Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Identificando habilidades de desenvolvedores de software por meio de suas contribuições em repositórios

Lucas de Pace Ribeiro

JUIZ DE FORA
FEVEREIRO, 2022

Identificando habilidades de desenvolvedores de software por meio de suas contribuições em repositórios

Lucas de Pace Ribeiro

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientadora: Alessandreia Marte de Oliveira Julio

Coorientador: Gleiph Ghiotto Lima de Menezes

JUIZ DE FORA

FEVEREIRO, 2022

IDENTIFICANDO HABILIDADES DE DESENVOLVEDORES DE SOFTWARE POR MEIO DE SUAS CONTRIBUIÇÕES EM REPOSITÓRIOS

Lucas de Pace Ribeiro

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Alessandreia Marte de Oliveira Julio
Doutora em Computação - IC/UFF

Gleiph Ghiotto Lima de Menezes
Doutor em Computação - IC/UFF

Fabrício Martins Mendonça
Doutor em Ciência da Computação - DCC/UFMG

Leonardo Vieira dos Santos Reis
Doutor em Ciência da Computação - DCC/UFMG

JUIZ DE FORA
11 DE FEVEREIRO, 2022

A meu pai Arnaldo, à minha mãe Adelina e aos familiares pelo respeito e apoio incondicional a todas minhas decisões.

Resumo

Identificar habilidades é um processo importante para qualquer área no auxilio da tomada de decisão. Na Ciência da Computação em específico, a escolha do desenvolvedor de *software* adequado influencia diretamente na longevidade e prosperidade de uma empresa. Mesmo com tamanha importância, o julgamento de habilidades ainda possui subjetividade humana e pouca informação embasada no que diz respeito a mensurar o conhecimento em bibliotecas específicas, por exemplo. Esta monografia propõe e implementa uma abordagem para identificar habilidades de desenvolvedores de *software* em bibliotecas de Java analisando suas contribuições ao longo do tempo em repositórios *Git*. A abordagem utiliza analisadores de linguagens, o que possibilita o trabalho ser expandido para outras linguagens no futuro. Para mostrar o funcionamento da abordagem, foi elaborado um exemplo de utilização com o repositório do *JUnit4*, um *framework* do Java para auxiliar na criação de testes unitários. Foi possível obter algumas informações do repositório *JUnit4*, como por exemplo as bibliotecas mais utilizadas e os desenvolvedores que mais interagiram com cada uma delas.

Palavras-chave: Identificação de Habilidades em Java, Experiência em Bibliotecas, Tomada de Decisão, Seleção de Candidatos, Competências Técnicas, ANTLR

Abstract

Identifying skills is an important process for any area in helping decision making. In Computer Science in particular, the choice of the appropriate *software* developer directly influences the longevity and prosperity of a company. Even with such importance, the assessment of skills still has human subjectivity and little grounded information with regard to measuring knowledge in specific libraries, for example. This monograph proposes and implements an approach to identify skills of *software* developers in Java libraries by analyzing their contributions over time in *Git* repositories. The approach uses parsers, which allows the work to be expanded to other languages in the future. To show how the approach works, an example of use with the repository of *JUnit4*, a Java *framework* to help in the creation of unit tests. It was possible to obtain some information from the *JUnit4* repository, such as the most used libraries and the developers who most interacted with each one of them.

Keywords: Java Skill Identification, Libraries Expertise, Decision-making, Candidate Selection, Hard Skills, ANTLR

Conteúdo

Lista de Figuras 5

Lista de Tabelas 6

Lista de Abreviações 8

1 Introdução 9
- 1.1 Apresentação do tema . . . 9
- 1.2 Contextualização . . . 9
- 1.3 Justificativa . . . 10
- 1.4 Objetivo . . . 10
- 1.5 Metodologia . . . 11
- 1.6 Organização da monografia . . . 11

2 Fundamentação teórica 12
- 2.1 Sistemas de controle de versões . . . 12
- 2.2 Analisadores de linguagens . . . 17
- 2.3 Considerações finais . . . 21

3 Revisão da literatura 23
- 3.1 Desenvolvimento . . . 23
- 3.2 Trabalhos relacionados . . . 25
  - 3.2.1 Abordagens utilizando aprendizado de máquina . . . 25
  - 3.2.2 Abordagens analisando códigos-fonte . . . 28
  - 3.2.3 Abordagens multi-linguagens . . . 32
- 3.3 Comparativo . . . 34
- 3.4 Considerações finais . . . 36

4 Identificação de habilidades de desenvolvedores 37
- 4.1 Abordagem proposta . . . 37
  - 4.1.1 Identificação de padrões . . . 37
  - 4.1.2 Coleta de dados . . . 38
  - 4.1.3 Análise dos dados . . . 38
- 4.2 Implementação e utilização dos *softwares* . . . 40
  - 4.2.1 Tecnologias utilizadas . . . 41
  - 4.2.2 Limitações . . . 42
  - 4.2.3 Coleta de dados . . . 43
  - 4.2.4 Obtendo métodos de classes na linguagem Java . . . 43
  - 4.2.5 Processamento de dados . . . 44
- 4.3 Exemplo de utilização . . . 46
- 4.4 Considerações finais . . . 54

5 Conclusão e trabalhos futuros 55

Bibliografia 57

Lista de Figuras

2.1 Exemplo de um histórico de *commits* . . . 15
2.2 Repositório centralizado - Adaptado de (MENEZES, 2011) . . . 15
2.3 Repositório distribuído - Adaptado de (MENEZES, 2011) . . . 16
2.4 Exemplo do analisador léxico e do analisador sintático processando a entrada “int i = 0;“ - Adaptado de Parr (2013) . . . 19
2.5 Saída do ANTLR para a entrada mostrada na Listagem 2.4 utilizando a gramática do Java 9 . . . 20

3.1 Exemplo de nuvem de palavras navegável . . . 33

4.1 Informações utilizadas para gerar a visualização dos dados . . . 41
4.2 Tela inicial para a escolha do repositório a ser analisado . . . 47
4.3 Resultado parcial do processamento do repositório *JUnit* . . . 48
4.4 Tela inicial do visualizador . . . 49
4.5 Estatísticas parciais do uso de classes do *JUnit* . . . 50
4.6 Lista de interações com a classe *java.util.ArrayList* por desenvolvedor . . . 51
4.7 Lista de categorias . . . 51
4.8 Cadastro de categorias . . . 52
4.9 Estatísticas de uso de categorias . . . 52
4.10 Interação de desenvolvedores com a categoria *Hamcrest* . . . 53
4.11 Utilização das classes da categoria *Hamcrest* . . . 54

Lista de Tabelas

3.1 Palavras-chave 24
3.2 *String* de busca da Scopus 24
3.3 *String* de busca da IEEEXplore 24
3.4 Tabela comparativa dos trabalhos relacionados 35

Lista de Listagens

2.1 Exemplo de versão inicial de um artefato 13
2.2 Exemplo de versão modificada do artefato 13
2.3 Exemplo de diferença entre a versão inicial e alterada 14
2.4 Exemplo de código a ser processado 18
2.5 Exemplo simplificado de uma saída de tokens 18
2.6 Exemplo de gramática 19
2.7 Exemplo de código a ser processado 21
2.8 Exemplo de método sobrescrito do *Visitor* 21
4.1 Exemplos de declarações de variáveis 38
4.2 Exemplo de alteração no código: importação de uma classe 39
4.3 Exemplo de alteração no código: utilização de métodos 40
4.4 Exemplo de uma criação de arquivo 45
4.5 Método resgatado na classe *File* 45
4.6 Informações obtidas utilizando o ANTLR 46
4.7 Linhas alteradas do arquivo *src/main/java/org/junit/Assert.java* 48

Lista de Abreviações

ANTLR	ANother Tool for Language Recognition
CSS	Cascading Style Sheets
GMA	General Mental Ability
HTML	HyperText Markup Language
LUIS	Microsoft Language Understanding Intelligent Service
PLN	Processamento de Linguagem Natural
SCV	Sistemas de Controle de Versões
SQL	Structured Query Language
TI	Tecnologia da Informação
UI	User Interface
UX	User Experience
XML	Extensible Markup Language

1 Introdução

1.1 Apresentação do tema

A perspectiva de vida de uma empresa está diretamente ligada a funcionários eficazes e dinâmicos que são peças-chave para alcançar o sucesso. Por isso, refinar o processo de seleção para buscar profissionais com alta competência é crucial para qualquer negócio (ACIKGOZ, 2018).

Na busca por selecionar os melhores profissionais, são utilizados métodos para reconhecer as aptidões de um candidato. Alguns desses métodos são considerados tradicionais, pois são amplamente utilizados para seleção de candidatos de qualquer área, como é o caso das entrevistas e o *Curriculum Vitae*(ACIKGOZ, 2018). Estes métodos tradicionais não buscam avaliar diretamente a sua capacidade cognitiva, o que acaba prejudicando a seleção de bons profissionais. Mensurar a habilidade mental geral (*General Mental Ability ou GMA*) de uma pessoa é uma maneira eficiente de prever o seu desempenho no trabalho (SCHMIDT; JOHN, 1998). Afinal, como indicam os autores, a GMA está ligada diretamente a predisposição de aprendizado na execução de uma tarefa.

1.2 Contextualização

Nas vagas de emprego direcionadas a desenvolvedores de *software* é grande a procura por profissionais que tenham conhecimento prévio em alguma linguagem de programação (PHP, Python, JavaScript, etc) e habilidades relacionadas ao desenvolvimento de *software*, como conhecimento em banco de dados, interface de usuário (UI), experiência do usuário (UX), domínio de bibliotecas, etc (MONTANDON et al., 2020). E mesmo diante de competências tão específicas, a avaliação no mercado de Tecnologia da Informação (TI) normalmente se mantém nas metodologias tradicionais. Como os métodos usados para identificar as habilidades de desenvolvedores tendem a ser os tradicionais, a seleção de candidatos de TI está sujeita a subjetividade.

As entrevistas, por exemplo, podem ser feitas por pessoas sem conhecimentos o suficiente para avaliar tecnicamente o candidato e, mesmo que com profissionais de TI presentes, esta é uma forma muito superficial e imprecisa para aferir a capacidade de um aspirante à vaga, pois a avaliação fica aberta a subjetividade humana (CHIAVENATO, 1999). Já os testes lógicos e teóricos esboçam uma tentativa de mensurar o conhecimento, porém estão longe de ser o ideal. Uma prova em um curto espaço de tempo, bem como entrevistas, podem ser prejudicadas por fatores externos, como por exemplo: indisposição, doença, preocupações momentâneas, nervosismo (TADAIESKY, 2008). Além disso, esse tipo de teste é usado para avaliar conhecimentos específicos e pontuais, não considerando a capacidade cognitiva como foi proposto por Schmidt e John (1998) como a melhor forma de avaliar um candidato.

1.3 Justificativa

A justificativa para a realização desta monografia se deve a uma possível superficialidade quando avaliados os conhecimentos de um desenvolvedor de *software* a partir das abordagens tradicionais. Entrevistas, questionários e avaliação de *Curriculum Vitae*, por exemplo, contém subjetividade humana (CHIAVENATO, 1999). Por isso, semi-automatizar a identificação de habilidades de um desenvolvedor, considerando as linguagens que ele trabalha e o conjunto de tecnologias que ele domina, pode ser de grande valia para os recrutadores que buscam os profissionais mais qualificados ou para a tomada de decisão dentro de uma empresa na alocação de desenvolvedores em projetos, por exemplo. Dessa forma, a avaliação das habilidades de um candidato e sua capacidade cognitiva não estaria relacionada à subjetividade humana, podendo ser consolidada com dados.

1.4 Objetivo

Esta monografia objetiva criar uma abordagem para apoiar a identificação de habilidades de desenvolvedores de *software* afim de buscar os candidatos com maior aptidão para uma tecnologia requisitdada por uma vaga de emprego. Como objetivo geral pode ser mencionada a revisão da literatura que apresenta as características dos trabalhos relacionados,

bem como as abordagens utilizadas, as ferramentas desenvolvidas e as linguagens que podem ser analisadas. Além disso, outro objetivo é aplicar a abordagem em um projeto real, apresentar um exemplo de utilização e demonstrar os resultados obtidos.

1.5 Metodologia

Primeiramente foi realizada uma revisão da literatura sobre a identificação de habilidades de desenvolvedores de *software*. Essa revisão mostrou um comparativo entre as diferentes abordagens. Foram apresentadas as principais características de cada uma, demonstrando os pontos em comum e os divergentes. Além disso, foram apresentadas as ferramentas utilizadas e as limitações de cada trabalho.

Em seguida, foi proposta uma abordagem para a identificação de habilidades de desenvolvedores. Para tornar a abordagem viável, foi desenvolvido um *software* que teve seus detalhes de implementação e execução apresentados. Para finalizar, um exemplo de utilização da ferramenta foi elaborado a partir de um projeto de código aberto: o *JUnit*4[1].

1.6 Organização da monografia

Esta monografia está organizada em 4 capítulos, além desta introdução. O Capítulo 2 apresenta conceitos importantes para a compreensão da abordagem como: sistemas de controle de versões e análise de linguagens. O Capítulo 3 apresenta uma revisão da literatura sobre a identificação de habilidades de desenvolvedores apresentando diferentes abordagens, tais como: a verificação das bibliotecas importadas em um projeto e a análise de *Curriculum Vitae* de forma automatizada. O Capítulo 4 descreve a abordagem proposta nesta monografia e apresenta um exemplo de utilização da ferramenta desenvolvida. Por fim, o Capítulo 5 apresenta as conclusões e as sugestões de trabalhos futuros.

[1] https://junit.org/junit4/

2 Fundamentação teórica

Neste capítulo são apresentados alguns conceitos para melhor compreensão desta monografia e dos trabalhos relacionados. A Seção 2.1 discute os sistemas de controle de versões, abordando conceitos como repositórios e *commits*. A Seção 2.2 apresenta itens relacionados à análise de linguagens, tais como análise léxica, análise sintática e a ferramenta ANTLR (ANother Tool for Language Recognition)[2]. Por fim, a Seção 2.3 apresenta as considerações finais do capítulo.

2.1 Sistemas de controle de versões

Os Sistemas de Controle de Versões (SCV) são capazes de armazenar e gerenciar alterações realizadas em artefatos ao longo do desenvolvimento de um *software*. Os SCVs possibilitam, por exemplo: reverter um artefato ou projetos inteiros para versões anteriores, comparar as alterações que ocorreram no decorrer do tempo e registrar o autor de cada modificação (CHACON; STRAUB, 2014).

Para compreender melhor as funcionalidades de um SCV e demonstrar um modelo de evolução, foi criado um exemplo composto pela Listagem 2.1 e a Listagem 2.2. As listagens apresentam trechos de código de uma aplicação na linguagem Java que calcula e exibe na tela uma quantidade de elementos da sequência de Fibonacci[3]. Esta sequência é uma sucessão de números em que cada número começa com 0 e 1 seguido da soma dos dois elementos anteriores. A saída do programa da Listagem 2.1 exibe a posição e o valor dos 5 primeiros elementos dessa sequência . A Listagem 2.2 é uma versão alterada da Listagem 2.1 onde são exibidos os 10 primeiras posições e seus respectivos valores. Além disso, na Listagem 2.2 há o acréscimo de linhas em branco.

[2]https://github.com/antlr/antlr4

[3]https://www.math.hkust.edu.hk/ machas/fibonacci.pdf

```
1  public class Fibonacci {
2
3      public static long fibo(int elemento) {
4          if (elemento < 2) {
5              return elemento;
6          } else {
7              return fibo(elemento - 1) + fibo(elemento - 2);
8          }
9      }
10     public static void main(String[] args) {
11      for (int i = 0; i < 5; i++) {
12             System.out.print("(" + i + "):" + Fibonacci.fibo(i) +
                   "\t");
13         }
14     }
15 }
```

Listagem 2.1: Exemplo de versão inicial de um artefato

```
1  public class Fibonacci {
2
3
4      public static long fibo(int elemento) {
5          if (elemento < 2) {
6              return elemento;
7          } else {
8              return fibo(elemento - 1) + fibo(elemento - 2);
9          }
10     }
11
12     public static void main(String[] args) {
13
14         for (int i = 0; i < 10; i++) {
15             System.out.print("(" + i + "):" + Fibonacci.fibo(i) +
                   "\t");
16         }
17     }
18 }
```

Listagem 2.2: Exemplo de versão modificada do artefato

Para manter um histórico das alterações feitas ao longo do tempo em um artefato, como a mudança da Listagem 2.1 para a Listagem 2.2, os SCVs utilizam os ***commits***. O *commit* é o estado de um projeto em um momento específico. Para salvar um novo estado deve-se criar um novo *commit* (CHACON; STRAUB, 2014). Qualquer alteração desse estado (modificação ou exclusão, por exemplo) não representa mais o *commit*.

Supondo um *commit* inicial realizado no artefato da Listagem 2.1 e outro *commit*

efetuado no artefato da Listagem 2.2, é possível obter a diferença entre os estados de cada um destes artefatos. Para mostrar essa diferença, os SCVs utilizam o *diff*: algoritmo para comparação de dois arquivos diferentes(HUNT; MCILROY, 1976). A saída do *diff* do artefato da Listagem 2.1 e do artefato da Listagem 2.2 pode ser observada na Listagem 2.3. Na Listagem 2.3 uma inserção de linha é representada pelo símbolo + destacada na cor verde, enquanto que a remoção de uma linha é representada pelo símbolo -, destacada na cor vermelha. Observa-se que o *diff* marcou como alterações os acréscimos de linhas em branco, pois para analisar as diferenças, o comando considera apenas as estruturas textuais do código. Ou seja, o *diff* textual não considera a sintaxe da linguagem.

```
1  public class Fibonacci {
2
3  -
4  +
5      public static long fibo(int elemento) {
6          if (elemento < 2) {
7              return elemento;
8          } else {
9              return fibo(elemento - 1) + fibo(elemento - 2);
10         }
11     }
12 -
13 +
14     public static void main(String[] args) {
15 - for (int i = 0; i < 5; i++)
16 +
17 + for (int i = 0; i < 10; i++)
18             System.out.print("(" + i + "):" + Fibonacci.fibo(i) +
                   "\t");
19         }
20     }
```

Listagem 2.3: Exemplo de diferença entre a versão inicial e alterada

A Figura 2.1 apresenta um exemplo de histórico de *commits*. Na figura, os *commits* são representados por círculos. O círculo verde representa a versão atual do projeto enquanto que os círculos cinzas são versões anteriores. Cada *commit* é identificado por um *hash* representado por um valor hexadecimal de 40 caracteres, abreviado na imagem para facilitar a visualização. Observa-se na Figura 2.1 que o *commit* de *hash e137e9b* está ligado ao *commit f0ddffb* por uma seta, representando que o *commit e137e9b* é o pai do *commit f0ddffb*, ou seja, o *commit f0ddffb* foi criado após uma alteração no *com-*

mit e137e9b. Dessa forma, tomando como base o exemplo anterior da Listagem 2.1 e da Listagem 2.2, pode se dizer que o artefato da Listagem 2.1 é pai da Listagem 2.2. Esse histórico de *commits* montado ao longo do tempo facilita, por exemplo, a rastreabilidade de modificações de artefatos. Por exemplo, é possível saber qual a modificação feita na Listagem 2.1 que resultou na Listagem 2.2.

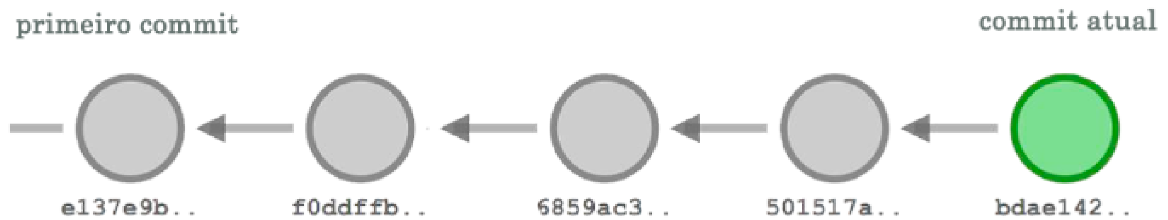

Figura 2.1: Exemplo de um histórico de *commits*

O responsável por monitorar todas as mudanças feitas no projeto, criar e armazenando o histórico de *commits* é o **repositório**. Para cada *commit* realizado, é criado uma nova versão do projeto que pode ser armazenada no repositório. Dessa forma, as mudanças podem ser visualizadas e resgatadas a qualquer momento (CEDERQVIST, 1992).

Repositórios tem diferentes tipos de arquiteturas. Uma dessas arquiteturas é a **centralizada**, representada na Figura 2.2. Nesse tipo de arquitetura há um repositório remoto central, representado pelo cilindro. Além disso, diversos desenvolvedores podem alterar os artefatos do repositório por meio do espaço de trabalho. Então, um *commit* realizado no Espaço de Trabalho 1, por exemplo, está armazenado no mesmo repositório que um *commit* realizado no Espaço de Trabalho 2.

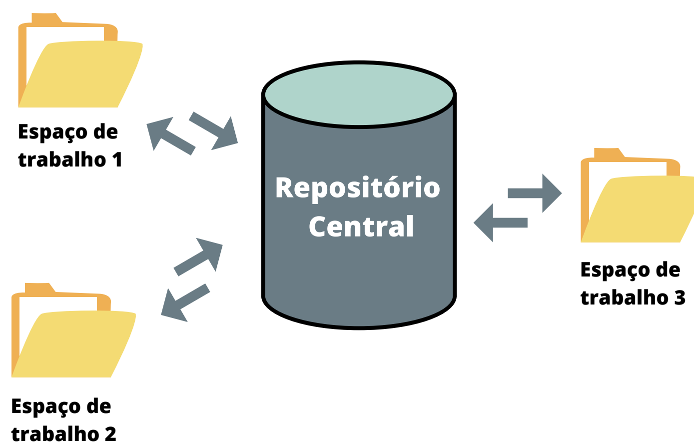

Figura 2.2: Repositório centralizado - Adaptado de (MENEZES, 2011)

Outra forma de organizar os repositórios é através da arquitetura **distribuída**. Nesta arquitetura, cada desenvolvedor tem um repositório local e todas as suas funcionalidades de forma individual no seu espaço de trabalho, como mostrado na Figura 2.3. Para efetuar *commits*, deve-se criar uma cópia do repositório do servidor no espaço de trabalho local. Somente após realizar os *commits* no repositório local é possível modificar o repositório do servidor com os novos *commits*. Para isso é necessário executar o comando *push*, enquanto que para resgatar os *commits* realizados no repositório do servidor e atualizar o seu repositório local deve-se executar o comando *pull*.

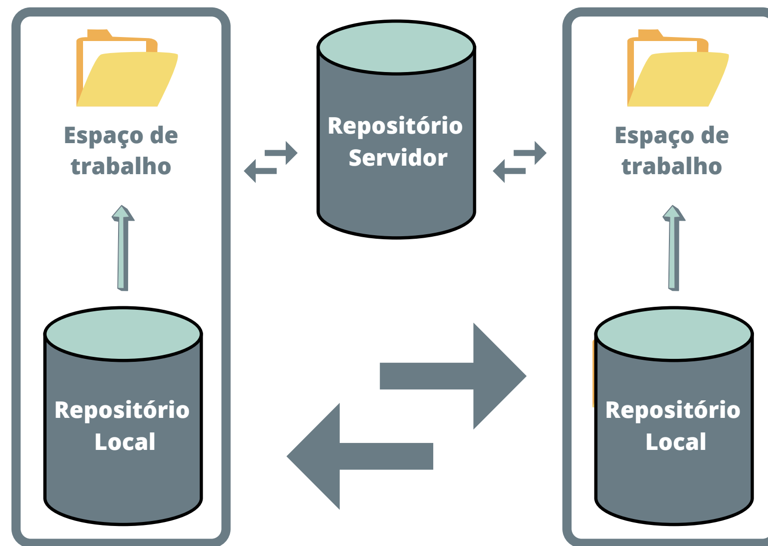

Figura 2.3: Repositório distribuído - Adaptado de (MENEZES, 2011)

Um exemplo de SCV distribuído popular[4] é o *Git*[5]. Em repositórios *Git* e em outros SCV distribuídos, é usual que as alterações sejam salvas em uma pasta na raiz do projeto que contém os artefatos que serão monitorados. Para criar um repositório local, deve-se executar o comando de criação na pasta raiz do projeto. No caso do *Git*, o comando de criação é: *git init*. Após a criação do repositório, as alterações podem ser rastreadas e monitoradas.

Outra forma de interação inicial com repositórios é executando sua clonagem. Na execução da clonagem, uma cópia do repositório é criada localmente. Esta ação possibilita ter acesso a todo histórico de alterações do projeto desde sua criação, mesmo que os *commits* tenham sido feitos por outros desenvolvedores. Dessa forma, é possível

[4]https://octoverse.github.com
[5]https://git-scm.com

um trabalho colaborativo e em paralelo entre os desenvolvedores que estiverem conectados a um mesmo repositório. Para facilitar a conexão entre desenvolvedores e repositórios, existem plataformas que hospedam os repositórios de servidor em nuvem. Assim, tendo conexão com a Internet e permissão de acesso, é possível que vários desenvolvedores compartilhem um repositório e trabalhem paralelamente. Um exemplo de plataforma para hospedagem de repositório *Git* em nuvem é o *GitHub*[6], utilizada por mais de 56 milhões de desenvolvedores[7].

2.2 Analisadores de linguagens

Como as alterações detectadas pelo comando *diff* do *Git* são apenas textuais, não é possível saber quais as estruturas da linguagem foram modificadas em um *commit*. Por exemplo, na alteração que ocorreu no *for*, representado na Listagem 2.3 pelas linhas 15 e 17, não é possível saber se a modificação foi em uma constante, variável, função ou qualquer outro tipo de estrutura da linguagem.

Descobrir qual foi a estrutura da linguagem manipulada em uma região alterada a partir de um resultado textual do *diff* entre um *commit* e outro é uma tarefa difícil, afinal cada linguagem tem sua própria sintaxe e suas particularidades. Uma particularidade que pode se diferir de linguagem para linguagem é, por exemplo, a chamada de um método que pode receber como parâmetro uma variável, uma constante, outro método, etc. Essa ampla gama de possibilidades torna inviável uma abordagem mais superficial, como a utilização de expressões regulares. Para solucionar este problema é necessário utilizar analisadores de linguagens.

Analisadores de linguagem são ferramentas utilizadas para identificar se um artefato de entrada faz parte de uma linguagem. Para isso, o artefato de entrada deve obedecer às regras gramaticais da linguagem (PRICE; TOSCANI, 2011). Para realizar essa verificação, o processo passa por duas etapas: a análise léxica e a análise sintática.

A **análise léxica** é a primeira etapa da execução de um compilador. O processo consiste em, dada uma entrada composta por uma sequência de artefatos, transformá-

[6]https://github.com
[7]https://octoverse.github.com

la em uma representação mais fácil de ser interpretada pelo analisador sintático. Para isso, o analisador léxico lê todo o código agrupando os caracteres para formar padrões pré-estabelecidos, chamados de **lexemas** (PRICE; TOSCANI, 2011).

Um exemplo de lexema que pode ser observado na Listagem 2.4 é o *int*, palavra reservada pela linguagem para denotar tipos inteiros composta pela concatenação dos caracteres *i,n,t*. Este padrão de caracteres é identificado (utilizando expressões regulares) pela linguagem, formando um lexema. Alguns exemplos de classes de lexemas são: palavras reservadas, identificadores, operadores e números (PRICE; TOSCANI, 2011).

```
1     int i = 0;
```

Listagem 2.4: Exemplo de código a ser processado

Os lexemas são utilizados pelo analisador léxico para formar os ***tokens***. Um *token* pode ser representado por uma tupla formada pela classe do lexema e o seu valor, se houver (PRICE; TOSCANI, 2011). Os *tokens* gerados pela linguagem Java ao ler a linha 1 da Listagem 2.4 estão representados, de forma simplificada, na Listagem 2.5, onde a tupla <x,y>é composta por um x que representa a classe do lexema e um y que representa o seu valor, se houver. É possível observar na Listagem 2.5 uma tupla <constante,0>, indicando que a classe do lexema é uma constante e que o seu valor é 0.

```
1     <int,>, <identificador,i>, <=,>,  <constante,0>, <;,>
```

Listagem 2.5: Exemplo simplificado de uma saída de tokens

A **análise sintática** é o processo de verificar se uma sequência de *tokens* dada como entrada é válida para uma linguagem (AHO et al., 2019). A formalização da linguagem é descrita por meio de uma gramática livre de contexto. Caso a entrada seja válida, o analisador sintático agrupa os *tokens* gerando uma estrutura representada por uma árvore de derivação. Caso seja uma entrada inválida, o analisador sintático emite um erro.

Para exemplificar, a Figura 2.4 mostra o processo dos analisadores léxico e sintático a partir da gramática representada na Listagem 2.6. A entrada “int i = 0;”é processada e transformada em *tokens* pelo analisador léxico. O analisador sintático usa os *tokens*

para gerar a árvore de saída. Nesta árvore, o nó raiz é uma sentença abstrata que define objetivamente o seu intuito, no caso da Figura 2.4: sentença, que também pode ser entendida como instrução. Os nós folhas são os *tokens* de entrada e o nós internos são sentenças pertencentes a gramática que agrupam e identificam os seus nós filhos. No exemplo, atribuição representa o objetivo da sentença. Os seus filhos representam a derivação necessária para se alcançar a entrada a partir da gramática representada na Listagem 2.6.

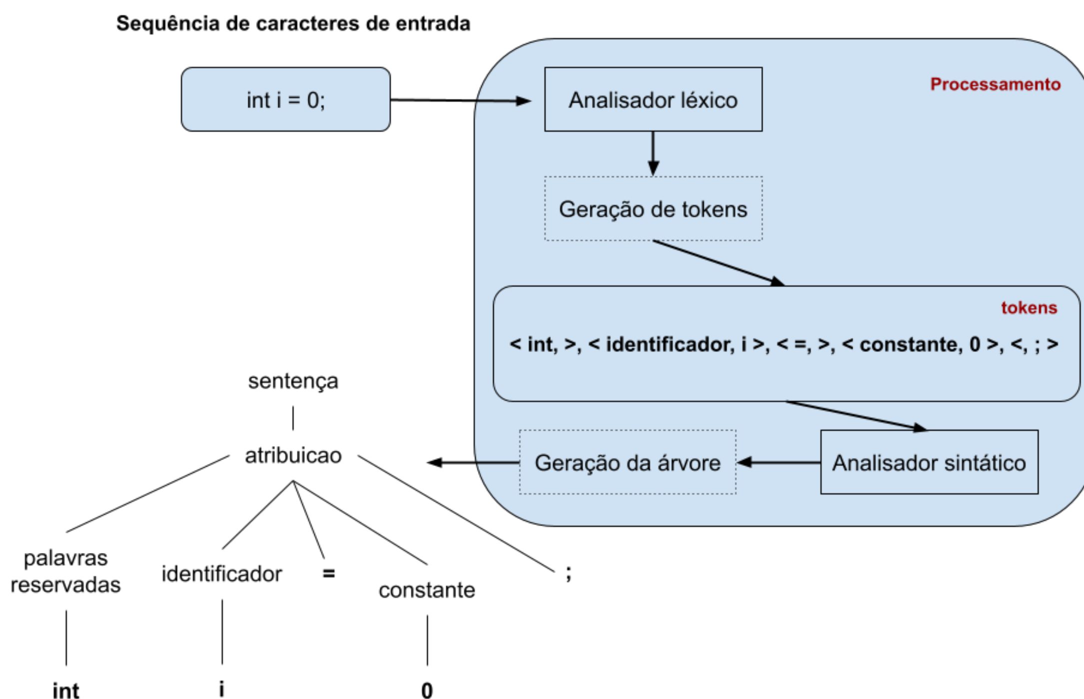

Figura 2.4: Exemplo do analisador léxico e do analisador sintático processando a entrada "int i = 0;" - Adaptado de Parr (2013)

```
1   atribuicao : PalavraReservada Identificador = Expressao ;
2   Expressao: int // valor aceito pela expresso regular: [-]?[0-9]+
```

Listagem 2.6: Exemplo de gramática

O **ANTLR** é uma ferramenta para gerar analisadores de linguagens. A partir de uma gramática, gera o analisador léxico e o sintático, criando automaticamente uma árvore para representar um artefato de entrada, desde que seja compatível com a gramática. O ANTLR pode ser usado na criação de linguagens, correspondência de padrões, ferramentas e *frameworks*. Foi usado entre outros projetos na ferramenta de

busca do Twitter[8] e no analisador sintático do C++ na IDE NetBeans[9] (PARR, 2013).

Várias das linguagens tem sua gramática disponibilizada pelo próprio ANTLR[10], podendo ser obtidas facilmente. Dessa forma, é possível implementar analisadores para diversas linguagens. Ao contrário do exemplo de gramática representada na Listagem 2.6, as regras de produções de linguagens de programação são mais completas e complexas.

Para simular uma execução real do ANTLR, foi utilizada a gramática do Java9[11] com o objetivo de gerar a árvore de derivação da Listagem 2.4. A Figura 2.5 representa, respeitando as regras da gramática e de forma abreviada, a saída gerada pelo ANTLR para o usuário. Pode ser observado que há um nó raiz, destacado em azul, que descreve de forma abstrata o que ocorre no seus nós filhos e que neste caso é uma declaração de variável local. Além disso, os nós folhas, destacados em vermelho, são resultantes da derivação de outros nós até alcançar a entrada do programa.

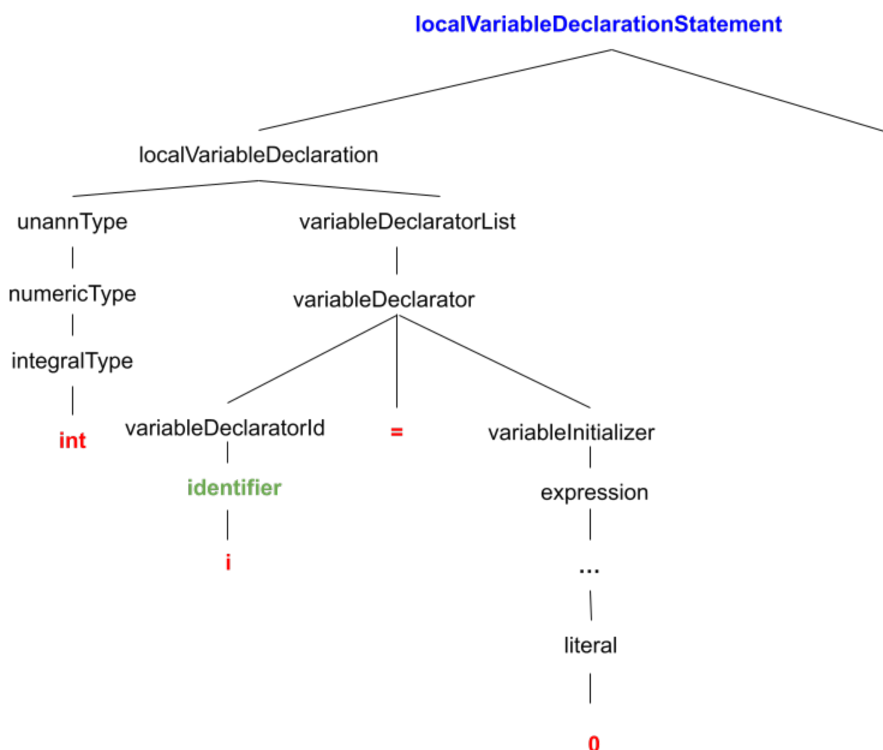

Figura 2.5: Saída do ANTLR para a entrada mostrada na Listagem 2.4 utilizando a gramática do Java 9

[8]https://twitter.com
[9]https://netbeans.apache.org
[10]https://github.com/antlr/grammars-v4
[11]https://github.com/antlr/grammars-v4/blob/master/java/java9/Java9Parser.g4

Além disso, é gerada pelo ANTLR uma interface que segue o padrão de projeto ***Visitor***[12]. O padrão facilita que métodos de uma classe sejam sobrescritos, se necessário, sem alterar outras partes do código. Tal interface gerada pelo ANTLR contém os métodos chamados quando um nó da árvore de saída é visitado. Portanto, é possível sobrescrever os métodos para extrair informações de um determinado nó e de seus nós filhos.

Para se obter todos os identificadores de um determinado código na gramática do Java9, por exemplo, deve-se executar uma ação para salvá-las ao alcançar o nó *identifier*, destacado em verde na Figura 2.5 (PARR, 2013). Para serem exibidos na tela todos os nomes de identificadores de um programa Java, como o da Listagem 2.7, o método do *Visitor* chamado quando um nó de identificação é percorrido deve ser sobrescrito (Listagem 2.8).

```
1 public class ClasseExemplo {
2     public static void main(String[] args) {
3         int i = 0;
4         for (int j = 0; j < 10; j++)
5             i += j;
6     }
7 }
```

Listagem 2.7: Exemplo de código a ser processado

```
1     @Override
2     public Object visitIdentifier(Java9Parser.IdentifierContext ctx) {
3         System.out.println(ctx.children.get(0).getText());
4         return visitChildren(ctx);
5     }
```

Listagem 2.8: Exemplo de método sobrescrito do *Visitor*

2.3 Considerações finais

O capítulo abordou os sistemas de controle de versões, discutindo como é armazenado e gerenciado alterações realizadas ao longo do tempo através dos *commits*. Uma sequência de *commits* proporciona um histórico de alterações de um projeto, monitorando quais as mudanças foram efetuadas de um estado do artefato para o outro e quem foi o autor. As

[12]https://refactoring.guru/design-patterns/book

mudanças identificadas e exibidas por um SCV levam em conta apenas a estrutura textual, não sendo considerada a estrutura da linguagem do artefato. Logo, para identificar quais as estruturas da linguagem foram alteradas é necessário analisar o código buscando identificar instruções (declarações de variáveis, chamada de métodos, etc). Para fazer isso de uma forma viável é possível utilizar ferramentas que geram analisadores léxicos e sintáticos a partir de uma gramática. Dessa forma é possível fornecer uma gramática e um artefato de entrada para os analisadores e obter como saída uma estrutura em árvore. Um exemplo de gerador de linguagens é o ANTLR, ferramenta capaz de gerar um *visitor* para cada nó da árvore possibilitando extrair suas informações.

3 Revisão da literatura

Este capítulo apresenta uma revisão da literatura relacionada à identificação de habilidades de desenvolvedores de *software*. Esta identificação pode ser usada tanto para avaliar candidatos em processos seletivos de empresas, quanto para a tomada de decisão na delegação de uma tarefa. Tal estudo se propõe a descrever as abordagens propostas bem como expor a motivação e os resultados alcançados por estes trabalhos. Para isso, este capítulo está dividido em quatro seções. A Seção 3.1 demonstra como foi desenvolvida a revisão da literatura, a Seção 3.2 descreve os trabalhos obtidos e a Seção 3.3 traz um comparativo entre eles. Por fim, a Seção 3.4 apresenta as considerações finais do capítulo.

3.1 Desenvolvimento

Para encontrar os trabalhos relacionados foram definidas palavras-chave que abordam o tema. A partir das palavras chaves, foi selecionada as bibliotecas digitais em que ocorreria a busca. Com as bibliotecas selecionadas, foi feita a *string* de busca para cada uma delas.

Para definir a *string* de busca foram utilizados 2 trabalhos previamente selecionados: Greene e Fischer (2016) e Montandon e Valente (2019). De cada trabalho foram extraídas palavras-chave que abordam o tema. Da busca inicial, foram extraídas as palavras-chave de mais trabalhos relevantes para esta monografia: Gajanayake et al. (2020) e Kourtzanidis, Chatzigeorgiou e Ampatzoglou (2020). As palavras mais relevantes foram selecionadas e seus sinônimos estão presentes na Tabela 3.1. É possível observar que cada termo utilizado tem um propósito. *Identifying*, por exemplo, é a ação de identificar habilidades, enquanto que utilizar um *framework* é uma habilidade. Os desenvolvedores podem ser referidos de algumas formas, por exemplo *Experts* e a palavra habilidade pode aparecer em outros trabalhos como *Expertise*.

A *string* de busca foi adaptada para ser utilizada em duas bibliotecas digitais: Scopus[13] e IEEExplore[14]. De acordo com Mourao et al. (2020), a Scopus é a biblioteca

[13]https://scopus.com
[14]https://ieeexplore.ieee.org

Tabela 3.1: Palavras-chave

Origem	**Palavras-chave**
Ação	Identifying / Mining / Explore / Quantify
Possível habilidades	Libraries / Frameworks / Languages / API
Pessoa	Developers / Experts / Specialist
Sinônimos de habilidade	Expertise / Skills / Ability

onde é encontrada a maior quantidade de trabalhos relevantes para a área de engenharia de software. Além disso, os autores afirmam que quando a pesquisa for complementada com uma busca utilizando a IEEExplore, a quantidade de resultados é ainda maior.

Para refinar os resultados foram feitos testes manuais de tentativa e verificação para encontrar as palavras que mais são relevantes para área. Além disso, algumas expressões regulares foram consideradas, podendo algumas serem parte de uma outra palavra completa, como por exemplo *develop** para encontrar trabalhos com *developer*, *developers* ou *development*. Embora as sintaxes das bibliotecas utilizadas sejam diferentes, alguns operadores como OR, AND, *, são semelhantes. Porém, a filtragem pela área, tópico, título, ano, autor, etc se mantêm distintos. Diante disso, a *string* de busca utilizada pela biblioteca Scopus está representada na Tabela 3.2.

Tabela 3.2: *String* de busca da Scopus

(TITLE (identify* OR mining OR abilit* OR expert* OR librar*) AND ABS ((identif* OR mining OR explor* OR quantify*) AND (developer* OR expert* OR specialist*) AND (expertise OR skill* OR abilit*) AND (librar* OR "programming language"OR framework* OR api))) AND (LIMIT-TO (SUBJAREA , "COMP"))

A *string* de busca utilizada na IEEEXplorer é mostrada na Tabela 3.3. Pela restrição imposta pela IEEEXplorer, a quantidade de operadores OR e operadores curingas (*) são limitadas. Para manter a qualidade de busca, foi feita uma adaptação no qual algumas palavras foram escritas sem curingas e algumas condições retiradas ou adaptadas.

Tabela 3.3: *String* de busca da IEEEXplore

(("Document Title":Identifying OR "Document Title": Identify OR "Document Title":mining OR "Document Title":ability OR "Document Title": abilities OR "Document Title":experts OR "Document Title": expertise OR "Document Title":library OR "Document Title": libraries)AND ("Abstract": identifying OR"Abstract": identify OR"Abstract": mining OR"Abstract": explor* OR"Abstract": quantify OR"Abstract": quantifying)AND ("Abstract": developer* OR"Abstract": expert* OR"Abstract": specialist*)AND ("Abstract": expertise OR"Abstract": ability* OR"Abstract": skill)AND ("Abstract"library OR"Abstract"libraries OR"Abstract": "programming language*"OR"Abstract": framework OR"Abstract": api))

Foram encontrados 292 trabalhos na Scopus e 55 na IEEEXplore, sendo que alguns deles eram duplicados (329 trabalhos diferentes) . Dentre os resultantes, foram excluídos trabalhos que não são escritos em português ou inglês ou que pertenciam as seguintes áreas: medicina, biologia, artes, física, química, economia, neurociência, energia, farmácia, agricultura e saúde. Por fim, após uma a leitura de títulos e resumos foram selecionados manualmente 7 trabalhos relevantes e de diferentes abordagens. Os trabalhos selecionados estão discutidos a seguir.

3.2 Trabalhos relacionados

Nesta seção são discutidos alguns pontos dos trabalhos relacionados tais como como: metodologia e ferramentas utilizadas, abordagem sugerida pelos autores, quais as habilidades avaliadas, objetivo do estudo e como se deu a validação da abordagem. Os trabalhos estão divididos em seções. A Seção 3.2.1 discute sobre os trabalhos que utilizaram aprendizado de máquina em sua proposta. A Seção 3.2.2 apresenta os trabalhos que fizeram uma análise de códigos-fonte enquanto a Seção 3.2.3 aborda os trabalhos que conseguiram propor uma solução para mais de uma linguagem de programação.

3.2.1 Abordagens utilizando aprendizado de máquina

Gajanayake et al. (2020) objetivam minimizar o tempo gasto nas avaliações de desenvolvedores de *software* e auxiliar os recrutadores na escolha dos melhores candidatos. Para isso, foi proposta uma forma de avaliação semelhante ao processo que ocorre no mercado de trabalho. Como afirmam os autores, é normalmente avaliado o *Curriculum Vitae* e, após esta primeira filtragem, os candidatos são convidados para uma entrevista. Na tentativa de automatizar este processo, foi feito o processamento de várias bases de informações, como: traços de personalidade, *Curriculum Vitae*, cartas de recomendação (cartas escritas por antigos chefes ou professores, por exemplo) e LinkedIn[15]. As bases de informações foram processadas de diferentes formas, abordadas a seguir.

Os traços de personalidade que foram levados em consideração seguem o modelo

[15]https://www.linkedin.com/

de avaliação OCEAN, que na Psicologia se refere a cinco fatores da personalidade: abertura a experiências, conscienciosidade, extroversão, neuroticismo e amabilidade (DANDANNAVAR et al., 2018). Para a avaliação da personalidade foi utilizada a transcrição do áudio de uma entrevista respondendo a perguntas específicas. Com o texto e utilizando *machine learning* foi possível estimar a personalidade do candidato. O algoritmo que apresentou o melhor resultado foi *logistic regression*. A acurácia variou entre 0.55 e 0.68, dependendo do fator de personalidade avaliado. Dessa forma, o melhor algoritmo não assegura a personalidade do candidato.

Para a análise do *Curriculum Vitae*, foram aplicadas no texto, técnicas neurolinguísticas para extrair detalhes do candidato como: nome, escolaridade, experiências de trabalho e tecnologias conhecidas. Esta identificação de habilidades teve uma média 0.7 de precisão. Com este resultado foi possível extrair as tecnologias chave do *Curriculum Vitae*, entretanto as habilidades menos citadas podem não aparecer no relatório final.

Na análise do *GitHub*[16], foram utilizadas API's do próprio site para apresentar as linguagens utilizadas pelo usuário em seus repositórios públicos.

Na avaliação da carta de recomendação foi feito um processamento do texto. Com o auxílio da biblioteca NLTK[17], o texto de entrada é extraído e após alguns tratamentos, uma nuvem de palavras explicitando pontos chaves do texto é formada.

Para extrair dados do *Linkedin*, foi desenvolvido um *software*, utilizando o *framework* Selenium[18], capaz de navegar pelos perfis dos desenvolvedores de forma automatizada. Após o tratamento do código da página, foram obtidas as informações textuais que foram utilizadas como entrada em uma rede neural. O modelo tem um processo de aprendizagem supervisionada que tenta categorizar as habilidades do desenvolvedor e verificar se estas suprem as habilidades requeridas para o cargo. Para este método de avaliação, a precisão alcançada foi de 0.4062, logo é menos preciso que a analise de outras bases de informaçõe. Os dados obtidos em cada método de avaliação é apresentada ao usuário separadamente.

Com os dados extraídos de diversas bases de informações e processados de forma

[16]https://github.com
[17]https://www.nltk.org
[18]https://www.selenium.dev

automatizada foi possível criar um resumo do perfil de um desenvolvedor. Dessa forma, pode ser útil para facilitar e tornar mais rápido a seleção de candidatos. Não foi feito nenhuma validação ou exemplo de utilização, porém os autores acreditam que as ferramentas satisfazem os candidatos e os recrutadores.

Em outra abordagem (MONTANDON; VALENTE, 2019), os autores buscam a identificação de habilidades de desenvolvedores de *software* para os componentes de *software* de terceiros. A hipótese do trabalho é que alguém que mantém um pedaço de código, possui habilidade nas tecnologias utilizadas para a sua implementação. Com essas ideias, foi dado enfoque a três bibliotecas de *JavaScript*: React[19], utilizado para a implementação de interfaces; Node-MongoDB[20], usado para se conectar com o banco de dados; e Socket.io[21], para comunicação em tempo real.

Para a coleta de dados foram analisados os 10 mil projetos mais populares de *JavaScript* presentes no *GitHub*. Em cada projeto foram buscados dois arquivos responsáveis por gerenciar dependências em repositórios desta linguagem: *package.json* e *bower.json*. Caso não fossem encontrados nenhum desses arquivos, o repositório era descartado. Analisando os arquivos, foi possível encontrar repositórios que utilizam as bibliotecas estudadas. A busca por especialistas nas bibliotecas é realizada através dos arquivos editados pelo desenvolvedor: quando o desenvolvedor edita um arquivo que importa uma biblioteca, ele é considerado um especialista.

De cada candidato foram extraídos atributos relacionados ao volume de participação no projeto: quantidade de *commits*, quantidade de *commits* importando alguma das bibliotecas estudadas e a quantidade de importações. Além de atributos relacionados à frequência (quantidade de dias desde a primeiro e a última importação, intervalo de dias entre um *commit* relevante e outro, etc) e consistência (quantidade de projetos que o desenvolvedor contribuiu e quantidade de projetos em que o candidato adicionou uma importação das bibliotecas relevantes durante todo o projeto).

A solução proposta para identificar os especialistas destas bibliotecas utiliza-se de aprendizado de máquina e algoritmos de clusterização. Os dados de entrada dos

[19]https://reactjs.org
[20]https://docs.mongodb.com/drivers/node/current/
[21]https://socket.io

softwares desenvolvidos são os dados coletados a partir das análises dos repositórios feitos anteriormente. O trabalho buscou responder duas perguntas: quão preciso o aprendizado de máquina é para identificar desenvolvedores habilidosos e qual atributo melhor distingue um programador iniciante de um especialista.

Para a validação dos resultados foi solicitado a todos os candidatos que respondessem uma única questão referente à biblioteca em que ele era candidato a especialista: qual sua habilidade, em uma escala de 1 (iniciante) a 5 (especialista), nesta biblioteca? A questão foi enviada para 8742 candidatos e o trabalho obteve 575 respostas. Com a auto-avaliação dos candidatos, foi possível obter uma base de dados para a execução da aplicação. A pesquisa mostrou, por exemplo, que 61% dos desenvolvedores se consideram especialista em React e 40% se consideram especialista na biblioteca Socket.io.

Foram utilizados dois algoritmos de aprendizagem: Random Forest (BREIMAN, 2001) e SVM (WESTON; WATKINS, 1999). Para avaliar o desempenho da solução proposta, foi utilizado o *F-Score*, pontuação calculada através da média harmônica entre a precisão e o *recall* (SASAKI, 2007). Utilizando os dados encontrados, foi alcançado o *F-Score* máximo de 0.56 no Node-MongoDB. Os autores argumentam que este resultado ruim é devido ao uso exclusivo do *GitHub* como base de dados. Para a clusterização dos dados foi utilizado o método k-means (MCQUEEN, 1967), evidenciando, por exemplo, que para o React e para o Node-MongoDB a frequência da atividade é o atributo mais importante, enquanto que para a Socket.io o que mais vale para o candidato se tornar um *expert* é a quantidade de projetos trabalhados.

3.2.2 Abordagens analisando códigos-fonte

Em Schuler e Zimmermann (2008), os autores extraíram informações buscando identificar as habilidades de desenvolvedores a partir de seus *commits*. O principal objetivo foi: encontrar dentre uma variedade de candidatos, qual a pessoa mais apropriada para manipular um certo arquivo de um projeto.

No trabalho foram abordados dois tipos de habilidades: a de implementação e a de uso. A habilidade de implementação diz respeito ao quanto o desenvolvedor conhece sobre a implementação de um método. Esta habilidade é detectada quando há alteração

dentro de um método, expressando que o desenvolvedor entende como ele funciona. Já a habilidade de uso é a capacidade do profissional utilizar e trabalhar com os métodos já criados, sendo detectada ao ocorrer uma inserção no código de uma chamada do método.

Para identificar as chamadas de métodos não foi utilizado nenhum processo de compilação nessa abordagem, tornando-a independente de compiladores e plataformas. Os autores optaram por analisar os arquivos individualmente a cada versão utilizado o APFEL(ZIMMERMANN, 2006). O APFEL é um *plugin* da IDE Eclipse[22] que busca, através da tokeninzação do código, criar um banco de dados com informações de um arquivo Java, armazenando, por exemplo, as variáveis utilizadas e as exceções. O resultado da utilização do *plugin* foi um processo menos custoso porém com limitações relacionadas à sintaxe. Não era possível, por exemplo, diferir tipos de variáveis passadas como parâmetro. Funções com o mesmo nome eram tratadas como idênticas e contabilizadas juntas, além do projeto ter sido limitado à linguagem Java.

Com as informações extraídas, foi criado para cada desenvolvedor um perfil que continha todos os métodos alterados e utilizados por ele, bem como uma frequência de mudança. Para identificar os desenvolvedores apropriados para uma tarefa, foi necessário montar manualmente uma base de dados contendo as funções que podem ser utilizadas no projeto. Por exemplo, se é necessário alguém com habilidades em SQL[23] (*Structured Query Language*), uma lista com o nome das funções deve ser extraída e comparada com cada perfil do candidato. Já para encontrar pessoas com habilidades semelhantes, foram comparados os perfis uns com os outros. Quanto mais alterações ou chamadas feita em um mesmo método, maior a semelhança entre os desenvolvedores. Com as habilidades e responsabilidades de cada um evidenciadas, o projeto pode auxiliar na comunicação entre quem cria e quem utiliza os métodos de uma aplicação.

Este trabalho foi avaliado utilizando o projeto ECLIPSE[24]. Como seu repositório é aberto, foi possível obter dados necessários para executar a aplicação. As informações extraídas condizem com o plausível, como é demonstrado pelo exemplo do desenvolvedor Erich Gamma, criador do *JUnit*[25]: seu perfil evidência métodos relacionados a testes

[22]https://www.eclipse.org/ide/
[23]https://www.iso.org/standard/63555.html
[24]eclipse github
[25]https://junit.org/junit5/

unitários da interface.

Em Kourtzanidis, Chatzigeorgiou e Ampatzoglou (2020), foi introduzida uma abordagem juntamente com uma ferramenta de código aberto RepoSkillMiner[26] buscando mensurar as habilidades de um desenvolvedor para alguns *frameworks* e tecnologias específicas. O *software* consegue trabalhar com três tecnologias na linguagem .NET e duas tecnologias *frameworks* de interface gráfica.

A abordagem propõe utilizar Processamento de Linguagem Natural(JAIN; KULKARNI; SHAH, 2018) (PLN) para avaliar se alguma das tecnologias definidas pelos autores foram utilizadas no código-fonte. O PLN aumenta o poder do aprendizado de máquina e das linguísticas computacionais tornando a linguagem humana processável e entendível pelo computador.

Foi utilizada a ferramenta PLN LUIS[27]. O *software* tem como proposta facilitar o desenvolvimento de programas que utilizem aprendizado de máquina. A partir de uma entrada de texto, por exemplo: onde é o restaurante mais próximo?, a ferramenta identifica a possível intenção do usuário (encontrar um local), e qual a precisão de sua resposta. As entradas que o LUIS consegue identificar partem de um modelo que o usuário do serviço define (WILLIAMS et al., 2015).

Embora o LUIS seja comumente utilizado para gerar aplicações de inteligência artificial de conversação, *chatbots* e identificação de voz, o serviço foi utilizado para tentar identificar qual *framework* é modificado em cada trecho de código.

Para treinar o LUIS, foram criados e adaptados 98 exemplos providos da documentação das 3 tecnologias escolhidas neste trabalho. O resultado final é, fornecidas algumas linhas de entrada, o retorno do LUIS estabelece a qual das 5 tecnologias aquela linha provavelmente pertence, bem como uma pontuação de sua precisão.

Para testar a ferramenta, foram utilizados 88 trechos de código disponíveis online[28]. O *software* obteve uma pontuação *F-Score* superior a 0.90 em todas tecnologias e, em uma delas, alcançou a pontuação 1.

Para um usuário ser avaliado, é necessário um repositório de entrada. A partir

[26]https://github.com/melkor54248/RepoSkillMiner
[27]https://www.luis.ai
[28]https://1drv.ms/u/s!Ak-X7Vy5IBQ3i_kM4CLWuUg6_UzTsw?e=3lqg5c

dele são coletadas informações como: todos os *commits* e, para cada *commit*, seu autor, arquivos modificados e regiões do código alterada. A identificação da linguagem de cada arquivo é feita pela sua extensão. Essa tarefa foi feita utilizando a API do *GitHub*. Esses dados são as informações utilizadas para construir a entrada do LUIS. A saída que o LUIS gera é um JSON[29] contendo qual a probabilidade de ser cada uma das 5 tecnologias.

Já a abordagem proposta por Oliveira, Pinheiro e Figueiredo (2020) avalia os profissionais utilizando o código-fonte de um *software*. Para tal, foi desenvolvida a ferramenta JExpert, utilizada buscar os melhores profissionais em 6 bibliotecas da linguagem Java que suportam uma arquitetura baseada em micro-serviços, sendo elas: SpringBoot[30], Apache Karaf[31], Spark[32], Netflix[33], JavaEE[34] e Swagger[35].

Para o desenvolvimento da ferramenta foram utilizados a linguagem Java e projetos com o SCV *Git* hospedados no *GitHub* para funcionar. Foi utilizada a API do *GitHub* para extrair projetos escritos em Java e que tinham como umas das palavras-chave *microservice*.

A ferramenta procura dentre os arquivos Java modificados que contenham a importação de uma das bibliotecas analisadas utilizando expressões regulares. Caso o arquivo importe alguma das bibliotecas, são contabilizadas a identificação do *commit* e a quantidade de importações relacionadas e não relacionadas à tecnologia em questão.

Com o objetivo de mensurar a habilidade de um desenvolvedor, a abordagem analisa três métricas: quantidade de *commits* que importam alguma das bibliotecas, quantidade de importações de cada biblioteca e a quantidade de linhas de código alteradas no *commit*. A heurística para calcular a interação do desenvolvedor com uma biblioteca é dada pela multiplicação da quantidade de linhas alteradas por *commit* com a quantidade de importações das bibliotecas estudadas dividido pela quantidade total de importações.

Para o experimento da ferramenta, foram analisados 1200 projetos e 797 desenvolvedores distintos. Ao todo, foram identificados 126 *experts*. Para ser considerado um

[29] https://www.json.org/json-en.html
[30] https://spring.io/projects/spring-boot
[31] https://karaf.apache.org
[32] https://spark.apache.org
[33] https://www.netflix.com/browse
[34] https://www.oracle.com/java/technologies/java-ee-glance.html
[35] https://swagger.io

expert pela ferramenta, deve-se ter uma pontuação que entre para os 25% melhores desenvolvedores em pelo menos duas métricas. A biblioteca com mais *experts* foi a Netflix com 64 desenvolvedores, enquanto foi encontrado apenas um *expert* utilizando a Karaf.

3.2.3 Abordagens multi-linguagens

Greene e Fischer (2016) propuseram uma análise independente de linguagem para identificar as habilidades de um candidato explorando suas contribuições em repositórios públicos. O trabalho procura auxiliar a tomada de decisão na escolha dos melhores candidatos para uma posição de desenvolvedor. O banco de repositórios analisado pelos autores é o *GitHub*.

A abordagem utilizada para extrair as habilidades dos desenvolvedores através de suas contribuições no *GitHub* ficou limitada à análise de extensões dos arquivos e palavras chaves encontradas nas mensagens de *commits* e do arquivo *ReadMe*. Os autores argumentam que analisar o conteúdo modificado pelo usuário e extrair informações relevantes é significativamente mais difícil.

Como resultado, foi obtida uma lista de 1000 perfis de desenvolvedores de *software* do *GitHub* que residem em uma certa localização (especificada no perfil da plataforma) através da API do *GitHub*[36]. Dessa forma foi possível extrair e clonar todos os repositórios de cada desenvolvedor e identificar cada *commit* do desenvolvedor. Para cada *commit* são obtidos os arquivos modificados por ele e a mensagem. Para cada repositório, são extraídos através também da API do *GitHub*, as linguagens utilizadas e o arquivo *ReadMe*.

Na abordagem, se o texto do *ReadMe* ou as mensagens dos *commits* contém a palavra "Java", por exemplo, é considerado que o repositório no qual o *ReadMe* está armazenado tem correlação com a linguagem "Java". Dessa forma, é atribuída esta habilidade aos desenvolvedores que colaboraram com o projeto mesmo que ele não tenha diretamente alterado uma linha de código da linguagem. A escolha de quais seriam as palavras-chave para verificação de habilidades foi feita de forma manual pelos autores. Foram incluídos linguagens de programação, *frameworks* Web, *frameworks de AJAX*, *frameworks* de mapeamento objeto-relacional e bibliotecas de interface.

Feita as correlações entre usuários e habilidades, foi escolhido exibir as informações

[36] https://docs.github.com/en/rest

através de uma nuvem de *tags* navegável, como mostra a Figura 3.1. Ao clicar na *tag* "Java", a próxima nuvem continha os desenvolvedores e habilidades mais proeminentes associados à *tag* "Java". O tamanho das palavras está associado à quantidade de aparições na análise e a sua cor está associada conforme o tipo de informação que ela representa, por exemplo: o nome de uma linguagem tem a cor azul enquanto o nome de usuário do desenvolvedor tem a cor verde. Ambos aparecem na mesma nuvem e são navegáveis.

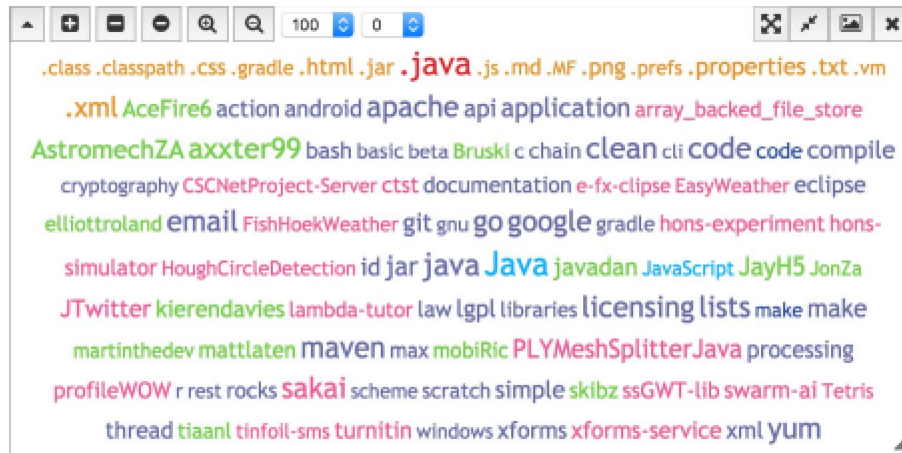

Figura 3.1: Exemplo de nuvem de palavras navegável

A validação do *software* se deu de maneira prática. Ele foi utilizado na recomendação de candidatos para duas empresas que buscam profissionais na cidade de Joanesburgo. As empresas deram um *feedback* de forma textual após as entrevistas respondendo se os candidatos recomendados supriam as suas necessidades. A resposta de ambas as empresas foi positiva.

Também considerando uma análise de código, Constantinou e Kapitsaki (2016) buscaram a identificação de habilidades de desenvolvedores utilizando plataformas sociais de códigos: *GitHub* e *Stack Overflow*[37]. O trabalho foi motivado pela falta de informação que esses *softwares* provêm a respeito das habilidades de um usuário ativo, por exemplo, a ausência de destaque nas linguagens mais utilizadas por um desenvolvedor ao longo do tempo. Diante disso, foi proposta uma abordagem que mensura a habilidade de um desenvolvedor baseada em sua atividade de desenvolvimento em uma linguagem. Para isso, foram extraídos todos os dados do usuário desde seu cadastro no *GitHub*.

Para cada repositório e ano foram extraídas as atividades de *commits* que mensu-

[37]https://stackoverflow.com

ram o quanto o desenvolvedor participou do desenvolvimento baseado na sua quantidade de *commits* e na quantidade total de *commits* do projeto. Também foi considerada a atividade de *commits* em um arquivo, que mede a porcentagem de arquivos distintos modificados pelo desenvolvedor em relação ao número de arquivos distintos modificados por todos os desenvolvedores do projeto. Outro fator considerado foi a quantidade de linhas alteradas no *commit* em relação ao total de linhas alteradas em todos os *commits* do projeto. Todos esses dados obtidos foram utilizados para cálculos propostos pelos autores.

Para os autores, outro indicativo da habilidade de um desenvolvedor é a constância de contato com a linguagem ao longo do tempo. Portanto, juntamente com os fatores citados anteriormente, foi considerada a frequência dos *commits*. Dessa forma, um aumento na frequência de *commits* em uma linguagem sinaliza um interesse maior por ela e deve ser recompensada, enquanto um decréscimo da frequência é punida pois pode ser indicação de que o desenvolvedor está se interessando mais por alguma outra linguagem, por exemplo.

A coleta dos dados inicial se deu por meio do projeto GHTorrent (GOUSIOS, 2013) e foi incrementada com o próprio *GitHub* para obter informações adicionais dos *commits*. As informações do *Stack Overflow* foram obtidas através do MSR 2015 (YING, 2015) e filtradas para exibir apenas usuários com respostas aceitas na plataforma e que desenvolveram em mais de uma linguagem de programação. A quantidade de respostas aceitas em uma linguagem demonstra um indicativo de aptidão do desenvolvedor para ela. Dessa forma, a heurística de análise de *commits* desenvolvida pode ser validada.

3.3 Comparativo

Esta seção efetua um comparativo entre os trabalhos discutidos anteriormente. Com o intuito de demonstrar mais explicitamente as semelhanças, peculiaridades e diferenças entre os trabalhos, a Tabela 3.4 apresenta características discutidas até o momento. Percebe-se a dificuldade de analisar o código-fonte de vários projetos considerando mais de uma linguagem. A dificuldade em tratar da sintaxe de cada linguagem torna o processo inviável por conta de sua complexidade e grande esforço manual. Dessa forma, normalmente se tem uma análise de código fonte de apenas uma linguagem ou o suporte a apenas uma linguagem ou poucas bibliotecas.

Tabela 3.4: Tabela comparativa dos trabalhos relacionados

	Análise de código-fonte	Suporte à múltiplas bibliotecas	Suporte a qualquer biblioteca de forma automatizada	Suporte a múltiplas linguagens	Utiliza-se de VCS	Utiliza-se de aprendizado de máquina	Análise de fatores externos ao projeto
Gajanayake et al. (2020)						✓	✓
Greene e Fischer (2016)		✓	✓	✓	✓		
Constantinou e Kapitsaki (2016)		✓	✓	✓	✓		
Schuler e Zimmermann (2008)	✓ Tokenização	✓			✓		
Montandon e Valente (2019)		✓			✓	✓	
Kourtzanidis, Chatzigeorgiou e Ampatzoglou (2020)	✓ PLN	✓			✓		
Oliveira, Pinheiro e Figueiredo (2020)	✓ Mineração de Texto	✓			✓		

Além disso, a utilização de SCV em abordagens que analisam projetos são unanimidade. Algumas utilizam as informações providas pela ferramenta como avaliador, pontuando a quantidade de *commits*, frequência de trabalho no projeto, etc. Entretanto outras utilizam o versionamento de arquivos para fazer a avaliação de um código-fonte.

Um grande fator limitador dos trabalhos é a baixa precisão na identificação real das alterações. Um exemplo disso, foi observado em Schuler e Zimmermann (2008), onde não foi possível diferir chamadas de funções com o mesmo nome e avaliar a habilidade do desenvolvedor em uma biblioteca específica. Além disso, em alguns dos trabalhos avaliados não são levados em consideração se as alterações feitas pelo desenvolvedor realmente envolvem a biblioteca utilizada. A alteração de um arquivo que importa uma biblioteca não confirma que o desenvolvedor tem habilidade nela, pois suas alterações não necessariamente estão relacionadas a ela (OLIVEIRA; PINHEIRO; FIGUEIREDO, 2020), (MONTANDON; VALENTE, 2019).

3.4 Considerações finais

A revisão da literatura descrita neste capítulo teve como finalidade detectar as principais características presentes na identificação de habilidades de desenvolvedores de *software*. Também, a partir dela foi possível identificar as semelhanças entre as abordagens, ferramentas utilizadas, etc . Além disso foi possível observar as limitações de cada uma, principalmente ligada às sintaxes de linguagens, e como se deu a validação ou experimentos de cada uma. Os trabalhos estudados possibilitaram identificar a necessidade de uma abordagem que analisa o código-fonte independente da linguagem ou biblioteca utilizada no projeto. Além disso, através deles surgiram ideias iniciais de como é possível validar uma ferramenta com tal propósito.

4 Identificação de habilidades de desenvolvedores

O objetivo deste capítulo é apresentar uma abordagem para identificar habilidades de desenvolvedores de *software* de forma semi-automatizada, além de discutir a implementação das ferramentas. Tal abordagem pode auxiliar na tomada de decisão em processos seletivos ou alocações de profissionais em projetos.

Para descrever a abordagem e a implementação, o capítulo foi dividido em quatro seções. A Seção 4.1 descreve a abordagem proposta por esta monografia e como foi realizada a identificação de padrões nas estruturas da linguagem, a coleta de dados e a análise de dados. Na Seção 4.2 são discutidos os seguintes detalhes da implementação: as decisões tomadas ao longo do desenvolvimento, as limitações do projeto, a coleta de dados na prática e o processamento dos dados. Na Seção 4.3 é mostrado um exemplo de utilização das ferramentas desenvolvidas utilizando o projeto JUnit4. Para finalizar, a Seção 4.4 apresenta as considerações finais do capítulo.

4.1 Abordagem proposta

A abordagem proposta nesta monografia busca encontrar as habilidades de desenvolvedores identificando quais as classes mais utilizadas por eles durante suas contribuições em repositórios *Git*. Através das informações extraídas, é possível quantificar a experiência do desenvolvedor em uma biblioteca ou área do conhecimento. Dessa forma, a tomada de decisão tem um embasamento, eliminando a subjetividade na identificação de habilidades.

4.1.1 Identificação de padrões

Identificar padrões de sintaxe de uma linguagem é uma tarefa difícil se for tratada de formas convencionais, por exemplo, utilizando expressões regulares. A complexidade envolvida no tratamento de declaração de variáveis é exemplificada na Listagem 4.1 que

mostra apenas algumas variações de declarações com o mesmo nome. Percebe-se que há várias formas de declarações em diferentes tipos de dados, o que aumenta a complexidade da identificação. Outros exemplos de situações que dificultam a identificação de padrões através de métodos tradicionais são as declarações de objetos e as chamadas de métodos.

```
int i = 0;
static int i = 0;
float i = 10.5f;
float i = (float) 10.5;
ArrayList i = new ArrayList<String>();
```

Listagem 4.1: Exemplos de declarações de variáveis

Para se ter uma solução precisa foi utilizado um gerador de analisador sintático a partir de uma gramática. Dessa forma, não é necessário tratar diretamente toda a sintaxe da linguagem e suas variações. Por exemplo, para encontrar uma declaração de variável na linguagem Java sem esses geradores, exigiria tratar e identificar todas as formas de declaração que a linguagem oferece. Porém, utilizando os geradores é possível fornecer a gramática da linguagem para identificar esses padrões sem implementar o analisador sintático.

4.1.2 Coleta de dados

Para utilizar analisadores de geradores sintáticos, é necessário ter os trechos de códigos escritos pelos desenvolvedores analisados. Por isso, é necessária uma forma de saber exatamente qual a alteração no código foi realizada e quem foi o autor. Essa análise é possível quando há o uso de SCV no projeto. Diante disso, são utilizados os *commits* pois estes suprem aos requisitos: estão associados a um desenvolvedor e contém os trechos de códigos alterados por ele. Para a coleta de dados é analisado todos os *commits* de um repositório. De cada *commit* são resgatados o autor e os arquivos modificados. Para cada arquivo modificado é resgatado todo o seu conteúdo e quais foram as linhas alteradas.

4.1.3 Análise dos dados

Com os trechos de código associados ao autor, é possível analisar os dados coletados e mensurar sua habilidade em uma determinada biblioteca. Para isso, pode-se verificar

se suas alterações interagiram com alguma classe pertencente àquela biblioteca. Para exemplificar, observa-se que na Listagem 4.1 na linha 5 ocorre uma interação com a classe *ArrayList*, da biblioteca *java.util*.

Para saber se houve uma interação com a classe é necessário analisar os dados coletados em busca de estruturas que comprovem a sua utilização. São exemplos de estruturas que comprovam a utilização da classe: a declaração de um objeto ou chamada de um método que pertence a classe. Estas estruturas podem ser identificadas com o auxílio de um analisador sintático.

Além disso, a análise de um arquivo pode promover falsos positivos se as estruturas analisadas forem escolhidas erroneamente. São exemplos desses falsos positivos as situações descritas nos *commits* da Listagem 4.2, onde é inserida apenas uma linha que importa a classe *ArrayList*. Porém, o código não apresenta usos do *ArrayList*. Com esta mudança, o autor, de acordo com a abordagem proposta nesta monografia, não demonstrou um indicativo de aptidão para utilizar a classe *java.util.ArrayList* e a biblioteca padrão do Java (*java.util*). Logo, se a estrutura de importação fosse contabilizada como interação com a classe, seria um falso positivo.

```
1 + import java.util.ArrayList;
2
3 public class Exemplo {
4     public static void main(String args[]) {
5       ArrayList<String> cars_name = new ArrayList<String>();
6     }
7 }
```

Listagem 4.2: Exemplo de alteração no código: importação de uma classe

Já na Listagem 4.3, a criação do objeto da classe importada *ArrayList* realizada na linha 6 não faz parte das linhas alteradas, bem como sua importação, realizada na linha 1. Porém, são utilizados os métodos *add* e *get*, nas linhas 7 e 8 respectivamente, que pertencem à classe. Mesmo que não tenha sido criado um objeto novo e nem tenha sido importada uma biblioteca ou classe nova na Listagem 4.3, o desenvolvedor demonstrou habilidade em utilizar a classe *java.util.ArrayList*. Consequentemente, alguma habilidade em utilizar a biblioteca *java.util*.

```
1  import java.util.ArrayList;
2
3  public class Exemplo {
4      public static void main(String args[]) {
5        ArrayList<String> cars_name = new ArrayList<String>();
6        + cars_name.add("lancer");
7        + System.out.println(cars_name.get(0));
8      }
9  }
```

Listagem 4.3: Exemplo de alteração no código: utilização de métodos

Para se ter uma análise mais precisa e evitar estes falsos positivos, foi utilizado um gerador de analisador sintático. Com os nós da árvore de saída resultante do analisador sintático, abordados na Seção 2.2, é possível identificar as estruturas que compõem o arquivo. Buscando pelas estruturas de chamadas de métodos, declarações de um objeto de uma classe e interações com o objeto declarado (como a modificação de um atributo do objeto) é possível restringir o que é considerado interação com a classe.

4.2 Implementação e utilização dos *softwares*

Para semi-automatizar a abordagem proposta foram desenvolvidas duas ferramentas: um *software desktop* e uma aplicação Web. O *software desktop* é capaz de identificar as classes que foram utilizadas pelos desenvolvedores nas linhas alteradas pelos *commits* de um repositório, por isso foi nomeado de **extrator**. Sabendo as classes utilizadas, é possível ranquear quais habilidades o desenvolvedor possui. Para uma melhor visualização dessas interações, foi desenvolvida a plataforma Web. Esta plataforma visa facilitar a visualização dos dados gerados pelo extrator, por isso foi nomeada **visualizador**.

As classes utilizadas no projeto podem ser agrupadas em categorias. A categoria é a representação de uma habilidade desejada, por exemplo: uma biblioteca ou uma área do conhecimento. Se deseja-se encontrar os desenvolvedores com mais habilidade em uma biblioteca, é possível agrupar as classes desta biblioteca em uma categoria. Se o objetivo é encontrar desenvolvedores com habilidades em banco de dados, por exemplo, é possível agrupar as classes relacionadas com esta área do conhecimento em um categoria.

Para facilitar o entendimento de quais as informações utilizadas pela plataforma

para gerar a visualização dos dados, tem-se o diagrama representado na Figura 4.1. É possível notar que desenvolvedores, categorias, *commits* e bibliotecas *imports* pertencem a um repositório. Cada desenvolvedor pode possuir diversos *commits*. Cada arquivo pertence a um *commit*, além de poder utilizar diversas bibliotecas. Uma biblioteca pode ser importada em vários arquivos, além ter a capacidade de pertencer a categorias.

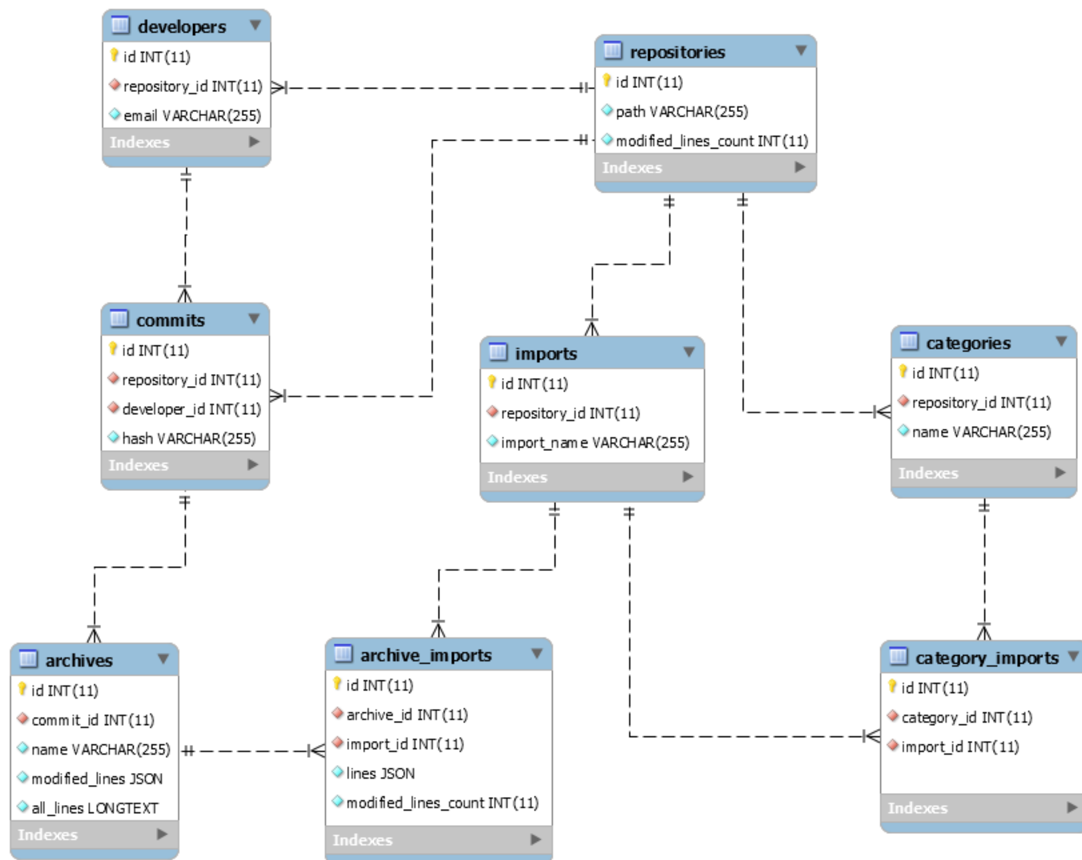

Figura 4.1: Informações utilizadas para gerar a visualização dos dados

4.2.1 Tecnologias utilizadas

A linguagem definida para o desenvolvimento do extrator foi Java, devido a sua compatibilidade com o gerador de analisador sintático escolhido, o ANTLR, discutido na Seção 2.2. No extrator foi realizada a construção de uma interface utilizando o *framework* Java Swing[38] para que o usuário possa escolher o repositório a ser analisado. O Maven[39] foi utilizado para gerenciar as dependências do extrator durante seu desenvolvimento. Já o

[38]https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
[39]https://maven.apache.org

SCV escolhido foi o *Git*, e o repositório do extrator está disponível online[40]. O visualizador foi construído utilizando HTML[41] (HyperText Markup Language), CSS[42] (Cascading Style Sheets), JavaScript[43], PHP[44] (com o *framework* Laravel[45]) e, para o banco de dados, o serviço MySQL[46]. Uma prévia da plataforma pode ser acessada online[47].

4.2.2 Limitações

Uma limitação inicial do extrator é capacidade de analisar somente a linguagem Java, utilizando a gramática do Java9[48] como entrada do ANTLR. Isso acontece pois cada linguagem requer um tratamento diferente no *visitor*. Porém, após o *visitor* ser implementado para uma linguagem, é possível adapta-lo seguindo a gramática de outra linguagem.

Além disso, em uma implementação inicial, só é possível analisar repositórios que utilizem o Maven. Isso é acontece pois há diferenças de como o usuário inclui uma biblioteca em cada gerenciador. No *Maven*, é utilizado um arquivo XML (Extensible Markup Language) para sinalizar as inclusões de bibliotecas externas, porém outros gerenciadores podem não utilizar este tipo de arquivo.

Outra limitação é a coleta de dados restrita ao SCV *Git*. Não foi possível generalizar a coleta de dados para outros SCV por causa da diferença na forma de resgatar as informações do projeto. Um comando do *Git* para visualizar os *commits* é diferente do comando utilizado em outro SCV. Por exemplo, para listar os 5 últimos *commits* no *Git* é necessário realizar o comando *git log -n 5*, enquanto que para o SCV Mercurial[49] o comando é *hg log –limit 5*. Portanto, para coletar dados de diferentes SCV, é necessário tratar cada um de forma individual. Além disso, um requisito para executar o extrator é ter o *Git* instalado pois os comandos são feitos na própria máquina que está executando a ferramenta.

[40]https://github.com/gleiph/developersSelection
[41]https://html.spec.whatwg.org/multipage/
[42]https://www.w3.org/Style/CSS/
[43]https://tc39.es/ecma262/
[44]https://www.php.net
[45]https://laravel.com
[46]https://www.mysql.com
[47]http://developer-selection.us-east-2.elasticbeanstalk.com
[48]https://github.com/antlr/grammars-v4/tree/master/java/java9
[49]https://mercurial-scm.org

4.2.3 Coleta de dados

Para a coleta de dados, o usuário fornece ao extrator um repositório de entrada. Com comandos *Git*, são recuperados todos os desenvolvedores que contribuíram para o repositório. Para cada desenvolvedor são recuperadas as informações de *commits*: o nome, todo o conteúdo dos artefatos modificados e as regiões de código alteradas. Cada artefato modificado é único, mesmo que este tenha o mesmo nome em diferentes *commits*.

Detalhes da implementação dos comandos para o SCV *Git* podem ser observados no arquivo Git.java[50]. Outra abordagem possível (presente em alguns trabalhos relacionados) é utilizar a API do GitHub, porém somente é possível fazer 5000 requisições para a API por hora[51]. Dessa forma, a API do *GitHub* é inviável para o *software* desenvolvido nesta monografia, pois em uma análise de um repositório com milhares de *commits*, esse limite seria ultrapassado.

4.2.4 Obtendo métodos de classes na linguagem Java

De acordo com a abordagem proposta, é necessário definir como ocorre a interação entre o desenvolvedor e uma classe, com o propósito de encontrar suas habilidades. A interação em um projeto com uma classe se dá por meio de chamadas de métodos e manipulação de atributos. Por exemplo, para instanciar um objeto é necessário chamar seu construtor e para manipulá-lo é necessário utilizar os métodos ou seus atributos diretamente. Por isso, para verificar se o desenvolvedor interagiu com a classe, é de grande valia ter mapeado todos os métodos possíveis de serem chamados no projeto e a qual classe ele pertence.

Para encontrar os métodos de cada biblioteca do Java é necessário localizar os arquivos *JAR* (Java ARchive) que compõem o projeto. Com os arquivos à disposição, é possível obter informações sobre os métodos: nomes de atributos das classes, bem como os nomes, os tipos de retornos e os parâmetros dos métodos. Para recuperar os arquivos *JAR*, duas abordagens foram usadas: uma para bibliotecas internas, como métodos do próprio java ou do projeto, e outra para externas, incluídas através do *Maven*, por exemplo.

Para tratar bibliotecas internas foi utilizada a *Reflection*[52], biblioteca do próprio

[50]https://github.com/gleiph/developersSelection/blob/master/src/main/java/git/Git.java
[51]https://docs.github.com/en/rest/overview/resources-in-the-rest-api
[52]https://www.javadoc.io/doc/org.reflections/reflections/0.9.10/org/reflections/Reflections.html

Java que é capaz de listar todos os métodos de uma classe. Há uma limitação da biblioteca: é possível analisar somente as classes que estejam no projeto em que a *Reflection* está sendo executada. Porém, não necessariamente as bibliotecas externas utilizadas no projeto analisado são importadas no extrator. Por isso, não é possível utilizá-la para listar os métodos externos, sendo necessária outra abordagem.

A solução para resolver este problema foi localizar onde está o arquivo *JAR* resultante da execução de cada biblioteca inserida no *Maven*. Para isso, o arquivo *pom.xml*, responsável por gerir as dependências, é analisado utilizando o SAX[53]. Dessa forma, é possível obter um caminho até o *JAR* de cada biblioteca, no qual é utilizado o comando *javap*[54] para listar as classes e os membros das classes (métodos e atributos, por exemplo). Assim sendo, é montada uma lista que contém todos os métodos das bibliotecas utilizáveis. Para mais detalhes sobre a implementação, o arquivo JarUtils.java[55] pode ser acessado.

4.2.5 Processamento de dados

O processamento inicial busca resgatar as classes que são importadas no arquivo, os métodos chamados, atributos de todas as classes do arquivo e as variáveis declaradas. Para isso, cada arquivo alterado pelo desenvolvedor de cada *commit* é passado como entrada do analisador sintático. Durante o processamento, para encontrar possíveis interações do usuário com a classe, o mecanismo *visitor* do ANTLR foi utilizado. Como discutido anteriormente, são criados diversos métodos que podem ser executados quando uma estrutura da linguagem é alcançada. É possível sobrescrever os métodos para manipular como o desenvolvedor quiser. Um exemplo de método sobrescrito do *visitor* é o *visitMethodInvocation*, visitado quando ocorre uma invocação de método. Para mais detalhes de todos os nós sobrescritos, o arquivo MyVisitor.java[56] pode ser consultado.

Para cada possível interação do desenvolvedor detectada pelo *visitor* com uma classe, é verificado se esta interação é realizada em uma linha alterada pelo *commit*. No caso de uma chamada de método detectada em uma linha alterada, são extraídos o seu nome, a classe a qual ela pertence, quantidade e tipos dos parâmetros. Foi possível

[53]https://docs.oracle.com/javase/7/docs/api/org/xml/sax/package-summary.html
[54]https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html
[55]https://github.com/gleiph/developersSelection/blob/master/src/main/java/Main/JarUtils.java
[56]https://github.com/gleiph/developersSelection/blob/master/src/main/java/myVisitor/MyVisitor.java

comparar essas informações (nome de método, parâmetros, tipo dos parâmetros, etc) do arquivo com o mapeamento de todos métodos dos arquivos *JAR* realizado anteriormente, já que ambos fornecem as mesmas informações. Para auxiliar a exemplificar os dados extraídos, a Listagem 4.4 mostra a criação de um arquivo nomeado *new_file* utilizando a biblioteca *File* e a invocação do método *getName()*.

```java

import java.io.File;

public class CriacaoArquivo {
    public static void main(String[] args) {
        File f = new File("new_file");
        System.out.println(f.getName());
    }
}
```

Listagem 4.4: Exemplo de uma criação de arquivo

Ao extrair os dados da classe *File* utilizando o comando *javap*, é obtido todas as informações de todos os seus métodos. As informações obtidas do método *getName()* estão representadas na Listagem 4.5. Ela é composta por um modificador de acesso, tipo do retorno e o nome do método (incluindo o caminho para a classe). Já a Listagem 4.6 exibe de forma intuitiva alguns dados obtidos com o auxilio do ANTLR, de forma direta (obtendo informação do nó) ou indireta (sendo necessário fazer tratamentos de texto). Há a classe do objeto, o tipo de retorno, o nome do invocador, o nome do método, os parâmetros e o modificador de acesso. Tratando as saídas é possível comprar ambas informações. No exemplo, o extrator conclui que a chamada do método *getName()* de um objeto *File* está associado a classe *java.io.File*. Todas informações obtidas com o auxílio do ANTLR relacionadas a uma chamada de método podem ser visualizadas na classe MethodInvocation[57].

```java
public java.lang.String java.io.File.getName()
```

Listagem 4.5: Método resgatado na classe *File*

[57]https://github.com/gleiph/developersSelection/blob/master/src/main/java/entities/MethodInvocation.java

```
Classe: java.io.File
Tipo de retorno: java.lang.String
Invocador: f
Nome: getName
Parametros:
Modificador de acesso: public
```

Listagem 4.6: Informações obtidas utilizando o ANTLR

Além disso, há interações com as classes que ocorrem de forma implícita ou que podem gerar falsos positivos. Cada uma destas situações foram tratadas individualmente, buscando aumentar a precisão do projeto. Um exemplo de situação é a chamada de um método que tem outro método como parâmetro. Para saber com exatidão a qual classe a chamada pertence, é preciso identificar quais os tipos dos parâmetros, e por isso é necessário descobrir o tipo de retorno do método. Este problema foi solucionado analisando o método chamado e procurando a qual classe ele pertence para então encontrar o tipo do retorno. Outro exemplo é tratar escopos de variáveis pois é necessário saber com exatidão o tipo da variável para que não se gere um falso positivo. Para tratar escopos de variáveis foi usda uma lista de listas: uma para definir os escopos e outra para definir as variáveis e seu valor em cada escopo. É possível observar este tratamento no *MyVisitor*[58].

4.3 Exemplo de utilização

Esta seção demonstra a utilização das ferramentas desenvolvidas no contexto desta monografia. Para isso foi escolhido o projeto *JUnit4*[59], *framework* para automatizar testes unitários em Java. O *JUnit4* tem seu código aberto e atende aos requisitos da aplicação de processamento de dados: utilizar o SCV *Git*, a linguagem Java e o gerenciador de dependências Maven. O repositório do projeto[60] conta com mais de dois mil *commits* e a contribuição de dezenas de desenvolvedores.

A Figura 4.2 mostra a interface extrator após o clique em *Local Repository*. Nesta janela o usuário escolhe um repositório que está no seu computador com o projeto *JUnit4*.

[58]https://github.com/gleiph/developersSelection/blob/master/src/main/java/myVisitor/MyVisitor.java
[59]https://junit.org/junit4/
[60]https://github.com/junit-team/junit4

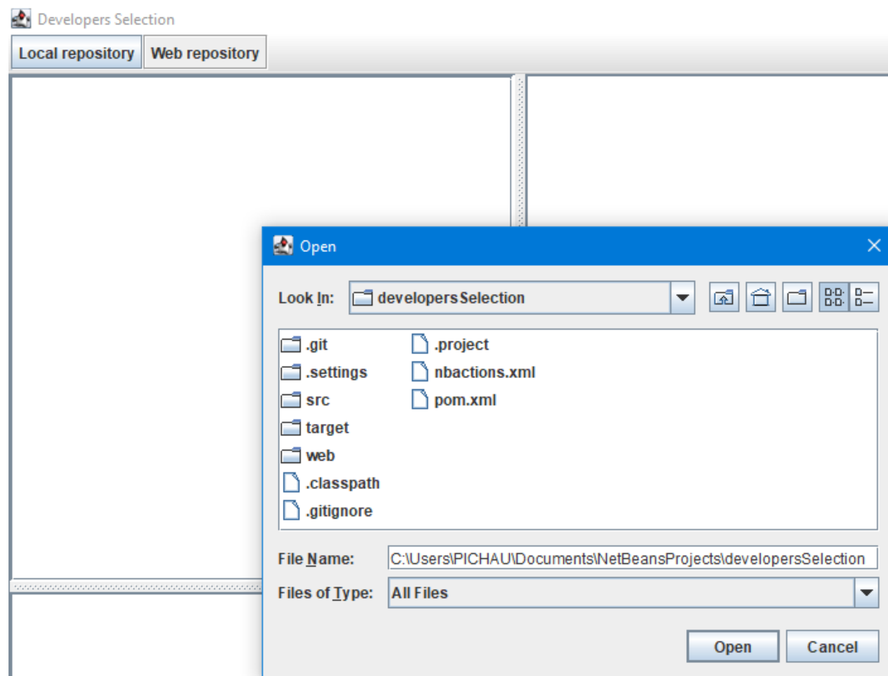

Figura 4.2: Tela inicial para a escolha do repositório a ser analisado

Ao selecionar o repositório, o processamento de dados é iniciado e quando finalizado é exibida uma árvore de arquivos, como mostrado à esquerda da Figura 4.3. É possível observar que os *e-mails* de todos os desenvolvedores que contribuíram para o repositório são exibidos. Ao selecionar algum desenvolvedor, os *hashs* dos *commits* feitos por ele aparecem, possibilitando clicar em um *commit* para visualizar os arquivos alterados que interagiram com alguma biblioteca. Por exemplo, na Figura 4.3 o desenvolvedor saff@redredrobot.home está selecionado no *commit 15815f...* realizado por ele onde ocorreu uma modificação em alguma biblioteca no arquivo *src/main/java/org/junit/Assert.java* e as alterações feitas estão exibidas à direita. As alterações feitas no *commit* são destacadas de verde, para facilitar a visualização.

Também são exibidas quais as classes do projeto que interagiram com as mudanças realizadas no arquivo e em que linha ocorreu a interação (canto inferior esquerdo da Figura 4.3). No arquivo selecionado *src/main/java/org/junit/Assert.java* é possível observar que houve alterações marcadas com a cor verde nas linhas 5 e 7, importando as classes. Porém as interações com essas classes só ocorreram a partir da linha 791, como mostrado no canto esquerdo inferior da Figura 4.3. Por isso as mudanças 5 e 7 não contabilizam, para a abordagem proposta, como uma demonstração de habilidade. As linhas de 791 a 798

estão representadas na Listagem 4.7. O arquivo completo, bem como as linhas de 791 a 798, e suas mudanças podem ser vistos no repositório online do *JUnit*[61].

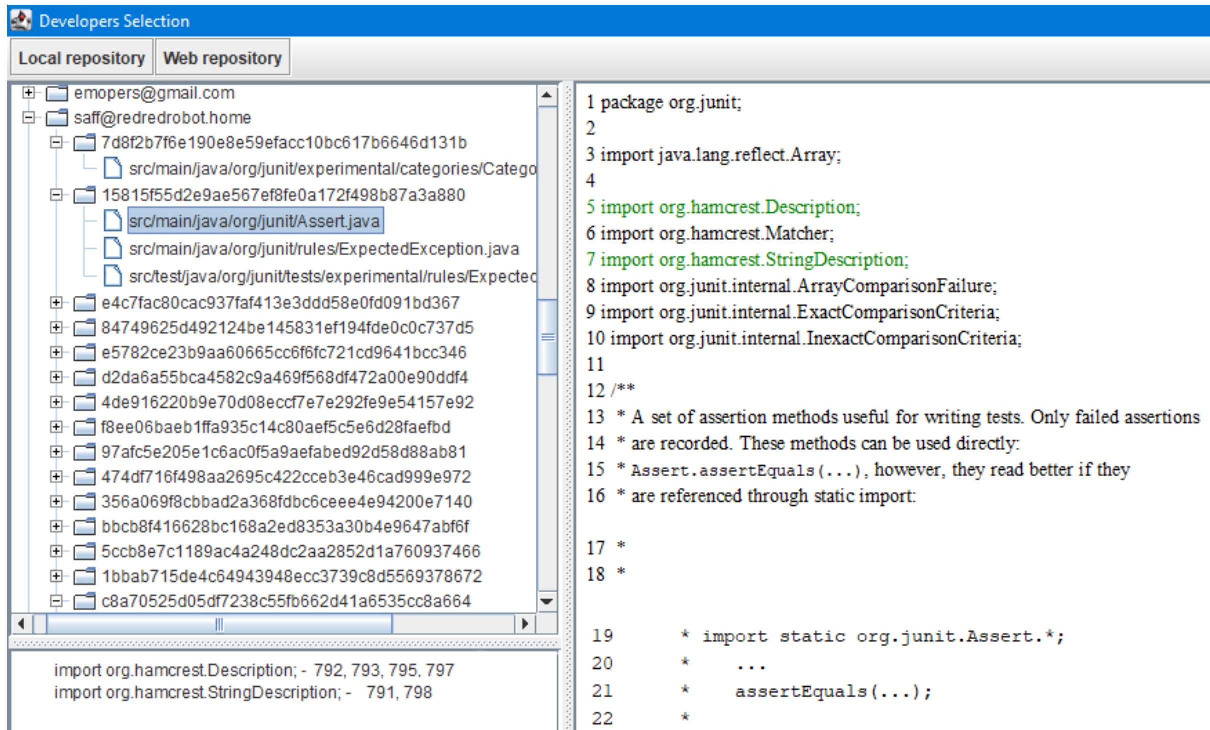

Figura 4.3: Resultado parcial do processamento do repositório *JUnit*

```
791 + Description description= new StringDescription();
792 + description.appendText(reason);
793 + description.appendText("/nExpected:  ");
794 + description.appendDescriptionOf(matcher);
795 + description.appendText("/n got:  ");
796 + description.appendValue(actual);
797 + description.appendText("/n");
798 + throw new java.lang.AssertionError(description.toString());
```

Listagem 4.7: Linhas alteradas do arquivo *src/main/java/org/junit/Assert.java*

Ao terminar o processamento, é criado um arquivo JSON que contém as informações extraídas do repositório como: desenvolvedores, *commits*, bibliotecas utilizáveis, arquivos (completo e as regiões alteradas) e as bibliotecas utilizadas em cada arquivo.

Ter um arquivo JSON das informações extraídas permite ao usuário armazená-las e construir suas próprias aplicações com os dados, caso seja necessário. Por exemplo, o arquivo JSON é utilizado para inserir as informações no banco de dados do visualizador

[61]https://github.com/junit-team/junit4/commit/15815f

que busca uma prover uma interface mais amigável com o usuário final, exibindo os dados de forma visual, intuitiva e com estatísticas gerais.

Ao entrar no visualizador é exibida a tela representada na Figura 4.4, na qual há uma lista arquivos JSON gerados pelos processamentos a serem carregados. Para exibir os dados na plataforma deve-se carregar o arquivo JSON desejado. O carregamento é o processo de salvar as informações obtidas pelo processamento no banco de dados do visualizador, pois dessa forma a exibição dos dados e das estatísticas é realizada de forma mais rápida e dinâmica. Para facilitar, o visualizador está disponibilizado online[62] já está com o arquivo JSON do *JUnit* carregado. Então, para acessar a exibição de dados basta clicar em visualizar ao lado do repositório que o usuário deseja abrir. Após o clique, o usuário é redirecionado para a exibição dos dados de desenvolvedores.

Lista de repositórios carregados

Show 10 entries Search:

ID	Caminho	Data	Ação
1	C:\Users\PICHAU\Documents\NetBeansProjects\junit4	2022-01-10 03:11:25	Visualizar

Showing 1 to 1 of 1 entries Previous 1 Next

Lista de repositórios a serem carregados

Nome	Ação
junit4-success.json	Carregar

Figura 4.4: Tela inicial do visualizador

A tela de exibição de dados de desenvolvedores está parcialmente representada na Figura 4.5, onde são exibidas as 10 classes mais utilizadas no projeto *JUnit*. Para cada classe é exibida: a quantidade de linhas alteradas que interagiram com a classe, o percentual que essas linhas representam dentre o total de linhas alteradas que interagem com alguma classe, a quantidade de arquivos que foram modificados e o percentual de

[62]http://developer-selection.us-east-2.elasticbeanstalk.com

declarações da classe dentre todas classes declaradas. A Figura 4.5 mostra que a classe *java.util.ArrayList* foi a mais utilizada em todo o projeto com 1400 linhas escritas que interagiram de alguma forma com a classe, o que representa 19.04% das interações feitas com todas as classes do projeto. Além disso, foram encontrados 881 arquivos alterados com alguma modificação que interagia com a classe, o que representa 26.6% dentre as importações feitas nos arquivos alterados.

Classe	Qtd. de linhas	Proporção de linhas alteradas	Qtd. arquivos modificados	Proporção de declarações	Interação por desenvolvedor
import java.util.ArrayList;	1400	19.04%	881	26.6%	Visualizar desenvolvedores
import java.lang.reflect.Method;	911	12.39%	277	8.4%	Visualizar desenvolvedores
import java.io.File;	767	10.43%	202	6.1%	Visualizar desenvolvedores
import java.io.ByteArrayOutputStream;	369	5.02%	139	4.2%	Visualizar desenvolvedores
import java.io.PrintWriter;	332	4.52%	40	1.2%	Visualizar desenvolvedores
import java.lang.reflect.Field;	329	4.47%	129	3.9%	Visualizar desenvolvedores
import java.io.StringWriter;	218	2.96%	101	3%	Visualizar desenvolvedores
import java.lang.reflect.InvocationTargetException;	203	2.76%	121	3.6%	Visualizar desenvolvedores
import java.io.PrintStream;	203	2.76%	64	1.9%	Visualizar desenvolvedores
import org.hamcrest.CoreMatchers;	197	2.68%	46	1.4%	Visualizar desenvolvedores

Figura 4.5: Estatísticas parciais do uso de classes do *JUnit*

Para saber os desenvolvedores que mais interagiram com a classe é possível clicar em visualizar desenvolvedores e as informações representadas na Figura 4.6 são exibidas: o e-mail do desenvolvedor e a quantidade de alterações feitas. Dessa forma, é possível observar com clareza quais desenvolvedores mais interagiram com uma classe específica. Todos os dados presentes na página podem ser filtrados para exibir apenas as interações de um único desenvolvedor. A página com as informações completas da análise por desenvolvedor do repositório do *JUnit* pode ser acessada online[63].

[63] http://developer-selection.us-east-2.elasticbeanstalk.com/statsOfDevelopers/1/-1

Utilização da classe: import java.util.ArrayList;

dsaff: 200
saff+aa@google.com: 152
mail@marcphilipp.de: 102
saff@google.com: 87
kcooney@google.com: 72
mark.michaelis@coremedia.com: 72
tibor17@lycos.com: 71
awulder@xebia.com: 67
nicobn@gmail.com: 66
david@saff.net: 51

Figura 4.6: Lista de interações com a classe *java.util.ArrayList* por desenvolvedor

Outra funcionalidade possível é criar uma categoria de classes. As categorias permitem criar uma habilidade desejada através da seleção das classes fazem parte dela. No caso da análise de repositório do *JUnit*, foram criadas três categorias de exemplo, como mostra a Figura 4.7. Cada categoria busca habilidades diferentes: a primeira, nomeada *Hamcrest*, agrupa todas as classes do projeto que interagem com a biblioteca *Hamcrest*, a segunda une todas as classes da biblioteca *java.util.concurrent* e a terceira une classes de diferentes bibliotecas.

Nome	Classes
Hamcrest	import org.hamcrest.BaseDescription; import org.hamcrest.CoreMatchers; import org.hamcrest.Description; import org.hamcrest.Matchers; import org.hamcrest.StringDescription;
java.util.concurrent	import java.util.concurrent.atomic.AtomicBoolean; import java.util.concurrent.atomic.AtomicInteger; import java.util.concurrent.atomic.AtomicLong; import java.util.concurrent.atomic.AtomicReference; import java.util.concurrent.ConcurrentHashMap; import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.CopyOnWriteArrayList; import java.util.concurrent.CountDownLatch; import java.util.concurrent.CyclicBarrier; import java.util.concurrent.ExecutionException; import java.util.concurrent.ExecutorService; import java.util.concurrent.locks.Lock; import java.util.concurrent.locks.ReentrantLock; import java.util.concurrent.TimeoutException; import java.util.concurrent.TimeUnit;
IO e awt	import java.io.ByteArrayInputStream; import org.hamcrest.BaseDescription; import org.hamcrest.CoreMatchers; import org.hamcrest.Description; import org.hamcrest.Matchers; import sun.reflect.generics.reflectiveObjects.NotImplementedException;

Figura 4.7: Lista de categorias

Para cadastrar uma nova categoria deve-se preencher um nome e quais as classes relacionadas a ela (Figura 4.8). Neste exemplo, tem-se uma categoria chamada *Java Lang*

que agrupa todas as classes da biblioteca *java.lang*. Essa categoria busca, por exemplo, encontrar os desenvolvedores com mais interações na biblioteca *java.lang*.

Figura 4.8: Cadastro de categorias

Com as categorias criadas, é possível obter estatísticas relacionadas a cada uma delas. Para cada categoria, os dados de todas as classes são agrupados, resultando em uma visão geral da habilidade criada (Figura 4.9). É possível notar que para cada categoria há a quantidade de linhas em que houve interação com alguma classe pertencente a ela. Por exemplo, na categoria *Hamcrest*, houve interação com alguma de suas classe em 268 linhas alteradas ao longo de todos os *commits*. Essas linhas alteradas estavam distribuídas em 81 versões de arquivos diferentes e dentre todas declarações, as classes que compõem a categoria *Hamcrest* representam 2.4%.

ID da categoria	Nome da categoria	Qtd de linhas alteradas	Qtd arquivos modificados	Proporção de declarações	Interação de desenvolvedores
4	java.util. concurrent	653	259	7.8%	Visualizar desenvolvedores
3	IO e awt	272	91	2.7%	Visualizar desenvolvedores
1	Hamcrest	268	81	2.4%	Visualizar desenvolvedores

Figura 4.9: Estatísticas de uso de categorias

Clicando em visualizar desenvolvedores são exibidas as informações da Figura 4.10. Esta figura representa a visualização da interação dos desenvolvedores na categoria

Hamcrest. Os e-mails dos desenvolvedores que interagiram com a categoria estão listados com a quantidade de interações feitas ao lado. É importante salientar que os e-mails no *Git* são definidos arbitrariamente pelo desenvolvedor e não há qualquer tipo de validação.

Categoria: Hamcrest

saff+aa@google.com: 42
mail@marcphilipp.de: 41
mark.michaelis@coremedia.com: 21
arjenwisse@gmail.com: 17
dsaff: 16
tibor17@lycos.com: 15
awulder@xebia.com: 14
kcooney@google.com: 14
nicobn@gmail.com: 14
pettermahlen@gmail.com: 14
kcooney@users.noreply.github.com: 12
paulduffin@google.com: 10
saff@redredrobot.local: 7
saff@google.com: 7
saff@dhcp-172-31-197-71.cam.corp.google.com: 7
kbeck: 6
saff@redredrobot.home: 5
pmahlen@shopzilla.com: 2
urs.metz@gmx.de: 1
coreyjv@gmail.com: 1
saff@dhcp-172-31-204-188.cam.corp.google.com: 1
saff@new-host-3.home: 1

Figura 4.10: Interação de desenvolvedores com a categoria *Hamcrest*

A plataforma permite também exibir dados das classes de apenas uma categoria. Alguns dados das classes da categoria *Hamcrest* estão representados na Figura 4.11. É possível observar que a classe *org.hamcrest.CoreMatchers* foi a mais utilizada, sendo utilizada em 197 linhas do projeto, o que representa 73.51% do total de linhas alteradas desta categoria. A classe foi utilizada em 46 versões de arquivos e a quantidade de sua declaração representa 1.4% das declarações totais no repositório.

Através dessas informações é possível que uma empresa tenha embasamento na tomada de decisão ao alocar um desenvolvedor em um projeto ou em uma seleção de candidatos para uma vaga de emprego. Por exemplo, caso a experiência com a biblioteca *Hamcrest* seja um requisito para a tomada de decisão, através da abordagem proposta foi possível mostrar que há indícios de que os desenvolvedores *saff+aa@google.com* e o *mail@marcphilipp.de* suprem tal requisito.

Classe	Qtd de linhas alteradas	Proporção de linhas alteradas	Qtd arquivos modificados	Proporção de declarações	Interação de desenvolvedores
import org.hamcrest. CoreMatchers;	197	73.51%	46	1.4%	Visualizar desenvolvedores
import org.hamcrest. StringDescription;	24	8.96%	16	0.5%	Visualizar desenvolvedores
import org.hamcrest. BaseDescription;	18	6.72%	9	0.3%	Visualizar desenvolvedores
import org.hamcrest. Description;	17	6.34%	7	0.2%	Visualizar desenvolvedores
import org.hamcrest. Matchers;	12	4.48%	3	0.1%	Visualizar desenvolvedores

Figura 4.11: Utilização das classes da categoria *Hamcrest*

4.4 Considerações finais

O capítulo apresentou a abordagem proposta por esta monografia para a identificação de habilidades de desenvolvedores: contabilizar a interação dos desenvolvedores com classes. Para semi-automatizar essa contabilização de interações, foi necessário utilizar o analisador de linguagens ANTLR para identificação de padrões. A identificação de padrões, por exemplo encontrar as chamadas de métodos, deve ocorrer em trechos de código em que houve interação com o desenvolvedor. Então, para resgatar as linhas de código alteradas por cada um dos desenvolvedores foi necessário analisar projetos que utilizem SCV *Git*. Para associar as interações a uma biblioteca ou área de conhecimento foi possível agrupar as classes em categorias criadas pelo próprio usuário.

Também foram apresentadas as ferramentas desenvolvidas: um *software desktop*, nomeado extrator, para o processamento de dados e uma plataforma Web para a exibição das estatísticas. Para demonstrá-las, foi feito um exemplo de utilização com o repositório do *JUnit*. Durante a exemplificação, foram ilustradas as interfaces de interações com o usuário. O capítulo também demonstrou alguns dos resultados alcançados após a análise do repositório do *JUnit*. Foi possível mostrar que é possível identificar os desenvolvedores que mais interagiram com cada biblioteca, proporcionando um embasamento para a tomada de decisão. Os resultados completos podem ser acessados online[64].

[64] http://developer-selection.us-east-2.elasticbeanstalk.com

5 Conclusão e trabalhos futuros

Esta monografia teve como objetivo auxiliar na identificação de habilidades de desenvolvedores. Para isso, foi proposta uma abordagem em que são analisados trechos de códigos e são quantificadas as interações do desenvolvedor com as bibliotecas utilizadas em um projeto com SCV. Dessa forma, é possível utilizar a quantidade de interações realizadas para embasar a tomada de decisão na hora de avaliar a habilidade de um desenvolvedor.

Foi desenvolvida uma ferramenta capaz de analisar repositórios *Git* que possuem arquivos em Java. A análise quantifica as interações dos desenvolvedores com bibliotecas presentes no projeto. Diferentemente de outros trabalhos, a abordagem de implementação utiliza geradores de linguagens. Esta abordagem permite analisar o código alterado pelo desenvolvedor e torna possível a expansão das linguagens e dos gerenciadores de dependências que podem ser analisados.

Também foi desenvolvida uma ferramenta Web, nomeada visualizador, para facilitar a visualização dos dados. A plataforma exibe as estatísticas da utilização de bibliotecas e a interação dos desenvolvedores com elas. Além disso é possível agrupar um conjunto de bibliotecas em categorias, possibilitando encontrar habilidades específicas, seja em uma área (*front-end*, banco de dados, etc) ou uma biblioteca específica. Essas funcionalidades permitiram visualizar estatísticas do repositório *JUnit*, encontrando, por exemplo, os desenvolvedores que mais interagiram com cada uma das classes e bibliotecas do projeto. Dessa forma, foram gerados parâmetros numéricos que podem auxiliar na tomada de decisão durante a escolha de desenvolvedores.

Como trabalhos futuros é possível expandir as linguagens que podem ser analisadas, os SCV que podem ser utilizados e os gerenciadores de dependências compatíveis. Quando houver mais tecnologias capazes de serem analisadas, também é possível a construção de uma interface que possibilita selecionar as configurações do repositório. Por exemplo, permitir o usuário selecionar a linguagem, o SCV e o gerenciador de dependência que o repositório utiliza. Para melhorar o tempo de processamento do extrator, pode ser usada uma abordagem incremental que analisa os trechos de códigos alterados, evitando

o processamento desnecessário das regiões não alteradas(GHEZZI; MANDRIOLI, 1979).

Para a melhoria da usabilidade das ferramentas, é possível criar um processamento em nuvem do repositório, onde a entrada de dados é feita pelo visualizador. Dessa forma, o usuário estaria isento de qualquer pré-requisito para a execução da ferramenta de processamento, não sendo necessário instalar um compilador Java ou o SCV *Git*. Outra funcionalidade que pode melhorar a usabilidade consiste em auxiliar o usuário a inspecionar manualmente as regiões de código alteradas que estão relacionadas a uma determinada classe. Por exemplo, apresentar todas as regiões de código que interagem com a classe *ArrayList*.

Bibliografia

ACIKGOZ, Y. Employee recruitment and job search: Towards a multi-level integration. *Human Resource Management Review*, v. 29, 02 2018.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. Identifying experts in software libraries and frameworks among github users. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2019. p. 276–287.

BREIMAN, L. Machine learning, volume 45, number 1 - springerlink. *Machine Learning*, v. 45, p. 5–32, 10 2001.

CEDERQVIST, P. Version management with cvs. 01 1992.

CHACON, S.; STRAUB, B. Pro git. In: APRESS (Ed.). [S.l.: s.n.], 2014.

CHIAVENATO. Planejamento, recrutamento e seleção de pessoal: como agregar talentos à empresa (4a ed.). *Săo Paulo: Atlas*, 1999.

CONSTANTINOU, E.; KAPITSAKI, G. M. Identifying developers' expertise in social coding platforms. In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.: s.n.], 2016. p. 63–67.

DANDANNAVAR, P. S.; MANGALWEDE, S. R.; KULKARNI, P. M.; SILVA, T. Social media text - a source for personality prediction. In: *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*. [S.l.: s.n.], 2018. p. 62–65.

GAJANAYAKE; HIRAS; ; GUNATHUNGA, P.; SUPUN, J.; ANURADHA; BANDARA, K. Candidate selection for the interview using github profile and user analysis for the position of software engineer. In: *2020 2nd International Conference on Advancements in Computing (ICAC)*. [S.l.: s.n.], 2020. v. 1, p. 168–173.

GHEZZI, C.; MANDRIOLI, D. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. 1, p. 58–70, jan 1979. ISSN 0164-0925. Disponível em: ⟨https://doi.org/10.1145/357062.357066⟩.

GOUSIOS, G. The ghtorrent dataset and tool suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 233–236. ISBN 978-1-4673-2936-1. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=2487085.2487132⟩.

GREENE, G. J.; FISCHER, B. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2016. p. 804–809.

HUNT, J. W.; MCILROY, M. D. *An Algorithm for Differential File Comparison*. [s.n.], 1976. Disponível em: ⟨https://www.cs.dartmouth.edu/~doug/diff.pdf⟩.

JAIN, A.; KULKARNI, G.; SHAH, V. Natural language processing. *International Journal of Computer Sciences and Engineering*, v. 6, p. 161–167, 01 2018.

KOURTZANIDIS, S.; CHATZIGEORGIOU, A.; AMPATZOGLOU, A. Reposkillminer: Identifying software expertise from github repositories using natural language processing. 09 2020.

MCQUEEN, J. Some methods for classification and analysis of multivariate observations. *Computer and Chemistry*, v. 4, p. 257–272, 01 1967.

MENEZES, G. G. L. de. *OURIÇO: UMA ABORDAGEM PARA MANUTENÇAO DA CONSISTÊNCIA EM REPOSITÓRIOS DE GERÊNCIA DE CONFIGURAÇAO*. Tese (Doutorado) — Gleiph Ghiotto Lima de Meneze, aug 2011.

MONTANDON, J.; POLITOWSKI, C.; SILVA, L.; VALENTE, M.; PETRILLO, F.; GUEHENEUC, Y.-G. What skills do it companies look for in new developers? a study with stack overflow jobs. *Information and Software Technology*, 09 2020.

MONTANDON, L. L. S. J. E.; VALENTE, M. T. Identifying experts in software libraries and frameworks among github users. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2019. p. 276–287.

MOURAO, E.; PIMENTEL, J. F.; MURTA, L.; KALINOWSKI, M.; MENDES, E.; WOHLIN, C. On the performance of hybrid search strategies for systematic literature reviews in software engineering. 04 2020.

OLIVEIRA, J.; PINHEIRO, D.; FIGUEIREDO, E. Jexpert: A tool for library expert identification. In: . [S.l.: s.n.], 2020. p. 386–392.

PARR, T. The definitive antlr 4 reference. In: PFALZER, S. D. (Ed.). [S.l.: s.n.], 2013.

PRICE, A. M. de A.; TOSCANI, S. S. Implementaçao de linguagens de programaçao: Compiladores. In: . [S.l.: s.n.], 2011.

SASAKI, Y. The truth of the f-measure. *Teach Tutor Mater*, 01 2007.

SCHMIDT, F.; JOHN, H. The validity and utility of selection methods in personnel psychology. *Psychological Bulletin*, v. 124, p. 262–274, 09 1998.

SCHULER, D.; ZIMMERMANN, T. Mining usage expertise from version archives. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2008. (MSR '08), p. 121–124. ISBN 9781605580241. Disponível em: ⟨https://doi.org/10.1145/1370750.1370779⟩.

TADAIESKY, L. T. Metodos de seleçao de pessoal: discussões preliminares sob o enfoque do behaviorismo radical. *Psicologia: Ciência e Profissao*, v. 28, 2008.

WESTON, J.; WATKINS, C. Multi-class support vector machine. In: . [S.l.: s.n.], 1999.

WILLIAMS, J. D.; KAMAL, E.; ASHOUR, M.; AMR, H.; MILLER, J.; ZWEIG, G. Fast and easy language understanding for dialog systems with Microsoft language understanding intelligent service (LUIS). In: *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. Prague, Czech Republic: Association for Computational Linguistics, 2015. p. 159–161. Disponível em: ⟨https://aclanthology.org/W15-4622⟩.

YING, A. T. T. Mining challenge 2015: Comparing and combining different information sources on the stack overflow data set. In: *The 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. to appear.

ZIMMERMANN, T. Fine-grained processing of cvs archives with apfel. In: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*. New York, NY, USA: Association for Computing Machinery, 2006. (eclipse '06), p. 16–20. ISBN 1595936211. Disponível em: ⟨https://doi.org/10.1145/1188835.1188839⟩.