Identifying Utility Functions in Java and JavaScript

Tamara Mendes
Departamento de
Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
Email: tamara.mendes@dcc.ufmg.br

Marco Tulio Valente
Departamento de
Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
Email: mtov@dcc.ufmg.br

Andre Hora
Departamento de
Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
Email: hora@dcc.ufmg.br

***Abstract*—Utility functions provide generic services that can be reused in different types of systems. Theoretically, they must be implemented in specific modules. However, it is common to find such functions implemented with domain specific functions, decreasing their chances of reuse. In this paper, we propose a set of heuristics to identify utility functions. With such heuristics, recommendations can be provided to move the functions to appropriate modules. In a survey conducted with 33 developers, the proposed heuristics showed a precision of 66% and 67% when applied to Java and JavaScript systems, respectively.**

***Index Terms*—Utility functions; Refactoring; Software Architecture; Modularization; Machine Learning;**

I. Introdução

Funções utilitárias são funções de propósito geral, que podem ser reusadas em diversas partes de um sistema. São usadas para auxiliar outras funções, fornecendo alguma funcionalidade comum, como conversões, manipulação de *string*, de datas, de estruturas de dados, dentre outras. Habitualmente, funções utilitárias são definidas em bibliotecas próprias, para facilitar o reuso. No entanto, frequentemente desenvolvedores implementam funções utilitárias em módulos projetados para conter funções de propósito específico, dificultando o reuso. Esse problema pode ser visto como uma forma de *Feature Envy* — tipo de *bad smell* onde um método parece estar mais interessado em outra classe do que na própria, ou seja, as responsabilidades do método estão mais relacionadas a outra classe [1].

O Código 1 exibe uma função utilitária para verificar o tipo de um objeto do sistema MOCHA (*framework* de teste para JavaScript) em JavaScript. Como um segundo exemplo, o Código 2 apresenta um método para manipulação de *strings* do sistema JGROUPS (um *toolkit* para comunicação em grupo) em Java. Essas funções fornecem serviços com propósitos genéricos e podem ser usadas em diferentes tipos de sistema[1]. Porém, ambos os códigos foram implementados em módulos não utilitários, junto a funções de propósito específico, diminuindo suas chances de reuso.

```
1 /* mocha/lib/browser/events.js */
2 function isArray(obj) {
3     return "[object Array]" == {}.toString.call(obj);
4 }
```

Código 1. Exemplo de função utilitária em JavaScript, do sistema MOCHA, implementada fora de um módulo utilitário.

[1]Neste artigo, usa-se a palavra "função" também para denotar métodos Java visando uniformizar os termos em Java e JavaScript.

```
1  /* jgroups/src/org/jgroups/conf/XmlConfigurator.java */
2  public static String replace(String input,
3          final String expr, String replacement) {
4      StringBuilder sb=new StringBuilder();
5      int new_index=0, index=0, len=expr.length(),
6              input_len=input.length();
7
8      while(true) {
9          new_index=input.indexOf(expr, index);
10         if(new_index == -1) {
11             sb.append(input.substring(index, input_len));
12             break;
13         }
14         sb.append(input.substring(index, new_index));
15         sb.append(replacement);
16         index=new_index + len;
17     }
18
19     return sb.toString();
20 }
```

Código 2. Exemplo de método utilitário em Java, do sistema JGROUPS, implementado fora de um módulo utilitário.

Neste artigo, propõe-se um conjunto de heurísticas para identificar funções utilitárias implementadas em módulos inadequados para duas linguagens populares: Java e JavaScript. O objetivo é alertar os desenvolvedores de que essas funções podem ser movidas para módulos apropriados, isto é, para módulos utilitários. Em um trabalho anterior [2], foi investigado o uso de técnicas de aprendizado de máquina para identificar funções utilitárias. Este artigo é uma extensão dessa pesquisa com três contribuições principais: (a) apresentação de uma nova avaliação (*cross-project*) também usando aprendizado de máquina; (b) proposta de um conjunto de heurísticas para identificação de funções utilitárias; e (c) apresentação de um *survey* com desenvolvedores de software para avaliar a precisão das funções identificadas por meio das heurísticas propostas.

O restante deste artigo é organizado conforme descrito a seguir. Na Seção II, apresenta-se o conjunto de dados utilizado na pesquisa assim como um estudo exploratório realizado. Na Seção III, é discutida uma investigação da classificação de funções utilitárias usando aprendizado de máquina. A Seção IV apresenta a proposta para identificar funções utilitárias por meio de heurísticas. As avaliações dessa abordagem são discutidas nas Seções V e VI. Nesta última, apresenta-se um *survey* realizado com 33 desenvolvedores. Possíveis ameaças à validade dos resultados apresentados neste artigo são discutidas na Seção VII. Os trabalhos relacionados são discutidos na Seção VIII. Finalmente, a Seção IX conclui o artigo.

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.16

II. Dataset

O *dataset* usado nesta pesquisa é formado por 106 sistemas de código aberto e quatro sistemas proprietários em Java e JavaScript. Para automatizar a identificação de bibliotecas utilitárias nos mesmos, convencionou-se que bibliotecas utilitárias são todos os arquivos que contêm a palavra *"util"* em seu nome ou diretório, como, por exemplo, `src/util/ByteConverter.java`, `src/ng-services/UtilDOM.js` e `src/util/ArquivoUtils.java`. Os demais módulos dos sistemas foram considerados de propósito específico, ou seja, não utilitários. Adicionalmente, os sistemas do *dataset* foram filtrados de modo que cada sistema usado neste artigo contivesse, no mínimo, 25 funções em módulos utilitários.

Para Java, inicialmente, foram considerados 111 sistemas do *Qualitas.class Corpus* [3]. Desses, foram usados 84 sistemas que possuem pelo menos 25 métodos utilitários. Esse *dataset* inclui sistemas como ArgoUML, Apache Collections e Tomcat. Para JavaScript, foram considerados inicialmente 100 projetos populares do *GitHub*[2], selecionados por ordem decrescente de estrelas (medida de popularidade entre os repositórios disponíveis na plataforma). 22 desses sistemas que possuem, no mínimo, 25 funções utilitárias foram considerados. Alguns exemplos desses sistemas são three.js, Bower e Mongoose. Pelo fato de JavaScript ser uma linguagem interpretada, bibliotecas de terceiros e arquivos do próprio projeto não possuem diferenciação formal. Foi usada, então, a ferramenta *Linguist*[3], que implementa uma heurística para remover as bibliotecas de terceiros dos projetos. Essa ferramenta já foi usada com o mesmo propósito em outros trabalhos [4]. Além disso, arquivos de teste foram desconsiderados em ambas as linguagens.

O *dataset* inclui também quatro sistemas proprietários de uma empresa de desenvolvimento de software brasileira: A e B são sistemas do domínio aeroespacial, C é um sistema de gestão de ativos e D é um sistema para controle de infraestrutura elétrica[4].

A. Estudo exploratório

Foram assumidas duas hipóteses nesta pesquisa: (a) existem funções utilitárias que não são implementadas em bibliotecas utilitárias e (b) a maioria das funções implementadas em bibliotecas utilitárias é, de fato, utilitária. A primeira suposição se baseia na ideia de que o problema de pesquisa ocorre na prática, ou seja, que os desenvolvedores algumas vezes implementam funções utilitárias em módulos não utilitários. A segunda suposição pretende mostrar a viabilidade de resolver o problema de pesquisa treinando um classificador com as funções implementadas em bibliotecas utilitárias e testando sua habilidade em identificar funções similares implementadas em outros módulos.

[2]http://github.com

[3]http://github.com/github/linguist

[4]Os sistemas tiveram seus nomes reais omitidos por questões de confidencialidade.

Para verificar essas suposições, um estudo exploratório foi realizado. As funções dos quatro sistemas proprietários foram analisadas manualmente pela primeira autora deste artigo, que é especialista nos mesmos, classificando cada uma como utilitária ou não. Este estudo foi apresentado anteriormente usando os sistemas A e B [2]. Neste artigo, ele é estendido usando mais dois sistemas. A Tabela I sumariza os resultados da análise. Para cada sistema, exibe-se o número de linhas de código (LOC); o número total de funções (NF); o número de funções implementadas em bibliotecas utilitárias (NFU); o número de falsos positivos (FP), ou seja, funções implementadas em bibliotecas utilitárias que não são funções utilitárias; e o número de falsos negativos (FN), isto é, funções utilitárias não implementadas em bibliotecas utilitárias.

Tabela I
Resultados do estudo exploratório

Sistema	Linguagem	LOC	NF	NFU	FP	FN
A	JavaScript	12,212	1,334	199	11	16
B	Java	60,184	6,905	388	17	14
C	Java	38,015	7,371	70	16	46
D	JavaScript	8,827	298	78	25	2

Respectivamente para os sistemas A, B, C e D, temos 16, 14, 46 e 2 funções utilitárias que *não* foram implementadas em bibliotecas utilitárias (coluna FN). Esses valores correspondem entre 0.2% e 1.4% das funções em módulos não utilitários, confirmando assim a suposição (a). Os dados também mostram que a maioria das funções em bibliotecas utilitárias é utilitária de fato, já que, respectivamente nos sistemas A, B, C e D, 94%, 95%, 77% e 67% das funções em módulos utilitários são realmente utilitárias (colunas NFU e FP). Apenas o sistema D apresentou muitos falsos positivos, ou seja, funções em módulos utilitários que não são utilitárias. Ainda assim, considerou-se verdadeira a suposição (b), pois o resultado obtido foi maior que 50%.

III. Predição Cross-Project Usando Aprendizado de Máquina

Uma primeira investigação da identificação de funções utilitárias usando aprendizado de máquina foi apresentada em um artigo resumido publicado recentemente [2]. No entanto, nesse artigo, foram apresentados resultados apenas para avaliações *intra-project*, isto é, o treinamento e a classificação das funções foram feitos por sistema. No presente artigo, esse estudo inicial é estendido, considerando agora uma avaliação *cross-project*, ou seja, funções de diferentes sistemas são usadas nas fases de treinamento e classificação do algoritmo de aprendizado de máquina. Considera-se essa nova avaliação mais próxima de um cenário prático de uso, pois um classificador gerado previamente, treinado com alguns sistemas, pode ser utilizado em outros sistemas sem a necessidade de realizar o treinamento em cada um deles. Além disso, nem todos sistemas possuem módulos utilitários de acordo com o critério adotado neste trabalho (apresentado na Seção II), o que inviabiliza o treinamento.

A. Algoritmo e Preditores

Assim como em [2], neste trabalho, o algoritmo de aprendizado de máquina adotado foi o *Random Forests* [5], pois ele é robusto a ruídos e flexível quanto ao número e tipos de preditores de entrada. Além disso, ele é largamente usado em outros trabalhos na área de Engenharia de Software [6]–[8]. Usou-se como entrada para o algoritmo uma lista de preditores baseados em métricas estáticas de código calculadas em nível de método (Java) e de função (JavaScript). Os preditores se baseiam em características intuitivas de funções utilitárias, como, por exemplo, o uso da palavra-chave *this*, o número de referências a variáveis globais e a dependência de outras funções que não sejam nativas da linguagem. Além dessas características, os preditores foram selecionados para incluir métricas básicas como: número de linhas de código e complexidade ciclomática; dados fornecidos diretamente pela AST (*Abstract Syntax Tree*), como número de parâmetros e uso do modificador *static*; e métricas de acoplamento.

As funções dos sistemas do *dataset* foram classificadas pelo *Random Forests* como utilitárias ou não, usando mais de um sistema em cada classificação. O algoritmo foi executado usando o pacote *randomForest* da ferramenta R [9] e configurado com o mesmo conjunto de preditores usado em [2] — 23 preditores para os sistemas em Java e 20 preditores para os sistemas em JavaScript.

B. Avaliação nos Sistemas Proprietários

Nos sistemas proprietários, o classificador foi treinado em um sistema e testado em outro, simulando a aplicação real do mesmo. Para Java, os resultados do sistema B foram obtidos treinando o classificador com o sistema C e vice-versa. De forma análoga, os resultados do sistema A foram obtidos treinando o classificador com o sistema D e vice-versa, ambos em JavaScript. As fases de treinamento e teste foram baseadas na classificação manual das funções feita por um especialista descrita na Subseção II-A. A Tabela II mostra os valores de verdadeiro positivo (VP), falso positivo (FP), verdadeiro negativo (VN), falso negativo (FN), precisão ($\frac{VP}{VP+FP}$) e *recall* ($\frac{VP}{VP+FN}$) para os quatro sistemas.

Tabela II
RESULTADOS DA VALIDAÇÃO CROSS-PROJECT NOS SISTEMAS PROPRIETÁRIOS

Sistema	VP	FP	VN	FN	Precisão	*Recall*
A	1	29	1101	203	0.03	0.004
B	0	5	6,515	385	0	0
C	0	5	7,266	100	0	0
D	0	6	237	55	0	0

O sistema A obteve precisão de 0.03 e *recall* de 0.004, com apenas um verdadeiro positivo. Os demais sistemas tiveram precisão e *recall* nulos. Esses resultados mostram que o *Random Forests* não apresentou um bom desempenho na validação *cross-project* quando aplicado nos sistemas apresentados. Isso indica que o algoritmo não foi capaz de generalizar a identificação de funções utilitárias, o que prejudica sua aplicação prática.

C. Avaliação nos Sistemas de Código Aberto

Na validação *cross-project* dos sistemas de código aberto, todas as funções de todos os sistemas foram adicionadas à entrada do *Random Forests*. Dessa forma, o classificador é treinado e testado com funções de vários projetos ao mesmo tempo. Basicamente, os dados utilizados se dividem em dois conjuntos: (a) funções implementadas em módulos utilitários e (b) funções implementadas fora dos mesmos. Esse foi o referencial usado para treinar e testar o classificador. Por esse motivo, normalmente existe ruído na entrada. De um lado, podem existir funções que não são utilitárias no conjunto (a); por outro lado, no conjunto (b), podem existir funções utilitárias. No entanto, como mostrado no estudo exploratório (Subseção II-A), ambos os casos normalmente não dominam os conjuntos. Além disso, o classificador *Random Forests* tolera certo nível de ruído nos dados [5].

Foi executada a *10-fold-cross validation*, que consiste em dividir aleatoriamente os exemplos da entrada do classificador em dez grupos (*folds*) mutuamente exclusivos, usando nove deles para treinamento e um para teste. Esse processo é repetido dez vezes de modo que todos os *folds* sejam usados no conjunto de teste. Obtém-se como resultado as medidas de AUC (*Area Under ROC Curve*), precisão e *recall*. AUC é a medida que se obtém calculando a área abaixo da curva ROC (*Receiver Operating Characteristic*). Ela revela a habilidade do algoritmo em classificar corretamente sua entrada, sendo que seus valores variam de 0 a 1.

A Tabela III exibe os resultados obtidos. Para os dados dos 84 sistemas em Java, a precisão foi de 0.39, o *recall* foi de 0.34 e o AUC foi de 0.74. Para os 22 sistemas em JavaScript, a precisão foi de 0.23, o *recall* foi de 0.25 e o valor do AUC foi de 0.70.

Tabela III
RESULTADOS DA AVALIAÇÃO CROSS-PROJECT NOS SISTEMAS DE CÓDIGO ABERTO

Linguagem	Precisão	*Recall*	AUC
Java	0.39	0.34	0.74
JavaScript	0.23	0.25	0.70

A Figura 1 ilustra a diferença dos resultados entre a abordagem por projeto, com entrada balanceada, conforme apresentada anteriormente [2] e a validação *cross-project*, que envolve vários projetos. Apesar dos valores de AUC serem próximos, a precisão e *recall* são consideravelmente inferiores na abordagem *cross-project*. Uma possível razão para tal diferença é que as características das funções utilitárias podem mudar de acordo com o sistema. Logo, o treinamento realizado com base nessas propriedades deixa de ser efetivo para classificar funções de outro sistema.

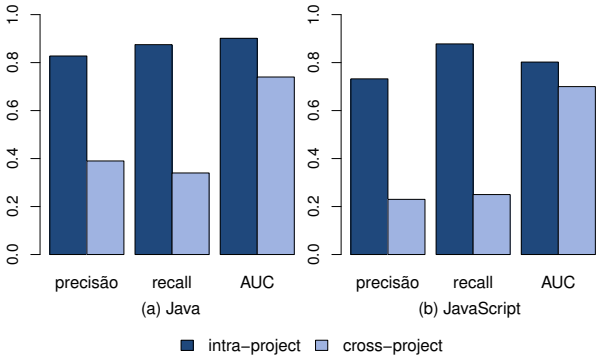

Figura 1. Resultados da validação *cross-project* comparados às avaliações por projeto.

Adicionalmente, foi calculada a importância dos preditores usando o índice *Gini* [10], obtido por meio da função *importance* do pacote *randomForest* em R [9]. Para Java, o preditor mais significativo foi LOC. Na sequência, os melhores preditores foram o método ser estático ou não, o número de acessos a campos da classe, a complexidade ciclomática e o número de parâmetros. Para os sistemas em JavaScript, o preditor mais significativo também foi LOC. Na sequência das primeiras posições, os melhores preditores foram o número de chamadas de função, o número de chamadas de função via propriedade, o número de referências a variáveis globais e a complexidade ciclomática.

Sumário: os resultados mostram que o algoritmo não foi capaz de generalizar a classificação de funções para diferentes sistemas, mostrando que uma solução usando *Random Forests* não apresenta resultados excelentes para identificar funções utilitárias definidas em módulos inapropriados. Apesar de obter resultados muito bons na identificação de funções nos módulos utilitários de cada sistema, com treinamento balanceado, como apresentado em [2], esses resultados não se mantêm em uma validação *cross-project*, a qual é mais próxima de um cenário de aplicação prática.

IV. Heurísticas

Em função dos resultados apresentados na seção anterior, decidiu-se investigar e propor heurísticas baseadas em um conjunto de características intuitivamente inerentes a funções utilitárias e que podem ser computadas por meio de análise estática. Esse conjunto foi levantado e aperfeiçoado observando os erros e acertos ao investigar tais características em alguns sistemas do *dataset*. A escolha das propriedades se fundamenta na ideia de que funções utilitárias tendem a ter um baixo acoplamento com as demais funções e não dependem de outros módulos ou funções do sistema. Normalmente, elas apenas processam e retornam algum dado. Elas têm propósito genérico e não implementam regras de negócio. Também *não* são implementadas usando determinados padrões e mecanismos como herança, por exemplo.

As características de funções utilitárias variam de acordo com o sistema. No entanto, existem algumas convenções gerais da comunidade de software. De acordo com algumas definições propostas na comunidade de programadores *Stack Overflow*[5], normalmente são funções globais que não precisam ser encapsuladas em objetos. Por esse motivo, em Java, costumam ser implementadas em métodos estáticos, sem a necessidade de instanciar uma classe [11]. Normalmente são funções com retorno e todos — ou quase todos — os objetos necessários são passados via parâmetro, portanto, não há necessidade de acessar variáveis globais e funções externas, na maioria das vezes. Quando acessam, geralmente são constantes declaradas no próprio módulo utilitário ou funções de bibliotecas externas. Inclusive, é comum que essas funções sejam usadas para encapsular funcionalidades de bibliotecas externas. Funções utilitárias também não devem implementar regras de negócio do sistema, pois a ideia é que elas disponibilizem serviços genéricos, que possam ser reusados em outros sistemas.

As heurísticas propostas nesta seção foram implementadas usando o *parser* do *Eclipse JDT*[6] para Java e o *Esprima*[7] para JavaScript, ferramentas responsáveis pela análise estática do código considerado. Depois de coletadas, as métricas são usadas em estruturas condicionais que definem se uma função é utilitária ou não. Essas regras são listadas a seguir, assim como alguns exemplos extraídos dos sistemas do *dataset* (Seção II).

A. Heurísticas para Java

Um método utilitário em Java *não* deve possuir as seguintes características:

(a) possuir corpo vazio;
(b) ser um método `get/set` usado apenas para recuperar e alterar atributos de uma classe, identificados por meio da seguinte heurística: seu nome se inicia com *"set"* e recebe um parâmetro, ou seu nome se inicia com *"get"*, não possui parâmetros e não possui tipo de retorno;
(c) possuir retorno do tipo `void`, pois se espera que um método utilitário retorne algo;
(d) apenas retornar o literal `null`, ou um literal booleano, numérico, *string*, ou um único caractere (um exemplo é mostrado no Código 3);
(e) apenas lançar uma exceção (um exemplo é mostrado no Código 4);
(f) ser abstrato;
(g) sobrescrever um método;
(h) ser implementado em uma subclasse;
(i) fazer referência à palavra-chave `this`, já que esta é usada normalmente para acessar atributos e métodos da própria classe e, idealmente, uma função utilitária não deve usar elementos externos;
(j) fazer referência a campos da classe;

[5]http://stackoverflow.com/questions/25060976
[6]http://www.eclipse.org/jdt
[7]http://esprima.org

(k) lançar uma exceção[8];
(l) possuir modificador diferente de `public`;
(m) instanciar classes do sistema;
(n) fazer chamadas a métodos do sistema.

```
1 /* apache/tools/ant/ProjectHelper.java */
2 protected Object initialValue(){
3   return ".";
4 }
```

Código 3. Exemplo de método não utilitário do sistema APACHE COLLECTIONS que apenas retorna um literal do tipo *string*.

```
1 /* argouml/kernel/MemberList.java */
2 public boolean containsAll(Collection<?> arg0){
3   throw new UnsupportedOperationException();
4 }
```

Código 4. Exemplo de método não utilitário do sistema ARGOUML que apenas lança uma exceção.

B. Heurísticas para JavaScript

Uma função utilitária em JavaScript *não* deve possuir as seguintes características:

(a) ser anônima (um exemplo é mostrado no Código 5);
(b) ser aninhada a funções não anônimas (um exemplo é mostrado no Código 6);
(c) ser uma função `get/set`, ou seja, seu nome se inicia com *"get"* ou *"set"*;
(d) possuir o corpo vazio;
(e) apenas retornar o literal `null` ou um literal booleano;
(f) fazer referência à palavra-chave `this`, pois com isso, subentende-se a existência de um construtor para o objeto e, normalmente, uma função utilitária não precisa de construtor;
(g) acessar elementos específicos do DOM (*Document Object Model*) usando o comando `document.getElementById("idDefinido")` ou uma seleção *jQuery*[9] do tipo `$("#idDefinido")`, passando como parâmetro uma *string* literal e não uma variável (um exemplo é mostrado no Código 7);
(h) usar variáveis globais não nativas da linguagem;
(i) fazer chamadas para outras funções, exceto nos seguintes casos: as funções são nativas da linguagem (por exemplo, `Math.sin()` e `parseInt()`); ou são locais (um exemplo é mostrado no Código 6); ou são chamadas via propriedade (um exemplo é mostrado no Código 8); ou são utilitárias, ou seja, obedecem a todos os critérios acima.

```
1 /* ace/lib/ace/ace.js */
2 editor.on("destroy", function() {
3     event.removeListener(window, "resize", env.onResize);
4     env.editor.container.env = null;
5 });
```

Código 5. Exemplo de função não utilitária e anônima do sistema ACE passada como parâmetro na chamada da função `on`.

[8]Apesar dessa característica sobrescrever a definida em (e), quando desconsiderada em uma possível configuração da ferramenta, produz resultados diferentes.

[9]http://jquery.com

```
1  /* brackets/src/editor/EditorManager.js */
2  function _handleRemoveFromPaneView(e, removedFiles) {
3      var handleFileRemoved = function (file) {
4          var doc = DocumentManager
5                  .getOpenDocumentForPath(file.fullPath);
6          if (doc) {
7              MainViewManager
8                      ._destroyEditorIfNotNeeded(doc);
9          }
10     };
11     if ($.isArray(removedFiles)) {
12         removedFiles.forEach(function (removedFile) {
13             handleFileRemoved(removedFile);
14         });
15     } else {
16         handleFileRemoved(removedFiles);
17     }
18 }
```

Código 6. Exemplo de função não utilitária pois é aninhada e chamada localmente no sistema BRACKETS. A função local `handleFileRemoved` (linha 3) encontra-se aninhada na função `_handleRemoveFromPaneView` (linha 2) e é chamada localmente em dois pontos (linhas 11 e 14).

```
1 /* ace/lib/ace/ext/settings_menu.js */
2 function showSettingsMenu(editor) {
3     var sm = document.getElementById("ace_settingsmenu");
4     if (!sm)
5     overlayPage(editor,
6             generateSettingsMenu(editor), "0", "0", "0");
7 }
```

Código 7. Exemplo de função não utilitária do sistema ACE que acessa o DOM com um literal *string* passado via parâmetro (linha 3).

```
1 /* leaflet/src/layer/tile/GridLayer.js */
2 createTile: function () {
3     return document.createElement("div");
4 }
```

Código 8. Exemplo de função não utilitária do sistema LEAFLET que faz uma chamada de função via propriedade. O objeto é `document` e a função é `createElement`.

As restrições apresentadas foram definidas para evitar falsas recomendações. No entanto, alguns sistemas apresentam melhores resultados quando determinadas restrições são desabilitadas. Isso porque as principais propriedades de funções utilitárias podem mudar significativamente de um projeto para outro. Por exemplo, cada sistema depende de diferentes bibliotecas; em alguns projetos Java, as funções utilitárias podem ser estáticas, mas em outros, não; os sistemas em JavaScript seguem paradigmas de modularização diferentes, desde programação procedural até orientação a objeto baseada em classes [12]. Por essa razão, o ideal é que um recomendador de funções utilitárias seja configurável, permitindo desabilitar algumas regras para melhor se adaptar ao contexto de um determinado sistema.

V. AVALIAÇÃO

As heurísticas apresentadas foram implementadas em um protótipo de ferramenta para identificação de funções utilitárias. Esse protótipo foi aplicado aos sistemas do *dataset* e os resultados obtidos são discutidos a seguir.

A. Avaliação nos Sistemas Proprietários

O protótipo de ferramenta que usa as heurísticas apresentadas foi aplicado com a configuração padrão (ou seja, incluindo todas as restrições) nos sistemas A e B. No sistema C ele foi configurado desabilitando a restrição de não lançar exceção,

pois muitos métodos dos módulos utilitários desse sistema possuem tal comportamento. O sistema D, em JavaScript, possui a característica de definir os módulos dentro de funções, logo, a restrição que desconsidera funções aninhadas impacta significativamente os resultados. Por esse motivo, a restrição de não ser aninhada à função não anônima foi desabilitada na configuração desse sistema. Os resultados obtidos são mostrados nas Tabelas IV e V.

Tabela IV
RESULTADO DA IDENTIFICAÇÃO DE FUNÇÕES UTILITÁRIAS EM MÓDULOS NÃO UTILITÁRIOS

Sistema	VP	FP	VN	FN	Precisão	Recall
A (JavaScript)	14	31	1088	2	0.31	0.87
B (Java)	4	4	6499	10	0.50	0.28
C (Java)	5	6	7249	41	0.45	0.11
D (JavaScript)	1	24	194	1	0.04	0.50

A Tabela IV mostra os resultados relativos à identificação de funções utilitárias fora dos módulos utilitários. Já a Tabela V mostra os resultados da identificação de todas as funções utilitárias do sistema, incluindo aquelas definidas em módulos utilitários (isto é, as heurísticas propostas foram aplicadas também na identificação de funções implementadas em módulos utilitários). Reporta-se o número de verdadeiros positivos (VP), falsos positivos (FP), verdadeiros negativos (VN), falsos negativos (FN), os valores de precisão e de *recall*.

Tabela V
RESULTADO DA IDENTIFICAÇÃO DE FUNÇÕES UTILITÁRIAS

Sistema	VP	FP	VN	FN	Precisão	Recall
A (JavaScript)	56	36	1094	148	0.60	0.27
B (Java)	64	4	6516	321	0.94	0.17
C (Java)	16	6	7265	84	0.72	0.16
D (JavaScript)	31	32	211	24	0.49	0.56

Na identificação de funções utilitárias em módulos não utilitários (Tabela IV), a melhor precisão foi de 0.50 no sistema B e a pior foi de 0.04 no sistema D. Já o melhor *recall* foi de 0.87 no sistema A e o mais baixo foi de 0.11 no sistema C. Na identificação das funções incluindo os módulos utilitários (Tabela V), a melhor precisão obtida foi de 0.94 no sistema B e a pior foi de 0.49 no sistema D. O melhor *recall* foi de 0.56 para o sistema D e o pior foi obtido no sistema C, no valor de 0.16. Portanto, o *recall* foi melhor nos sistemas em JavaScript. Além disso, incluindo todas as heurísticas na identificação em todos os módulos, o sistema C obteve precisão de 0.66 e *recall* de 0.11, e o sistema D apresentou precisão e *recall* nulos.

Apesar da acurácia na identificação de funções utilitárias em módulos de propósito específico ser inferior àquela que inclui os módulos utilitários, ambos os resultados são superiores aos obtidos usando aprendizado de máquina. A Figura 2 apresenta uma comparação entre os resultados obtidos nos sistemas proprietários usando as heurísticas propostas e usando um classificador *Random Forests* (validação *cross-project*). Esses valores se referem à identificação de funções utilitárias em todos os módulos dos sistemas. Observa-se que as heurísticas propostas apresentam melhores resultados em todos os quatro sistemas analisados.

Sumário: os resultados obtidos com as heurísticas propostas são superiores aos obtidos com a abordagem de aprendizado de máquina (Seção III). Apesar do *recall* baixo, o conjunto de heurísticas mostrou-se capaz de identificar funções utilitárias com uma precisão mínima de 49% e máxima de 94% (quando consideradas todas as funções de um sistema).

VI. SURVEY COM DESENVOLVEDORES

As funções encontradas por meio das heurísticas em módulos não utilitários são aquelas cuja movimentação deve ser sugerida para módulos utilitários (exemplos dessas funções são apresentados nos Códigos 2 e 1). Para contabilizar o total de recomendações corretas seria necessária uma análise manual para verificar se cada função é ou não utilitária. Devido às dificuldades de acesso a um especialista dos sistemas de código aberto considerados e ao grande número de sistemas e funções, essa análise não é trivial de ser realizada. Assim, para se obter uma estimativa da precisão das recomendações nos

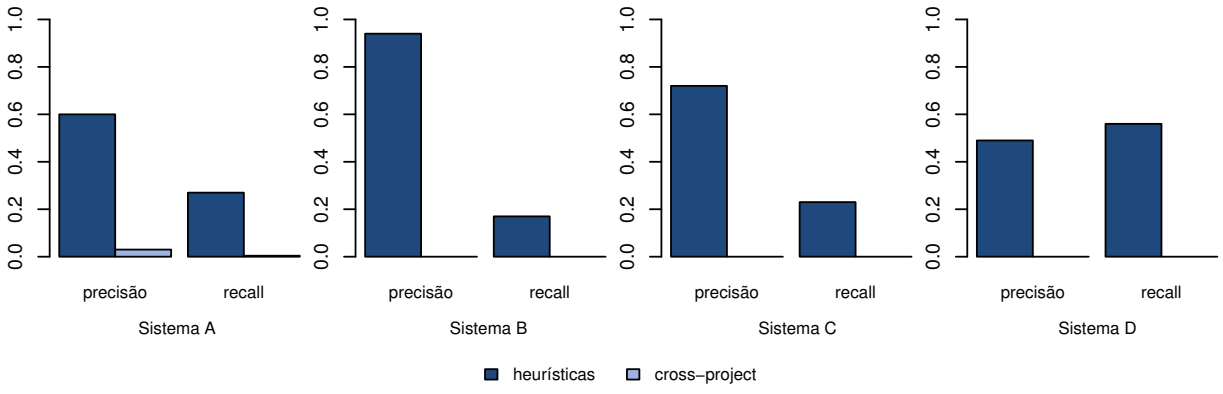

Figura 2. Resultados da avaliação das heurísticas comparados à validação *cross-project* nos sistemas proprietários.

sistemas de código aberto do *dataset* considerado, foi realizado um *survey*. Nele, parte das funções identificadas em módulos não utilitários foi julgada por desenvolvedores de software como utilitária ou não.

O *survey* foi realizado por meio de formulários do *Google Forms* enviados por e-mail. Em cada formulário, foram inseridas dez funções para que o participante as julgasse usando uma escala de zero a três, onde zero significa que a função certamente não é utilitária e três significa que certamente é utilitária. O número par de respostas foi usado para forçar o participante a definir o tipo da função. Assim, respostas com opção zero ou um foram contabilizadas como funções não utilitárias, e respostas com opção dois ou três foram consideradas como utilitárias. As escalas intermediárias foram usadas porque as funções não são avaliadas pelos especialistas dos sistemas e há certa subjetividade ao determinar se uma função é utilitária. Cada pergunta foi acompanhada do código fonte da função, o nome do arquivo, a linha onde a mesma foi implementada e um link para o arquivo.

Das dez funções de cada questionário, duas são *pontos de controle* para validar a resposta do participante. Elas foram escolhidas manualmente do *dataset* e são claramente de domínio específico, ou seja, dependem de entidades relacionadas ao domínio do sistema e implementam regras de negócio específicas. O Código 9 mostra um exemplo desse tipo de método usado nos formulários. Se o participante classificou uma dessas duas funções como utilitária, sua resposta foi descartada e o formulário foi reenviado a outro desenvolvedor. As demais oito funções foram escolhidas aleatoriamente, sem repetição, dentre as funções identificadas por meio das heurísticas.

```
1  /* compiere/base/src/org/compiere/model/MBankStatement.
       java */
2  @UICallout public void setC_BankAccount_ID(String
       oldC_BankAccount_ID, String newC_BankAccount_ID,
3                    int windowNo) throws Exception
4  {
5      if ((newC_BankAccount_ID == null) ||
6              (newC_BankAccount_ID.length() == 0))
7        return;
8      int C_BankAccount_ID = Integer.parseInt(
           newC_BankAccount_ID);
9      if (C_BankAccount_ID == 0)
10       return;
11     setC_BankAccount_ID(C_BankAccount_ID);
12     //
13     MBankAccount ba = getBankAccount();
14     setBeginningBalance(ba.getCurrentBalance());
15 }
```

Código 9. Exemplo de método claramente não utilitário usado nos formulários como ponto de controle.

O formulário também possui perguntas para que o participante informe seu nível de experiência em desenvolvimento de software e nas linguagens Java ou JavaScript. Em seu cabeçalho foi dada uma breve definição sobre funções utilitárias. Informou-se também o tempo estimado para respondê-lo (de 10 a 15 minutos). Foi ressaltada a importância de se consultar o arquivo onde a função foi implementada para entender o contexto em que a mesma se encontra.

A Tabela VI apresenta o número de funções identificadas por meio das heurísticas e o número de funções usadas nos formulários. Dentre as recomendações, 144 métodos Java escolhidos aleatoriamente foram distribuídos em 18 formulários, assim como 120 funções JavaScript foram divididas em outros 15 formulários. Isso correspondeu a 3.85% das recomendações para sistemas Java e 8.52% das recomendações em JavaScript. Os formulários foram enviados para desenvolvedores com conhecimento nas respectivas linguagens. Uma dada função foi enviada para um único participante, de forma que um número maior de funções fosse avaliado no *survey*. Logo, cada formulário foi reenviado a outra pessoa apenas em dois casos: (a) se o participante não respondeu dentro do prazo de aproximadamente uma semana, ou (b) se errou qualquer uma das duas funções de controle.

Tabela VI
NÚMERO DE FUNÇÕES IDENTIFICADAS POR MEIO DAS HEURÍSTICAS PROPOSTAS E DE FUNÇÕES AVALIADAS NOS FORMULÁRIOS

Linguagem	F. Identificadas	Formulários	F. avaliadas
Java	3733	18	144 (3.85%)
JavaScript	1407	15	120 (8.52%)

A. Resultados

A Tabela VII exibe o número total de formulários, o número de participantes para quem eles foram enviados, quantas respostas foram obtidas e quantas respostas tiveram erro nas funções de controle, ou seja, pelo menos uma das funções claramente não utilitárias recebeu resposta dois (possivelmente utilitária) ou três (certamente utilitária).

Tabela VII
NÚMERO DE FORMULÁRIOS ENVIADOS, RESPONDIDOS E DE RESPOSTAS COM ERRO NAS FUNÇÕES DE CONTROLE

Linguagem	Formulários	Envios	Respostas	Erros
Java	18	25	20	2
JavaScript	15	21	18	3

Para Java, 20 dos 25 desenvolvedores para quem os formulários foram enviados responderam, o que corresponde a uma taxa de resposta de 80%. Dois participantes erraram um dos métodos de controle. Em três formulários não houve resposta no primeiro envio. Em dois formulários os participantes erraram algum ponto de controle no primeiro envio e não houve resposta no segundo envio. Os formulários em JavaScript foram enviados para 21 desenvolvedores; destes, 18 responderam (taxa de resposta de 85%) e 3 erraram algum ponto de controle. Para dois formulários os participantes erraram algum ponto de controle no primeiro envio. Em um formulário não houve resposta no primeiro envio. Em um formulário o participante errou algum ponto de controle no primeiro envio e não houve resposta no segundo e terceiro envios.

As Figuras 3 e 4 apresentam a experiência dos participantes do *survey*. Neles foram contabilizados apenas os participantes com respostas válidas, ou seja, excluem-se aqueles que erraram as funções de controle.

Figura 3. Experiência dos participantes em desenvolvimento de software.

Experiência dos participantes em desenvolvimento de software: o nível de experiência dos participantes em desenvolvimento de software, informado pelos mesmos, é exibido na Figura 3. 18 participantes possuem mais de quatro anos de experiência, correspondendo a 54% dos participantes. Doze desenvolvedores possuem entre dois e quatro anos de experiência (36%), dois participantes possuem de um a dois anos e apenas um possui menos de um ano de experiência. Ou seja, 90% dos participantes possuem pelo menos dois anos de experiência em desenvolvimento de software.

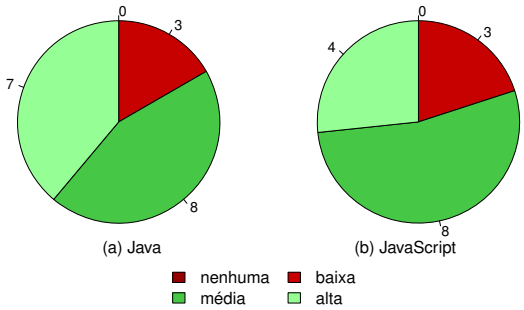

Figura 4. Experiência dos participantes nas linguagens Java e JavaScript.

Experiência dos participantes em Java e JavaScript: a Figura 4 exibe o nível de experiência dos participantes respectivamente em Java e JavaScript. Para Java, sete participantes informaram ter alto domínio em Java, o que corresponde a 38% dos participantes. Outros oito participantes (44%) declararam ter experiência média e três possuem baixa experiência. Nos formulários para JavaScript, oito participantes declararam ter média experiência, correspondendo a 53% dos participantes. Outros quatro participantes têm experiência alta e três possuem baixo domínio. Esses resultados mostram que a grande maioria dos participantes possui boa experiência em desenvolvimento de software e em ambas as linguagens, o que é importante para obter respostas mais confiáveis.

Funções de controle: a Tabela VIII apresenta as respostas obtidas nas funções de controle, aquelas claramente de domínio específico, utilizadas para validar as respostas. Para Java, no método de controle I, 16 participantes marcaram a opção zero (certamente não utilitário), correspondendo a 80% das respostas, dois participantes marcaram a opção 1 (possivelmente não utilitário) e dois erraram, respondendo que possivelmente é utilitário. Já o método de controle II foi indicado como não utilitário por 17 participantes, sendo 85% das respostas. Foi marcado como possivelmente não utilitário por dois participantes e apenas um errou, marcando a opção 2 (possivelmente utilitário). Para os formulários JavaScript, na primeira função, dez participantes marcaram a opção zero (certamente não utilitária), o que corresponde a 55% das respostas. Seis desenvolvedores marcaram a opção 1 (possivelmente não utilitária), sendo 33% das respostas, e dois erraram indicando-a como possivelmente utilitária. Na segunda função, 14 participantes, ou seja, 78% deles, a definiram como certamente não utilitária, três desenvolvedores a marcaram como possivelmente não utilitária e um errou, escolhendo a opção 2 (possivelmente utilitária). Esses resultados mostraram que a maioria dos participantes acertou os pontos de controle. No entanto, a primeira função de controle para os formulários JavaScript pode não ter sido uma boa escolha, dado que para uma parcela considerável dos participantes (44% marcaram as opções 1 ou 2) ela não é claramente de propósito específico.

Tabela VIII
NÚMERO DE RESPOSTAS PARA OS PONTOS DE CONTROLE

Linguagem	Função	É utilitária? 0 (não)	1	2	3 (sim)
Java	PC I	16	2	2	0
	PC II	17	2	1	0
JavaScript	PC I	10	6	2	0
	PC II	14	3	1	0

Precisão: os resultados da classificação das funções, segundo os participantes do *survey*, são mostrados na Figura 5. Respectivamente para Java e para JavaScript, 43% e 41% das funções foram classificadas como certamente utilitárias. Aquelas indicadas como possivelmente utilitárias são, respectivamente para Java e JavaScript, 23% e 26%; as possivelmente não utilitárias correspondem a 17% e 11%; e 17% e 22% das funções não são utilitárias segundo os participantes. Assim, conclui-se que, nesta amostra aleatória das funções detectadas pelas heurísticas propostas neste artigo, 66% dos métodos Java e 67% das funções em JavaScript podem ser considerados utilitários (respostas nos níveis 2 e 3 da escala adotada).

Sumário: o *survey* realizado com 33 desenvolvedores, avaliando uma amostra aleatória de 5.13% do total de recomendações, mostrou que 66% das recomendações de métodos em Java e 67% das recomendações de funções em JavaScript são corretas, isto é, tendem a representar funções utilitárias.

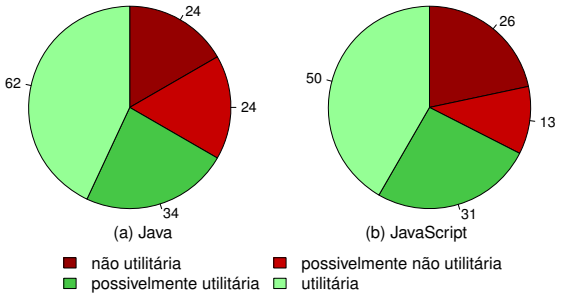

Figura 5. Classificação das funções de acordo com as respostas dos formulários.

VII. Ameaças à Validade dos Resultados

Uma ameaça aos resultados da avaliação dos sistemas de código aberto e ao estudo exploratório (Subseção II-A) é a suposição de que as bibliotecas utilitárias têm a palavra *"util"* em seu diretório. Existem funções utilitárias em arquivos que não obedecem a esse critério. Alguns exemplos dessas bibliotecas são `runtime/internal/Conversions.java` do sistema AspectJ em Java, que possui métodos de conversões de tipo; `artofillusion/math/FastMath.java` do sistema AOI que faz cálculos matemáticos simples como arredondamentos, potenciação, dentre outros; e `src/LiveDevelopment/Agents/DOMHelpers.js` do sistema Brackets em JavaScript que possui funções genéricas para manipular o DOM. Esse critério também afeta o número de sistemas no *dataset*, já que considerou-se apenas os que possuem pelo menos 25 funções em módulos utilitários. Acredita-se que é possível melhorar os resultados considerando um critério mais preciso para selecionar as bibliotecas utilitárias.

A classificação manual das funções utilitárias no estudo exploratório está sujeita a falhas, pois é uma análise parcialmente subjetiva, apesar de ter sido feita por uma especialista nos sistemas e nas linguagens. O risco dessas falhas pode ser diminuído incluindo a opinião de mais especialistas.

Em JavaScript, geralmente a importação de bibliotecas de terceiros se dá pela inclusão de um arquivo JavaScript contendo todo o código da biblioteca unificado. A remoção dessas bibliotecas dos projetos JavaScript de código aberto foi feita através de uma heurística implementada pela ferramenta *Linguist*. No entanto, não é garantido que esses arquivos sejam completamente removidos. O problema disso é que geralmente as funções não estão divididas em módulos nessas bibliotecas, logo as funções utilitárias estarão junto com todas as outras funções, mesmo que, originalmente, estivessem em módulos utilitários. Isso pode prejudicar o resultado das avaliações que se baseiam na suposição de que a maioria das funções utilitárias está em arquivos contendo a palavra *"util"* em seu nome ou diretório.

A análise das funções recomendadas no *survey* é, em certo grau, subjetiva, pois envolve a opinião de pessoas. No entanto, a experiência da maioria dos participantes aumenta a chance de se obter uma resposta tecnicamente mais precisa. Além disso, uma definição de funções utilitárias (conforme considerada nesta pesquisa) foi inserida nos formulários para que os participantes fizessem uso da mesma em suas respostas.

Devido ao grande número de funções identificadas, foi inviável avaliar cada uma delas, sendo os resultados do *survey* correspondentes a 3,8% e 8,5% das funções recomendadas para Java e JavaScript, respectivamente. Contudo, esta é uma amostra aleatória das funções e considera-se um resultado representativo do total.

VIII. Trabalhos Relacionados

Quando uma função utilitária é implementada em um módulo de propósito específico, caracterizando um *Feature Envy*, pode-se resolver o problema aplicando a refatoração *Move Method*. Alguns exemplos de sistemas de recomendação voltados para esse tipo de refatoração incluem o *JDeodorant*, o *JMove* e o *Methodbook*. *JDeodorant* [13] é um sistema que recomenda refatoração de *bad smells* como o *Feature Envy*. A refatoração *Move Method* foi proposta para remover esse tipo de *bad smell* [14]. Quando um método acessa mais atributos e métodos de uma classe do que da própria, ele pode ser recomendado para ser movido. Para que isso seja possível, o *JDeodorant* define um conjunto de precondições para verificar se a refatoração pode ser aplicada sem erros. São verificadas precondições para garantir que o sistema compile corretamente, para certificar que o comportamento do código seja preservado e que métricas de qualidade de projeto não sejam violadas. *JMove* [15] é um sistema que recomenda refatorações *Move Method* baseado na ideia de que os métodos de uma classe bem estruturada devem ter dependência de tipos similares. De modo simplificado, quando um método possui dependências mais similares a métodos de outras classes do que aos métodos da própria classe, recomenda-se a refatoração. O *Methodbook* [16] é outro sistema de recomendação de *Move Method*. Além de considerar as dependências estruturais entre os métodos (chamadas e atributos), ele também analisa a informação textual presente no código (comentários e identificadores) para mapear relações semânticas entre os métodos.

No entanto, esses sistemas foram propostos para mover qualquer tipo de método, sem tratar particularidades de funções utilitárias. Além disso, todos eles são implementados para sistemas Java e são fortemente dependentes de tipagem estática. Logo, teriam que ser adaptados para sistemas que usam linguagens dinâmicas, como JavaScript, o que possivelmente não é uma tarefa trivial. Em um trabalho futuro, pretende-se comparar os resultados obtidos nesta pesquisa com os resultados desses trabalhos.

Outros trabalhos em Engenharia de Software usam *Random Forests* para a identificação de características que mais influenciam no sucesso de um aplicativo *Android* [8]; para detecção de defeitos de software [17]–[19]; para prever o tempo de integração de *issues* resolvidas [6]; e para identificação de modificações não relacionadas em um *commit* [7]. Esses exemplos

mostram a variedade de aplicações do *Random Forests* nas pesquisas.

IX. Considerações Finais

Este trabalho apresentou um conjunto de heurísticas para identificar funções utilitárias baseadas em propriedades de código obtidas por meio de análise estática. Elas foram aplicadas a 84 sistemas Java e 22 sistemas em JavaScript de código aberto, e também em quatro sistemas proprietários em Java e JavaScript. Os resultados mostraram que essas heurísticas são capazes de identificar funções utilitárias com boa precisão. Em um *survey* realizado com 33 desenvolvedores de software obteve-se uma precisão 66% e 67%, respectivamente, para Java e JavaScript.

Os resultados obtidos com as heurísticas também são melhores que aqueles apresentados por técnicas de aprendizado de máquina. Apesar de obter bons resultados na identificação de funções nos módulos utilitários de cada sistema, com treinamento balanceado, essa abordagem não apresentou os mesmos resultados quando uma avaliação *cross-project* foi realizada.

Em trabalhos futuros planeja-se estender o protótipo implementado para uma ferramenta que possa ser usada nos ambientes integrados de desenvolvimento de software. Além disso, intenta-se aperfeiçoar as heurísticas propostas de forma a melhorar sua acurácia. Pretende-se também realizar uma avaliação das funções identificadas por meio das heurísticas considerando a opinião dos especialistas dos sistemas de código aberto.

Agradecimentos

Esta pesquisa é financiada pela FAPEMIG e pelo CNPq.

Referências

[1] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] T. Mendes, M. T. Valente, A. Hora, and A. Serebrenik, "Identifying utility functions using random forests," in *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 614–618.

[3] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class Corpus: A compiled version of the Qualitas Corpus," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–4, 2013.

[4] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[5] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan, "An empirical study of delays in the integration of addressed issues," in *30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 281–290.

[7] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2015, pp. 341–350.

[8] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? A case study on free android applications," in *301st International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 301–310.

[9] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[10] K. J. Archer and R. V. Kimes, "Empirical characterization of random forest variable importance measures," *Comput. Stat. Data Anal.*, vol. 52, no. 4, pp. 2249–2260, 2008.

[11] J. Bloch, *Effective Java*, 2nd ed. Prentice Hall PTR, 2008.

[12] L. Silva, M. Ramos, M. T. Valente, N. Anquetil, and A. Bergel, "Does Javascript software embrace classes?" in *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2015, pp. 73–82.

[13] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature envy bad smells," in *33rd IEEE International Conference on Software Maintenance (ICSM)*, 2007, pp. 519–520.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[15] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 232–241.

[16] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[17] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[18] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Guéhéneuc, "Can lexicon bad smells improve fault prediction?" in *19th Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 235–244.

[19] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *10th IEEE Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 409–418.