Better Similarity Coefficients to Identify Refactoring Opportunities

Arthur F. Pinto
Universidade Federal de Lavras
Departamento de Ciência da Computação
UFLA, Brasil
fparthur@posgrad.ufla.br

Ricardo Terra
Universidade Federal de Lavras
Departamento de Ciência da Computação
UFLA, Brasil
terra@dcc.ufla.br

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 paper, therefore, proposes new coefficients—based on genetic algorithms over a training set of ten 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, an improvement of 10.57% for the identification of Move Method refactoring opportunities, 3.17% for Move Class, and 0.30% for Extract Method. Moreover, we implemented a tool that relies on the proposed coefficients to recommend refactoring opportunities.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; *Maintaining software*; Software evolution;

KEYWORDS

Software Architecture, Structural Similarity, Code Refactoring, Move Class, Move Method, Extract Method

ACM Reference Format:
Arthur F. Pinto and Ricardo Terra. 2017. Better Similarity Coefficients to Identify Refactoring Opportunities. In *Proceedings of SBCARS 2017, Fortaleza, CE, Brazil, September 18–19, 2017,* 10 pages.
https://doi.org/10.1145/3132498.3132511

1 INTRODUÇÃO

Durante o desenvolvimento de software, vários problemas estão sujeitos a ocorrer na arquitetura do software. *Code Smells* (também chamado de *Bad Smells*) são definidos como quaisquer sintomas no código que possam indicar um desses problemas [5]. Muitas vezes, tais *Code Smells* implicam no estabelecimento de dependências desnecessárias. No entanto, como garantir o projeto arquitetural é de suma importância para manutenibilidade, reusabilidade, escalabilidade e portabilidade de sistemas de software [9], espera-se que os elementos de código de um projeto de software estejam localizados em entidades estruturalmente similares.

Deve-se destacar que este artigo leva em consideração dependências estruturais entre os elementos de código de um sistema para o cálculo de similaridade entre pacotes, classes, métodos e blocos. Dependências estruturais ocorrem quando uma unidade de compilação depende de outra em tempo de compilação ou de vinculação [4]. Dentre os diversos tipos de dependência existentes em uma estrutura de código orientada a objetos, pode-se citar como mais relevantes: acesso de métodos e atributos (*access*), declaração de variáveis (*declare*), criação de objetos (*create*), extensão de classes (*extend*), implementação de interfaces (*implement*), ativação de exceções (*throw*) e uso de anotações (*useannotation*).

Uma estrutura de código com bons índices de similaridade influencia diretamente na qualidade de software, uma vez que evidencia o alto grau de coesão e baixo acoplamento, afetando positivamente na arquitetura e nas características de manutenção do software.

A fim de assegurar a similaridade estrutural entre as entidades de código e evitar a ocorrência de *Code Smells*, deve-se verificar a similaridade das dependências entre seus elementos (seja no nível de pacote, classe, método ou bloco) e, quando necessário, conduzir refatorações. Isso implica no deslocamento de métodos e classes, e na extração de um bloco de código de um método (gerando um novo método), por intermédio de refatorações como *Move Class*, *Move Method* e *Extract Method* [5].

A Figura 1 ilustra um exemplo de um sistema que implementa uma arquitetura MVC (*Model-View-Controller*), composta por três camadas *Modelo*, $Visão$ e *Controle*. É possível perceber que C_2 está mal localizada na camada de controle (*Controle*), pois depende de elementos gráficos enquanto demais classes da camada possuem dependências a elementos de manipulação de requisição e respostas (e.g., `HttpRequest` e `HttpResponse`). Portanto, é possível sugerir mover tal classe (*Move Class*) para uma camada estruturalmente similar que, nesse cenário, seria a camada de visão ($Visão$). Tal similaridade estrutural ocorre devido ao fato de C_2 e as classes da camada de visão ($V_1, \ldots, V_n$) possuírem dependências a elementos gráficos em comum (e.g., `JPanel` e `JLabel`). Dessa forma, a refatoração não somente garantiria uma arquitetura com entidades devidamente localizadas, como passaria a respeitar uma arquitetura MVC.

Diversos coeficientes foram propostos para o cálculo de similaridade entre entidades de código. Entretanto, muitas vezes, a utilização de tais coeficientes não garante de fato uma precisão satisfatória.

SBCARS 2017, September 18–19, 2017, Fortaleza, CE, Brazil

ACM ISBN 978-1-4503-5325-0/17/09…$15.00
https://doi.org/10.1145/3132498.3132511

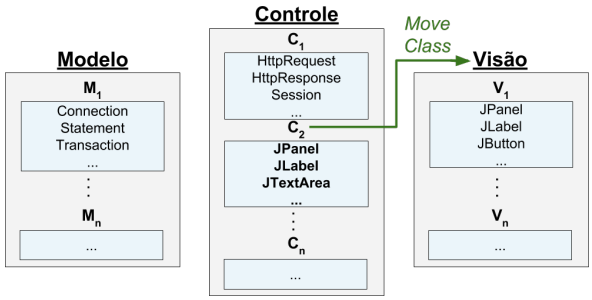

Figura 1: Exemplo de Refatoração *Move Class*

Ademais, os principais coeficientes vigentes na literatura não foram projetados para a análise estrutural de um sistema de software. Por exemplo, o coeficiente *Jaccard*, um dos mais utilizados em Engenharia de Software, foi inicialmente concebido para comparar a similaridade entre espécies florais em diferentes distritos [7].

Diante disso, este artigo tem como principal objetivo propor novos coeficientes de similaridade para uma identificação mais precisa de oportunidades de refatoração *Move Class*, *Move Method* e *Extract Method*. Isso possibilita (i) localizar, com maior precisão, entidades indevidamente posicionadas sobre a arquitetura de um sistema e (ii) alavancar a precisão de ferramentas de identificação de oportunidades de refatoração baseadas em similaridade estrutural. É importante mencionar, entretanto que os novos coeficientes são específicos a essas três refatorações, uma vez que são as mais largamente utilizadas pelos desenvolvedores [13].

Primeiramente, é analisada a precisão de 18 coeficientes de similaridade em dez sistemas (*training set*) do *Qualitas.class Corpus* [16]. Em segundo lugar, o coeficiente *Simple Matching* é adaptado por meio de algoritmos genéticos de forma a gerar três novos coeficientes (*PTMC*, *PTMM* e *PTEM*) que possuam maior precisão na identificação de, respectivamente, oportunidades de refatoração *Move Class*, *Move Method* e *Extract Method*. Em terceiro lugar, os coeficientes propostos são comparados com os existentes em outros 101 sistemas. Os resultados indicam, em relação ao melhor coeficiente analisado, uma melhoria de 10,57% para identificação de oportunidades *Move Method*, 3,17% para *Move Class* e 0,30% para *Extract Method*. Por fim, é implementada uma ferramenta que identifica oportunidades de refatoração baseadas nos coeficientes propostos.

O restante deste artigo está organizado como descrito a seguir. A Seção 2 introduz conceitos fundamentais ao estudo. A Seção 3 descreve a metodologia utilizada. A Seção 4 apresenta uma análise dos coeficientes existentes, tendo como objetivo a seleção do coeficiente a ser adaptado. A Seção 5 propõe três novos coeficientes para identificação de oportunidades de refatoração. A Seção 6 apresenta uma avaliação dos coeficientes propostos em 101 sistemas. A Seção 7 descreve a implementação de uma ferramenta para identificação de oportunidades de refatoração baseadas nos coeficientes propostos. A Seção 8 discute os trabalhos relacionados. Por fim, a Seção 9 apresenta as considerações finais, bem como trabalhos futuros.

2 BACKGROUND

No intuito de prover o conhecimento necessário para a concepção e compreensão deste artigo, são apresentados os conceitos fundamentais para o mesmo. A Seção 2.1 diz respeito ao processo de refatoração, o qual envolve métodos para reestruturação do código-fonte. A Seção 2.2 trata a respeito de similaridade estrutural, além de apresentar os principais coeficientes vigentes na literatura. Por fim, a Seção 2.3 introduz algoritmos de otimização, com foco em algoritmos genéticos, além de apresentar um exemplo ilustrativo de otimização.

2.1 Refatoração

Refatoração de código é o processo de mudança de um sistema de software de forma que preserve o comportamento externo do código e aperfeiçoe sua estrutura interna [5]. Por meio da refatoração, torna-se possível tratar a ocorrência de diferentes *Code Smells*. Embora certos processos de refatorações levam em consideração outros aspectos (e.g., semântica), este estudo foca exclusivamente em *Code Smells* que se manifestem estruturalmente no código fonte.

Diversas técnicas podem ser utilizadas para refatoração de código. Como este artigo foca sua análise exclusivamente em classes, métodos e blocos de um sistema, serão utilizadas as refatorações *Move Class*, *Move Method* e *Extract Method* a fim de tratar a ocorrência de *Code Smells*. Enquanto a aplicação de *Move Class* e *Move Method* consiste, respectivamente, na simples movimentação de uma classe para outro pacote e na movimentação de um método para outra classe, *Extract Method* é realizado por meio da extração de um bloco de código para um novo método que será gerado, substituindo o trecho extraído com uma chamada para o referente novo método.

Embora a aplicação de *Move Class* para o reposicionamento de uma classe em seu devido pacote não trate de nenhum *Code Smell* específico, é fundamental para evitar a ocorrência de qualquer *Code Smell*, visto que a indevida localização de uma classe contribuirá para que sua estrutura interna também seja inadequada, ou seja, acarretará em métodos e blocos inapropriadamente posicionados.

Já o *Move Method* possibilita tratar os seguintes *Code Smells*: *Large Class*, em que o reposicionamento dos métodos indevidos irá reduzir o tamanho da referente classe; *Divergent Change* e *Shotgun Surgery*, em que será possível posicionar os métodos que necessitem de alterações em uma classe específica que não ocasione na necessidade de alterações; *Feature Envy*, em que os métodos que fazem uso excessivo de propriedades internas de outra classe poderão ser movidos para a outra classe em questão; e *Refused Bequest*, em que os métodos da superclasse, que não possuírem propriedades em comum com todas as subclasses, poderão ser movidos para as subclasses que os utilizam.

A refatoração *Extract Method*, por sua vez, proporciona meios de se tratar de *Long Method*, onde a extração dos blocos de código em novos métodos irá reduzir o tamanho do método em questão. Ademais, ainda que *Extract Method* trate diretamente apenas de *Long Method*, indiretamente pode ser utilizado para o tratamento de outros *Code Smells* (e.g., *Large Class*, *Divergent Change*, *Shotgun Surgery*, *Feature Envy* e *Refused Bequest*), visto que após a extração do bloco de código em um novo método, esse método pode ser movido para outra classe por meio do *Move Method*. Dessa forma, é possível tratar dos referentes *Code Smells* em situações em que ocorrem em apenas determinados blocos de código.

2.2 Similaridade Estrutural

A fim de identificar a ocorrência de *Code Smells* e, consequentemente, oportunidades de refatoração de código, é fundamental a análise da similaridade estrutural das entidades de código presentes em um projeto de software. Similaridade, no contexto de arquitetura de software, refere-se à relação entre as propriedades compartilhadas entre duas ou mais entidades presentes na estrutura do código.

Para o cálculo dos índices de similaridade, diversos coeficientes foram propostos. A Tabela 1 representa os principais coeficientes de similaridade propostos na literatura [17].

Tabela 1: Coeficientes de similaridade

Coeficiente	Definição	Escala
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*]

O símbolo * indica a similaridade máxima

Para a compreensão de cada coeficiente de similaridade, considere duas entidades de código A e B. Levando em consideração que este artigo analisa dependências estruturais entre entidades de código, tem-se as seguintes variáveis:

a = quantidade de dependências em ambas entidades,

b = quantidade de dependências exclusivas da entidade A,

c = quantidade de dependências exclusivas da entidade B, e

d = quantidade do restante do universo de dependências considerado.

Com o objetivo de ilustrar o cálculo de similaridade estrutural entre duas entidades de código, o coeficiente *Jaccard* e *Simple Matching* foram aplicados em dois métodos do sistema MyAppointments, um sistema de controle e gerenciamento de compromissos [9]. Dessa forma, foram selecionados os métodos `loadAppointments`, responsável por carregar os compromissos do sistema, e `getAppointmentRowAsDate`, responsável por retornar a data do compromisso de uma linha específica do conjunto de dados. Ambos os métodos estão presentes na mesma classe de controle (`AgendaController`). O Código 1 apresenta a implementação de ambos os métodos `loadAppointments` e `getAppointmentRowAsDate`.

Com base na análise de ambos os métodos e as dependências estruturais presentes em suas estruturas, tem-se que:

- $a = 2$, visto que ambos os métodos acessam métodos de `AgendaView` e `DateUtils` (destacados pela cor vermelha);
- $b = 4$, visto que o método `loadAppointments` possui as seguintes dependências exclusivas: o lançamento de uma exceção do tipo `Exception`, a declaração de `List`, a declaração do tipo `Appointment` e o acesso aos métodos de `AgendaDAO` (destacadas pela cor azul);

```
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 }
```

Código 1: Fragmento da classe AgendaController

- $c = 1$, visto que a única dependência exclusiva do método `getAppointmentRowAsDate` é a declaração do tipo `Date` como retorno (destacada pela cor verde);
- $d = 32$, visto que ao considerar o sistema como um todo, são utilizados outros 32 diferentes tipos de dependências distintas.

Deve-se ressaltar que dependências a tipos primitivos (e.g., `int`, `char`, `byte`, etc.), a classes *Wrappers* (e.g., `Integer`, `Long`, `Character`, etc.) e ao tipo `String` são desconsideradas durante a análise, uma vez que praticamente todas as classes estabelecem dependências com esses tipos, não contribuindo para o cálculo de similaridade.

Portanto, ao se aplicar, como exemplo, o coeficiente de *Jaccard* e *Simple Matching*, encontra-se os seguintes índices de similaridade:

$$Jaccard = \frac{a}{a + b + c} = \frac{2}{2 + 4 + 1} = 0{,}28 \quad (1)$$

$$Simple\ Matching = \frac{a + d}{a + b + c + d} = \frac{2 + 32}{2 + 4 + 1 + 32} = 0{,}87 \quad (2)$$

De tal forma, é importante mencionar que somente o valor bruto não indica o nível de similaridade com exatidão, pois devem-se observar e comparar os demais valores de similaridade do sistema por completo. Por exemplo, 0,28 (*Jaccard*) pode ser considerado um alto valor de similaridade se a média de similaridade do sistema for 0,12. Da mesma forma, 0,87 (*Simple Matching*) pode ser considerado um baixo valor.

2.3 Algoritmos de Otimização

Otimização diz respeito ao processo de buscar e comparar soluções para determinado problema, maximizando e/ou minimizando os valores das variáveis que compõem a função objetivo até que seja encontrada a melhor solução possível [1]. Entretanto, em muitos casos, um problema não possui solução ótima, o que resulta na busca por soluções que atendam o objetivo desejado de forma satisfatória.

Esse conceito, por sua vez, pode ser aplicado para adaptar e melhorar os coeficientes de similaridade abordados na seção anterior.

Para a busca da solução de problemas de otimização, diversas abordagens podem ser aplicadas, envolvendo diferentes tipos de algoritmos. Dentre as diversas abordagens, a aplicação de um algoritmo genético simples mostrou ser capaz de encontrar soluções satisfatórias de forma eficaz. Portanto, tornou-se a abordagem escolhida para aplicação neste artigo.

2.3.1 Algoritmos Genéticos. Um algoritmo genético define um conjunto de candidatos à solução proposta e, a cada iteração (cada geração do algoritmo genético), seleciona e combina os candidatos mais aptos, podendo sofrer leves alterações. Assim, ao fim de toda execução, o conjunto solução será composto por candidatos com relativo grande potencial de otimizar a função objetivo [14].

Considerando o problema de otimizar os coeficientes de similaridade, o algoritmo define pesos a cada variável da fórmula do coeficiente escolhido e a cada iteração busca adaptar cada peso visando otimizar o resultado da função objetivo, selecionando por fim o conjunto dos melhores resultados. Por exemplo, considerando o coeficiente *Jaccard* (apresentado na Tabela 1), e os pesos $P_{a'}$, $P_{a''}$, P_b e P_c correspondentes, respectivamente, às variáveis a do numerador e a, b e c do denominador. Assim, tem-se como resultado:

$$Coeficiente\ Resultante\ =\ \frac{(P_{a'} * a')}{(P_{a''} * a'')\ +\ (P_b * b)\ +\ (P_c * c)} \tag{3}$$

Durante a execução de um algoritmo genético, são definidos parâmetros como função objetivo (ou função de *fitness*), população inicial, número de gerações, operador de seleção, operador de cruzamento e operador de mutação. A função objetivo representa o dado ou função que se pretende otimizar. A população inicial estipula o número inicial de possíveis candidatos ao pesos pretendidos em busca de otimizar a função objetivo. Número de gerações diz respeito a quantidade de vezes o algoritmo genético irá repetir, i.e., iterar adaptando os pesos almejados. O operador de seleção irá escolher candidatos mais aptos para a realização do cruzamento entre eles, gerando assim um ou mais candidatos que possam apresentar maior aptidão para o conjunto solução. Por fim, o(s) candidato(s) gerado(s) pode(m) sofrer uma pequena mutação, i.e., uma leve alteração em seu valor, o que previne a estagnação do mesmo, além de possibilitar que se chegue em qualquer ponto do espaço de busca.

Dentre os diferentes operadores de seleção, cruzamento e mutação, este artigo utiliza respectivamente, *Torneio Binário*, *Cruzamento Binário Simulado* e *Mutação Polinomial*. O método *Torneio Binário* seleciona, dentre todos os possíveis candidatos gerados até o momento, dois dos indivíduos que apresentam maior resultado em relação à função objetivo para que os mesmos sejam cruzados. O *Cruzamento Binário Simulado* ocorre por meio do cruzamento de dois indivíduos, combinando suas representações binárias, de forma a gerar dois novos indivíduos. Tal combinação considera uma probabilidade definida, analisando se cada índice binário deve ser combinado ou não. Por fim, a *Mutação Polinomial* também considera uma probabilidade definida a fim de alterar um ou mais índices binários dos indivíduos resultantes do cruzamento. Em ambos os operadores de cruzamento e mutação, deve ser estabelecido um índice de distribuição com o propósito de avaliar a diversidade das soluções selecionadas no espaço de busca, o que garante a seleção de indivíduos mais heterogêneos.

2.3.2 Exemplo Ilustrativo. Para melhor compreensão da abordagem de algoritmos de otimização e dos conceitos de algoritmos genéticos, esta seção apresenta a aplicação de um algoritmo genético visando otimizar o coeficiente de similaridade *Jaccard* em um pequeno exemplo de refatoração *Move Class*.

De tal modo, suponha um sistema com dois pacotes $pkg1$ e $pkg2$ tendo seu conjunto de classes e suas dependências, respectivamente, como $pkg1$ = {A={X,Y,W,Z}, B={X,Y,Z,L}, C={X,Y,W,K}} e $pkg2$ = {D={X,W,R,T}, E={R,T,Z}, F={X,Z,R,T,M}}, conforme Figura 2.

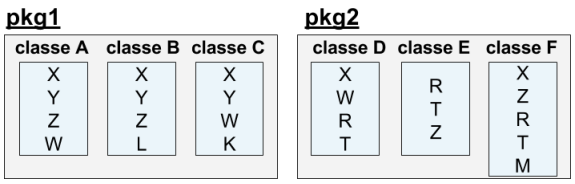

Figura 2: Exemplo Ilustrativo

Suponha que sabe-se que a arquitetura atual do sistema é a ideal e pretende-se utilizar um coeficiente de similaridade em que não haja sugestões de refatoração *Move Class*, ou seja, mantenha a arquitetura atual e garanta bons índices de similaridade entre as classes de um mesmo pacote. Para isso, pretende-se adaptar um coeficiente de forma a maximizar a similaridade de $sim(A, B)$, $sim(A, C)$, $sim(B, C)$, $sim(D, E)$, $sim(D, F)$ e $sim(E, F)$. Ao mesmo tempo, espera-se minimizar a similaridade de $sim(A, D)$, $sim(A, E)$, $sim(A, F)$, $sim(B, D)$, $sim(B, E)$, $sim(B, F)$, $sim(C, D)$, $sim(C, E)$ e $sim(C, F)$. Entretanto, somente o valor bruto não indica o nível de similaridade com exatidão, pois deve-se observar e comparar os demais valores de similaridade do sistema por completo. Dessa forma, pretende-se obter a maior diferença possível entre a média aritmética das similaridades que deseja-se maximizar e da que deseja-se minimizar.

Levando em consideração o coeficiente *Jaccard* como exemplo, decide-se aplicar o algoritmo genético definindo pesos para cada variável do coeficiente, conforme previamente descrito na Equação 3. Dessa forma, foi utilizada as configurações apresentadas na Tabela 2, a qual foi obtida após uma série de tentativas que buscavam melhorar os coeficientes, levando em consideração a disposição dos recursos computacionais e o tempo de execução.

Tabela 2: Configuração do Algoritmo Genético

Função objetivo:	Quantidade de similaridades otimizadas	**Operador de cruzamento:**	Cruzamento Binário Simulado
Tamanho da população:	1.200	**Probabilidade de mutação:**	0,6
Representação da população:	$\{x \in \mathbb{R} \mid -2 \leq x \leq 2\}$	**Probabilidade de cruzamento:**	0,9
Número de gerações:	150	**Índice de distribuição de mutação:**	20,0
Operador de seleção:	Torneio Binário	**Índice de distribuição de cruzamento:**	20,0
Operador de mutação:	Mutação Polinomial		

Nesse cenário, foi encontrado como possível conjunto solução, os pesos 6, 07 ($P_{a'}$), 0, 01 ($P_{a''}$), 9, 1 (P_b) e 9, 1 (P_c), respectivamente às variáveis a do numerador e a, b e c do denominador. O comparativo das similaridades decorrentes do coeficiente *Jaccard* e o coeficiente resultante é apresentado na Tabela 3.

Tabela 3: Comparativo de similaridade entre *Jaccard* e Coeficiente Resultante

Maximizar				
Objetivo	***Jaccard***	**Similaridade**	**Coeficiente Resultante**	**Similaridade**
sim(A,B) =	$\frac{3}{3+1+1}$ =	0,6	$\frac{6,07*3}{0,01*3+9,1*1+9,1*1}$ =	0,9989
sim(A,C) =	$\frac{3}{3+1+1}$ =	0,6	$\frac{6,07*3}{0,01*3+9,1*1+9,1*1}$ =	0,9989
sim(B,C) =	$\frac{2}{2+2+2}$ =	0,3333	$\frac{6,07*2}{0,01*2+9,1*2+9,1*2}$ =	0,3333
sim(D,E) =	$\frac{2}{2+2+1}$ =	0,4	$\frac{6,07*2}{0,01*2+9,1*2+9,1*1}$ =	0,4444
sim(D,F) =	$\frac{3}{3+1+2}$ =	0,5	$\frac{6,07*3}{0,01*3+9,1*1+9,1*2}$ =	0,6663
sim(E,F) =	$\frac{3}{3+0+2}$ =	0,6	$\frac{6,07*3}{0,01*3+9,1*0+9,1*2}$ =	0,9939
Média		**0,5055**	**Média**	**0,7393**

Minimizar				
Objetivo	***Jaccard***	**Similaridade**	**Coeficiente Resultante**	**Similaridade**
sim(A,D) =	$\frac{2}{2+2+2}$ =	0,3333	$\frac{6,07*2}{0,01*2+9,1*2+9,1*2}$ =	0,3333
sim(A,E) =	$\frac{1}{1+3+2}$ =	0,1667	$\frac{6,07*1}{0,01*1+9,1*3+9,1*2}$ =	0,1333
sim(A,F) =	$\frac{2}{2+2+3}$ =	0,2857	$\frac{6,07*2}{0,01*2+9,1*2+9,1*3}$ =	0,2667
sim(B,D) =	$\frac{2}{2+2+2}$ =	0,3333	$\frac{6,07*2}{0,01*2+9,1*2+9,1*2}$ =	0,3333
sim(B,E) =	$\frac{0}{0+4+3}$ =	0,0	$\frac{6,07*0}{0,01*0+9,1*4+9,1*3}$ =	0,0
sim(B,F) =	$\frac{1}{1+3+4}$ =	0,125	$\frac{6,07*1}{0,01*1+9,1*3+9,1*4}$ =	0,0953
sim(C,D) =	$\frac{1}{1+3+3}$ =	0,1429	$\frac{6,07*1}{0,01*1+9,1*3+9,1*3}$ =	0,1111
sim(C,E) =	$\frac{1}{1+3+2}$ =	0,1667	$\frac{6,07*1}{0,01*1+9,1*3+9,1*2}$ =	0,1334
sim(C,F) =	$\frac{2}{2+2+3}$ =	0,2857	$\frac{6,07*2}{0,01*2+9,1*2+9,1*3}$ =	0,2667
Média		**0,2044**	**Média**	**0,1859**

É possível observar que o coeficiente resultante da otimização mostrou ser muito mais eficiente que o coeficiente *Jaccard*, visto que obteve uma diferença entre a média das similaridades que deveriam ser maximizadas e a média das similaridades que deveriam ser minimizadas de 0, 55, em contrapartida ao *Jaccard* que obteve uma diferença de apenas 0, 30. Portanto, o coeficiente resultante demonstrou ser o mais adequado e preciso para a análise de similaridade no exemplo apresentado.

3 METODOLOGIA

Tendo como objetivo a criação de três novos coeficientes de similaridade que sejam mais eficientes na identificação, respectivamente, de oportunidades de refatoração *Move Class*, *Move Method* e *Extract Method*, este artigo adota uma metodologia baseada na seleção de um coeficiente para ser adaptado por meio da análise dos principais coeficientes de similaridade existentes (Seção 4), na proposta dos três novos coeficientes após a adaptação do respectivo coeficiente selecionado (Seção 5) e na avaliação dos mesmos (Seção 6). Por fim, os coeficientes propostos são aplicados por meio de uma implementação de um *plug-in* para o IDE Eclipse (Seção 7).

Para isso, o cálculo de similaridade, utilizado em cada etapa da metodologia, é realizado comparando determinada classe a determinado pacote, bem como comparando determinado método a determinada classe, além de analisar a similaridade resultante de uma determinada classe após a extração de determinado bloco para geração de um novo método. Tal processo é detalhado a seguir.

3.1 Cálculo de Similaridade

Classe-Pacote: A similaridade de uma classe com um pacote é dada por meio da média aritmética da similaridade entre a respectiva classe e as demais classes presentes no pacote. Isso ocorre devido ao fato de que caso fosse calculada apenas a similaridade da classe com o pacote de forma geral, o conjunto de dependências do pacote poderia resultar em uma elevada similaridade ainda que as classes não fossem similares, apresentando baixos índices. Consequentemente, é possível sugerir refatorações *Move Class* para pacotes que possuam de fato classes similares.

Método-Classe: A mesma abordagem utilizada para identificação de oportunidades de refatoração *Move Class* é aplicada para a análise de similaridade entre métodos e classes, em que é considerada a média da similaridade entre determinado método e os demais métodos da classe.

Bloco-Método: Inicialmente é calculada a similaridade de cada método de uma classe com os demais métodos da mesma classe, obtendo assim, as similaridades internas da classe. Em seguida, para cada método serão analisadas todas as possibilidades de extração de seus blocos a fim de gerar um novo método (*Extract Method*), ou seja, para cada bloco extraído, serão recalculadas as similaridades da classe após extração, incluindo o novo método gerado. Por fim, as médias aritméticas de ambos os conjuntos de valores são comparadas a fim de verificar se deve ou não ser sugerida a refatoração *Extract Method*.

Dessa forma, os coeficientes são analisados e avaliados utilizando os sistemas do *Qualitas.class Corpus* [16], base de dados contendo 111 sistemas orientados a objetos, mais de 18 milhões de LOC (*Lines Of Code*), 200 mil classes compiladas e 1,5 milhão de métodos compilados. Tendo como base o fato de que o *Qualitas.class Corpus* é composto por sistemas mais maduros e estáveis, assume-se que a estrutura atual dos sistemas possua o mais elevado índice de similaridade em comparação às possíveis refatorações de código. Em suma, pretende-se que a estrutura seja mantida e nenhuma refatoração seja realizada. Assim, torna-se possível avaliar a precisão de cada coeficiente de similaridade tendo em vista cada sistema analisado.

Regras de Execução: Para evitar a ocorrência de falsos positivos nas recomendações de refatoração resultantes, após uma série de testes e execuções experimentais, foram definidas nove regras de execução para a implementação da solução proposta:

(1) *A entidade sob análise é desconsiderada enquanto o sistema busca por oportunidades de refatoração.* Quando é calculada a similaridade entre uma classe A e seu respectivo pacote Pkg, o sistema considera Pkg como sendo $Pkg - \{A\}$. Nesse caso, entidades localizadas isoladamente são totalmente descartadas;

(2) *Pacotes e classes de teste não são consideradas*, visto que a maioria dos sistemas organizam suas classes de teste em um único pacote. Logo, esse pacote contém classes relacionadas a diferentes partes do sistema, isto é, não estão estruturalmente relacionados. Para tal, é utilizada uma abordagem que desconsidera pacotes e classes que contenham o texto "`test`" (caixa alta ou caixa baixa) em qualquer parte de seu nome. E, embora nem todo pacote ou classe de teste contenha o texto "`test`" e nem todo pacote ou classe que o contenha é de fato um pacote ou classe de teste, o ganho em precisão é maior do que a perda quando pacotes ou classes de teste forem erroneamente detectados;

(3) *Dependências triviais são ignoradas.* São filtradas dependências como as estabelecidas com tipos primitivos e de *wrappers* (e.g., `int` e `java.lang.Integer`), dependências de `java.lang.String` e `Java.lang.Object`, etc. Uma vez que a grande maioria dos elementos de código estabelecem dependências com esses tipos, eles não contribuem de fato para o cálculo de similaridade;

(4) *Não são analisadas entidades de classe, método ou bloco que estabelecem dependências com menos de três tipos.* Embora possam existir entidades com poucas dependências que deveriam ser refatoradas, essas entidades contém pouca informação para o cálculo de similaridade ou para fazer qualquer inferência com base nas suas dependências estruturais. Entretanto, deve-se ressaltar que, embora essas entidades não são analisadas se devem ser refatoradas, elas ainda são consideradas como possível destino de refatoração, caso possuam ao menos alguma dependência;

(5) *Não são analisadas entidades que não estejam co-localizadas com pelo menos duas outras entidades analisáveis.* Por exemplo, um pacote que possua apenas duas classes não provê informações estruturais suficientes para recomendar ou não a movimentação de suas classes. Novamente, esse critério desconsidera apenas a análise de tais entidades, mas ainda podem ser consideradas como destino de refatoração;

(6) *Não será possível extrair o primeiro bloco de um método.* A extração do primeiro bloco de um método não ocasionará uma mudança de similaridade. Uma vez que a extração de um bloco resultará na extração de seus blocos internos, tal operação apenas recriará o respectivo método. Nessa situação, o ideal seria mover o método;

(7) *Qualquer dependência presente em uma entidade interna de classe, método ou bloco será atribuída às suas entidades externas.* Tendo em vista que uma entidade de código interna (e.g., classes aninhadas, métodos e blocos internos, etc.) está presente dentro do escopo da(s) entidade(s) externa(s), qualquer dependência estabelecida deve ser atribuída às entidades externas;

(8) *Métodos e blocos pertencentes a um tipo* `Interface` *serão desconsiderados.* Devido à própria natureza de interfaces serem compostas por membros abstratos, essas não devem ser movidas ou extraídas, pois tal ação acarretaria em uma série de conflitos na estrutura do projeto; e

(9) *Métodos construtores e seus respectivos blocos serão desconsiderados.* Considerando a obrigatoriedade de construtores em uma classe, a movimentação desse tipo de método é desconsiderada, embora pretende-se avaliar a possibilidade de extração de seus blocos internos.

4 ANÁLISE E COMPARAÇÃO DE COEFICIENTES

Esta seção analisa e compara os coeficientes de similaridade existentes a fim de encontrar o coeficiente mais adequado a ser adaptado. Dessa forma, pretende-se propor novos coeficientes de similaridade que obtenham maiores taxas de precisão. Assim, são analisados 18 coeficientes de similaridade (Tabela 1) em dez sistemas selecionados de forma completamente aleatória (*training set*) do *Qualitas.class Corpus*, conforme pode ser observado na Tabela 4.

Tabela 4: Sistemas-Alvo

Sistema	Versão	Sistema	Versão
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

Todos os coeficientes de similaridade analisados foram aplicados para cada entidade dos sistemas-alvo selecionados, analisando se uma entidade possui o maior grau de similaridade encontrado com a entidade a qual deveria estar posicionada. Por exemplo, para uma classe *A*, cada coeficiente é aplicado comparando *A* com cada pacote no sistema, buscando verificar se a maior taxa de similaridade (*Top #1*) encontrada corresponde ao pacote em que *A* está, de fato, posicionada. Essa avaliação também analisa a segunda (*Top #2*) e a terceira (*Top #3*) maior taxa a fim de verificar se o coeficiente aplicado possui ao menos resultados próximos ao desejado. Por fim, é calculada a média aritmética nos 10 sistemas do *training set* considerando os *Top #1*, *Top #2* e *Top #3*.

Cada coeficiente foi avaliado separadamente em relação aos tipos de refatoração de código *Move Class*, *Move Method* e *Extract Method*. A Tabela 5 apresenta a precisão da similaridade dos coeficiente abordados em relação a identificação de oportunidades de refatoração *Move Class*, *Move Method* e *Extract Method*. Refatorações *Extract Method* levam em consideração apenas uma única taxa de acertos, visto que para refatorações *Extract Method* é analisado somente se o método deve ser extraído ou não, portanto é descartada a possibilidade acertar na segunda ou terceira tentativa (*Top #2* ou *Top #3*). Por restrições de espaço, a tabela detalhada com os resultados de cada sistema está publicamente disponível em [10].

Após a análise e comparação dos principais coeficientes existentes, *Simple Matching* foi selecionado para ser adaptado a fim de gerar um novo coeficiente. Tal escolha se deu devido ao fato de que o coeficiente *Simple Matching* possui uma estrutura fácil de ser adaptada, definindo pesos para as variáveis, o que contribui para a obtenção de bons resultados na proposta de novos coeficientes. Em contrapartida, grande parte dos coeficientes analisados já possuíam pesos definidos. Embora não tenha sido o coeficiente com melhores resultados, a possível definição de pesos do *Simple Matching* permite simular coeficientes como *Sokal and Sneath 2* que foi o melhor coeficiente em *Move Method* e segundo melhor em *Move Class*[1], bem como *Russell and Rao*, que obteve maior precisão em *Extract Method*. Ademais, o coeficiente considera o universo de dependências, ao contrário dos coeficientes *Jaccard*, *Sorenson*, *Ochiai*, *PSC*, *Dot-product* e *Sokal and Sneath 2*.

[1]PSC foi o melhor coeficiente para *Move Class*, contudo não foi selecionado por não ser linear.

Tabela 5: Precisão de similaridade dos 18 coeficientes de similaridade em relação ao *training set*

MOVE CLASS			
Coeficiente	**Top1**	**Média Top2**	**Top3**
Baroni-Urbani and Buser	27,83%	41,87%	47,63%
Dot-product	43,59%	55,9%	62,26%
Hamann	20,35%	26,13%	29,41%
Jaccard	45,66%	58,3%	64,26%
Kulczynski	40,85%	55,1%	62,16%
Ochiai	43,47%	56,58%	62,83%
Phi	43,28%	56,65%	62,64%
PSC	49,19%	62,18%	69,82%
Relative Matching	34,20%	46,73%	53,52%
Rogers and Tanimoto	20,37%	26,31%	29,56%
Russell and Rao	39,30%	53,28%	60,07%
Simple matching	20,07%	26,03%	29,39%
Sokal and Sneath	19,70%	25,65%	28,94%
Sokal and Sneath 2	47,00%	59,69%	65,51%
Sokal and Sneath 4	41,15%	55,08%	62,18%
Sokal binary distance	21,87%	28,35%	31,93%
Sorenson	43,59%	55,9%	62,26%
Yule	20,00%	29,84%	35,69%

MOVE METHOD			
Coeficiente	**Top1**	**Média Top2**	**Top3**
Baroni-Urbani and Buser	16,06%	22,39%	27,24%
Dot-product	29,09%	38,2%	44,78%
Hamann	17,86%	24,07%	28,54%
Jaccard	32,71%	42,55%	48,89%
Kulczynski	23,36%	32,61%	39,45%
Ochiai	27,26%	36,52%	43,19%
Phi	27,23%	36,52%	43,08%
PSC	33,70%	44,22%	51,16%
Relative Matching	23,36%	31,66%	36,94%
Rogers and Tanimoto	18,07%	24,25%	28,85%
Russell and Rao	30,33%	40,12%	46,29%
Simple matching	17,94%	24,15%	28,48%
Sokal and Sneath	16,88%	23,31%	27,76%
Sokal and Sneath 2	35,74%	45,65%	51,74%
Sokal and Sneath 4	23,33%	32,49%	39,32%
Sokal binary distance	21,20%	27,96%	33,25%
Sorenson	29,09%	38,2%	44,78%
Yule	9,22%	15,13%	20,1%

EXTRACT METHOD	
Coeficiente	**Média Acertos**
Baroni-Urbani and Buser	52,57%
Dot-product	58,00%
Hamann	15,59%
Jaccard	62,09%
Kulczynski	66,52%
Ochiai	62,00%
Phi	61,94%
PSC	66,18%
Relative Matching	69,17%
Rogers and Tanimoto	15,56%
Russell and Rao	74,79%
Simple matching	15,57%
Sokal and Sneath	15,47%
Sokal and Sneath 2	67,49%
Sokal and Sneath 4	66,28%
Sokal binary distance	21,75%
Sorenson	58,00%
Yule	55,44%

5 PROPOSTA DE NOVOS COEFICIENTES

Após a seleção do coeficiente *Simple Matching* para ser adaptado, é aplicado um algoritmo genético ao referente coeficiente, tendo como *training set* dez sistemas do *Qualitas.class Corpus*.

Para a execução da proposta deste artigo, foi definido como objetivo de otimização, maximizar a precisão de acerto do algoritmo selecionado. A configuração utilizada pelo algoritmo genético é a mesma aplicada no exemplo ilustrativo da Seção 2.3.2 (ver Tabela 2). É importante clarificar que cada novo coeficiente proposto é resultante de uma execução diferente do algoritmo genético.

Após gerado o conjunto de soluções, na maioria dos casos são apresentadas várias possibilidades que resultam na mesma taxa de precisão. Logo, é selecionada uma única solução que apresente maior distância entre a média dos índices de similaridade que deseja maximizar com a média do índices que deseja minimizar, visto que uma maior distância indica que o coeficiente tenderá a maximizar e minimizar as similaridades de forma correta em outros sistemas.

Assim, considerando a possibilidade de definição de pesos às variáveis do *Simple Matching*, foi simulado inicialmente como candidatos do algoritmo genético os pesos do coeficiente *Sokal and Sneath 2* para refatorações *Move Class* e *Move Method*, bem como o *Russell and Rao* para refatorações *Extract Method*, o que facilita a seleção de novos candidatos, visto que estavam entre os melhores resultados em suas respectivas refatorações. Posteriormente, o algoritmo genético foi executado sobre os dez sistemas do *training set* a fim de obter pesos mais adequados para cada variável do coeficiente e formar os novos coeficientes a serem avaliados nos demais 101 sistemas (*test set*).

Dessa forma, foram propostos os seguintes coeficientes, onde cada um corresponde a um tipo de refatoração de código: *PTMC* para operações de *Move Class*, *PTMM* para operações de *Move Method* e *PTEM* para operações de *Extract Method*. A execução do algoritmo genético sobre o *Simple Matching* no *training set* definido resultou na primeira versão dos três coeficientes almejados. Os resultantes pesos $P_{a'}$, $P_{d'}$, $P_{a''}$, P_b, P_c e $P_{d''}$, correspondentes às variáveis a e d do numerador e a, b, c e d do denominador, são reportados na Tabela 6.

Tabela 6: Pesos dos coeficientes propostos

Coeficiente	$P_{a'}$	$P_{d'}$	$P_{a''}$	P_b	P_c	$P_{d''}$
$PTMC$	5,13e-7	3,73e-7	8,71e-6	2,0	0,17	3,9e-7
$PTMM$	1,62e-10	1,15e-10	6,18e-7	1,62	8.16e-4	3,55e-10
$PTEM$	1,63	0,08	1,0	9,93	0,03	1,48

Dessa forma, ao atribuir cada peso à variável correspondente do *Simple Matching*, tem-se os seguintes coeficientes propostos:

$$PTMC = \frac{5,13e{-}7a + 3,73e{-}7d}{8,71e{-}6a + 2b + 0,17c + 3,9e{-}7d} \quad (4)$$

$$PTMM = \frac{1,62e{-}10a + 1,15e{-}10d}{6,18e{-}7a + 1,62b + 8,16e{-}4c + 3,55e{-}10d} \quad (5)$$

$$PTEM = \frac{1,63a + 0,08d}{a + 9,93b + 0,03c + 1,48d} \quad (6)$$

6 AVALIAÇÃO DOS COEFICIENTES PROPOSTOS

A fim de avaliar a eficiência dos coeficientes propostos, foi realizada uma avaliação em que a precisão dos respectivos coeficientes é comparada com outros 18 coeficientes existentes na literatura (Tabela 1), envolvendo os demais 101 sistemas da base *Qualitas.class Corpus* (*test set*). Cada coeficiente proposto é analisado e comparado de acordo com sua respectiva refatoração de código, ou seja, esta seção apresenta uma comparação diferente para *Move Class*, *Move Method* e *Extract Method*.

Para a avaliação dos resultados, foi adotada a mesma abordagem de análise descrita na Seção 4. Dessa forma, a Tabela 7 apresenta os resultados das médias das taxas de precisão dos coeficientes analisados para refatorações *Move Class* (incluindo *PTMC*), *Move Method* (incluindo *PTMM*) e *Extract Method* (incluindo *PTEM*). Novamente, por restrições de espaço, a tabela detalhada abrangendo os resultados de cada um dos 101 sistemas está publicamente disponível em [10]. Os resultados são reportados a seguir.

Move Class: Pode-se observar que o coeficiente proposto se destacou aos demais ao atingir média aproximada de 58,20%, 70,82% e 77,22% em relação às taxas de similaridade *Top #1*, *Top #2* e *Top #3*. *PSC* apresentou a segunda melhor média de precisão dentre os coeficientes analisados. Para *Top #1*, *Top #2* e *Top #3*, *PSC* obteve, respectivamente, os valores de precisão de 55,03%, 69,20% e 76,36%. Mais importante, *PTMC* mostrou ser 3,17% superior em relação ao *Top #1*, 1,62% em relação ao *Top #2* e 0,86% em relação ao *Top #3* do *PSC*. É possível observar, também acréscimo de 31,61%, 34,56% e 33,52% em relação aos valores de, respectivamente, *Top #1*, *Top #2* e *Top #3* do *Simple Matching*, coeficiente do qual *PTMC* foi adaptado.

Move Method: Pode-se observar que *PTMM* superou os demais coeficientes nos três casos de taxa de similaridade (*Top #1*, *Top #2* e *Top #3*), alcançando, respectivamente, média de aproximadamente 51,36%, 60,93% e 66,41%. O segundo melhor coeficiente foi *Sokal and Sneath 2* em que considerando os três casos de taxa de similaridade (*Top #1*, *Top #2* e *Top #3*), o coeficiente atingiu, respectivamente, os valores de precisão de 40,79%, 51,86% e 58,58%. Portanto pode-se concluir que *PTMM* foi 10,57% melhor em relação ao *Top #1*, 9,07% melhor em relação ao *Top #2* e 7,83% melhor em relação ao *Top #3*. Além disso, mais uma vez ocorreu um acréscimo significativo no coeficiente proposto em relação ao *Simple Matching*, havendo aumento de 28,44%, 30,90% e 32,21% em relação aos valores de *Top #1*, *Top #2* e *Top #3*, respectivamente.

Extract Method: Pode-se observar que *PTEM* apresentou uma média de aproximadamente 86,88%, superando o segundo melhor coeficiente (*Russell and Rao*), o qual apresentou uma média de aproximadamente 86,58%. Dessa forma, *PTEM* mostrou ser 0,30% melhor em na precisão de identificação de oportunidades de refatoração *Extract Method*. E, embora tal diferença na média seja relativamente pequena, *PTEM* não foi o melhor em apenas cinco dos 101 sistemas analisados. Essa pequena melhora é esperada devido ao fato de o cálculo da similaridade de blocos e métodos considerarem apenas os métodos da própria classe (como mencionado na Seção 3.1) o que implica em um espectro de avaliação reduzido. É relevante observar também o acréscimo de 75,05% em relação a média obtida pelo *Simple Matching*, coeficiente do qual *PTEM* foi geneticamente adaptado.

Conforme observado, os coeficientes *PTMC*, *PTMM*, e *PTEM* apresentaram melhor eficiência aos demais coeficientes, entretanto deve-se ressaltar que é possível aumentar ainda mais as precisões dos coeficientes propostos. Tal fato é possível por meio de abordagens que visam aprimorar o conjunto solução resultante do algoritmo genético. Devido ao fato de o algoritmo genético ser não-determinístico, planeja-se como trabalho futuro aplicar o Teste de Tukey [6] sobre sua execução, aplicando repetitivas execuções do algoritmo em diferentes grupos de configurações do mesmo, diminuindo assim, a taxa de erro e aumentando o intervalo de confiança.

Tabela 7: Precisão de similaridade dos 19 coeficientes de similaridade em relação ao *test set*

MOVE CLASS			
Coeficiente	**Top1**	**Top2 Média**	**Top3**
Baroni-Urbani and Buser	39,32%	52,24%	60,42%
Dot-product	50,50%	64,50%	72,29%
Hamann	26,43%	36,16%	43,43%
Jaccard	52,62%	66,53%	73,98%
Kulczynski	48,58%	63,33%	71,48%
Ochiai	50,40%	64,48%	72,32%
Phi	50,56%	64,39%	72,31%
PSC	55,03%	69,20%	76,36%
Relative Matching	39,65%	55,29%	64,70%
Rogers and Tanimoto	26,78%	36,37%	43,55%
Russell and Rao	45,11%	61,79%	70,81%
Simple matching	26,59%	36,26%	43,70%
Sokal and Sneath	26,23%	35,92%	43,09%
Sokal and Sneath 2	54,32%	68,02%	75,22%
Sokal and Sneath 4	48,80%	63,53%	71,54%
Sokal binary distance	29,39%	38,72%	46,09%
Sorenson	50,69%	64,76%	72,61%
Yule	30,79%	43,01%	51,62%
PTMC	58,20%	70,82%	77,22%

MOVE METHOD			
Coeficiente	**Top1**	**Top2 Média**	**Top3**
Baroni-Urbani and Buser	21,97%	29,98%	35,43%
Dot-product	34,24%	45,34%	52,32%
Hamann	22,96%	30,05%	34,19%
Jaccard	37,84%	49,01%	55,94%
Kulczynski	28,61%	40,16%	47,54%
Ochiai	32,61%	43,98%	51,13%
Phi	32,50%	43,92%	51,01%
PSC	38,98%	51,06%	58,31%
Relative Matching	27,59%	37,43%	43,64%
Rogers and Tanimoto	23,17%	30,31%	34,54%
Russell and Rao	34,62%	45,74%	52,49%
Simple matching	22,92%	30,03%	34,20%
Sokal and Sneath	22,32%	29,23%	33,33%
Sokal and Sneath 2	40,79%	51,86%	58,58%
Sokal and Sneath 4	28,47%	40,07%	47,54%
Sokal binary distance	26,18%	33,64%	38,21%
Sorenson	34,24%	45,34%	52,32%
Yule	14,21%	22,07%	27,83%
PTMM	51,36%	60,93%	66,41%

EXTRACT METHOD	
Coeficiente	**Acertos Média**
Baroni-Urbani and Buser	61,74%
Dot-product	69,48%
Hamann	11,86%
Jaccard	71,74%
Kulczynski	77,62%
Ochiai	73,11%
Phi	72,70%
PSC	75,35%
Relative Matching	81,51%
Rogers and Tanimoto	12,02%
Russell and Rao	86,58%
Simple matching	11,83%
Sokal and Sneath	11,66%
Sokal and Sneath 2	74,37%
Sokal and Sneath 4	76,66%
Sokal binary distance	18,03%
Sorenson	69,48%
Yule	65,50%
PTEM	86,88%

Ameaças à Validade: Embora as regras de execução foram propostas após uma série de testes e execuções experimentais, não se pode afirmar que as regras de execução descritas na Seção 3.1

são completas, visto que ainda é possível a ocorrência de falsos positivos (validade interna). Ademais, como é comum em estudos empíricos em Engenharia de Software, os resultados não podem ser extrapolados (validade externa). Embora o *test set* conte com um número razoável de 101 sistemas, é importante a avaliação dos novos coeficientes em cenários reais de desenvolvimento.

7 SISTEMA DE RECOMENDAÇÃO

Tendo como objetivo aplicar os novos coeficientes propostos neste artigo para a identificação de oportunidades de refatoração, desenvolveu-se um protótipo de um *plug-in* para o IDE Eclipse. A ferramenta, denominada *AIRP*[2], é composta por uma arquitetura baseada em quatro módulos principais:

- ***Módulo de Extração de Dependências***: Responsável por identificar e armazenar todas as dependências de uma classe, método ou bloco. Ou seja, identifica o conjunto de tipos com os quais uma entidade de código estabelece dependência estrutural. Isso inclui chamada de métodos, acesso a atributos, instanciação, declaração de variáveis, anotações, etc.;
- ***Módulo de Cálculo de Similaridade***: Realiza o cálculo da similaridade estrutural de determinada entidade de código presente no sistema-alvo. Tal cálculo é efetuado utilizando a fórmula do coeficiente de similaridade escolhido, realizando a comparação entre as dependências previamente armazenadas de determinada classe, método ou bloco com a entidade (pacote, classe ou método) na qual se encontra;
- ***Módulo de Recomendação***: Após o cálculo de similaridade, seu resultado é comparado com um índice mínimo de aceitação especificado pelo usuário. Caso o resultado seja menor que esse *threshold* definido pelo usuário, é apresentada uma lista de sugestões de possíveis refatorações *Move Class*, *Move Method* ou *Extract Method*. Em seguida, as recomendações são reportadas para a análise dos usuários, sendo ordenadas pelo seu índice de similaridade; e
- ***Módulo de Visualização***: Com o objetivo de fornecer mais detalhes a respeito da organização estrutural do sistema alvo, é gerado um grafo da arquitetura implementada. Tal grafo reportará possíveis refatorações de *Move Class*, *Move Method* e *Extract Method*, bem como a similaridade resultantes de cada refatoração. Dessa forma, torna possível uma maior compreensão por parte do usuário a respeito do processo realizado pela ferramenta, podendo observar o reposicionamento das entidades envolvidas e a arquitetura resultante de cada recomendação. A Figura 3 ilustra um protótipo do grafo de oportunidades de refatoração.

A Figura 3 ilustra um cenário hipotético onde a classe *B* possui um valor da média de similaridade 0,12 maior com as classes de *pkg*2, o *métodoA*1 possui um valor da média de similaridade 0,2 maior com os métodos da classe *C*, *métodoC*1 possui um valor de média 0,23 maior com a classe *D* e, por fim, ao ser extraído o segundo bloco de *métodoA*1, a média de similaridade do respectivo método aumentaria em 0,3. Assim, *AIRP* realiza sugestões de refatoração *Move Class* (mover a classe *B* para *pkg*2), *Move Method* (mover *métodoA*1 para a classe *C* e mover *métodoC*1 para a classe *D*) e *Extract Method* (extrair o segundo bloco de *métodoA*1 para um novo método).

[2]Publicamente disponível em: https://github.com/rterrabh/AIRP

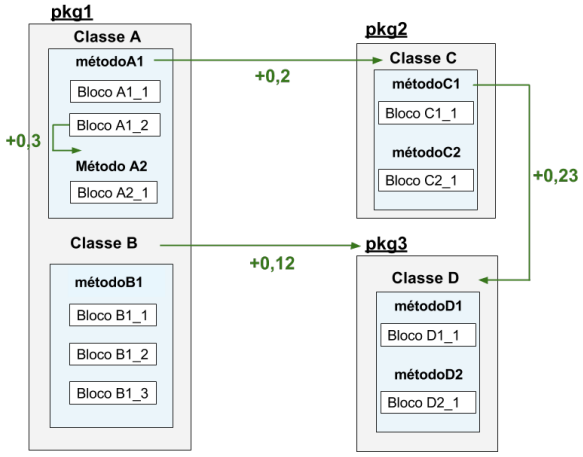

Figura 3: Grafo de oportunidades de refatoração

8 TRABALHOS RELACIONADOS

Ainda que apenas um estudo a respeito da proposta de novos coeficientes de similaridade foi encontrado, alguns trabalhos apresentam metodologias e técnicas para a identificação de oportunidades de refatoração, adaptações de métricas existentes ou estudos empíricos a respeito dos conceitos abordados neste artigo. Como o objetivo deste artigo é aprimorar a identificação de oportunidades de refatoração por meio de coeficientes de similaridade, não são considerados trabalhos que não utilizem tais coeficientes. Os trabalhos mais relevantes a este artigo são apresentados, de acordo com suas categorias, a seguir.

Proposta de Coeficientes de Similaridade Estrutural: Naseem et al. propõem um novo coeficiente de similaridade voltado para sua aplicação em algoritmos de clusterização [8]. O novo coeficiente proposto, por sua vez, é uma adaptação do coeficiente *Jaccard*, denominado *Jaccard-NM*, cuja principal diferença é a adição de uma nova variável que leva em consideração o universo de fatores analisados, i.e., todos os fatores existentes no conjunto em que as entidades analisadas estão presentes. É importante ressaltar que o novo coeficiente de similaridade proposto é comparado somente ao coeficiente *Jaccard*, do qual foi adaptado e em apenas três sistemas. Em contrapartida, este artigo conta com uma análise aprofundada dos coeficientes propostos para uma identificação mais precisa em relação aos demais coeficientes existentes.

Estudos Empíricos: Terra et al. realizam uma robusta avaliação em 111 sistemas da base *Qualitas.class Corpus*, envolvendo 18 dos principais coeficientes de similaridade [17]. Tendo em vista o propósito do estudo, o mesmo torna-se fundamental para a realização deste artigo, considerando que os coeficientes utilizados e suas análises, bem como a abordagem envolvendo dependências estruturais, atuaram como o principal embasamento para a proposta dos novos coeficientes, assim como suas avaliações.

Szõke et al. apresentam um estudo de caso onde é discutido se refatorações automáticas de código realmente melhoram a

manutenibilidade de sistemas de software [15]. Embora o estudo reporte que refatorações usualmente impactam positivamente na manutenibilidade, efetuá-las sem a devida análise pode impactar negativamente o sistema. Ainda que seja de grande importância investigar o real uso de refatorações automáticas, este artigo propõe coeficientes que permitem identificar de forma mais precisa oportunidades de refatoração que podem ou *não* ser passíveis de refatoração automática.

Revisões Sistemáticas de Literatura: Dallal apresenta uma revisão sistemática da literatura a respeito de refatoração de código [2]. O estudo aborda 47 trabalhos sobre os tipos de atividades de refatoração, as diferentes abordagens para identificar oportunidades de refatoração, bem como os *data sets* e os meios utilizados para avaliá-las. Essa revisão teve grande importância neste artigo, visto que as análises realizadas permitem detectar as abordagens mais adequadas a serem utilizadas, além de compreender melhor o processo de identificação de oportunidades de refatoração de código, assim como sua realização e sua avaliação.

Ferramentas para Identificação de Oportunidades de Refatoração: Sales et al. descreveram a proposta da ferramenta JMove, um *plug-in* do IDE Eclipse que indica oportunidades de refatoração *Move Method* baseado no coeficiente de similaridade estrutural *Sokal and Sneath 2* [11]. A ferramenta, entretanto, envolve análises apenas de *Move Method*. Mais importante, o uso do coeficiente *PTMM*, proposto neste artigo, poderia melhorar a precisão do JMove em 10,57%.

Silva et al. propuseram a ferramenta JExtract, um *plug-in* do IDE Eclipse voltado para a identificação de oportunidades de refatoração *Extract Method* baseado no coeficiente de similaridade *Kulczynski* [12]. JExtract, por sua vez, foca apenas na refatoração *Extract Method*. Similarmente, o uso do coeficiente *PTEM*, proposto neste artigo, poderia melhorar a precisão do JExtract em 9,26%.

Fokaefs et al. apresentam a ferramenta JDeodorant, que identifica *Code Smells* e aplicam diferentes técnicas de refatoração de código a fim de tratá-los [3]. JDeodorant não utiliza a comparação de similaridade entre as dependências de um projeto. Em contrapartida, utiliza o coeficiente *Jaccard* para calcular apenas a similaridade entre os atributos e métodos da classes analisadas, o que pode afetar sua eficiência uma vez que *Jaccard*—ainda que seja um dos coeficientes mais utilizados em Engenharia de Software—não apresenta boa precisão quando comparado aos demais coeficientes existentes.

Embora os trabalhos analisados demonstrem foco na identificação de oportunidades de refatoração de código ou em abordagens que envolvam coeficientes de similaridade estrutural, não foram encontrados estudos que propusessem novos coeficientes visando maior precisão na identificação de tais oportunidades. Dessa forma, a proposta de novos coeficientes, tendo, como embasamento estatístico, robustas análises e comparações entre os principais coeficientes de similaridade existentes focados em três tipos diferentes de refatoração, ressalta a originalidade deste estudo.

9 CONCLUSÃO

Este artigo apresenta a proposta de três novos coeficientes de similaridade estrutural a fim de prover meios mais precisos na identificação de oportunidades de refatoração de código: *PTMC*, para identificação de oportunidades de refatoração *Move Class*; *PTMM*, para *Move Method*; e *PTEM*, para *Extract Method*. Os coeficientes propostos apresentam melhor precisão em relação aos coeficientes até então mais precisos utilizados para identificação de oportunidades de refatoração. Em termos numéricos, *PTMC*, *PTMM* e *PTEM* apresentam melhores precisões, respectivamente, de 3,17%, 10,57% e 0,30% em relação aos segundos melhores coeficientes.

Como trabalhos futuros, pretende-se: (i) considerar pesos nos expoentes das variáveis; (ii) utilizar mais coeficientes na etapa de análise; (iii) aplicar técnicas de validação cruzada; (iv) realizar um estudo comparativo entre técnicas para identificação de oportunidades de refatoração que usem ou não coeficientes de similaridade; (v) propor um coeficiente único que atenda a outros tipos de refatoração; e (vi) medir o grau de aceitação dos novos coeficientes em cenários reais.

AGRADECIMENTOS

Este trabalho é apoiado pelo CNPq (projeto n^o 460401/2014-9), CAPES e FAPEMIG.

REFERÊNCIAS

[1] Monica Chis. 2010. *Evolutionary Computation and Optimization Algorithms in Software Engineering: Applications and Techniques.* Information Science Reference.
[2] Jehad Al Dallal. 2015. Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review. *Information and Software Technology* 58 (2015), 231–249.
[3] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Identification and Application of Extract Class Refactorings. In *33rd International Conference on Software Engineering (ICSE).* 1037–1039.
[4] Martin Fowler. 2004. *UML Distilled: a Brief Guide to the Standard Object Modeling Language.* Addison-Wesley.
[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
[6] Winston Haynes. 2013. *Tukey's Test.* Springer New York, USA. 2303–2304 pages.
[7] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New Phytologist* 11, 2 (1912), 37–50.
[8] Rashid Naseem, Onaiza Maqbool, and Siraj Muhammad. 2010. An Improved Similarity Measure for Binary Features in Software Clustering. In *2nd International Conference on Computational Intelligence, Modelling and Simulation (CIMSiM).* 111–116.
[9] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. 2010. Static Architecture Conformance Checking: An Illustrative Overview. *IEEE Software* 27, 5 (2010), 132–151.
[10] Arthur F. Pinto and Ricardo Terra. 2017. Dados das Avaliações Reportadas no Artigo Submetido ao SBCARS'17. (2017). https://github.com/rterrabh/2017_sbcars
[11] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending Move Method Refactorings Using Dependency Sets. In *20th Working Conference on Reverse Engineering (WCRE).* 232–241.
[12] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending Automated Extract Method Refactorings. In *22nd International Conference on Program Comprehension (ICPC).* 146–156.
[13] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *24th International Symposium on Foundations of Software Engineering (FSE).* 858–870.
[14] S.N. Sivanandam and S. N. Deepa. 2007. *Introduction to Genetic Algorithms.* Springer Science & Business Media.
[15] Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. 2015. Do automatic refactorings improve maintainability? An industrial case study. In *31st International Conference on Software Maintenance and Evolution (ICSME).* 429–438.
[16] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *17th Asia Pacific Software Engineering Conference (APSEC).* 336–345.
[17] Ricardo Terra, Joao Brunet, Luis Miranda, Marco Tulio Valente, Dalton Serey, Douglas Castilho, and Roberto S. Bigonha. 2013. Measuring the Structural Similarity between Source Code Entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE).* 753–758.