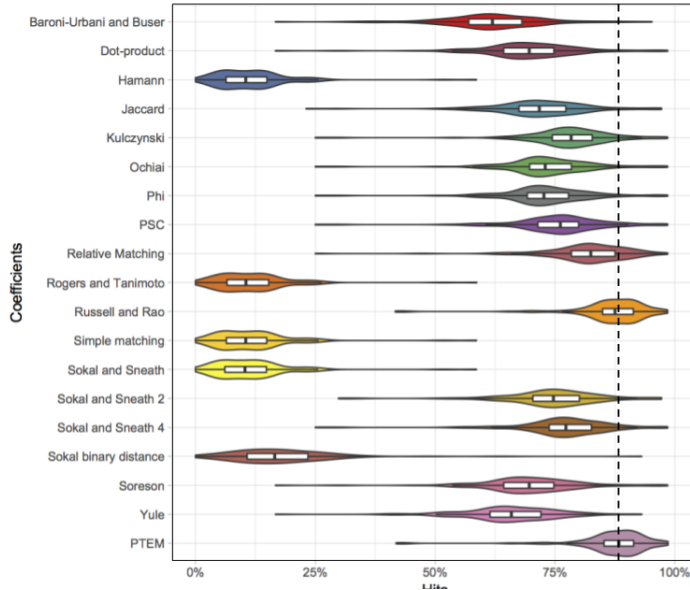


Fig. 6: Extract Method Plot

7 Recommendation System

Aiming to apply the new coefficients proposed in this article to identify refactoring opportunities, we developed *AIRP*⁴, a plug-in prototype for IDE Eclipse.

For a better understanding of the proposed tool, we executed *AIRP* in an example involving a modified version (in order to create the refactoring suggestions) of *myAppointments* (Passos et al 2010), a simple personal information management system with modules *Model*, *View*, *Controller*, and *Util* (following a MVC architecture).

Figure 7 illustrates three examples of suggestions for refactoring identified by *AIRP*. For the modified version of *myAppointments*, class *AngendaDAO* has a similarity mean value 0.22 higher with the classes of package *myappointments.model.domain* (*Move Class*), method *loadDriver()* from class *DB* has a similarity mean value 0.14 higher with methods from *StringUtils* (*Move Method*), and finally, method *actionPerformed* will result in a similarity mean value 0.15 higher with methods from class *AngendaView* if its second inner block from the second block is extracted to a new method (*Extract Method*).

In order to provide the mentioned results, *AIRP* is composed of an architecture based on four main modules:

⁴ Publicly available from: <https://github.com/rterrabh/AIRP>

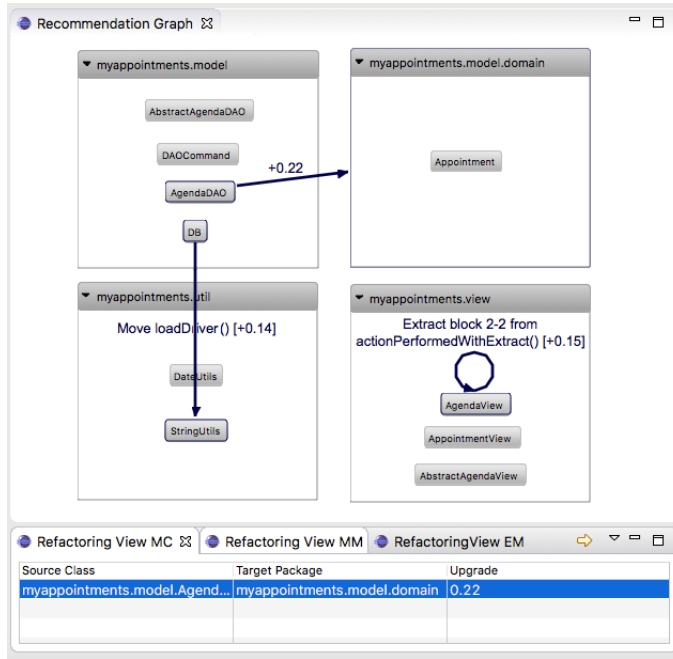


Fig. 7: Refactoring opportunity graph

- **Dependency Extraction Module:** Responsible for identifying and storing all dependencies of a class, method, or block. In other words, it identifies the set of types at which a code entity establishes structural dependency. This includes method calling, attribute access, instantiation, variable declaration, annotation, etc. In order to perform such extraction, we used a syntactic analysis through an AST (*Abstract Syntax Tree*) that checks each element in the source code and, if it refers to a dependency to other entities, stores it according to its package, class, method, and/or block.;
- **Similarity Calculation Module:** It calculates the structural similarity of a given code entity in the target system. This calculation is performed using the formula of the chosen similarity coefficient, making a comparison between the previously stored dependencies of a given class, method, or block with its respective entity (package, class or, method), as described in the methodology of this article (See Section 3.1). By default, it uses the *PTMC*, *PTMM* and *PTEM* coefficients, although it is possible to select any of the other 18 coefficients presentes in Table 1;
- **Recommendation Module:** This module calls the previous one to calculate the resulting similarity of every possible *Move Class*, *Move Method*, and *Extract Method* refactorings in the code. After calculating similarity, its results are compared with a minimum acceptance index of improvement specified by the user. If the result is higher than this

user-defined *threshold*, a list of suggestions is presented with possible refactoring opportunities. Afterwards, it reports the recommendations for the user analysis, being ordered by their similarity index improvement. If the user does not specify a minimum acceptance index, all refactorings that leads to an improvement are displayed; and

- **Visualization Module:** In order to provide more details on the structural organization of the target system, this module generates a refactoring opportunity graph, as illustrated in Figure 7, using the Zest⁵ library. This graph contains the implemented architecture and reports all refactorings in the list of suggestions from previous module. The module also highlights the selected refactoring by the user, showing more details about it, and its respective resulting similarity upgrade. In this way, it makes possible a greater understanding by the user regarding the process performed by the tool, being able to observe the repositioning of involved entities and the architecture resulting from each recommendation.

8 Related Work

Although it was found only one study regarding the proposal of new similarity coefficients, some studies present methodologies and techniques for the identification of refactoring opportunities, adaptations of existing metrics or empirical studies regarding the concepts discussed in the present article. Considering that the objective of this article is to improve the identification of refactoring opportunities through similarity coefficients, we did not considered studies that do not use such coefficients. The most relevant studies to this article are presented below, according to their categories.

Proposal of structural similarity coefficients: Naseem et al (2010) propose a new similarity coefficient aimed at its application in clustering algorithms. In turn, the new proposed coefficient is an adaptation of the *Jaccard* coefficient, called *Jaccard-NM*, whose main difference is the addition of a new variable that considers the universe of analyzed factors, i.e., all the existing factors in the set in which analyzed entities are present. It is important to emphasize that the proposed new similarity coefficient is compared only to the *Jaccard* coefficient, from which it was adapted and in only three systems. On the other hand, this article has an in-depth analysis of the proposed coefficients for a more precise identification in relation to other existing coefficients.

Empirical studies: Terra et al (2013) perform a robust evaluation in 111 systems of the base *Qualitas.class Corpus*, involving 18 of the main similarity coefficients. Considering the study purpose, it is fundamental to the accomplishment of this article, considering that the used coefficients and their

⁵ <https://www.eclipse.org/gef/zest/>

analyses, besides the approach involving structural dependencies, acted as the main basis for the proposal of new coefficients, as well as their evaluations.

Szöke et al (2015) present a case study where it is discussed whether automatic code refactorings actually improve maintainability of software systems. Although the study reports that refactorings usually have a positive impact on maintainability, their execution without proper analysis can negatively impact the system. Although it is extremely important to investigate the actual use of automatic refactorings, this article proposes coefficients that identify more precisely refactoring opportunities that may or *may not* be amenable to automatic refactoring.

Systematic reviews of literature: Dallal (2015) presents a systematic review of the literature regarding code refactoring. The study addresses 47 studies on the types of refactoring activities, the different approaches to identify refactoring opportunities, as well as the data sets and the means used to evaluate them. Such review was highly relevant in this article since the performed analyses brought to us the most appropriate approaches to be considered, as well as to better understand the process of identifying code refactoring opportunities, as well as their accomplishment and evaluation.

Tools for identifying refactoring opportunities: Terra et al (2018) described the proposal of JMove tool, an Eclipse IDE plug-in indicating *Move Method* refactoring opportunities based on the structural similarity coefficient *Sokal and Sneath 2*. The tool, however, involves analysis only of *Move Method*. More importantly, the use of the *PTMM*, coefficient, proposed in this article, could improve accuracy of JMove up to 14.74%.

Silva et al (2014) proposed JExtract tool, an Eclipse IDE plug-in aimed at identifying *Extract Method* refactoring opportunities based on *Kulczynski* similarity coefficient. JExtract, in turn, focuses only on the *Extract Method* refactoring. Similarly, the use of the *PTM* coefficient, proposed in this article, could improve accuracy of JExtract around 9.96%.

Tsantalis and Chatzigeorgiou (2009b,a) present the JDeodorant tool, which identifies Code Smells and applies different code refactoring techniques in order to treat them. JDeodorant does not use the similarity comparison between the dependencies of a project. On the other hand, it uses the *Jaccard* coefficient to calculate only the similarity between attributes and methods of the analyzed classes, which can affect its effectiveness since *Jaccard*—although it is one of the most used coefficients in Software Engineering—does not present good accuracy in relation to other existing coefficients.

Although the analyzed studies show a focus on the identification of code refactoring opportunities or approaches involving structural similarity coefficients, it was not found studies that propose new coefficients aiming at greater accuracy in the identification of such opportunities. Thus, the proposal of new coefficients, based on statistics, robust analyses and comparisons between the main existing similarity coefficients focused on three different types of refactoring, emphasizes the originality of this study.

9 Conclusion

Identifying code refactoring opportunities is essential for a developer to maintain a well-defined software architecture, as well as high cohesion and low coupling. Among the existing techniques, several use the structural similarity calculation. However, the main similarity coefficients proposed in literature are not designed to deal with object-oriented code structures, resulting in low precision and non-realistic rates.

In order to treat this problem, this article presents the proposal of three new structural similarity coefficients in order to provide more precise ways to identification of code refactoring opportunities. For this purpose, we analyzed the main similarity coefficients proposed in literature in a training set of 10 well-designed systems and then, we selected the most adequate coefficient to be adapted and improved. After that, we conducted an experiment with a genetic algorithm and multiple comparisons to propose new coefficients. Finally, we analyzed the proposed coefficients in another 101 well-designed systems.

This approach resulted in the following similarity coefficients: *PTMC*, for identification of *Move Class* refactoring opportunities; *PTMM*, for *Move Method*; and *PTEM*, for *Extract Method*. The proposed coefficients are more accurate than the hitherto accurate coefficients used to identify refactoring opportunities. In numerical terms, *PTMC*, *PTMM* and *PTEM* showed a statistically significant improvement from 5.23% to 6.81%, 12.33% to 14.79%, and 0.25% to 0.40%, respectively, in relation to the second best coefficients. These results can impact in many techniques that use structural similarity calculation, for example, the precision of the tools JMove and JExtract could be improved up to 14.74% and 9.96%, respectively. We also presented *AIRP*, a plug-in for IDE Eclipse to identify refactoring opportunities, that implements the proposed coefficients.

As future studies, it is intended to: (i) to use more coefficients in the analysis stage; (ii) apply cross-validation techniques; (iii) perform a comparative study among techniques in order to identify refactoring opportunities that use or not similarity coefficients; (iv) propose a single coefficient that meets other types of refactoring; and (v) measure the acceptance degree of new coefficients in real scenarios.

References

- Chis M (2010) Evolutionary Computation and Optimization Algorithms in Software Engineering: Applications and Techniques. Information Science Reference
- Dallal JA (2015) Identifying refactoring opportunities in object-oriented code: A Systematic Literature Review. Information and Software Technology 58:231–249
- Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley
- Jaccard P (1912) The distribution of the flora in the alpine zone. New Phytologist 11(2):37–50
- Naseem R, Maqbool O, Muhammad S (2010) An improved similarity measure for binary features in software clustering. In: 2nd International Conference on Computational Intelligence, Modelling and Simulation (CIMSIM), pp 111–116

- Passos L, Terra R, Diniz R, Valente MT, Mendonça N (2010) Static architecture conformance checking: An illustrative overview. *IEEE Software* 27(5):132–151
- Pinto AF, Terra R (2017) Better similarity coefficients to identify refactoring opportunities. In: 11th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), pp 1–10
- Silva D, Terra R, Valente MT (2014) Recommending automated Extract Method refactorings. In: 22nd International Conference on Program Comprehension (ICPC), pp 146–156
- Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of GitHub contributors. In: 24th International Symposium on Foundations of Software Engineering (FSE), pp 858–870
- Sivanandam S, Deepa SN (2007) Introduction to Genetic Algorithms. Springer Science & Business Media
- Szöke G, Nagy C, Hegedűs P, Ferenc R, Gyimóthy T (2015) Do automatic refactorings improve maintainability? an industrial case study. In: 31st International Conference on Software Maintenance and Evolution (ICSME), pp 429–438
- Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The Qualitas Corpus: A curated collection of Java code for empirical studies. In: 17th Asia Pacific Software Engineering Conference (APSEC), pp 336–345
- Terra R, Brunet J, Miranda L, Valente MT, Serey D, Castilho D, Bigonha RS (2013) Measuring the structural similarity between source code entities. In: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp 753–758
- Terra R, Valente MT, Miranda S, Sales V (2018) JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138:19–36
- Tsantalis N, Chatzigeorgiou A (2009a) Identification of extract method refactoring opportunities. In: 13th European Conference on Software Maintenance and Reengineering (CSMR), pp 119–128
- Tsantalis N, Chatzigeorgiou A (2009b) Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering (TSE)* 35(3):347–367
- Verzani J (2014) Using R for Introductory Statistics. Chapman and Hall/CRC, USA