INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Fórmulas de Legibilidade de Software

CLÁUDIO DANIEL TAVARES PINTO
Outubro de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

FÓRMULAS DE LEGIBILIDADE DE SOFTWARE

Cláudio Daniel Tavares Pinto

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Prof. Dra. Isabel Sampaio

Júri:

Presidente:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, outubro de 2018

Aos meus “paitrocinadores” e à “Ti’ Palmira”.

Resumo

Legibilidade é um conceito que tem vindo a ser estudado há vários séculos, surgindo desses estudos inúmeras fórmulas de legibilidade de texto, cujos resultados têm vindo a ser comprovados, conferindo-lhes assim credibilidade e aceitação no meio em que se apresentam.

Com a evolução da tecnologia, surgiram as linguagens de programação e, como consequência, o conceito de legibilidade de software começou a merecer a atenção de investigadores. Vários estudos foram realizados nesta área, tendo já sido propostas algumas fórmulas de legibilidade de software que visam quantificar este conceito subjetivo. Estas fórmulas têm focos e aplicabilidades distintas. Por um lado, podem ser utilizadas por iniciantes na área do desenvolvimento de software, de forma a permitir aos mesmos ter uma noção do que é software legível desde os primeiros estágios de aprendizagem. Por outro lado, empresas de desenvolvimento de software e respetivos profissionais podem recorrer a elas, de forma a estimar se o software que produzem apresenta um valor de legibilidade aceitável para os padrões estipulados.

Neste trabalho é realizado, em primeiro lugar, um levantamento das fórmulas de legibilidade de software propostas até à data, sendo elas posteriormente analisadas em detalhe, de modo a perceber o foco de cada uma. Com vista a oferecer aos desenvolvedores uma forma de avaliar a legibilidade do código em tempo de desenvolvimento, são então implementadas três das fórmulas de legibilidade recolhidas num *plugin* para o Ambiente de Desenvolvimento Integrado (IDE) NetBeans. Por fim, este *plugin* é testado junto de atuais alunos e profissionais graduados, de modo a perceber se a utilização do mesmo durante o processo de desenvolvimento de software pode contribuir para a melhoria da legibilidade do código produzido.

Palavras-chave: Legibilidade de Software, Fórmulas de Legibilidade, Qualidade de Software, Métricas de Legibilidade

Abstract

Readability is a concept that has been studied for several centuries, and these studies resulted in countless text readability formulas, whose results have been proven, thus giving them credibility and acceptance in the field where they belong.

With the evolution of technology, programming languages emerged and, consequently, the concept of software readability started to deserve the attention of researchers. Several studies have been done in this area, having some software readability formulas already been proposed that aim to quantify this subjective concept. These formulas focus on different aspects of the code and their applications differ. On one hand, they can be used by software development novices to allow them to have a sense of what is readable software since the earliest stages of learning. On the other hand, software houses and their professionals may use them to estimate if the written software has an acceptable readability value, comparing to the defined standards.

In this thesis, firstly, a survey of the proposed software readability formulas up to the date is done, which are then analyzed in detail, to allow the understanding of the focuses of each one of them. To provide the developers with a way to evaluate the readability of code in development time, three of the collected software readability formulas are implemented in a plugin for the NetBeans Integrated Development Environment (IDE). Finally, this plugin is tested with current students and graduated professionals, in order to understand if its use during the software development process can contribute to the improvement of the readability of the produced code.

Keywords: Software Readability, Readability Formulas, Software Quality, Readability Metrics

Agradecimentos

À Professora Doutora Isabel Sampaio, orientadora desta tese, desejo agradecer, não só pela sugestão do tema, mas também por toda a ajuda prestada ao longo deste trabalho.

A todos os familiares, amigos, colegas e professores que de alguma forma contribuíram para o meu sucesso e me apoiaram, não só durante o mestrado, mas também ao longo da licenciatura.

Aos bravos amigos e companheiros que encontrei, em 2012, nas turmas 1DG e 1DH, logo no início desta etapa, e que me acompanharam até esta reta final. Ao António Nunes, ao André Conceição, ao Nuno Duarte, ao Henrique Sampaio, ao Luís Resende e ao Rui Moreira, um muito obrigado!

À Real República dos LyS.O.S. e à Tuna Académica do ISEP por me terem feito crescer como pessoa e por me mostrarem que tão importante como o conhecimento adquirido durante o percurso académico são as vivências e as amizades que se criam no seu decorrer.

À Lala um obrigado muito especial, não só pelo carinho, compreensão e paciência, mas também por muitas vezes acreditar mais em mim que eu próprio.

À minha avó materna pelo sorriso na cara que faz sempre que me vê e à minha irmã pela contagiante boa disposição.

Por fim, e acima de tudo, o meu profundo agradecimento aos meus pais pela educação que me deram, tornando-me na pessoa que sou hoje, e por todo o apoio prestado, não só no meu percurso académico, mas a todos os níveis. Sem eles, teria sido extremamente difícil ter conseguido atingir tudo o que consegui até agora e tudo o que ainda estou certo de conseguir.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Gráficos

Lista de Códigos

Acrónimos e Símbolos

Lista de Acrónimos

ANTLR	*ANother Tool for Language Recognition*
API	*Application Programming Interface*
ASL	*Average Sentence Length*
ASCII	*American Standard Code for Information Interchange*
AST	*Abstract Syntax Tree*
ASW	*Average Number of Syllables per Word*
AWL	*Average Word Length*
B&W	*Buse & Weimer*
BSAD	*Blank Space After Directive Statements*
CLI	*Coleman-Liau Index*
CR	*Code Readability*
CRT	*Code Readability Tool*
DRY	*Don't Repeat Yourself*
FRES	*Flesch Reading Ease Score*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
JDK	*Java Development Kit*
KISS	*Keep It Simple, Stupid*
LSA	*Latent Semantic Analyisis*
n1	Número de operadores únicos
N1	Número total de operadores
n2	Número de operandos únicos
N2	Número total de operandos

IPFCR	*Impact of Programming Features on Code Readability*
IU	Interface do Utilizador
JAR	*Java ARchive*
JDK	*Java Development Kit*
LL	*Lines Length*
LOC	*Lines of Code*
LOM	*Lines with Comments*
LPP	*Lines per Page*
NCD	Novo Modelo de Desenvolvimento de Conceito
NLAS	*Breaking the Line After Semicolon*
NOBL	*Number of Blank Lines*
NOCL	*Presence of Comment Lines in the Program*
NOM	*Number of Methods*
PHD	*Posnett Hindle Devanbu*
POGJE	*Properties of "Good" Java Examples*
PPA	*Partial Program Analysis*
SLOC	*Source Lines of Code*
SRES	*Software Readability Ease Score*
WCMR	*Word Concreteness and Memory Retention*

Lista de Símbolos

$\sum$	Somatório
$\lvert \ \rvert$	Módulo
$\in$	Elemento

1 Introdução

A indústria procura constantemente formas de tornar os seus processos mais eficazes e eficientes, de modo a aumentar a produtividade e diminuir os custos de produção (Jeremy, 2013). As empresas de desenvolvimento de software não fogem a esta regra.

Sendo que a fase de manutenção de software tende a ser penosa (Buse and Weimer, 2010), seria interessante procurar atenuar este problema de alguma forma. A legibilidade de software tem um papel preponderante na fase de manutenção do mesmo (Sampaio and Barbosa, 2016), dado que a dificuldade de perceção do código está diretamente relacionada com o tempo que a manutenção do software leva (Tashtoush et al., 2013).

De modo semelhante, a legibilidade do código-fonte afeta também a forma como iniciantes na área do desenvolvimento de software interpretam o código que está incluído, por exemplo, em manuais ou tutoriais *online* (Börstler, Caspersen and Nordström, 2015).

Apesar da iniciativa de muitos investigadores em estudar a área da legibilidade de software, ainda há muito por fazer (Sampaio, 2017). Vários estudos foram já realizados e também já foram feitas propostas de formas de avaliar a legibilidade de software, porém, no que toca à implementação de ferramentas de avaliação de legibilidade, são poucas as contribuições significativas (Sampaio, 2017; Buse and Weimer, 2010; Abbas, 2010).

Procurando colmatar esta falha, neste trabalho será feito um levantamento do que foi estudado nesta área e, após realizar uma análise comparativa entre as várias fórmulas existentes, proceder-se-á à implementação de uma ferramenta de avaliação de legibilidade de software com recurso às fórmulas de legibilidade propostas ao longo dos últimos anos que se verifiquem mais adequadas.

A melhoria da legibilidade tem como objetivos primordiais a redução dos tempos leitura e de compreensão do software, o que consequentemente levará à redução dos custos do desenvolvimento e manutenção do software.

Sendo a ferramenta proposta destinada para toda a comunidade de desenvolvedores, vale enfatizar que ela permitirá a iniciantes no desenvolvimento de software serem capazes de avaliar o código que escrevem e perceber se o mesmo se apresenta ou não legível. Isto irá dar-lhes, desde o início do seu percurso nesta área, uma noção sobre como deve estar estruturado o código, de forma a torná-lo legível.

Em última instância, esta vai acabar por ser uma contribuição para as boas práticas da engenharia de software.

Este capítulo introdutório serve para introduzir e contextualizar o problema, abordar os objetivos que se pretendem alcançar com este trabalho e apresentar a organização do documento.

1.1 Contextualização e Problema

Legibilidade de software é um tópico que tem merecido a atenção de vários investigadores no decorrer dos últimos anos. Acredita-se que a legibilidade de código-fonte afeta diretamente a qualidade do software e a produtividade do desenvolvimento (Aggarwal, Singh and Chhabra, 2002; Tashtoush et al., 2013).

Se o código-fonte for legível, a equipa de manutenção do software terá menos dificuldades quando considerar necessário efetuar algum tipo de alteração ao código que foi escrito por outra pessoa. Por outro lado, código pouco legível é um obstáculo à produtividade e entra em discordância com as boas práticas da engenharia de software.

Fórmulas de legibilidade de texto são utilizadas há dezenas de anos e a sua viabilidade tem vindo a ser comprovada (Buse and Weimer, 2010). Seria, portanto, pertinente pensar que este tipo de avaliação pode ser aplicado a código-fonte, uma vez que código é, à semelhança de texto, uma forma de comunicar informação.

Nesse sentido, ao longo das últimas décadas, vários estudos na área da legibilidade de software têm sido realizados, tendo já sido propostas algumas fórmulas que pretendem estimar o quão legível um pedaço de código é. Ainda que a legibilidade seja um conceito subjetivo, testes conduzidos por vários investigadores demonstram que o nível de assertividade que as fórmulas de legibilidade de software propostas apresenta é bastante aceitável, encorajando a sua utilização (Börstler, Caspersen and Nordström, 2015; Buse and Weimer, 2010).

Estando a legibilidade diretamente ligada à manutenibilidade (Oak, 2014), se for possível encontrar um método que permita obter um *feedback* assertivo e imediato sobre o quão legível é o código a ser escrito, será possível reduzir o tempo gasto na manutenção do software e, por conseguinte, haverá um aumento da manutenibilidade e uma consequente redução dos custos do desenvolvimento do software (Aggarwal, Singh and Chhabra, 2002).

1.2 Objetivos

Tendo em conta o que foi apresentado no subcapítulo anterior, é possível concluir a importância que a legibilidade tem no desenvolvimento de software e o consequente interesse que isso tem para desenvolvedores e organizações.

O grande foco deste trabalho é criar uma ferramenta que permita a avaliação da legibilidade de software em tempo de desenvolvimento, de forma a possibilitar aos desenvolvedores produzirem código mais legível. Este é o grande objetivo deste trabalho, e ele pode ser decomposto em dois objetivos mais específicos, que são o levantamento das fórmulas de legibilidade propostas e a implementação de uma ferramenta de avaliação de legibilidade.

Para o primeiro objetivo, será efetuado um estudo exaustivo das fórmulas de legibilidade propostas. Sendo que já existem vários estudos relevantes na área, irão ser analisadas as várias fórmulas de legibilidade teorizadas[1] e será feita uma comparação entre todas elas, de forma a escolher a fórmula, ou fórmulas, que serão de interesse implementar na solução pretendida.

Numa segunda fase, será desenhada e efetivada a implementação de uma ou mais fórmulas de legibilidade num *plugin* para um IDE. Isto permitirá obter um *feedback* constante e em tempo real da legibilidade do código a ser produzido.

Este segundo objetivo subdivide-se em dois objetivos ainda mais específicos: o primeiro é possibilitar desenvolvedores a repensar ou reestruturar o código com baixa classificação de legibilidade no momento em que as ideias ainda estão bem formuladas nas suas mentes. O segundo é fomentar em iniciantes na área do desenvolvimento de software, nomeadamente alunos do primeiro ano de cursos de informática, a prática da escrita de código legível, oferecendo-lhes um método que lhes permita ter a noção, logo desde o início da sua aprendizagem, das diferenças entre código legível e código pouco legível.

O desenvolvimento desta ferramenta de avaliação de legibilidade de código-fonte pretende ser, por consequência, uma contribuição para as boas práticas da engenharia de software.

1.3 Organização do Documento

Este documento é composto por cinco capítulos. No presente capítulo é feita uma introdução ao problema estudado e são abordados os objetivos que se pretendem alcançar.

No segundo capítulo, intitulado Contexto, é elaborada a análise de valor e o estado da arte. Em primeiro lugar é efetuada, como o próprio nome indica, uma análise aos benefícios e custos que o desenvolvimento da ferramenta proposta traz. No estado da arte descreve-se a legibilidade, fala-se um pouco da história, as vantagens e críticas apontadas às fórmulas de

[1] Neste documento utiliza-se a expressão "fórmulas teorizadas" como sinónimo de "fórmulas propostas".

legibilidade e são apresentadas algumas fórmulas de legibilidade de texto. É neste subcapítulo que são identificadas as fórmulas de legibilidade de código-fonte teorizadas até ao momento.

O terceiro capítulo é designado Análise e Comparação das Fórmulas de Legibilidade e Decisões de Implementação. Nele são detalhadas todas as fórmulas de legibilidade de código-fonte recolhidas. As características avaliadas por cada uma das fórmulas e os respetivos resultados são também discutidos, de forma a possibilitar uma comparação entre todas elas. Esta comparação permitirá escolher quais as fórmulas que serão implementadas na ferramenta de legibilidade proposta. Neste capítulo são realizadas também as decisões sobre o rumo a tomar na implementação da ferramenta, nomeadamente as fórmulas que serão implementadas, a linguagem de programação contemplada e o IDE escolhido.

No quarto capítulo, intitulado Conceção, Implementação e Avaliação, é detalhada a forma como foi implementado o *plugin*. Aqui é especificada a forma como foi feita a análise do código, é detalhado o funcionamento da ferramenta, a estrutura e documentação do código e os testes efetuados ao mesmo. Neste capítulo também são examinadas as especificações de cada uma das fórmulas implementadas. É desenvolvida uma análise aprofundada sobre as variáveis que cada fórmula contempla e é detalhada a implementação da análise de cada uma delas na ferramenta de legibilidade. Ao longo deste capítulo é também demonstrada a forma como foi desenhada a solução implementada e, por fim, é apresenta a avaliação realizada à mesma.

O quinto e último capítulo, designado Conclusões e Trabalho Futuro, apresenta as conclusões gerais obtidas com a realização do trabalho e são apresentadas algumas propostas de investigação e implementação futuras.

2 Contexto

A fase de manutenção de sistemas de software de larga escala tende a durar muito mais tempo do que todas as fases anteriores do ciclo de vida de um software (Aggarwal, Singh and Chhabra, 2002). Será então lógico argumentar que, se for possível reduzir esse tempo, o custo total do desenvolvimento do software, por conseguinte, pode ser reduzido.

Além disso, já foi demonstrada, em estudos, a importância de alunos serem capazes de escrever código legível, fomentando assim o desenvolvimento de código com qualidade, visto esta ser uma das características que mais influencia a qualidade do software. Isto demonstra que, em complemento a um bom ensino e à utilização de exemplos, a disponibilização de ferramentas de legibilidade terá um papel preponderante na conceção de software de qualidade por parte de alunos desde a sua formação, tornando-os, posteriormente, melhores profissionais. (Sampaio, 2017)

A legibilidade pode ser definida como o julgamento humano sobre o quão fácil um texto é percebido (Buse and Weimer, 2010). Transportando isto para a área da engenharia de software, investigadores afirmam que a legibilidade de código-fonte é o julgamento pessoal de um programador sobre o quão fácil um bloco de código é percebido (Buse and Weimer, 2010).

Somando a ideia consensual de que a legibilidade é uma característica essencial e determinante para a qualidade do código, às afirmações de alguns investigadores de que ler código é a componente da manutenção de software que mais tempo consome (Deimel Jr., 1985; Rugaber, 2000), e que boas práticas para o desenvolvimento de código legível são características fulcrais que devem ser adotadas por alunos, de modo a torná-los mais preparados para enfrentar o mercado de trabalho (Sampaio, 2017), seria então interessante encontrar uma forma de notificar em tempo real os desenvolvedores sobre o quão legível é o código que estão a escrever, de modo a reduzir o tempo que terceiros levam a ler e perceber o código escrito pelos primeiros.

Este capítulo serve para dar um contexto global da área em estudo e do estado atual em que ela se encontra. Em primeiro lugar, é feita uma identificação das funções e uma relação das

mesmas com os custos que a sua implementação implicará. Para isso, é realizada, em 2.1 a análise de valor deste trabalho. De seguida, em 2.2 é apresentado o estado da arte, ou seja, procura-se investigar tudo o que já foi estudado na área da legibilidade de software e que seja de interesse para este trabalho. Neste subcapítulo procura-se dar especial ênfase a estudos que envolvam a teorização de fórmulas de legibilidade de software.

2.1 Análise de Valor

Neste capítulo serão apresentados alguns argumentos que procuram sustentar a validade deste estudo para a área. Aqui serão apresentados alguns dos conceitos abordados no módulo de competências "Análise de Valor", procurando-se enquadrar esses mesmos conceitos neste trabalho.

2.1.1 Novo Modelo de Desenvolvimento de Conceito (NCD)

No NCD são definidos cinco pontos-chave. São eles a identificação e análise da oportunidade, a seleção e criação da ideia e a definição do conceito. Cada um destes pontos é tratado nas seguintes secções.

2.1.1.1 Oportunidade

A legibilidade de código-fonte tem um grande impacto na hora de efetuar alterações ao código de um software. A fase de manutenção de software tende a demorar mais que todas as outras fases (Aggarwal, Singh and Chhabra, 2002), e algo que influencia esta demora é o facto de muitas vezes o código escrito não ser tão fácil de compreender como o desejável.

A legibilidade de software afeta também a qualidade do ensino, uma vez que o código escrito por um aluno é muitas vezes lido por colegas de grupo ou professores (Sampaio, 2017). Além disso, o nível de legibilidade de excertos de código de exemplo incluídos em manuais tem um grande impacto sobre a facilidade que iniciantes no desenvolvimento de software, terão ao interpretar estes componentes de ensino (Börstler, Caspersen and Nordström, 2015; Abbas, 2010).

Será então interessante pensar numa forma de manter os desenvolvedores informados sobre a legibilidade do código que estão a escrever, preferencialmente em tempo real. Assim, de modo a tentar oferecer uma ferramenta de auxílio à programação, uma análise tem vindo a ser feita sobre as várias fórmulas de legibilidade de código-fonte propostas, com a finalidade de implementar um *plugin* num IDE que permita obter um *feedback* constante sobre o valor da legibilidade do código escrito.

2.1.1.2 Ideia

De momento, são poucas as ferramentas que permitem avaliar a legibilidade de código-fonte. A aparentemente mais relevante de todas permite até avaliar vários ficheiros Java de uma só

vez, contudo, apenas permite avaliar a legibilidade do código-fonte de um software depois de implementadas as funcionalidades.

Nenhuma das ferramentas permite saber em tempo real qual a estimativa da legibilidade do código a ser escrito. Esta abordagem permitirá aos desenvolvedores avaliarem o código enquanto este está a ser desenvolvido, sem para isso necessitarem de sair do IDE e abrir uma nova aplicação.

2.1.1.3 Conceito

Desenvolvimento de um *plugin* que permita obter um *feedback* sobre a legibilidade do código a ser escrito com base numa ou mais fórmulas de legibilidades que se mostrem válidas.

Este *plugin* permitirá aos desenvolvedores de software saberem em tempo real se o código que estão a escrever é de fácil compreensão ou não, permitindo assim efetuar alterações enquanto as ideias estão ainda bem estruturadas nas suas mentes.

2.1.2 Valor

Os investigadores Susana Nicola, Eduarda Ferreira e João Ferreira dizem que o conceito valor tem vindo a ser definido em diferentes contextos teóricos como necessidade, desejo, interesse, padrão/critério, crenças, atitudes e performances. Dizem ainda que a criação de valor é chave para qualquer negócio. (Nicola, Ferreira and Ferreira, 2012)

Nos seguintes subcapítulos serão apresentadas as propostas de valor que o trabalho realizado oferece.

2.1.2.1 Valor Para o Cliente

Através do desenvolvimento de uma ferramenta capaz de avaliar a legibilidade de software, procura-se com este trabalho encontrar uma forma de permitir aos desenvolvedores escreverem código mais legível. No caso da aplicação desta ferramenta num projeto com várias pessoas, isso irá permitir que o posterior tempo de manutenção do software seja reduzido.

Um outro caso em que a utilização desta ferramenta se pode mostrar útil é com pessoas que estejam a iniciar a aprendizagem da programação, como por exemplo alunos do primeiro ano de cursos como Engenharia Informática. Se estes tiverem forma de obter um *feedback* sobre a legibilidade do código que estão a desenvolver, poderão perceber, logo desde o início, se o código que escrevem é legível ou não. Isto fará com que desde os primeiros estágios de aprendizagem eles consigam ter perceção da diferença entre código pouco e muito legível.

Para isso, será estudada a possibilidade de implementar um *plugin* para um IDE que permita a avaliação da legibilidade do código-fonte em tempo de desenvolvimento.

Código legível vai de encontro às boas práticas do desenvolvimento de software e é um grande passo dado no sentido de os responsáveis pela manutenção de software não perderem demasiado tempo a perceber o que determinado excerto de código faz (Aggarwal,

Singh and Chhabra, 2002). Além disso, código legível é também uma característica que influencia positivamente a qualidade do ensino do desenvolvimento de software (Börstler, Caspersen and Nordström, 2015).

Com a implementação de um *plugin* será possível obter, em tempo real, o valor de legibilidade estimado pelas fórmulas de legibilidade implementadas, sendo que nos casos onde se apresentem classificações de legibilidade reduzidas, o desenvolvedor poderá, no mesmo momento, enquanto ainda tem a ideia do que pretende fazer bem formulada na sua mente, repensar a solução.

2.1.2.2 Valor Percetível

Aqui são apresentadas as diferenças entre os benefícios e os sacrifícios do ponto de vista dos clientes.

Na Tabela 1 é apresentada a perspetiva longitudinal do valor com os benefícios e sacrifícios para o cliente neste contexto.

Tabela 1 – Perspetiva longitudinal de valor

	Benefícios	**Sacrifícios**
Antes da Compra		
Transação	Não apresenta custos de aquisição.	
Após a Compra	Não apresenta custos de manutenção.	Adaptação à programação com *feedback* contínuo.
Após a Utilização	Avaliação contínua da legibilidade do código-fonte de um software em desenvolvimento; Melhorias na legibilidade do código-fonte; Redução do tempo gasto na manutenção de software; Redução de custos na manutenção de software; Melhorias na qualidade do ensino do desenvolvimento de software.	

2.1.3 Proposta de Valor

Desenvolver um *plugin* para um IDE, de forma a obter um *feedback* contínuo e em tempo real sobre a legibilidade do código-fonte a ser escrito. Desta forma, será possível oferecer um método que fomente o desenvolvimento de software legível, levando à redução do tempo de manutenção de software e, por conseguinte, reduzir os custos do desenvolvimento do mesmo. Este *plugin* poderá também ser utilizado por alunos, de modo a que estes sejam capazes de perceber se a forma como escrevem e estruturam código segue um padrão que o torna legível, facilitando assim o trabalho de professores e colegas de grupo quando tiverem de analisar código escrito por esses alunos (Sampaio, 2017).

Poucas aplicações do género estão disponíveis, pelo que a implementação desta ferramenta aparenta ser uma das pioneiras na área.

2.1.4 Quadro de Modelo de Negócios

A Figura 1 mostra o *Business Model Canvas*, ou Quadro de Modelo de Negócios, elaborado para este trabalho.

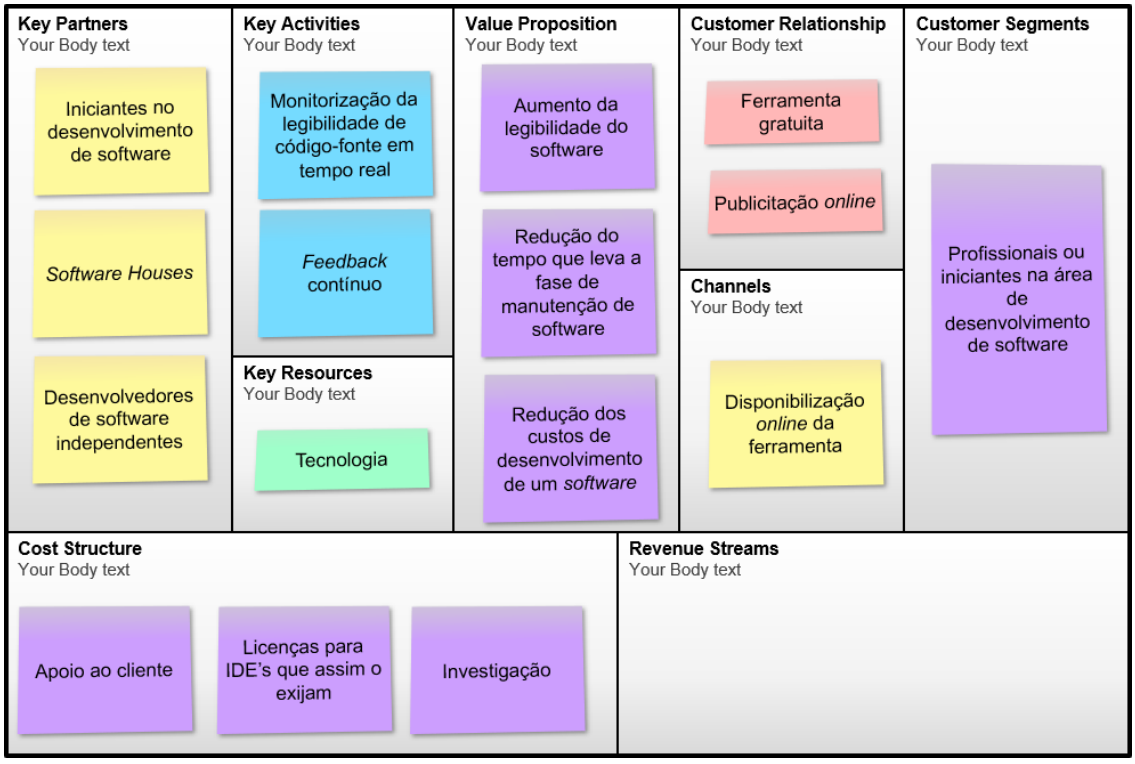

Figura 1 – Quadro de Modelo de Negócios (Canvas)

A Figura 1 permite concluir que a ferramenta proposta tem como parceiros todos os desenvolvedores independentes e empresas de desenvolvimento de software. Também são considerados parceiros, iniciantes ao desenvolvimento de software, como por exemplo alunos do primeiro ano de cursos da área da informática.

Uma ferramenta capaz de avaliar a legibilidade de software será desenvolvida, recorrendo às fórmulas de legibilidade de software teorizadas. Isto permitirá aos desenvolvedores terem um *feedback* contínuo sobre a legibilidade do código que está a ser escrito, permitindo assim um julgamento mais acertado sobre as decisões a tomar.

Este *plugin* será disponibilizado gratuitamente, pelo que não existirá fluxo de receitas.

2.2 Estado da Arte

A legibilidade começou a ser estudada há alguns séculos atrás, tendo sido as primeiras fórmulas de legibilidade propostas em finais do século XIX (My Byline Media, 2018b). Alguns anos depois do aparecimento da informática e das linguagens de programação, investigadores começaram a estudar fórmulas e métricas de avaliação de legibilidade de código-fonte. Estas fórmulas, tendo começado a aparecer no início do século XXI, já conseguem dar resultados bastante conclusivos sobre a legibilidade do software, resultados esses que entram em concordância com a opinião de profissionais da área no que toca às características que um excerto de código deve ter para ser considerado legível (Buse and Weimer, 2010).

Este capítulo serve para fazer o levantamento do que já foi estudado na área da legibilidade de software. Nele são abordados alguns dos estudos que mais impacto tiveram nesta área, no entanto, o foco principal são aqueles onde efetivamente foram propostas fórmulas de legibilidade de software.

Sendo assim, em 2.2.1 é apresentada uma introdução sobre o que é a legibilidade de um modo geral, a sua importância e algumas limitações que ela apresenta. É, desta forma, abordado o papel da legibilidade na análise de texto.

Alguns exemplos de fórmulas de legibilidade de texto são apresentados em 2.2.2, com o intuito de possibilitar o enquadramento sobre o que é avaliar algo tão subjetivo como a legibilidade e também porque algumas pesquisas na área da legibilidade de código-fonte fazem referência a estas fórmulas. Aqui também serão abordadas as limitações e críticas apontadas às fórmulas de legibilidade.

Em 2.2.3, é então introduzido o conceito de legibilidade de software. Em primeiro lugar é apresentada uma definição do que é a legibilidade de código-fonte e em seguida são abordados vários pontos em torno dela. Neste subcapítulo são apresentadas as fórmulas de legibilidade de software propostas até à data e são superficialmente abordados alguns estudos efetuados na área da legibilidade, mas que não resultaram em propostas de fórmulas. Também são referidas as ferramentas que foram desenvolvidas com vista a avaliar a legibilidade de código-fonte.

2.2.1 Legibilidade

O texto escrito é utilizado há milhares de anos para comunicar opiniões, conhecimento, informação, entre muitas outras coisas. Porém, a forma como se apresentam as ideias – a linguagem usada, o comprimento das frases, o uso da gramática – pode ser um impedimento à boa compreensão do texto por parte do leitor. As fórmulas de legibilidade foram a forma encontrada para tentar resolver este problema, conseguindo algumas delas classificar muito bem quando comparadas com outras medidas psicométricas, como por exemplo o teste de leitura (Sampaio, 2017).

2.2.1.1 O que é a Legibilidade?

Em traços gerais, legibilidade é o julgamento humano sobre o quão fácil um texto é percebido (Buse and Weimer, 2010). Ao longo da história, contudo, foram sendo dadas diferentes propostas para a definições de legibilidade.

Nas palavras do professor de psicologia George Klare (1963), legibilidade é "a facilidade da perceção ou compreensão relacionada com o estilo de escrita". Esta definição apenas se foca no estilo de escrita, e ignora outros pontos importantes, tais como o conteúdo, a coerência, e a organização do texto (DuBay, 2004).

Em 1969, o criador da fórmula de legibilidade SMOG, G. Harry McLaughlin, descreveu a legibilidade como sendo "o grau que uma determinada classe de pessoas acha certa matéria de leitura atrativa e compreensível" (DuBay, 2004).

Possivelmente, a definição mais detalhada seja a dada por Edgar Dale e Jeanne Chall, em 1949, que define legibilidade como sendo "a soma total (incluindo todas as interações) de todos aqueles elementos que se encontram dentro de um dado pedaço de material impresso que afeta o sucesso que um grupo de leitores tem com ele. O sucesso é a medida em que os leitores entendem o texto, o leem a uma velocidade ideal e o acham interessante" (DuBay, 2004).

2.2.1.2 História da Legibilidade

O estudo da legibilidade começou no final do século XIX nos Estados Unidos da América e tem continuado até aos dias de hoje (My Byline Media, 2018b). Os estudos desta área podem ser divididos em duas categorias: os estudos da legibilidade clássicos e os novos estudos da legibilidade (DuBay, 2004).

Os estudos da legibilidade clássicos abordam os estudos pioneiros da legibilidade, que começaram no final do século XIX e foram até aos anos 40 do corrente século, com a publicação das populares fórmulas de Flesch e Dale-Chall. Durante este período, a preocupação das editoras, educadores e professores estava centrada em encontrar métodos práticos para fazer corresponder a dificuldade de textos com as habilidades de leitores. (DuBay, 2004)

Por seu turno, os novos estudos da legibilidade tiveram início nos anos 50. Novos avanços transformaram o estudo da legibilidade, incluindo um novo teste de compreensão de leitura e

as contribuições da linguística e psicologia cognitiva. Os investigadores exploraram de que forma o interesse dos leitores, a motivação e o conhecimento anterior afetam a legibilidade. Por sua vez, estes estudos permitiram a criação de novas e melhores fórmulas de legibilidade. (DuBay, 2004)

2.2.1.3 Importância

A legibilidade mostra-se presente nas mais distintas áreas. Pode-se falar em legibilidade em relação à escrita de livros, de artigos, de *newsletters* e de contratos de trabalho, só para enumerar alguns exemplos.

Sendo a legibilidade definida como o julgamento humano sobre o quão fácil um texto é percebido, como foi apresentado em 2.2.1.1, torna-se mais ou menos clara a sua relevância. É impraticável o leitor necessitar de reler várias vezes um texto, porque a sua construção frásica é de difícil perceção, ou porque as palavras escolhidas não foram as mais indicadas para os leitores-alvo. Numa análise descomedida, se um livro para crianças for escrito da mesma forma que um artigo científico, subentende-se que a sua leitura e consequente compreensão por parte do público-alvo tornar-se-á demasiado penosa, senão impossível.

Uma outra demonstração da importância da legibilidade é o facto de instituições governamentais dos Estados Unidos da América utilizarem métodos de medição de legibilidade para avaliar se os seus documentos alcançam o valor padrão de legibilidade exigido (Buse and Weimer, 2010).

Legibilidade é, portanto, um fator-chave na comunicação visual no meio que nos cerca. Se a forma que um texto é escrito não é coerente com as capacidades de leitura do público-alvo, este texto deve ser repensado e reestruturado.

2.2.2 Fórmulas de Legibilidade de Texto

As fórmulas de legibilidade são um dos métodos utilizados para medir ou prever o nível de dificuldade de um texto. Estas fórmulas surgiram para possibilitar a classificação da legibilidade. O resultado obtido depende do tipo de abordagem de cada fórmula, mas por norma é uma estimativa do número de anos de escolaridade que uma pessoa necessita ter para compreender um dado texto, ou seja, o nível mínimo de escolaridade (Zamanian and Heydari, 2012; Kondru, 2006).

A primeira fórmula de legibilidade de texto apareceu em 1923 e foi proposta por Bertha A. Lively e Sidney L. Pressey (Zamanian and Heydari, 2012; DuBay, 2004). Em 1980 já existiam mais de 200 fórmulas de legibilidade (DuBay, 2004) e em 2000 já existiam cerca de 1000 estudos sobre a validade e o mérito das fórmulas de legibilidade de texto (Klare, 2000).

Há uma imensidão de fórmulas de legibilidade disponíveis. Cada uma usa a sua abordagem própria para avaliar o texto. A *Flesch Reading Ease Readability Formula* (FRES), o Índice de FOG, o *Automated Readability Index* (ARI), o SMOG, o Índice de Coleman-Liau e o *Flesch-Kincaid Grade Level* são apenas algumas das muitas fórmulas de legibilidade de texto, estando

esta última em uso há mais de 50 anos e tendo sido integrada em editores de texto populares como o Microsoft Office Word. Além disso, o *Flesch-Kincaid Grade Level* é um teste de legibilidade padrão em algumas instituições governamentais nos Estados Unidos da América. (Buse and Weimer, 2010)

2.2.2.1 Algumas Fórmulas de Legibilidade

De forma a ter uma visão geral sobre quais as características do texto que estas fórmulas abordam e qual o seu fundamento geral, de seguida são apresentadas e analisadas superficialmente algumas fórmulas de legibilidade de texto.

Tabela 2 – Algumas fórmulas de legibilidade de texto (adaptado de (Sampaio, 2017))

Ano	**Autor(es)**	**Fórmula**
1948	Rudolph Flesch	*Flesch Reading Ease Readability Formula* (FRES)
1952	Robbert Gunning	Índice de FOG
1967	R.J. Senter e E.A. Smith	*Automated Readability Index* (ARI)
1969	Harry McLaughlin	SMOG
1975	J. Peter Kincaid	*Flesch-Kincaid Grade Level*
1975	Meri Coleman e T.L. Liau	Índice Coleman-Liau

Na Tabela 2 estão expostos alguns exemplos de fórmulas de legibilidade de texto. Mostra-se pertinente abordar estas fórmulas por vários motivos, sendo o principal o facto de certos investigadores se terem apoiado, ou até mesmo baseado, em algumas delas para desenvolver os seus estudos na área da legibilidade de software (Börstler, Caspersen and Nordström, 2015).

Considerada como uma das mais antigas e precisas fórmulas de legibilidade, a fórmula FRES avalia o nível de escolaridade mínima que o leitor deve ter para perceber um dado texto. Tornou-se a fórmula de legibilidade padrão em muitas agências governamentais nos EUA, incluindo o Departamento de Defesa dos Estados Unidos (My Byline Media, 2018e). A facilidade de leitura (*Reading Ease* - RE) é calculada através da média do comprimento de frases (*Average Sentence Length* - ASL) e do número médio de sílabas por palavra (*Average Number of Sylables per Word* - ASW) (My Byline Media, 2018e). Mais de meio século depois, acabou por ser um pilar de sustentação para o desenvolvimento da fórmula de legibilidade de código-fonte *Software Readability Ease Score* (SRES) (Börstler, Caspersen and Nordström, 2007).

$$RE = 206.835 - (1.015 \times ASL) - (84.6 \times ASW) \quad (1)$$

, onde RE é uma posição numa escala entre 0 e 100, sendo o intervalo entre 0 e 30 considerado muito difícil e o intervalo de 70 a 100 considerado acessível a adultos. ASL é o comprimento médio das frases medido em palavras ($\frac{número\_de\_palavras}{número\_de\_frases}$) e ASW é a média de sílabas por palavra ($\frac{número\_de\_sílabas}{número\_de\_palavras}$) (Sampaio, 2017).

O Índice de Fog é uma métrica desenvolvida por Robert Gunning que calculada a legibilidade adicionando a média do comprimento de frases à percentagem de palavras difíceis (*Percentage of Hard Words* - PHW).

$$FOG = 0.4(ASL + PHW) \quad (2)$$

, onde FOG é o índice de FOG, ASL é a média do comprimento das palavras e PHW é a percentagem de palavras difíceis (com mais de duas sílabas) (Sampaio, 2017).

O *Automated Readability Index* é um índice de legibilidade que se baseia nos rácios de dificuldade de frases e de palavras. Ao contrário de outras fórmulas, esta ignora a contagem silábica no seu cálculo. Ao invés, confia no fator de caracteres por palavra (My Byline Media, 2018c).

Esta fórmula foi desenvolvida em 1967 especificamente para a marinha norte-americana (*United States Navy*) e recorre à dificuldade das palavras e à dificuldade das frases para calcular o valor de legibilidade. A dificuldade das palavras é calculada através do número de caracteres por palavra e a dificuldade das frases é calculada recorrendo ao número de palavras existentes numa frase. (Sampaio, 2017)

$$Grau = 0.50X1 + 4.71X2 - 21.43 \quad (3)$$

, onde X1 é o número de palavras por frase ($\frac{número\_de\_palavras}{número\_de\_frases}$) e X2 é o número de caracteres por palavra ($\frac{número\_de\_caracteres}{número\_de\_palavras}$).

O *SMOG Grading*, ou simplesmente SMOG, é uma medida de legibilidade que avalia o tempo (em anos de escolaridade) necessários por uma pessoa para ler e compreender um texto. Harry McLaughlin criou esta fórmula como melhoria em relação a outras fórmulas de legibilidade (My Byline Media, 2018f). McLaughlin verificou a existência de uma correlação negativa quase perfeita entre a quantidade de polissílabos e as medições obtidas por si ao longo dos seus estudo (Sampaio, 2017).

$$SMOG = 3 + \sqrt{número\_de\_polissílabos} \quad (4)$$

, onde SMOG é o valor de legibilidade e número_de_polissílabos é, como o nome indica, o número de polissílabos contidos no texto.

A *Flesch-Kincaid Grade Level Readability Formula* surgiu da necessidade da marinha norte-americana de avaliar a dificuldade dos seus manuais de treinamento (Colmer, 2018; Sampaio, 2017). Apoiado por Fishburne, Rogers e Chissom, John P. Kincaid desenvolveu esta fórmula que teve como base outras três fórmulas de legibilidade já existentes: ARI, Índice de FOG e FRES (Sampaio, 2017).

$$Grau = (0.39 \times ASL) + (11.8 \times ASW) - 15.59 \quad (5)$$

, onde ASL é o comprimento médio das frases medido em palavras ($\frac{número\_de\_palavras}{número\_de\_frases}$) e ASW é a média de sílabas por palavra ($\frac{número\_de\_sílabas}{número\_de\_palavras}$) (Sampaio, 2017).

O Índice Coleman-Liau é um teste de legibilidade proposto pelos linguistas Meri Coleman e T. L. Liau que procura avaliar a usabilidade de um texto. Coleman disse que criou a fórmula como uma de muitas formas de ajudar o Escritório de Educação dos Estados Unidos a calibrar a legibilidade de todos os livros escolares do sistema público (My Byline Media, 2018d; Sampaio, 2017). A fórmula proposta é a seguinte:

$$CLI = 0.0588 \times 448L - 0.296S - 15.8 \quad (6)$$

, onde CLI é o valor do índice, L é o número médio de letras por 100 palavras e S é o número médio de frases por 100 palavras.

2.2.2.2 Limitações e Críticas às Fórmulas de Legibilidade

Devido à sua possível falibilidade, alguns investigadores e autores apontam limitações e críticas à utilização das fórmulas de legibilidade. Em 2012, uma revisão da literatura levada a cabo por Mostafa Zamanian e Pooneh Heydari permitiu constatar a existência de um grupo de autores que se posicionam a favor das fórmulas de legibilidade e um outro grupo de autores que considera que estas fórmulas apresentam diversas limitações (Sampaio, 2017; Zamanian and Heydari, 2012).

Uma crítica apontada às fórmulas de legibilidade, é o facto de que "legibilidade" é um conceito diferente de "capacidade de perceber". No caso de não se saber o público-alvo, a utilização de fórmulas de legibilidade pode não se mostrar eficaz. Em acréscimo, as fórmulas de legibilidade não são capazes de avaliar o conhecimento prévio que o leitor-alvo tem sobre o tema em causa (My Byline Media, 2018a).

Muitas grandes obras da literatura falham nos testes de legibilidade devido à sua construção frásica e ao léxico utilizado, porém tal não significa que esses trabalhos tenham qualidade de escrita inferior (My Byline Media, 2018a).

Outro problema apontado às fórmulas de legibilidade, é o vasto número de fórmulas existente. Isto faz com que a probabilidade de obter grandes variações nos resultados para um mesmo texto seja elevada. (My Byline Media, 2018a).

Uma crítica que J. Peter Kincaid, criador do teste de legibilidade Flesch-Kincaid, assinalou, é que se podem reorganizar algumas palavras de um texto de forma aleatória e continuar a obter o mesmo valor de legibilidade do texto original (McClure, 1987).

Richard Kern acusa as fórmulas de legibilidade de serem contraprodutivas quando utilizadas como guias de reescrita de material avaliado com baixo valor de legibilidade por alguma fórmula (Kern, 1980).

Uma outra crítica feita pelos mais puristas, é a aplicação de fórmulas matemáticas a literatura. Esta ideia por si só não favorece a aceitação de alguns autores e investigadores (My Byline Media, 2018a).

De um modo geral, as críticas apontadas não retiram a utilidade às fórmulas nem à sua utilização, mas permitem reconhecer as suas limitações (Sampaio, 2017).

2.2.3 Legibilidade de Software

Em 2011, Daryl Posnett, Abram Hindle e Premkumar Devanbu definiram legibilidade de software como sendo a propriedade que influencia o quão facilmente um excerto de código consegue ser lido e compreendido (Posnett, Hindle and Devanbu, 2011).

Comunicar instruções a uma máquina pode ser comparado à comunicação de informação em formato de texto. Um software, em termos de conjunto de instruções, é um tipo especial de texto com semântica e regras sintáticas próprias, definidas pela linguagem de programação utilizada (Abbas, 2010). Na escrita de código, tal como na escrita de texto, existem diversas formas de apresentar a mesma ideia, sendo umas mais apropriadas que outras. Se uma instrução, ainda que conforme, for demasiado complexa, seguindo um fluxo de ideias muito embaraçado e pouco lógico, provavelmente a sua legibilidade e consequente compreensibilidade serão reduzidas.

Existem vários fatores que influenciam a legibilidade do software, tendo sido alguns deles já estudados por diversos autores e ponderados nas fórmulas de legibilidade de código-fonte propostas. Investigadores já utilizaram abordagens em que procuram seguir a lógica de fórmulas de legibilidade de texto e transformá-las em fórmulas de legibilidade de software, recorrendo à ideia de que o código de um programa pode ser tratado como texto em linguagem natural (Börstler, Caspersen and Nordström, 2015). Esses investigadores dizem que se for possível encontrar as equivalências para o software das características de texto avaliadas por uma fórmula, então podem ser desenvolvidas fórmulas equivalentes para avaliar a legibilidade de um programa (Abbas, 2010).

Outros já propuseram, por exemplo, a introdução de medidas de complexidade para procurar estimar a legibilidade do software, defendendo que, não sendo conceitos análogos, a legibilidade e complexidade estão intimamente inter-relacionadas (Posnett, Hindle and Devanbu, 2011).

2.2.3.1 Legibilidade, "Leiturabilidade" e Complexidade

Quando se fala em legibilidade de software, é difícil não mencionar outros pontos que estão de uma forma ou de outra ligados a este conceito, pontos esses que por vezes podem ser confundidos. Sendo assim, mostra-se necessário fazer uma distinção entre os conceitos de legibilidade, "leiturabilidade" e complexidade de software.

Foi apresentada em 2.2.3 a definição de legibilidade de software. Como sugere a definição apresentada, verifica-se que legibilidade está relacionada com as regras semânticas e sintáticas do texto – no caso, do código – e respetiva construção. Por seu turno, a "leiturabilidade" está relaciona com aspetos visuais do texto, ou seja, características como o tipo de letra, ou o espaçamento entre parágrafos, por exemplo.

Importa referir que o termo "leiturabilidade" não está contemplado no dicionário da língua portuguesa, sendo o seu uso desencorajado (Rocha, 2009), no entanto, é a tradução mais aproximada do termo em inglês *legibility*.

Já complexidade de software é uma propriedade do código intrínseca ou essencial baseada no domínio do problema e não pode ser evitada por completo em todos os cenários, ao contrário da legibilidade, que é uma propriedade acidental que pode ser evitada independentemente do domínio ou da complexidade do problema (Börstler, Caspersen and Nordström, 2007; Abbas, 2010). Legibilidade é uma medida estática baseada nos elementos individuais e independentes do código, como por exemplo os identificadores, declarações, comentários, indentação e estilo do programa, ao passo que a complexidade depende tanto dos componentes estáticos como das interações dinâmicas entre os vários componentes do programa (Abbas, 2010).

2.2.3.2 A Importância da Legibilidade no Ciclo de Vida do Software e no Ensino

As fórmulas de legibilidade têm vindo a ser utilizadas desde meados do século XIX para avaliar a dificuldade de leitura de textos e a sua importância e eficácia têm vindo a ser comprovadas ao longo do tempo, porém só mais recentemente se começou a estudar a sua aplicação para a avaliação da legibilidade de código-fonte.

Estudos indicam que 70% do custo do ciclo de vida de um software centra-se na fase de manutenção (Boehm and Basili, 2001). Há quem garanta que a legibilidade do código e da própria documentação têm um papel preponderante na duração e consequentes custos da manutenção de um projeto de software (Aggarwal, Singh and Chhabra, 2002). De todas as atividades que englobam a fase da manutenção, ler código é a que mais tempo consome, segundo pesquisas feitas (Rugaber, 2000; Deimel Jr., 1985). Uma outra evidência da importância da legibilidade do código-fonte, foi a proposta de Elshoff e Marcotty de adicionar uma nova fase no ciclo de desenvolvimento, onde o objetivo seria tornar o código mais legível (Elshoff and Marcotty, 1982), o que vai, de certa forma, de encontro com a sugestão de Knight e Myers de que uma fase da inspeção de software deveria ser uma verificação da legibilidade, de forma a certificar a manutenibilidade, portabilidade e reusabilidade do código (Knight and Myers, 1991). Também Nuzhat J. Haneef sugeriu uma medida que entra em concordância com as medidas de Elshoff e Marcotty e de Knight e Myers. Esta investigadora sugeriu adicionar

um grupo de trabalho dedicado à legibilidade e à documentação à equipa de desenvolvimento de software (Haneef, 1998).

Estes são alguns aspetos que comprovam a importância da legibilidade ao nível do mercado de trabalho, no entanto, dando alguns passos atrás no nível de profissionalização, pode-se verificar que a legibilidade também é de suma importância durante a aprendizagem do desenvolvimento de software. É natural que um profissional interprete o código de uma forma diferente de, por exemplo, um aluno de um curso de informática, e que tenha mais facilidade em compreender o que determinado excerto de código pretende fazer. A maioria das fórmulas de legibilidade propostas não tem em conta as diferenças de conhecimento que os desenvolvedores têm mediante o nível em que se encontram na carreira, no entanto já foram feitos alguns estudos neste sentido e até já foi proposta uma fórmula que se foca exclusivamente na legibilidade dos excertos de código incluídos em manuais de introdução à programação (Börstler, Caspersen and Nordström, 2015; Abbas, 2010).

A legibilidade de software mostra-se assim importante de duas formas distintas. Por um lado, se as organizações que produzem excertos de código de exemplo, expostos em sites, blogues ou livros, conseguirem escrever esse mesmo código de forma legível para os iniciantes no desenvolvimento de software, então estes terão mais facilidade durante a sua fase de aprendizagem. Por outro lado, se os mesmos conseguirem ter uma noção da legibilidade do código que produzem, poderão desde o início do seu percurso adquirir uma maior sensibilidade e visão crítica sobre a qualidade do código que escrevem, facilitando assim o seu progresso enquanto desenvolvedores de software e podendo-os tornar melhores profissionais mais rapidamente.

2.2.3.3 Fórmulas de Legibilidade de Software

À semelhança do que aconteceu com o texto em linguagem natural, a aplicação de fórmulas de legibilidade em código de software também tem vindo a ser estudada.

Várias fórmulas e métricas foram sendo propostas no decorrer das últimas duas décadas, sendo os resultados de algumas delas bastante promissores. Ainda que legibilidade seja um conceito subjetivo, os resultados obtidos por algumas das fórmulas propostas entram em concordância com a opinião de vários profissionais da área sobre o que é considerado um excerto de código legível, pelo que se pode olhar para elas como possíveis auxiliares à programação (Buse and Weimer, 2010; Tashtoush et al., 2013).

A Tabela 3 apresenta as fórmulas de legibilidade de software propostas ao longo dos últimos anos.

Tabela 3 – Fórmulas de legibilidade de código-fonte

Ano	**Autores**	**Nome**
2002	K.K. Aggarwal, Y. Singh e J.K. Chhabra	*Comments Ratio*

Ano	Autores	Nome
2011	Daryl Posnett, Abram Hindle e Prem Devanbu	Posnett Hindle Devanbu (PHD)
2012	Rajendar Namani, Kumar J	*Code Readability*
2013	Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi e Maryan Yatim	*Impact of Programming Features on Code Readability* (IPFCR)
2015	Jürgen Börstler, Michael E. Caspersen e Marie Nordström	*Software Readability Ease Score* (SRES)
2017	Weifeng Xu, Dianxiang Xu e Lin Deng	*Word Concreteness and Memory Retention* (WCMR)

Como mostra a Tabela 3, a primeira fórmula de legibilidade de código-fonte foi proposta no ano de 2002. Desde então, foram efetuados vários outros estudos na área da legibilidade de software, resultando alguns deles nas fórmulas que se apresentam nesta tabela.

Verifica-se que a Tabela 3 apresenta um reduzido número de fórmulas. Isto deve-se ao facto de que a legibilidade de software é ainda uma área pouco explorada (Sampaio, 2017), apesar do crescente interesse verificado nos últimos anos. É, contudo, motivador constatar que novas fórmulas foram propostas em anos recentes. Também é interessante observar que as várias fórmulas de legibilidade propostas abordam características bastante distintas do código. Tal será aprofundado nos próximos capítulos.

De notar que a Tabela 3 apenas se foca em apresentar as fórmulas de legibilidade de software propostas. Existem diversos estudos nesta área que levaram à criação de modelos ou métricas de legibilidade, mas que não resultaram em fórmulas propriamente ditas. Também importa referir que as fórmulas apresentadas nesta tabela se focam unicamente na legibilidade. Fórmulas que avaliam, por exemplo, a complexidade do software, como é o exemplo das fórmulas desenvolvidas por Maurice Halstead (mais informação em 3.1.2), não foram consideradas para esta tabela.

2.2.3.4 Outros Estudos de Interesse Efetuados na Área da Legibilidade de Software

Diversos estudos na área da legibilidade de software foram desenvolvidos. Estes variam o seu foco de estudo entre vários temas relacionados com a legibilidade do código. Alguns procuram mostrar o peso que determinadas características do código-fonte acarretam na legibilidade, enquanto outros, por exemplo, procuram provar as vantagens da adição de novas fases ao ciclo de vida do software.

De seguida, são listados alguns estudos que, não introduzindo novas fórmulas de legibilidade, se mostram interessantes referir, já que alguns deles foram citados em artigos onde foram propostas fórmulas, ou até mesmo são pilares de sustentação de fórmulas que vieram a ser desenvolvidas em anos subsequentes.

- *Beauty and the Beast Toward a Measurement Framework for Example Program Quality* (2007) (Börstler, Caspersen and Nordström, 2007)
  Neste estudo conduzido por Jürgen Börstler, Michael E. Caspersen e Marie Nordström é discutida a compreensibilidade de programas exemplo de um ponto de vista cognitivo e de medição. Os investigadores argumentam que medidas comuns de software não são suficientes para capturar todos os aspetos relevantes da compreensibilidade do software.

  Os autores acabam por propor e discutir uma nova medida de legibilidade de software chamada SRES. Mais tarde, em 2015, no artigo *"Beauty and the Beast: on the readability of object-oriented example programs"*, os mesmos autores acabam por desenvolver uma fórmula de legibilidade baseada nestas medidas.

- *The Effect of Identifier Naming on Source Code Readability and Quality* (2009) (Butler, 2009)
  Estudo efetuado por Simon Butler, onde se procura relacionar a qualidade dos nomes dados aos identificadores e a facilidade na compreensão do software. O autor propõe-se a descrever os métodos utilizados para avaliar a qualidade dos identificadores e a propor o desenvolvimento de uma forma de compreensão mais sofisticada da qualidade de nomes de identificadores.

- *Relating Identifier Naming Flaws and Code Quality: An Empirical Study* (2009) (Butler et al., 2009) e *Exploring the Influence of identifier names on code quality: An Empirical Study* (2010) (Butler et al., 2010)
  Nestes dois estudos levados a cabo por Simon Butler, Michel Wermelinger, Yijun Yu e Helen Sharp é abordada a natureza da qualidade dos identificadores, se os identificadores aderem ou não ao estilo das instruções e a sua relação com a qualidade e a legibilidade.

- *Learning A Metric for Software Readability* (2010) (Buse and Weimer, 2010)
  Este estudo levado a cabo por Raymond Buse e Westley Weimer é uma melhoria ao estudo "*A Metric for Software Readability*", publicado pelos mesmos investigadores em 2008. É uma das publicações mais citadas nos estudos sobre a legibilidade de software, uma vez que é neste artigo que é descrita a primeira métrica geral de legibilidade de código-fonte. Esta métrica foi a base de sustentação para o desenvolvimento da fórmula de legibilidade PHD, em 2011.

- *Refactoring Code To Increase Readability And Maintainability: A Case Study* (2014) (Dibble II and Gestwicki, 2014)

Estudo levado a cabo por Christopher Dibble II e Paul Gestwicki, onde mostram o impacto que a refatoração de código tem na legibilidade e manutenibilidade do software. Os investigadores utilizaram um videojogo educativo com 2823 linhas de código. Este jogo foi desenvolvido recorrendo à linguagem de programação C#. A refatoração do código ocorreu em duas fases, sendo a primeira a utilização de um aplicativo que analisa o código, identifica potenciais problemas e dá recomendações para a resolução dos mesmos. A segunda fase foi a execução de uma análise manual a todos os ficheiros de código, com vista a identificar oportunidades de refatoração não identificadas pela ferramenta utilizada na primeira fase.

- *Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability* (2014) (Wang, Pollock and Vijay-Shanker, 2011)
  Estudo realizado por Xiaoran Wang, Lori Pollock, e K. Vijay-Shanker, onde os autores constroem o que, segundo eles, é o primeiro sistema de inserção de linhas em branco em código-fonte com a finalidade de aumentar a legibilidade do código e localizar pontos para a inserção de documentação interna.

- *Analyzing Program Readability Based on WordNet* (2015) (Liu, Sun and Duan, 2015)
  Estudo desenvolvido por Yangchao Liu, Xiaobing Sun e Yucong Duan, onde sugerem a utilização do WordNet[2] para analisar a legibilidade de código. Os autores focaram-se em analisar a legibilidade de um programa utilizando a correspondência de sinónimos do WordNet, de forma a ultrapassar o problema da ambiguidade semântica em estudos anteriores.

- *Impact and Comparison of Programming Constructs on JAVA and C# Source Code Readability* (2015) (Batool et al., 2015)
  Estudo efetuado por Aisha Batool, Muhammad Habib ur Rehman, Aihab Khan e Amsa Azeem, onde os autores selecionam algumas partes do código que afetam a legibilidade e calculam o seu valor para as linguagens C# e Java. Os resultados obtidos com os testes efetuados sugerem que Java é uma linguagem de programação mais legível que C#. Estes resultados estão em concordância com um inquérito que os autores realizaram com alguns profissionais na área da programação (Batool et al., 2015).

2.2.4 Ferramentas Para a Avaliação da Legibilidade de Software

Como foi mencionado anteriormente, a legibilidade de software mostra-se importante, quer a nível profissional, quer a nível académico. Como tal, a utilização de ferramentas que recorram à utilização das fórmulas ou modelos de legibilidade pode ser uma mais valia para a área da engenharia de software.

[2] Base de dados léxica de inglês, onde substantivos, verbos, adjetivos e advérbios são agrupados em conjuntos de sinónimos cognitivos, cada qual expressando um conceito distinto. É uma ferramenta que pode ser aplicada na linguística computacional e em processamento de linguagem natural. (Princeton University, 2018)

Existe um grande leque de opções no que toca a ferramentas que aplicam as fórmulas propostas para a avaliação da legibilidade de texto. O mesmo não acontece com a avaliação de código-fonte. Algumas ferramentas foram já desenvolvidas nesse sentido, porém, são poucas e apresentam ainda várias limitações.

De seguida, serão apresentadas e analisadas as ferramentas que foram desenvolvidas. Procura-se com isto perceber quais as falhas que elas apresentam e assim poder projetar uma solução que as colmate.

2.2.4.1 Source Code Readability Metric

Esta ferramenta foi desenvolvida por Raymond P. L. Buse, um dos autores de "*Learning a Metric for Code Readability*", estudo publicado em 2010 onde é apresentada uma métrica de legibilidade de software.

O *Source Code Readability Metric* recorre a ferramentas e recursos externos para efetuar a recolha da informação necessária para calcular a legibilidade. Utiliza a *Application Programming Interface* (API) Weka[3] para realizar tarefas de mineração de dados e o analisador de código FindBugs[4]. O autor implementa nesta ferramenta o modelo de legibilidade que foi proposto no estudo referido.

Esta ferramenta apresenta diversas limitações. Uma delas é o facto de não apresentar uma interface do utilizador (IU). Todo o funcionamento ocorre através da linha de comandos, pelo que a sua utilização é pouco intuitiva.

Uma consequente desvantagem que a falta de uma IU acarreta, é o facto de não ser possível alterar linhas de código anteriores. Isto é, ao introduzir o código desejado, a inserção de uma nova linha é efetuada premindo a tecla *Enter*. Isto faz com que o cursor da consola inicie numa nova linha, impossibilitando a alteração das linhas previamente inseridas.

Por fim, outro problema que a ferramenta apresenta, é ela não ser capaz de carregar um ficheiro Java. Qualquer excerto de código a ser analisado deve ser inserido diretamente na consola.

A Figura 2 apresenta a utilização do *Source Code Readability Metric*, recorrendo à linha de comendos.

[3] Coleção de algoritmos de *machine learning*.
[4] Ferramenta para análise estática de programas escritos em Java. Analisa o software em busca de defeitos no código. (Sprunck, 2012)

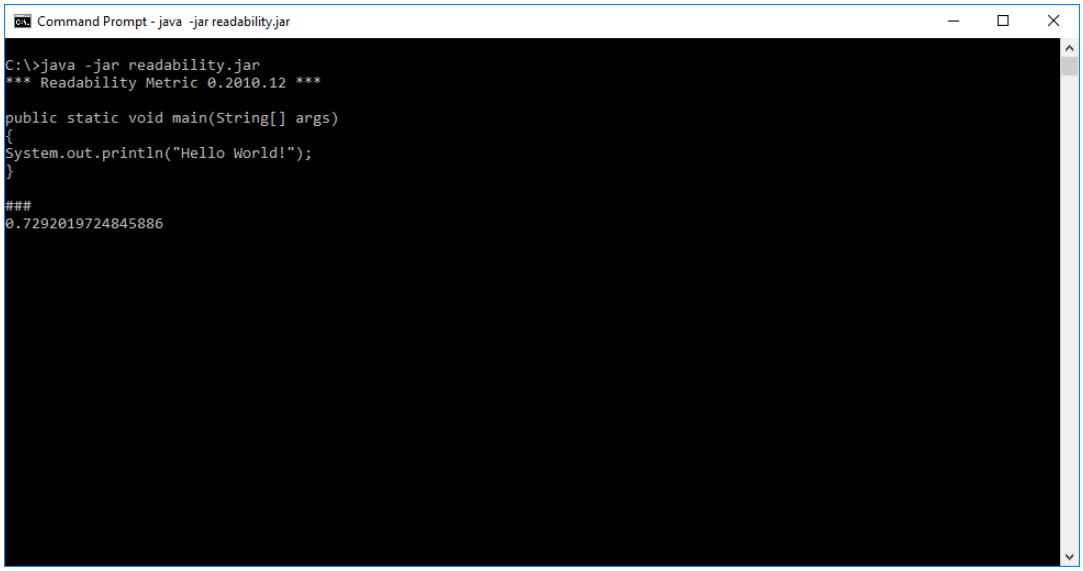

Figura 2 – Funcionamento da Source Code Readability Metric

2.2.4.2 Properties of "Good" Java Examples (POGJE)

Nadeem Abbas, na sua tese de mestrado intitulada *"Properties of 'Good' Java Examples"*, implementa uma ferramenta intitulada POGJE. Nesta ferramenta, Abbas aplica algumas medidas de Halstead e também a métrica de legibilidade SRES, proposta por Jürgen Börstler, Michael E. Caspersen e Marie Nordström em 2007 (Börstler, Caspersen and Nordström, 2007) (de notar que estes investigadores voltaram a publicar um artigo em 2015, onde acabam por desenvolver a fórmula de legibilidade SRES, que segue os mesmos princípios destas métricas, como pode ser visto no ponto 3.1.5). Ainda que não calcule diretamente o resultado de legibilidade mediante a fórmula SRES, esta ferramenta é capaz de determinar as características do código necessárias para o cálculo das diferentes variáveis que compõe o cálculo do SRES.

O POGJE apresenta-se como a ferramenta de legibilidade de software mais interessante desenvolvida até ao momento, uma vez que ao contrário da *Source Code Readability Metric*, apresenta uma interface que facilita a interação entre do utilizador e a aplicação. Uma outra característica positiva do POGJE, é este permitir a avaliação de múltiplos ficheiros Java em simultâneo.

Uma possível limitação que o POGJE apresenta, é o facto de assumir que os ficheiros Java carregados não apresentam erros. O autor não especifica como se comporta a aplicação no caso de serem inseridos ficheiros com erros sintáticos ou semânticos.

Uma outra limitação que esta ferramenta possui, é o facto de não poder ser utilizada durante o processo de desenvolvimento de software. A necessidade de sair do IDE e abrir uma outra ferramenta para analisar o código de um projeto mostra-se como um possível inconveniente para o desenvolvedor.

Um dado que importa referir, é que Nadeem Abbas foi orientado por Jürgen Börstler durante o desenvolvimento da sua tese, sendo este último coautor da fórmula de legibilidade de software SRES.

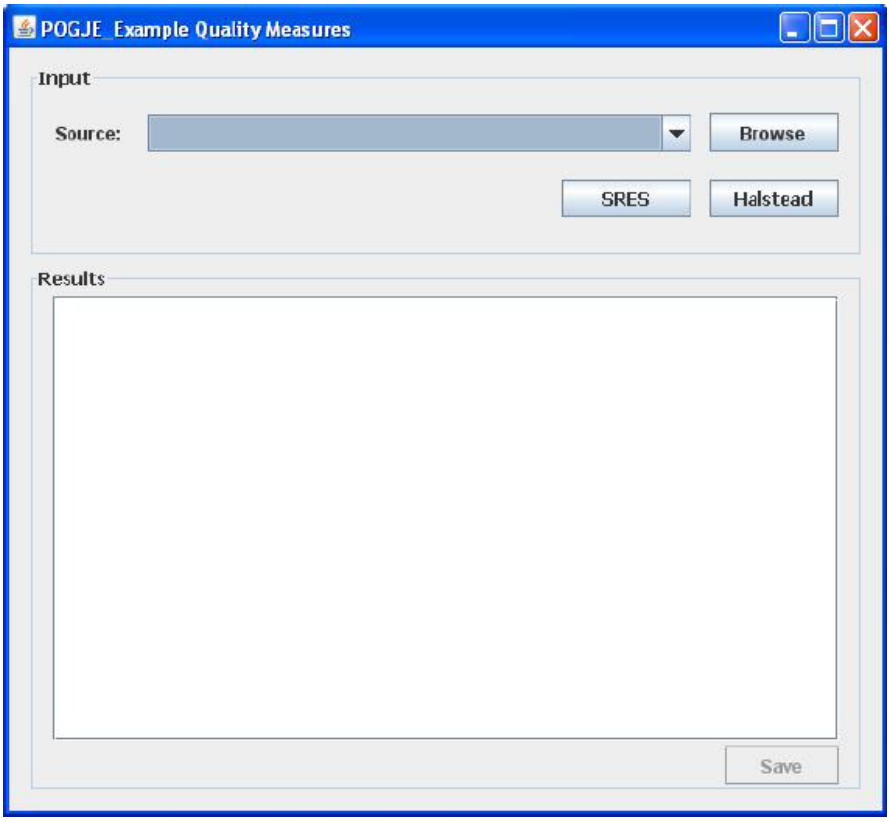

Figura 3 – IU do POGJE (retirado de (Abbas, 2010))

2.2.4.3 Code Readability Tool (CRT)

Aquando da realização do estudo que levou à publicação do artigo "*Impact of Programming Features on Code Readability*", Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi e Maryan Yatim desenvolveram o CRT, uma ferramenta que permite avaliar a legibilidade de código Java através da fórmula *Impact of Programming Features on Code Readability* (IPFCR).

No artigo em que a ferramenta é mencionada, não é apresentada qualquer imagem do seu funcionamento, nem uma ligação que permita descarregá-la e testá-la, pelo que não é possível perceber se ela possui uma IU, ou se o seu funcionamento recorre à utilização da linha e comandos. Do mesmo modo, o artigo também não faz referência à quantidade de ficheiros que a aplicação permite avaliar em simultâneo.

Não havendo uma descrição detalhada do funcionamento do CRT, nem sendo possível testá-lo, não é assim possível perceber quais as características positivas e negativas que ele apresenta.

3 Análise e Comparação das Fórmulas de Legibilidade e Decisões de Implementação

Algumas fórmulas de legibilidade de software foram propostas ao longo das últimas duas décadas, sendo que cada uma utiliza uma abordagem distinta para estimar a legibilidade do código. Estas diferenças na abordagem à avaliação da legibilidade mostram-se interessantes, já que permitem obter diferentes perspetivas sobre o que pode ser considerado um excerto de código legível de um excerto de código pouco legível.

Será expectável que a solução ideal para o problema apresentado neste documento passe por implementar as seis fórmulas teorizadas na ferramenta de legibilidade proposta, já que desta forma haverá um maior leque de opções por onde os desenvolvedores se poderão orientar, de modo a avaliar a legibilidade do código que produzem. Contudo, devido a constrangimentos de tempo, serão implementadas apenas três fórmulas nesta primeira versão da ferramenta. Este é um número que parece razoável para o trabalho que se apresenta.

Como já foi referido, cada fórmula utiliza uma abordagem própria para o cálculo da legibilidade e, como tal, a decisão sobre quais as primeiras fórmulas a serem integradas no *plugin* terá de ser criticamente ponderada.

Depois de apresentadas no capítulo anterior as fórmulas de legibilidade de código-fonte propostas, este capítulo serve para, numa primeira instância, analisar cada uma das fórmulas, de modo a auferir a abordagem tomada por cada uma e os aspetos do código que elas avaliam. Numa segunda fase, é efetuada uma comparação entre todas as fórmulas. Isto facilitará a decisão de quais as fórmulas a implementar na ferramenta. Por fim, são abordadas as decisões de implementação, como por exemplo o tipo de ferramenta que vai ser implementada, as fórmulas de legibilidade que esta contemplará e para que plataforma ela será direcionada.

3.1 Análise

Como mostra a Tabela 3, desde 2002, seis fórmulas de legibilidade de software foram propostas. Nos próximos subcapítulos, cada uma dessas fórmulas de legibilidade é apresentada e analisada. São explorados vários aspetos das mesmas, como por exemplo a abordagem tomada, a noção de legibilidade que os seus autores procuraram atribuir às fórmulas, as características do código que cada uma delas avalia e até mesmo os cálculos necessários para determinar o resultado das várias variáveis envolvidas em cada uma das fórmulas. Procura-se com isto ter uma visão crítica sobre cada uma das fórmulas e perceber quais as grandes diferenças entre elas.

3.1.1 Comments Ratio

A demora que leva a fase de manutenção de grandes sistemas de software comparativamente com as fases anteriores motivou K.K. Aggarwal, Y. Singh e J.K. Chhabra a sugerir um modelo de manutenção de código-fonte. Este modelo, apresentado no artigo *"An Integrated Measure of Software Maintainability"*, avalia três aspetos importantes do software (Aggarwal, Singh and Chhabra, 2002):

1. Legibilidade do código-fonte;
2. Qualidade da documentação;
3. Compreensibilidade do software.

Este modelo mostra-se importante no âmbito deste trabalho pelo primeiro aspeto do software abordado pelo mesmo.

Os autores argumentam que qualquer tipo de manutenção vai, em última instância, resultar em alterações no software. Eles dizem que perceber a lógica de milhares de linhas de código é quase impossível se este não estiver devidamente comentado. Desta forma, chegam à conclusão de que a legibilidade de código-fonte pode ser estimada pela percentagem de linhas com comentários em relação ao total de linhas de código (Aggarwal, Singh and Chhabra, 2002). Assim, chegam à definição de uma fórmula de legibilidade que se baseia no rácio de comentários e que se traduz na seguinte fórmula:

$$CR = LOC/LOM \tag{7}$$

, onde CR é o rácio de comentários, LOC representa o número total de linhas de código (incluindo comentários) e LOM representa o número total de linhas com comentários.

3.1.2 PHD

Posnett Hindle Devanbu, ou PHD, é uma fórmula de legibilidade desenvolvida por Daryl Posnett, Abram Hindle e Premkumar Devanbu, sendo ela um melhoramento às métricas de

legibilidade de software desenvolvidas por Buse e Weimer (Posnett, Hindle and Devanbu, 2011). Esta fórmula foi proposta no artigo *"A Simpler Model of Software Readability"*.

Os autores têm três objetivos principais com o desenvolvimento desta fórmula:

1. Simplificar o modelo de legibilidade de Buse e Weimer (B&W) e melhorar a sua performance;
2. Clarificar e fortalecer as suas bases teóricas nas ideias clássicas da engenharia de software, no que respeita ao tamanho e à complexidade;
3. Melhorar o modelo, recorrendo a noções simples de informação teórica de entropia.

Eles detalham de que forma recorrem ao tamanho, à complexidade e à entropia para calcular a legibilidade do código. De seguida é feita uma abordagem ao que os autores referem.

- Tamanho e a Legibilidade
  Os autores debruçaram-se sobre os excertos de código apresentados por Buse e Weimer, para o desenvolvimento desta fórmula. Eles notaram que os excertos que Buse e Weimer utilizaram variam de tamanho, pelo que sugerem que o tamanho dos excertos pode ter influência na sua legibilidade. Os autores da fórmula PHD posicionam-se a favor da utilização do fator tamanho num modelo de legibilidade. Para sustentar esta posição, eles apresentam uma correlação de Spearman[5] entre as várias características utilizadas por Buse e as medidas de tamanho (linhas, palavras e caracteres). Os resultados obtidos estão apresentados na Figura 4, e demonstram que as características apresentadas não são independentes do tamanho.

Metrics	lines	words	characters
avg math	0.45	**0.66**	0.51
avg comment	0.57	**0.63**	0.49
max idents	0.30	**0.66**	**0.65**
max word	0.34	0.49	0.54
max line length	0.11	0.44	**0.62**
max occurrences char	0.47	**0.62**	**0.84**

Figura 4 – Correlação de Spearman entre as características apresentadas por Buse e o tamanho (retirado de (Posnett, Hindle and Devanbu, 2011))

  Eles observam com isto que o número de linhas não se correlaciona tão significativamente como o número de palavras e caracteres.

  Os autores apresentam ainda uma correlação de Spearman entre as três medidas de tamanho e a média das classificações atribuídas por quem votou na legibilidade dos excertos. Essa correlação é apresentada na Figura 5.

[5] Correlação que mede a força e direção de uma associação entre duas variáveis classificadas (Lund Research Ltd., 2018).

Metrics	lines	words	characters
mean voter score	0.232	−0.002	−0.202

Figura 5 – Correlação de Spearman entre as três medidas de tamanho e a média das votações (retirado de (Posnett, Hindle and Devanbu, 2011))

Os resultados apresentados na Figura 5 levou-os a crer que a baixa correlação entre cada métrica individual e a média das classificações indica que cada uma dessas métricas por si só não é um bom preditor. Isto também lhes permitiu observar que a correlação positiva entre o número de linhas e a média das classificações sugere que excertos com mais linhas são mais legíveis.

Posnett e os restantes investigadores concluem que, embora o tamanho não represente totalmente a legibilidade, os resultados obtidos mostram que esta medida é um preditor significativo de legibilidade e que deve ser incluído em qualquer modelo que procure perceber quais os fatores que influenciam a legibilidade.

- Halstead e a Legibilidade
  Outro ponto abordado foi a relação entre a complexidade do software e a legibilidade do mesmo. As medidas de Halstead são métricas de complexidade de software criadas por Maurice Howard Halstead, em 1977 (Hamer and Frewin, 1982), e foi sobre elas que Posnett e os restantes investigadores incidiram.

  Estas métricas são calculadas através da contagem do número total operadores, do número único de operadores, do número total de operandos e do número único de operandos. Estes quatro valores são então combinados, de forma a permitir o cálculo de vários aspetos relacionados com a complexidade de software.

  Um dos aspetos avaliados por esta métrica é o volume do programa, que procura medir o conteúdo de informação do código-fonte combinando as contagens totais com as contagens únicas de operadores e operandos. Posnett e os seus colegas afirmam que pode ser conjeturada uma relação próxima entre esta medida e a legibilidade.

  Os cálculos para as métricas de Halstead são apresentados de seguida:

  - Comprimento do Programa

    $$N = N1 + N2 \quad (8)$$

    , onde N é o comprimento do programa, N1 é o número total de operadores e N2 é o número total de operandos.

- Vocabulário do Programa

$$n = n1 + n2 \quad (9)$$

, onde n é o vocabulário do programa, n1 é o número de operadores únicos e n2 é o número de operandos únicos.

- Volume

$$V = N \log_2 n \quad (10)$$

, onde V é o volume, N é o comprimento do programa e n é o vocabulário do programa.

- Dificuldade

$$D = \frac{n1}{2}\frac{N2}{n2} \quad (11)$$

, onde D é a dificuldade, n1 é o número de operadores únicos, N2 é o número total de operandos e n2 é o número de operadores distintos.

- Esforço

$$E = DV \quad (12)$$

, onde E representa o esforço, D a dificuldade e V o volume.

Depois de efetuados vários estudos entre a relação destas métricas e a legibilidade de software, Posnett e os restantes investigadores chegaram a algumas conclusões. Uma delas foi que N, n e V estão altamente ligadas. Isso indica que incluir mais do que uma destas variáveis no modelo proposto não vai influenciar significativamente o resultado de legibilidade obtido pelo mesmo.

Outra conclusão a que chegaram foi de que D e E não contribuem significativamente para os resultados obtidos, pelo que a sua inclusão no modelo proposto foi descartada.

Os investigadores também notaram que o número de linhas de um excerto está positivamente relacionado com a legibilidade, à semelhança do que demonstram os modelos desenvolvidos aquando da comparação entre o tamanho e a legibilidade. Concluíram então que adicionar o número de linhas de um excerto ao modelo melhora os resultados obtidos pelo mesmo.

Outra descoberta que os investigadores fizeram foi que o aumento do número total de *tokens*[6], ou do número de *tokens* únicos leva a uma diminuição na legibilidade no caso de o tamanho do fragmento se manter constante. Por outro lado, ao aumentar o tamanho do fragmento, há uma maior área para os fragmentos existentes se espalharem e, portanto, a legibilidade aumenta.

Uma última observação constatada por estes investigadores é que um excerto que contenha muitos elementos distintos, sejam eles operadores ou operandos, terá um volume maior que um excerto com número parecido de elementos totais, mas menos elementos distintos. Os resultados indicam que a identidade dos elementos é menos importante que a cardinalidade e densidade dos mesmos.

Em suma, concluem que o volume das medições de Halstead adiciona um poder considerável ao modelo, e que quando combinado com medidas simples de tamanho, superam o modelo de Buse e Weimer.

- Entropia e a Legibilidade
  A entropia, ou a quantidade de informação num conjunto de dados, pode ser calculada através da contagem de termos (*tokens* ou *bytes*) totais e únicos. Posnett, Hindle e Devanbu abordam estes dois tipos de termos no seu estudo.

  Os autores procuram responder a duas questões durante o estudo da influência da entropia na legibilidade. Primeiro, pretendem saber se a entropia ao nível dos *bytes* contribui para o modelo de legibilidade. Depois, procuram perceber se o volume de Halstead pode ser substituído pela entropia de *tokens* no modelo de legibilidade proposto.

  Quanto à primeira questão, e depois de analisados os modelos produzidos, os autores verificaram que a correlação bruta entre a entropia de *bytes* e a legibilidade é baixa e negativa, sugerindo que um aumento na informação do conteúdo irá reduzir de alguma forma a legibilidade. Referem ainda que adicionar a entropia de *bytes* ao modelo, que até então apenas engloba o volume de Halstead e o tamanho, aumenta a performance do mesmo. Com isto, concluem que embora a entropia de *bytes* aumente o nível de previsão do modelo, o seu impacto ao nível de performance é reduzido quando comparado com outros preditores.

  Em relação à segunda pergunta à qual Posnett e a sua equipa tentam responder, eles afirmam que o volume de Halstead se debruça sobre a contagem e frequência de *tokens* e que, nesse caso, a entropia ao nível de *tokens* deve ser considerada. Os investigadores calcularam a entropia de *tokens* Java nos excertos disponíveis e compararam esses valores com os resultados obtidos pelo volume de Halstead. Com

[6] São os vários elementos do código que são identificados pelo compilador. São o elemento mais pequeno de um programa com significado para este. Os *tokens* dividem-se em cinco categorias: variáveis, palavras-chave, caracteres especiais, literais e operadores. (Thakur, 2018)

isso, concluíram que ao utilizar a entropia de *tokens* ao invés do volume, é produzido um modelo cuja performance é comparável ao de Buse, com a exceção de que as medidas de entropia de *tokens* têm um coeficiente positivo e não negativo, como acontece com o volume. Verificam então que a entropia de *tokens* e o volume de Halstead se correlacionam negativamente. Posnett e os seus colegas concluem que a entropia de *tokens* explica os dados tão bem como a característica dos *tokens* avaliada pelo modelo de Buse, mas não tão bem como o volume de Halstead.

A fórmula utilizada para o cálculo da entropia é a seguinte:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{13}$$

, onde X é um documento e x_i é um termo em X.

A variável $p(x_i)$ apresentada na fórmula da entropia é uma função de probabilidade que estima a probabilidade de o evento x_i acontecer e calcula-se recorrendo à seguinte equação:

$$p(x_i) = \frac{count(x_i)}{\sum_{j=1}^{n} count(x_j)} \tag{14}$$

A avaliação de todos estes aspetos permitiu concluir a fórmula de legibilidade pretendida. Em suma, esta fórmula engloba as seguintes três variáveis: o volume de Halstead, o número de linhas e a entropia.

A fórmula proposta pelos investigadores baseia-se em dois cálculos: o cálculo da variável de regressão, através de uma equação de regressão linear, e a aplicação da variável logística que retorna o valor final da legibilidade.

As três características do código que os autores consideraram pertinentes adicionar à fórmula servem para calcular a variável de regressão. Esses valores combinados com os respetivos coeficientes dão origem à seguinte fórmula de regressão linear:

$$z = 8.87 - 0.033Volume + 0.40Linhas - 1.5Entropia \tag{15}$$

, onde z é o valor da variável de regressão.

Depois de calculado o valor de z, é então possível calcular o valor final da legibilidade, recorrendo à função logística[7] representada pela equação 16.

[7] Função que apresenta um formato de "S".

$$Legibilidade = \frac{1}{1 + e^{-z}} \quad (16)$$

, onde z é a variável de regressão.

A função logística apresentada retorna um valor entre 0 e 1, pelo que o resultado da legibilidade calculado pela fórmula PHD vai sempre variar o seu valor dentro deste intervalo. Quanto mais próximo de um for o valor de legibilidade, mais legível é o excerto de código em causa, segundo a fórmula PHD.

3.1.3 Code Readability

No artigo *"A New Metric for Code Readability"*, Rajendar Namani e Kumar J. desenvolveram uma fórmula que dizem permitir avaliar a legibilidade de código-fonte (Namani and Kumar, 2012). Entre outras, eles recorreram a propriedades como o número de linhas de código, o comprimento das linhas e a presença de comentários no código para estimar o quão legível este é.

Segundo os autores, esta fórmula pretende avaliar a legibilidade de código escrito em C#, ainda que no artigo onde ela é proposta, sejam apresentados resultados obtidos com esta fórmula nas linguagens C, C# e JavaScript. Uma outra observação que eles fazem é que o resultado obtido por esta fórmula é a percentagem de legibilidade do código.

A fórmula sugerida é a seguinte:

$$CR = LOC + LL + NOCL + NOBL + NLAS + BSAD + NOM \quad (17)$$

, onde CR é o valor de legibilidade de código, LOC é o número de linhas de código, LL é o comprimento das linhas, NOCL é a presença de linhas com comentários no programa, NOBL é o número de linhas em branco, NLAS representa a quebra de linha depois de um ponto e vírgula, BSDAS representa o espaço em branco após declarações diretivas e NOM é o número de métodos.

O artigo publicado pelos autores da fórmula apresenta, contudo, vários problemas e discordâncias. Tais problemas passam, por exemplo, por frases mal construídas, imagens com baixa resolução que pretendem apresentar resultados obtidos, impossibilitando assim a sua análise, ou o facto de os autores não referirem como são avaliados alguns parâmetros, como por exemplo o comprimento das linhas de código. Para além disso, não é justificada a forma como foram decididas as características do código incluídas na fórmula. Todavia, o grande problema que a fórmula proposta apresenta, e o que faz com que ela seja, de facto, inviável, é que ela deveria, segundo os autores, calcular uma percentagem, contudo, o seu cálculo é feito exclusivamente através de somas, pelo que o resultado nunca será o sugerido por Rajendar e Kumar.

3.1.4 IPFCR

Num estudo intitulado *"Impact of Programming Features on Code Readability"* realizado por Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi e Maryan Yatim, os investigadores procuram desenvolver uma nova abordagem que pretende avaliar a influência e o efeito de várias características de programação sobre a legibilidade de código.

A par da idealização das várias fórmulas que integram a abordagem do IPFCR, os autores desenvolveram uma ferramenta de legibilidade de código chamada CRT, onde implementam a lógica da abordagem proposta. Esta ferramenta, implementada em C#, permite avaliar a legibilidade de código escrito na linguagem Java.

Os autores tentam com esta abordagem avaliar a influência que várias características que podem ser recolhidas automaticamente do código têm na legibilidade do mesmo. Foi realizado um questionário com vários profissionais da área da programação, de forma a auferir o nível de satisfação dos mesmos em relação à influência na legibilidade de várias características. As respostas foram analisadas e serviram para estimar o peso que cada uma delas tem na legibilidade do código. Para cada característica foi desenvolvida uma fórmula dedicada que usa os pesos anteriormente definidos. O CRT permite extrair essas características do código e aplicá-las às respetivas fórmulas, recorrendo também aos pesos estimados para cada uma delas. Por fim, a soma do valor de todas as fórmulas é o resultado da legibilidade.

A Figura 6, adaptada do artigo onde esta abordagem foi proposta, apresenta o fluxo de trabalho do IPFCR.

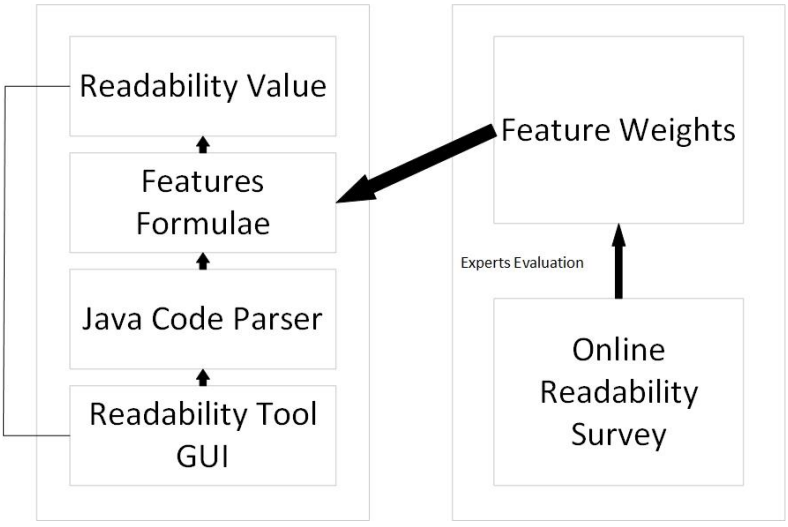

Figura 6 – Fluxo de trabalho do IPFCR (adaptado de (Tashtoush et al., 2013))

Esta abordagem sugere que várias características do código têm pesos diferentes na legibilidade. Desse modo, tal como foi referido anteriormente, os autores desenvolveram uma fórmula para cada uma dessas características, onde o seu peso é tido em conta como uma das variáveis incluídas no cálculo.

As várias fórmulas teorizadas pelos autores apresentam-se de seguida:

- Média das variáveis com nomes significativos (AVMN)

$$AVMN = (\frac{VMN}{V}) \times MNW \quad (18)$$

, onde VMN é a média de variáveis com nomes significativos, V é o número de variáveis e MNW é o peso dos nomes significativos.

- Média de funções com nomes significativos (AFMN)

$$AFMN = (\frac{FMN}{F}) \times MNW \quad (19)$$

, onde FMN é o número de funções com nomes significativos, F é o número de funções e MNW é o peso dos nomes significativos.

- Linhas em branco por linha de código (WLC)

$$WLC = (\frac{BL1}{BL2}) \times SW \quad (20)$$

, onde BL1 é o número de locais que têm uma linha em branco, LB2 é o número de locais onde são precisas linhas em branco e SW é o peso do espaçamento.

- Indentações (I)

$$I = \frac{I1}{I2} \times IW \quad (21)$$

, onde I1 é o número de locais que têm indentações, I2 é o número de locais onde são precisas indentações e IW é o peso das indentações.

- Comentários (CM)

$$Cm = \left(\frac{Cml}{CL}\right) + (\frac{Cml}{0.2 \times CW}) \quad (22)$$

, onde Cml é o número de linhas com comentários, CL é o número de linhas de código e CW é o peso dos comentários.

- *Scope* (S)

$$S = (\frac{MSV}{MaxSV}) \times SpW \quad (23)$$

, onde MSV é a mediana do volume de *scopes*, MaxSV é o volume máximo de *scopes* e SpW é o peso do *scope*.

- Comprimento das Linhas (LL)

$$LL = \left(\frac{MLL}{MaxLL}\right) \times LLW \quad (24)$$

, onde MLL é a mediana do comprimento das linhas, MaxLL é o comprimento máximo das linhas e LLW é o peso do comprimento das linhas.

- Fórmulas Aritméticas (ArF)

$$ArF = e^{-(\frac{Fn}{CL})\times FW} \quad (25)$$

, onde Fn é o número de fórmulas, CL é o número de linhas de código e FW é o peso das fórmulas.

- Média de *if-else* (AvIe)

$$AvIe = (\frac{IeS}{CL}) \times IeW \quad (26)$$

, onde IeS é o número de declarações *if*, CL é o número de linhas de código e IeW é o peso dos *if-else*.

- Imbricação de Condições (Ni)

$$Ni = e^{(-MaxNiD)\times NiW} \quad (27)$$

, onde MaxNiD é a profundidade máxima dos *if* imbricados e NiW é o peso dos *if* imbricados.

- Média de ciclos *for* (AvFL)

$$AvFL = (\frac{FL}{L}) \times FLW \quad (28)$$

, onde FL é o número de ciclos *for*, L é o número de outros ciclos e FLW é o peso dos ciclos *for*.

- Ciclos Imbricados (NL)

$$NL = e^{(-MaxNLD)\times NLW} \quad (29)$$

, onde MaxNLD é a profundidade máxima de um ciclo imbricado e NLW é o peso de um ciclo imbricado.

- Funções Recursivas (RF)

$$RF = e^{(-R)\times RW} \quad (30)$$

, onde R é o número de recursividades e RW é o peso da recursividade.

- *Arrays* (AR)

$$Ar = (\frac{A}{Is}) \times AW \quad (31)$$

, onde A é o número de *arrays*, Is é o número de identificadores e AW é o peso dos *arrays*.

- Distribuição de Classes (CD)

$$CD = (\frac{MCV}{MaxCV}) \times CW \quad (32)$$

, onde MCV é a mediana do volume de classes, MaxCV é o volume máximo das classes e CW é o peso das classes.

- Herança (In)

$$In = (\frac{InC}{C}) \times InW \quad (33)$$

, onde InC é o número de classes herdadas, C é o número de classes e InW é o peso da herança.

- Sobreposição (Or)

$$Or = (\frac{PVFD}{FD}) \times OrW \quad (34)$$

, onde PVFD é o número de funções herdadas de declarações de funções puras virtuais, FD são todas as declarações de funções e OrW é o peso da sobreposição.

- Consistência (Cn)

$$Cn = (\frac{Com}{LSc}) \times CnW \quad (35)$$

, onde Com é o número de vírgulas, LSc é o número de linhas de código e CnW é o peso da consistência.

O valor final da legibilidade é então a soma do resultado de todas as fórmulas apresentadas anteriormente, traduzindo-se na equação 36.

$$\begin{aligned} Legibilidade = {} & AVMN + AFMN + WLC + I + Cm + S + LL + ArF \\ & + AvIe + Ni + AvFL + NL + RF + Ar + CD + In + Or \\ & + Cn \end{aligned} \quad (36)$$

3.1.5 SRES

Software Readability Ease Score, ou simplesmente SRES, é uma fórmula de legibilidade proposta por Jürgen Börstler, Michael E. Caspersen e Marie Nordström baseada na *Flesch Reading Ease Readability Formula*, também conhecida por FRES. Ela tem como foco principal avaliar a legibilidade de exemplos de código direcionados para iniciantes à programação. Nadeem Abbas diz que o SRES é uma extensão orientada para software do FRES (Abbas, 2010). A idealização desta fórmula foi inicialmente introduzida por Börstler, Caspersen e Nordström, em 2007, no artigo *"Beauty and the Beast Toward a Measurement Framework for Example Program Quality"* e, posteriormente efetivada, no artigo *"Beauty and the Beast: on the readability of object-oriented exemple programs"*, publicado oito anos depois pelos mesmos autores.

Seguindo a ideia de Flesch, o SRES pode ser definido interpretando os lexemas[8] de uma linguagem de programação como sílabas, as suas declarações como palavras e as unidades de abstração como frases. Segundo estes três investigadores, pode-se, portanto, argumentar que quanto menores forem as médias do tamanho das palavras e das frases, mais fácil é reconhecer pedaços de compreensão relevantes (*chunks*), tornando assim o código mais legível. Börstler, Caspersen e Nordström afirmam que *chunking*[9] apropriado é altamente relevante para a compreensão de excertos de código de exemplo (Börstler, Caspersen and Nordström, 2015).

Para medir o SRES, os autores procuraram seguir uma abordagem semelhante à utilizada no índice de facilidade de Flesch, no entanto, dado que neste caso a avaliação da legibilidade é orientada para software, as variáveis foram ligeiramente adaptadas a este contexto. Uma das variáveis utilizadas é, tal como no FRES, o comprimento médio das frases (*Average Sentence Length* - ASL), no entanto, a segunda variável utilizada não é a média de sílabas por palavra, mas sim o comprimento médio das palavras (*Average Word Length* - AWL). Em traços gerais, estas duas variáveis são calculadas da seguinte forma (Börstler, Caspersen and Nordström, 2015):

- ASL corresponde ao número médio de palavras por declaração ou bloco (delimitado por chavetas e pontos e vírgulas).
- AWL corresponde à média do comprimento dos lexemas em número de caracteres.

Tendo isto em conta, foi proposta a fórmula apresentada na equação 37.

$$SRES = ASL - 0.1AWL \tag{37}$$

, onde SRES é o valor de legibilidade, ASL é o comprimento médio das frases e AWL é o comprimento médio das palavras.

[8] Em programação, um lexema é uma sequência de caracteres alfanuméricos num *token*.
[9] Processo de reorganizar informação de muitos *bits* de baixo nível de informação, em poucos *chunks* com muitos *bits* de informação.

Börstler e os restantes investigadores defendem que o AWL deve ser razoavelmente alto, já que identificadores mais compridos transportam mais significado. Eles também dizem que, segundo alguns estudos, identificadores mais longos ajudam na compreensão do programa. Por seu lado, o ASL deve ser pequeno, visto ser o fator mais importante para o *chunking*. Sendo assim, os investigadores acabaram por concluir que o ASL é mais importante que o AWL, já que a maioria das palavras é familiar, mas a gramática não, pelo que ler "frases" de um programa é mais difícil que ler as suas "palavras" (Börstler, Caspersen and Nordström, 2015).

Os autores admitem que a legibilidade tem outras variáveis associadas, como por exemplo a indentação ou os comentários, no entanto, o SRES foca-se nas propriedades inerentes que não podem ser facilmente modificadas sem reestruturar o código, isto é, eles pretendem que esta fórmula ignore propriedades que estão mais ligadas à "leiturabilidade" do que à legibilidade (Börstler, Caspersen and Nordström, 2015).

3.1.6 WCMR

"Word Concreteness and Memory Retention" é o nome do artigo onde Weifeng Xu, Dianxiang Xu e Lin Deng propuseram uma abordagem para a medição automatizada da legibilidade de código-fonte. Esta abordagem apresenta quatro fórmulas de legibilidade distintas que possibilitam estimar a legibilidade do software com diferentes níveis de abstração (Xu, Xu and Deng, 2017):

1. Legibilidade de uma variável;
2. Legibilidade de um método;
3. Legibilidade de uma classe;
4. Legibilidade do programa.

A base lógica do WCMR centra-se na legibilidade das variáveis incluídas no código. O cálculo da legibilidade de um método baseia-se na legibilidade das variáveis nele contidas, ao passo que o cálculo da legibilidade de uma classe se baseia na legibilidade dos métodos que a compõe. A mesma lógica aplica-se à avaliação da legibilidade de um programa.

Sendo assim, esta abordagem classifica as variáveis em duas categorias: unigramas e multigramas. Como contextualização para os cálculos que se apresentam mais à frente neste subcapítulo, é importante perceber a diferença entre estes dois conceitos:

- Unigrama é uma variável definida com uma única palavra (ex: "*class*").
- Multigrama é um nome composto, que consiste em múltiplos unigramas. Por exemplo, "*classLoader*" consiste na aglutinação das palavras "*class*" e "*load*".

Para além disso, o WCMR recorre a dois conceitos diferentes para estimar a legibilidade das variáveis. São eles a percetibilidade de palavras e a retenção de memória (Xu, Xu and Deng, 2017). À semelhança dos conceitos apresentados anteriormente, estes também são importantes para a compreensão dos cálculos mais à frente apresentados.

- Percetibilidade de palavras é a facilidade de memorizar o significado de variáveis quando estas são vistas pela primeira vez.
- Retenção da memória representa até que ponto o significado das variáveis é retido.

Os cálculos da percetibilidade e da retenção representam os dois termos incluídos na expressão do cálculo da legibilidade de uma variável, pelo que é necessário fazer referência a eles antes de introduzir as fórmulas de legibilidade propostas.

- Cálculo da Percetibilidade
  Tal como foi referido anteriormente, um dos pontos necessários para calcular a legibilidade é a percetibilidade das variáveis. A fórmula que permite calcular a percetibilidade de uma variável é a seguinte:

$$Percetibilidade(v) = \begin{cases} fc(v), v \text{ é } um\ unigrama \\ duc(x) \times \big(1 + \Delta cgr(v)\big), v \text{ é } um\ multigrama \end{cases} \quad (38)$$

  , onde v é uma variável, $fc(v)$ é o valor de percetibilidade de um unigrama v, $duc(x)$ é a percetibilidade do unigrama dominante x e $\Delta cgr(v)$ é a taxa de ganho de percetibilidade do multigrama v.

  Verifica-se com esta fórmula a necessidade de introduzir mais dois cálculos diferentes:

  - Percetibilidade de um Unigrama Dominante

$$duc(multigrama) = \max_{i \in |n|} fc(wi) \quad (39)$$

    , onde $|n|$ é o tamanho do multigrama e $fc(wi)$ é o valor de percetibilidade de cada palavra unigrama wi.

  - Taxa de Ganho de Percetibilidade

$$\Delta cvg(multigrama) = \sum_{i,j \epsilon |n|, i \neq j} rSim(wi, wj) \times disSimC(wi, wj) \quad (40)$$

    , onde $|n|$ é o tamanho do multigrama, i e j são índices, wi e wj são palavras, $rSim(wi, wj)$ é a similaridade relacional do par de palavras wi e wj e $disCSim(wi, wj)$ é a dissimilaridade da similaridade concetual entre o par de palavras wi e wj.

    Esta última fórmula apresenta mais um cálculo que ainda não foi introduzido, que é o cálculo da dissimilaridade da similaridade concetual. Os autores não apresentam uma definição para este conceito, mas olhando para o seu nome, verifica-se que este cálculo procura verificar a diferença entre a similaridade concetual de duas palavras. O seu cálculo é apresentado na fórmula 41.

$$disSimC(w1, w2) = 1 - cSim(w1, w2) \quad (41)$$

, onde w1 e w2 são palavras e $cSim(w1, w2)$ é a similaridade concetual do par de palavras w1 e w2.

Importa referir que $fc(x)$ representa valores constantes previamente determinados e listados. No caso de x não pertencer à listagem definida, o seu valor de percetibilidade é igual a zero. Por seu lado, $cSim(x1, x2)$ e $rSim(x1, x2)$ podem ser calculados recorrendo a ferramentas de análise semântica latente (*Latent Semantic Analysis* - LSA).

- Cálculo da Retenção

  Para além do cálculo da percetibilidade, é também necessário calcular a retenção das variáveis. A fórmula utilizada para esse fim é a seguinte:

$$retenção(v) = a \times \left(\frac{Distância(v, m)}{LPP}\right)^{-b} \quad (42)$$

, onde v é uma variável, m é um método, a é uma constante com valor 0.85, b é também uma constante com valor 0.12 e LPP é o número de linhas por página. É um valor constante que representa o número de linhas de código padrão por página (p.e. 30 para alguns IDE's Java populares). (Xu, Xu and Deng, 2017)

À semelhança do cálculo da percetibilidade, é necessário introduzir alguns cálculos intermédios:

  - Distância de Leitura de uma Variável

$$\begin{aligned} Distância(v, m) &= primeiraDistânciaDU \times \big(1 - \Delta dvu(v, m)\big) \\ &= \big(uso(v, 1, m) - Def(v, m)\big) \times (1 - \Delta dvu(v, m)) \end{aligned} \quad (43)$$

, onde v é uma variável, m é um método e $\Delta dvu(v, m)$ é a densidade de uso de v em m.

Este cálculo leva à introdução de mais duas fórmulas:

  - Primeira Distância DU – diferença de linhas entre a primeira declaração de uma variável e o seu primeiro uso

$$primeiraDistânciaDU = uso(v, 1, m) - Def(v, m) \quad (44)$$

, onde v é uma variável, m é um método, $uso(v, 1, m)$ é o valor da linha onde v é utilizada pela primeira vez em m e $Def(v, m)$ é a linha onde é declarada a variável v no método m.

- Densidade de Uso de uma Variável

$$\Delta dvu(v,m) = \frac{n}{vida\acute{U}til(v,m)} = \frac{n}{uso(v,n,m) - Def(v,m) + 1} \quad (45)$$

, onde v é uma variável, m é um método, n é o número total de utilizações de v, $vida\acute{U}til(v,m)$ é a diferença de linhas entre a definição e a última utilização de v em m, $uso(v,1,m)$ é o valor da linha onde v é utilizada pela primeira vez em m e $Def(v,m)$ é a linha onde é declarada a variável v no método m.

As fórmulas apresentadas anteriormente representam todos os cálculos intermédios necessários para obter a legibilidade das variáveis. Sendo assim, é então possível apresentar as fórmulas que permitem calcular a legibilidade do código:

- Legibilidade de uma Variável

$$legibilidade(v,m) = Percetibilidade(v) \times Retenção(v) \quad (46)$$

, onde v é uma variável e m é um método.

- Legibilidade de um Método (ou construtor)

$$legibilidade(m,c) = \frac{\sum_{i\in1}^{n} legibilidade(vi,m)}{|V|} \quad (47)$$

, onde m é um método (ou um construtor), c é uma classe, V são todas as variáveis utilizadas em m (incluindo variáveis locais, parâmetros, constantes, variáveis de instância e variáveis de classe) e vi é uma variável em V.

- Legibilidade de uma Classe

$$legibilidade(c,P) = \frac{\sum_{i\in1}^{n} legibilidade(mi,c)}{|M|} \quad (48)$$

, onde c é uma classe, P é um programa orientado a objetos, M são todos os métodos e construtores utilizados em C e mi é um método ou construtor em M.

- Legibilidade de um Programa

$$legibilidade(P) = \frac{\sum_{i\in1}^{n} R(ci,P)}{|C|} \quad (49)$$

, onde P é um programa orientado a objetos, C são todas as classes contidas no programa, ci é uma classe dentro de C e $R(ci,P)$ é o valor de legibilidade da classe ci dentro do programa P.

3.2 Comparação Entre as Fórmulas de Legibilidade

Como já foi mencionado, a solução mais completa passaria por implementar todas as fórmulas de legibilidade propostas, no entanto, nesta primeira fase, tal não se mostra viável. Desse modo, terá de se proceder a um processo seletivo que permita escolher quais as fórmulas mais indicadas para serem implementadas na primeira versão da ferramenta de legibilidade proposta.

Ao longo deste subcapítulo será feita uma análise crítica ao método de avaliação de legibilidade de cada fórmula, com o propósito de permitir uma comparação entre elas. Uma vez que as fórmulas apresentadas avaliam características distintas do código e seguem abordagens muito díspares umas das outras, uma comparação direta entre elas é difícil, já que os resultados obtidos por cada fórmula traduzem a legibilidade do código com pontos de vista diferentes. Assim, esta análise procura permitir diferenciar os vários aspetos de cada fórmula, possibilitando então a escolha das fórmulas a implementar na ferramenta de legibilidade.

De forma a ter uma visão mais detalhada e aprofundada sobre cada fórmula, numa primeira fase são investigadas as características abordadas por cada uma delas e são analisados os resultados e conclusões que estas obtiveram.

Posteriormente, são analisadas as linguagens de programação que cada fórmula de legibilidade permite avaliar. Isto possibilitará, não só perceber as linguagens em que cada fórmula se foca, mas também ajudar a decidir qual vai ser a direção seguida aquando da implementação da ferramenta de legibilidade, isto é, qual ou quais as linguagens de programação que irão ser consideradas.

Importa notar que, tal como se provou em 3.1.3, o cálculo da fórmula *Code Readability* não permite obter resultados fidedignos, pelo que esta será desconsiderada para a implementação do *plugin* e, como tal, será descartada das análises efetuadas neste capítulo e nos capítulos subsequentes.

3.2.1 Características do Código Abordadas e Resultados Obtidos

A legibilidade é um conceito subjetivo (Posnett, Hindle and Devanbu, 2011; Tashtoush et al., 2013) e, como tal, diferentes investigadores têm opiniões distintas sobre os fatores que a influenciam. Desse modo, as fórmulas propostas abordam a legibilidade de várias formas, seguindo cada uma diferentes critérios.

Vários estudos foram feitos em torno das características do código que mais influenciam a legibilidade. Num artigo publicado em 2016 por Isabel Sampaio e Luís Barbosa, é feita uma revisão de literatura, onde estes investigadores procuram apresentar as práticas que influenciam a legibilidade do software e é enfatizada a importância do ensino das mesmas por parte dos docentes (Sampaio and Barbosa, 2016). Estas boas práticas envolvem várias

características das linguagens de programação, características essas que são utilizadas pelos diferentes autores das fórmulas de legibilidade de software apresentadas.

Com vista a possibilitar uma comparação entre as várias fórmulas, é preciso em primeiro lugar perceber quais as características do código que cada uma delas contempla no cálculo da legibilidade. Neste subcapítulo é realizada uma recolha detalhada das características do código que cada uma das fórmulas aborda e são comentados os resultados e as conclusões que os testes efetuados pelos autores permitiram alcançar, de forma a tentar identificar possíveis discordâncias entre fórmulas e também observar eventuais correlações entre as características avaliadas.

Importa referir que existem dois tipos de correlações: correlações positivas e correlações negativas. Correlação positiva é a relação entre duas variáveis, em que ambas se movem em conjunto (Limited Liability Company, 2018b). Por exemplo, quanto mais tempo uma pessoa passa a correr numa passadeira, mais calorias ela queima. Aqui, as variáveis "tempo" e "calorias" têm uma correlação positiva (LoveToKnow Corp., 2018b). Já uma correlação negativa é a relação entre duas variáveis, em que quando uma delas aumenta, a outra diminui e vice-versa (Limited Liability Company, 2018a). Um exemplo de uma correlação negativa é a situação em que quando um carro diminui a velocidade, o tempo de viagem aumenta. Neste caso, as variáveis "velocidade" e "tempo" têm uma correlação negativa (LoveToKnow Corp., 2018a).

3.2.1.1 Comments Ratio

Como foi descrito no ponto 3.1.1, o *Comments Ratio* é uma fórmula proposta por K.K. Aggarwal, Y. Singh e J.K. Chhabra que se baseia no rácio entre o número de linhas com comentários e o número de linhas de código para avaliar a legibilidade do software.

Apesar de proporem uma fórmula para avaliar a legibilidade do código, o foco geral dos autores no estudo que levou ao desenvolvimento desta fórmula foi o de criar um modelo integrado, com o propósito de estimar a manutenibilidade do software, com base, não só na avaliação da legibilidade do código-fonte, mas também na qualidade da documentação e na compreensão do software. Como tal, não foram realizados testes específicos para comprovar a validade do *Comments Ratio* como avaliador de legibilidade. Os autores apresentam, contudo, o seguinte gráfico que ilustra os valores do rácio de comentários (CR) que representam boa, média e má legibilidade:

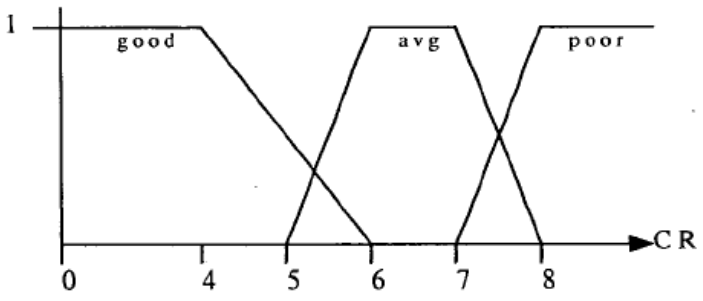

Gráfico 1 – Interpretação dos valores do CR (retirado de (Aggarwal, Singh and Chhabra, 2002))

Sumário das características avaliadas

- Linhas de Código;
- Linhas com Comentários.

3.2.1.2 PHD

Sendo esta uma fórmula que se baseou nas métricas desenvolvidas por Buse e Weimer, ela assenta na mesma ideia de legibilidade proposta por estes investigadores. Como tal, as características que Posnett e a sua equipa integram na sua fórmula de legibilidade, procuram melhorar este modelo, ao nível dos resultados obtidos e de performance, recorrendo a um modelo mais simplificado.

Uma das variáveis que é incluída nos cálculos do PHD é o volume de Halstead. Como se pode observar na fórmula do volume apresentada na equação 10, esta fórmula recorre ao cálculo do comprimento e do vocabulário do programa. Por sua vez, estes dois cálculos requerem a quantidade de operadores e de operandos incluídos no código. De um modo geral, operadores são símbolos capazes de manipular valores e operandos numa expressão (Techopedia Inc., 2018). Por seu lado, operandos são termos utilizados para descrever qualquer objeto capaz de ser manipulado (Computer Hope, 2017).

Os investigadores afirmam que o volume calculado pelas métricas de Halstead acrescenta poder explanatório considerável ao modelo e que, quando combinado com uma medida simples de tamanho, consegue superar o modelo desenvolvido por Buse e Weimer. Como tal, argumentando que o tamanho está diretamente relacionado com a legibilidade do código, os autores adicionaram o número de linhas de código ao cálculo da fórmula, de forma a determinar o tamanho do programa. (Posnett, Hindle and Devanbu, 2011)

Por fim, foram ainda acrescentadas medidas de entropia ao modelo proposto que, de acordo com os autores, melhoram o poder preditivo do modelo, tendo pouco impacto na sua performance. Eles provam que ao adicionar o cálculo da entropia ao modelo proposto o faz superar o modelo de Buse e Weimer em todas as medidas de performance por si testadas (Posnett, Hindle and Devanbu, 2011). De forma a calcular esta medida, os autores recorreram à distribuição de *tokens* e *bytes* no código-fonte. Neste contexto, *bytes* podem ser entendidos como sinónimo de caracteres contidos no código.

Em suma, segundo os resultados obtidos por Posnett e os seus colegas, é demonstrado que com apenas três variáveis, esta fórmula supera o modelo de Buse e Weimer enquanto classificador de legibilidade para pequenos excertos de código. Os autores dizem ainda que não afirmam que este simples modelo capture toda a essência da legibilidade, nem que possa ser aplicado levianamente a qualquer pedaço de código. Ao invés, eles observam que os resultados obtidos se enquadram com os dados recolhidos razoavelmente bem, e que, com um número significativamente menor de variáveis, acaba por ser um modelo mais parco de legibilidade dentro do limite do contexto estudado, isto é, recorre a menos preditores para obter os resultados do modelo proposto por Buse e Weimer. (Posnett, Hindle and Devanbu, 2011)

Sumário das características avaliadas

- Operadores;
- Operandos;
- *Tokens*;
- *Bytes*;
- Linhas de Código.

3.2.1.3 IPFCR

Um diferencial da fórmula IPFCR em relação a outras fórmulas de legibilidade, é que para cada característica é atribuído um peso definido através de inquéritos realizados com desenvolvedores de software. Os autores afirmam que as características que eles incluem na sua abordagem podem ser facilmente extraídas do código (Tashtoush et al., 2013).

Depois de implementar a lógica desta abordagem no CRT, os investigadores passaram à fase de testes. Na primeira fase, recorreram a três códigos Java diferentes, retirados de *websites* da área. Nestes testes, os autores procuraram usar várias versões de cada código para poderem focar em uma ou duas características específicas, comparando cada versão com a anterior. Os códigos utilizados foram implementações do algoritmo de Dijkstra, do jogo Sudoku e de um utilitário de data e hora. Este primeiro teste permitiu concluir que a utilização de classes não balanceadas diminui a legibilidade do código.

A segunda fase de testes envolveu o estudo de 50 códigos Java, de forma a examinar as mudanças na legibilidade baseadas em cada característica do código. Esta fase de testes permitiu concluir que o valor de legibilidade não diminui significativamente com o número de linhas de código, o que indica que essa característica tem baixo impacto na avaliação da legibilidade do código-fonte, como se pode verificar no Gráfico 2.

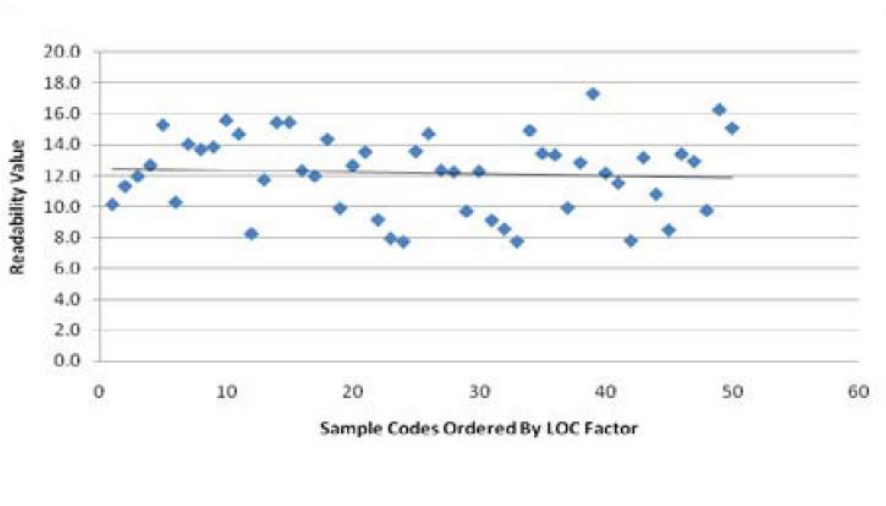

Gráfico 2 – Variação do valor de legibilidade em função no número de linhas de código (retirado de (Tashtoush et al., 2013))

Por outro lado, os investigadores concluíram que o valor de legibilidade aumenta significativamente com a adição de comentários e no caso de os nomes dados a funções e a métodos serem concordantes com o contexto em que se apresentam. O gráfico que permite verificar o impacto dos comentários na legibilidade de código é apresentado no Gráfico 3.

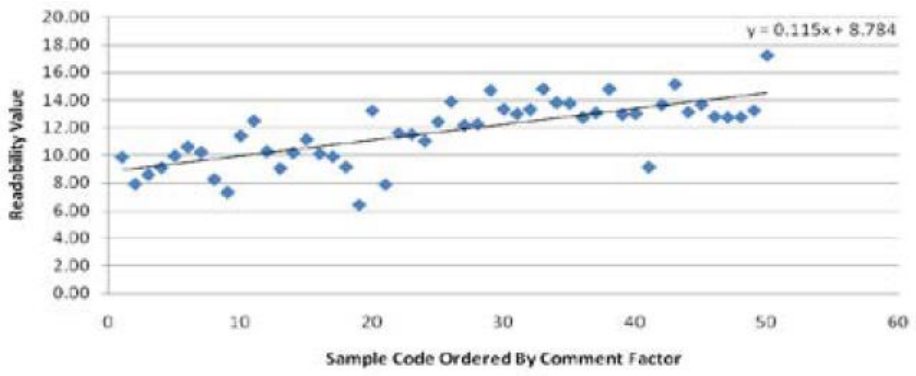

Gráfico 3 – Valores do CRT para as amostras de código, ordenados pelo fator "comentários" (retirado de (Tashtoush et al., 2013))

O Gráfico 4 sumariza as correlações entre as várias características de código avaliadas e a legibilidade do mesmo. Quanto maior for o valor de F, mais positiva é a correlação entre a respetiva característica e a legibilidade e vice-versa.

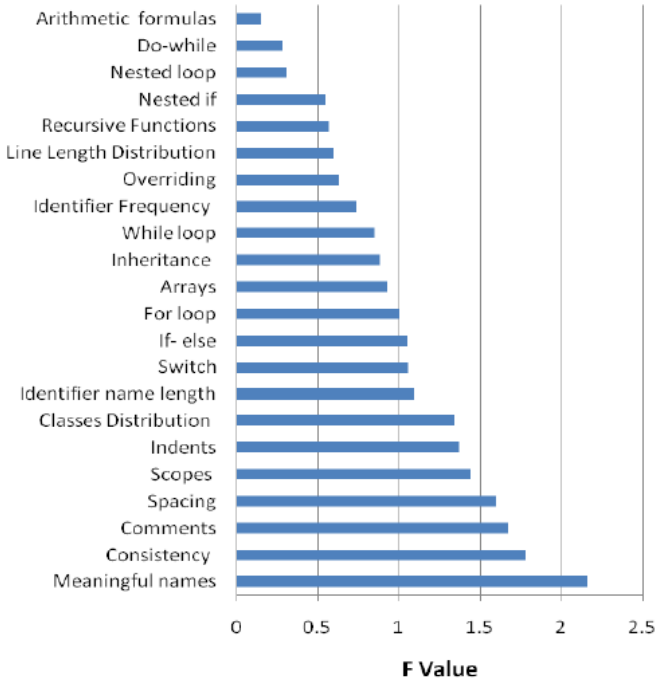

Gráfico 4 – Correlação entre as características avaliadas e a legibilidade (retirado de (Tashtoush et al., 2013))

O Gráfico 4 permite concluir que nomes de identificadores com significado, consistência e comentários são as três características com maior correlação positiva com a legibilidade. Isto

significa que estas são as três características que mais promovem a boa legibilidade do código, segundo esta abordagem. Por seu turno, fórmulas aritméticas, ciclos *do-while* e ciclos imbricados são as três características com maior correlação negativa com a legibilidade. Isto indica que estas são as três características que mais prejudicam a legibilidade do código.

Sumário das características avaliadas

- Nomes de Métodos;
- Nomes de Variáveis;
- Número de Linhas em Branco;
- Indentação;
- Número de Linhas com Comentários;
- Número de Fórmulas Aritméticas;
- Número de Condições *If-Else*;
- Número de Linhas com Código;
- Número de Ciclos *For*;
- Número de Ciclos Além do Ciclo *For*;
- Número de Métodos Recursivos;
- Número de *Arrays*;
- Número de Identificadores;
- Número de Linhas de Código.

Dois reparos devem ser feitos à listagem anterior. Em primeiro lugar, importa referir que há uma diferença entre "linhas com código" e "linhas de código". Os autores não apresentaram detalhadamente estas duas características, no entanto, nos cálculos que apresentam, fazem distinção entre duas variáveis diferentes: *"number of code lines"*, que tratam com o identificador CL e *"number of lines in the source code"*, que tratam com o identificador LSc. Desta forma, percebe-se que há diferença entre as duas variáveis e subentende-se que o identificador CL se refere a linhas de código físicas (apresentado na listagem como "linhas de código") e o identificador LSc se refere a linhas de código lógicas (apresentado na listagem como "linhas com código"). Uma definição mais aprofundada destes dois tipos de linhas de código é apresentada em 4.2.1.

O segundo reparo a fazer é que procuraram referir-se apenas características concretas do código. Por exemplo, um dos cálculos que é feito nesta abordagem é o comprimento das linhas. No entanto, essa característica não foi listada, uma vez que os autores recorrem à mediana do comprimento das linhas e ao máximo comprimento das linhas. Em última instância, eles recorrem às linhas de código como característica avaliada. O mesmo acontece com outros cálculos apresentados.

3.2.1.4 SRES

Apoiados na fórmula de legibilidade de texto *The Flesch Reading Ease Score*, que se centra no comprimento médio das frases e na média de sílabas por palavra, Börstler, Caspersen e

Nordström desenvolveram o SRES, que se baseia na interpretação dos lexemas de uma linguagem de programação, nas suas declarações e nas unidades de abstração.

Uma consideração importante sobre esta fórmula de legibilidade é que os autores dizem que o SRES não tenta fazer o cálculo pelo tamanho, uma vez que o seu foco é a legibilidade. Na opinião dos mesmos, programas maiores levam mais tempo a ser lidos, mas tal não significa necessariamente que sejam mais difíceis de ler (Börstler, Caspersen and Nordström, 2015).

Importa notar que apesar de poder ser aplicada a qualquer pedaço de código Java, esta abordagem é especialmente orientada para avaliar a legibilidade de excertos de código de exemplo, incluídos, por exemplo, em manuais escolares.

Uma outra consideração a ter em conta é que, para o cálculo desta fórmula, os autores assumem que o código está bem indentado, formatado e tem um nível apropriado de comentários (Abbas, 2010).

A nível de resultados, a primeira observação que deve ser feita, é que nos testes efetuados, o SRES é aplicado a 21 exemplos de código Java retirados de livros de introdução à programação. Os dados recolhidos pela pesquisa levada a cabo pelos investigadores comparam o resultado de várias outras fórmulas de legibilidade de código-fonte com os resultados obtidos pelo SRES.

Uma conclusão a que os autores chegaram é a que não há correlação significativa entre o tamanho (número de declarações) e as medidas de legibilidade SRES, B&W e PHD. Isto contradiz as afirmações de Posnett e da sua equipa, quando dizem que o número de linhas de um excerto de código está positivamente associado à legibilidade.

Os resultados também permitiram concluir que PHD é a única fórmula testada que apresenta uma correlação significativa com a densidade de comentários. Isto deve-se ao fator “linhas” contemplado por esta fórmula. Segundo os autores, o fator “linhas de código” na fórmula PHD torna-a muito sensível a comentários e a linhas em branco. Isto torna a fórmula mais sensível a problemas de “leiturabilidade” (“coisas que afetam os olhos dos leitores”) do que o SRES, que procura capturar fatores inerentes da legibilidade de código (“coisas que afetam as mentes dos leitores”) (Börstler, Caspersen and Nordström, 2015).

Por fim, os autores afirmam que o SRES pode ser útil para ajudar educadores na seleção e desenvolvimento de programas de exemplo adequados, acrescentando que como a legibilidade do código é um fator importante na manutenção de software, seria interessante investigar a utilidade do SRES em prever vários aspetos de manutenibilidade do mesmo.

Os investigadores concluem as suas argumentações, afirmando que os resultados obtidos são promissores, já que o SRES atua tão bem ou melhor que as medidas de B&W e PHD em exemplos de código de Java retirados de livros.

Sumário das características avaliadas

- Declarações;
- Lexemas.

3.2.1.5 WCMR

Esta abordagem avalia a legibilidade do software calculando a legibilidade de todas as variáveis declaradas no código-fonte. Para avaliar a legibilidade das variáveis, o WCMR determina o quão facilmente o significado do nome das variáveis é memorizado e a rapidez com que ele é esquecido com o passar do tempo.

A legibilidade dos vários níveis de abstração é obtida, na sua génese, através da legibilidade das variáveis, isto porque, tal como já foi referido, a legibilidade de um método é avaliada segundo a média da legibilidade das variáveis nele contidas. Por sua vez, a legibilidade de uma classe é determinada através da legibilidade dos métodos que a compõe, e a legibilidade de um programa é avaliada segundo a legibilidade de todas as classes.

De forma a testar a validade desta abordagem, os investigadores utilizaram 14 aplicações de código aberto com mais de meio milhão de linhas de código e 10,000 *warnings* de defeitos. Os últimos foram reportados através da ferramenta FindBugs.

Os autores concluem que os resultados obtidos nos testes permitem concluir que o WCMR se correlaciona negativamente com as taxas gerais de avisos de defeitos no código, em particular com avisos de más práticas de programação, vulnerabilidade de código e avisos de *bugs*. Os investigadores acrescentam ainda que esta correlação é muito significativa. Dito por outras palavras, quanto maior for o número de defeitos, menor vai ser o valor do WCMR (baixa legibilidade).

Isso pode ser demonstrado com os resultados obtidos pelos autores, representados no Gráfico 5.

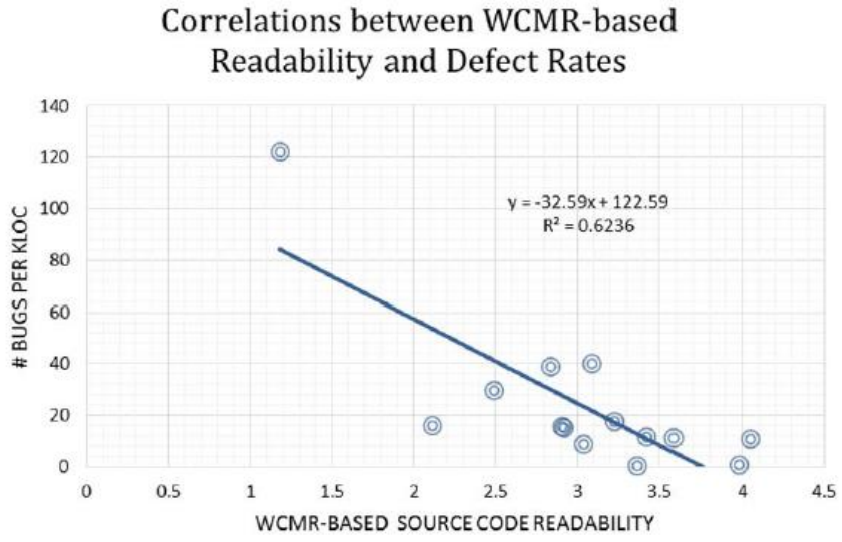

Gráfico 5 – Correlação ente o número de defeitos e o valor do WCMR (retirado de (Xu, Xu and Deng, 2017))

Uma observação que os investigadores fazem, é que a abordagem tomada pode subestimar alguns valores de legibilidade, uma vez que ela recorre a um dicionário para estimar a

percetibilidade de palavras, dicionário esse que não contempla termos e calões utilizados pela comunidade de desenvolvimento de software.

Sumário das características avaliadas

- Nomes de Variáveis.

A Tabela 4 sumariza as características do software utilizadas por cada uma das fórmulas propostas para calcular a legibilidade.

Tabela 4 – Características do código-fonte utilizadas por cada fórmula de legibilidade

Fórmula / Característica	Comments Ratio	PHD	IPFCR	SRES	WCMR
Arrays			✓		
Bytes		✓			
Ciclos Além do Ciclo For			✓		
Ciclos For			✓		
Condições If-Else			✓		
Declarações				✓	
Identificadores			✓		
Indentação			✓		
Lexemas				✓	
Linhas com Comentários	✓		✓		
Linhas com Código			✓		
Linhas de Código	✓	✓	✓		
Linhas em Branco			✓		
Métodos			✓		
Operadores		✓			
Operandos		✓			
Tokens		✓			
Variáveis					✓

3.2.2 Linguagens de Programação Abordadas

A legibilidade de software, como tem vindo a ser referido ao longo deste documento, é influenciada por diversas características do código, porém, algo que merece ser mencionado, é que as próprias linguagens de programação possuem características que as tornam mais ou

menos legíveis (Chisnall, 2006). Será então, de certa forma espectável, que uma fórmula de legibilidade aplicada a código escrito em duas linguagens de programação diferentes apresente valores de legibilidade distintos, dependendo do quão legível a linguagem em questão é, isto no caso de essa fórmula ser capaz de avaliar mais de uma linguagem de programação. As fórmulas de legibilidade apresentadas neste documento não abordam, contudo, as diferenças de legibilidade entre as várias linguagens de programação. Além de que a maioria das fórmulas é destinada especificamente para uma só linguagem.

A seguinte listagem apresenta as linguagens de programação às quais cada fórmula se propõe a avaliar:

- *Comments Ratio*: Aggarwal e a sua equipa não fazem qualquer tipo de referência às linguagens de programação às quais esta fórmula se destina, porém, como ela apenas aborda o número de linhas de código e o número de linhas com comentários, é intuitivo pensar que ela pode ser aplicada a qualquer tipo de linguagem de programação que suporte comentários.
- PHD: Posnett, Hindle e Devanbu não referem diretamente as linguagens de programação às quais esta fórmula se propõem a avaliar, contudo dois reparos devem ser feitos. O primeiro reparo é que esta fórmula assenta nas métricas de Buse e Weimer, métricas essas que são destinadas a código Java. O segundo reparo é que quando os autores falam na comparação do volume de Halstead com entropia de *tokens*, dizem que calcularam a entropia de *tokens* Java para efetuar tal comparação (Posnett, Hindle and Devanbu, 2011). Deste modo, será de prever que esta fórmula seja mais indicada para avaliar a legibilidade de código escrito em Java. A sua utilização com outras linguagens de programação, contudo, pode ser viável dadas as variáveis que englobam este cálculo não serem exclusivas da linguagem Java.
- IPFCR: destinada à linguagem de programação Java.
- SRES: destinada à linguagem de programação Java.
- WCMR: sendo esta uma fórmula que avalia unicamente a legibilidade dos nomes atribuídos a variáveis, pode-se entender que ela permite avaliar a legibilidade de qualquer linguagem de programação. Contudo, importa notar que Weifeng Xu e os restantes investigadores referem que ela é indicada para avaliar linguagens de programação orientadas a objetos. Além disso, esta fórmula foi testada pelos seus autores em 14 programas de código aberto, todos eles escritos em Java. Sendo assim, subentende-se que essa é a linguagem que melhor se adequa a esta fórmula.

Estas constatações são pontos fulcrais para decidir quais linguagens de programação devem ser contempladas aquando da implementação da ferramenta de legibilidade proposta. Isto traz várias implicações. Uma delas é ao nível da própria implementação, visto que linguagens diferentes possuem sintaxe e semântica diferentes, pelo que a análise do código não pode ser efetuada da mesma forma. Outra implicação é que caso se utilize uma fórmula preparada para avaliar uma linguagem de programação específica, numa outra linguagem diferente, o resultado da legibilidade poderá ser falacioso.

3.3 Decisões de Implementação

Assentando na análise realizada ao longo deste capítulo, podem assim ser tomadas as decisões em relação à implementação da ferramenta de legibilidade. Estas decisões abrangem três aspetos distintos. São eles a linguagem de programação que vai ser suportada pela ferramenta de legibilidade, o IDE para o qual vai ser desenvolvido o *plugin* e as fórmulas de legibilidade que irão ser implementadas. Esses pontos são abordados ao longo dos próximos três subcapítulos.

3.3.1 Escolha da Linguagem de Programação Suportada

A primeira decisão a ser tomada tem a ver com a linguagem de programação que o *plugin* vai suportar. Verificou-se com a listagem feita em 3.2.2 que todas as fórmulas de legibilidade permitem avaliar código Java, sendo que apenas três das fórmulas são capazes de avaliar a legibilidade de outras linguagens.

A implementação da mesma fórmula para linguagens diferentes não pode ser efetuada exatamente da mesma forma, uma vez que, como já foi referido, linguagens diferentes possuem sintaxe e semântica distintas, então, a análise do código escrito nas várias linguagens terá de recorrer a bibliotecas ou gramáticas diferentes.

Sendo assim, e não descartando a inclusão de mecanismos que permitam avaliar a legibilidade de outras linguagens em versões posteriores da ferramenta de legibilidade, mostra-se mais oportuno nesta primeira versão contemplar apenas uma linguagem, sendo ela a linguagem de programação Java.

3.3.2 Escolha do IDE

Como já foi referido, a ideia deste trabalho passa por implementar um *plugin* que, depois de instalado no IDE, ficará nele integrado, possibilitando a avaliação da legibilidade do código a ser escrito, sem que para tal seja necessário sair do ambiente de desenvolvimento e abrir uma nova aplicação. Desse modo, ter-se-á de decidir qual o IDE que será suportado.

É vasto o número de IDE's para o desenvolvimento de aplicações Java. Um estudo foi feito em 2014 pelo site *codeanywhere.com* teve em vista perceber quais os IDE's e editores de código mais utilizados nesse ano. Esse estudo baseou-se nas respostas de 2000 desenvolvedores e permitiu perceber que os três principais ambientes de desenvolvimento são o Eclipse, o NetBeans e o IntelliJ (Burazin, 2015).

Os resultados obtidos estão sumarizados no Gráfico 6.

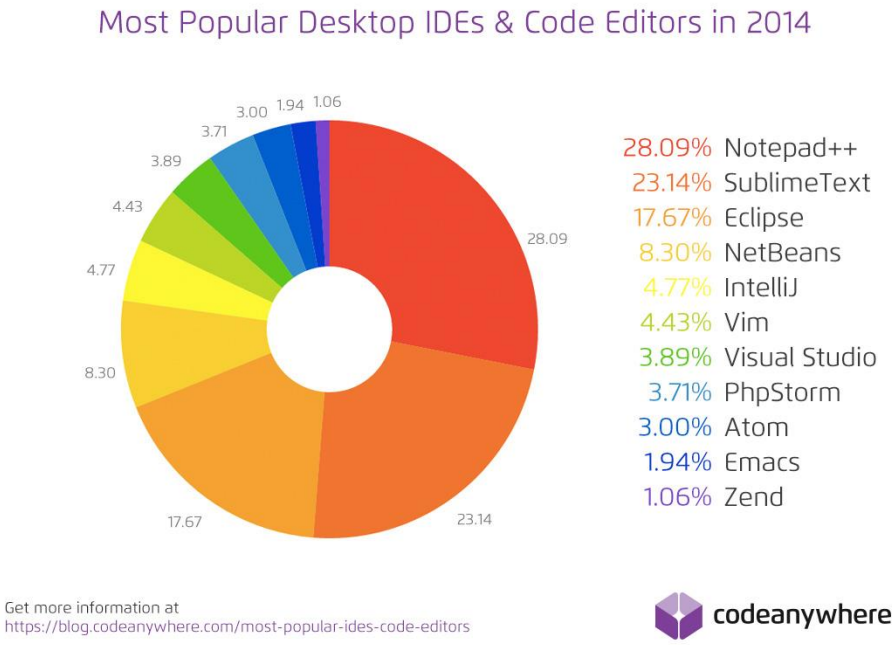

Gráfico 6 – IDE's e editores de texto mais utilizados em 2014 (retirado de (Burazin, 2015))

O Notepad++ e o Sublime Text apresentam uma maior percentagem de utilizadores, no entanto, estes dois programas encaixam na categoria de editores de código e não na categoria de IDE's.

Verifica-se então, que desenvolver um *plugin* para qualquer um dos três IDE's apresentados como mais utilizados se mostra uma opção viável. Ainda que, de acordo com o Gráfico 6, o Eclipse seja o IDE ao qual mais desenvolvedores recorrem, a decisão passa por ser implementar um *plugin* para o NetBeans. Isto permitirá aos alunos do primeiro ano da Licenciatura em Engenharia Informática (LEI) do Instituto Superior de Engenharia do Porto (ISEP) poderem utilizar a ferramenta, uma vez que o NetBeans é o IDE recomendado ao longo do primeiro ano da LEI.

3.3.3 Fórmulas de Legibilidade a Implementar

Decidida a linguagem de programação e o IDE no qual o *plugin* será implementado, é então necessário escolher quais as fórmulas de legibilidade que serão integradas na ferramenta a implementar, tendo por base as conclusões obtidas com as análises anteriormente efetuadas.

SRES é a primeira fórmula a ser considerada. O facto de já ter sido implementada e estar disponibilizada em formato *Java ARchive* (JAR), possibilita a sua fácil integração com a ferramenta de legibilidade proposta, pelo que só por si, este já é um argumento que sustenta a sua escolha. Porém, também deve ser tido em conta que, como já foi referido, esta fórmula se baseia na ideia de avaliação de legibilidade da fórmula FRES e que esta é uma das fórmulas mais aceites de entre todas as fórmulas de legibilidade de texto (Sampaio, 2017). A

legibilidade de texto e a legibilidade de software são dois mundos distintos, pelo que tal não significa que a SRES seja uma fórmula tão credível no meio da engenharia de software como o FRES é no seu meio, no entanto, segundo os testes realizados pelos seus autores, esta fórmula correlaciona-se bem com várias métricas de software, sendo uma delas a noção humana de qualidade de código em linguagens orientadas a objetos (Börstler, Caspersen and Nordström, 2015). Tendo isto em conta, esta será uma das fórmulas a integras na ferramenta a implementar.

A segunda fórmula a ser equacionada é o *Comments Ratio*. O rácio de comentários é uma métrica que não é de comum aceitação entre a comunidade de desenvolvedores de software como quantificador de legibilidade. Vários desenvolvedores defendem que o código deve ser autoexplicativo, e que, se este tiver demasiados comentários, a legibilidade pode inclusive ficar comprometida (Iacovelli, 2018). Contudo, vários autores dizem que, apesar de a afirmação anterior não estar necessariamente errada, a legibilidade está diretamente relacionada com os comentários feitos ao código. Na revisão de literatura efetuada pelos investigadores Isabel Sampaio e Luís Barbosa, é mencionada a utilização consistente e controlada de comentários como sendo uma das práticas que promove legibilidade do código (Sampaio and Barbosa, 2016). A acrescentar a esta afirmação, Brijendra Singh e Shikha Gautam afirmaram numa revisão realizada em 2016 sobre o impacto do processo de desenvolvimento de software na qualidade de código, que a legibilidade do software está geralmente relacionada com os comentários e com os *standards* de nomenclatura (Singh and Gautam, 2016). Mais um fator que mostra a importância da inclusão desta fórmula no *plugin* a implementar, é o facto de ela poder complementar os resultados obtidos pelo SRES. Isto é, Nadeem Abbas, o desenvolvedor que implementou o *Software Reading Ease Score*, considera que os comentários, quando utilizados com moderação, contribuem para a legibilidade do código, mas que este fator não é considerado pelo SRES. Ele diz que esta fórmula assume que o código está bem indentado, formatado e que tem um número apropriado de comentários incluído (Abbas, 2010). Sendo assim, ao utilizar o resultado obtido pelo SRES com o resultado obtido pelo *Comments Ratio*, pode então chegar-se a uma conclusão mais realista sobre a legibilidade do código.

Tendo em conta os factos apresentados, verifica-se então que esta é uma das fórmulas que deve constar no *plugin* a implementar. Importa notar, contudo, que segundo o intervalo de valores apresentado no Gráfico 1, Aggarwal e a sua equipa sugerem que quanto maior for o número de linhas com comentários no código, mais legível este será. Isto entra em discordância com o que foi apresentado anteriormente e é um ponto que deve ser tido em consideração. Ou seja, os valores de legibilidade obtidos com a utilização desta fórmula devem ser criticamente julgados como bons ou maus. Um valor do *Comments Ratio* para um projeto não terá o mesmo significado que o mesmo valor para outro projeto. Isto deve ser ponderado pelo desenvolvedor.

PHD é a terceira fórmula a ser considerada. Esta fórmula apresenta a limitação de ser projetada para avaliar apenas pequenos excertos de código. Esta limitação poderia ser suficiente para a descartar do conjunto de fórmulas a implementar, no entanto, um fator que

deve ser tido em conta é que o artigo no qual esta fórmula foi proposta é um dos mais citados no meio dos estudos de legibilidade de software. A adicionar a isso, esta fórmula baseia-se nas métricas que Buse e Weimer propuseram em 2010 no artigo *"Learning a Metric for Code Readability"*, sendo este o artigo talvez mais influente no meio. Uma outra característica que a fórmula PHD apresenta em relação a outras fórmulas, é que ela recorre, entre outras medidas, a métricas de complexidade para estimar a legibilidade do código. Esta é uma abordagem que mais nenhuma fórmula tomou, e que se provou eficaz com os resultados obtidos por Posnett e os seus colegas (Posnett, Hindle and Devanbu, 2011). Em acréscimo, tal como Börstler, Caspersen e Nordström referiram, sendo esta uma fórmula sensível ao tamanho por considerar o número de linhas de código como fator integrante dos seus cálculos, ela recorre também a questões de "leiturabilidade" para estimar a legibilidade do software. O SRES não avalia esta característica e o *Comments Ratio*, apesar de também ser sensível ao tamanho do código, é uma fórmula diferente, visto que apenas recorre à presença de comentários para estimar a legibilidade.

A fórmula PHD apresenta, assim, vários diferenciais em relação a outras fórmulas de legibilidade. No seio do mercado de trabalho, esta fórmula pode não se mostrar tão relevante, no entanto, quando utilizada por estudantes que estejam a ser introduzidos à programação, o PHD pode ser interessante para estimar a legibilidade de pequenos excertos de código que estes escrevem. Nesse caso, esta será outra das fórmulas incluídas na ferramenta de legibilidade.

O IPFCR é uma fórmula que se mostra interessante pelo facto de avaliar múltiplas características do código e por atribuir pesos a cada uma delas. Esta fórmula será certamente considerada numa versão futura do *plugin*, no entanto, como já foi referido, serão apenas consideradas três fórmulas para esta primeira fase da implementação, já que é difícil perceber se o tempo disponível permitiria implementar mais fórmulas.

À semelhança da fórmula IPFCR, a WCMR também será futuramente equacionada, no entanto, ela apresenta uma possível limitação que pode ser suficiente para inviabilizar a sua implementação. Esta fórmula requer a utilização de vários dicionários, de forma a avaliar a percetibilidade e a retenção dos nomes atribuídos às variáveis. Por exemplo, um dos dicionários compila a abreviação de várias palavras - p.e. *"src"* é, normalmente, uma abreviação de *"source"*. Este dicionário contém mais de 100,000 entradas. Para além deste, mais dois dicionários são utilizados, perfazendo um total de cerca de 200,000 entradas. Colocar em memória tantas entradas e percorrê-las em busca de uma em específico não é eficiente e iria tornar a análise do código demasiado lenta. Isto demonstra que esta fórmula pode ser pertinente para ser implementada numa ferramenta que avalie a legibilidade de software, mas não num *plugin* que pretende analisar o código em tempo de desenvolvimento.

Com isto, conclui-se que as fórmulas cuja implementação se mostra mais pertinente para a versão inicial do *plugin* a implementar são a *Comments Ratio*, a SRES e a PHD.

3.3.3.1 Algumas Considerações Sobre as Fórmulas Escolhidas

A primeira consideração que deve ser tomada é que, tal como tem vindo a ser referido ao longo deste documento, todas as fórmulas de legibilidade de software referidas apresentam abordagens distintas e avaliam características diferentes do código, pelo que não há duas fórmulas que, quando adicionadas à ferramenta de legibilidade, se sobreponham e avaliem a mesma coisa, tornando os resultados redundantes. Tendo isto em conta, verifica-se que as fórmulas que serão implementadas poderão ser utilizadas em conjunto para auferir a legibilidade do código com diferentes perspetivas e complementando os resultados umas das outras.

A fórmula de legibilidade SRES, por exemplo, como avalia o comprimento médio de palavras e de declarações, permite aos desenvolvedores perceber se o código que escrevem é demasiado comprido para os padrões estipulados. Esta fórmula permite avaliar a legibilidade com uma perspetiva essencialmente ligada ao comprimento do código.

Por seu turno, a fórmula PHD permite ter uma noção da legibilidade de código de um ponto de vista relacionado com a complexidade e com a "leiturabilidade" do código, algo que as outras fórmulas não contemplam. Como já foi referido, complexidade, "leiturabilidade" e legibilidade são características distintas, mas que, como foi provado com os estudos realizados em torno desta fórmula, pode ajudar a quantificar a legibilidade do software.

Por fim, o *Comments Ratio* é uma fórmula que se abstrai das características semânticas e sintáticas do código e se foca mais numa característica que serve como complemento de compreensão do mesmo. Além disso, como foi mencionado em 3.3.3, os resultados obtidos por esta fórmula podem ser relacionados com os resultados obtidos pelo SRES, de forma a complementar esta última.

Outra consideração que deve ser tomada, é que tendo em conta as três fórmulas de legibilidade escolhidas, verifica-se assim que a utilização desta primeira versão do *plugin*, apesar de orientada para qualquer tipo de desenvolvedor, parece ser especialmente indicada para iniciantes à programação. Isto porque, em primeiro lugar, a fórmula de legibilidade SRES foi especialmente desenhada para avaliar a legibilidade de excertos de código de exemplo, contidos em manuais de ensino.

Em segundo lugar, a limitação de tamanho da fórmula PHD também é um fator que desencoraja a sua utilização a nível profissional. Não permitindo a avaliação de grandes e complexos algoritmos, pode ser, contudo, utilizada por iniciantes, de forma a avaliar a legibilidade de pequenos excertos de código.

4 Conceção, Implementação e Avaliação

Com este trabalho foi desenvolvido um *plugin* para o IDE NetBeans intitulado *Readability Checker*. Este *plugin* permite avaliar a legibilidade de código-fonte recorrendo a três fórmulas de legibilidade distintas. Neste capítulo é detalhado todo o processo de desenvolvimento do *Readability Checker* e, em conjunto, é feita uma análise ao *design* da solução.

Em 4.1 é feita uma análise aprofundada ao *plugin* desenvolvido. Aqui são detalhadas as principais tecnologias utilizadas no desenvolvimento do *Readability Checker*, as especificações, a estrutura geral do código e o funcionamento da ferramenta. Neste subcapítulo também é abordada a forma como foi documentado o *plugin* desenvolvido e os testes realizados.

Nos subcapítulos 4.2, 4.3 e 4.4 é efetuada uma análise detalhada à conceção e implementação de cada uma das três fórmulas de legibilidade escolhidas para integrar neste *plugin*. Nestes subcapítulos são pormenorizadas as variáveis que cada uma das fórmulas contempla, os cálculos envolvidos e é abordada a forma como tudo isso é tratado pelo *Readability Checker*. Também são apresentados alguns exemplos breves que pretendem demonstrar a forma como cada uma das fórmulas aborda a análise do código.

Finalmente, em 4.5, é feita a avaliação à solução efetivada. Neste subcapítulo são abordadas as hipóteses avaliadas, a metodologia utilizada para avaliar essas hipóteses e também é feita uma análise aos resultados obtidos.

Todas as decisões, dificuldades e pressupostos tomados são detalhados, explicados e justificados ao longo dos diferentes subcapítulos.

4.1 Readability Checker

O *Readability Checker* é um *plugin* escrito em Java para o IDE NetBeans. Ele tem o propósito de analisar ficheiros Java e avaliar a sua legibilidade, recorrendo a três fórmulas de legibilidade de software distintas. No momento, a aplicação tem implementadas as fórmulas *Comments Ratio*, SRES e PHD e a sua utilização foi testada na versão 8.2 do NetBeans.

Este *plugin* oferece total suporte para a versão 11 do Java (ou Java 1.11) para a fórmula *Comments Ratio*. Quanto às fórmulas SRES e PHD, o *plugin* suporta totalmente a versão 1.5 desta linguagem. A avaliação de código escrito com características introduzidas em versões mais recentes da linguagem pode não ser possível. O motivo pelo qual estas duas últimas fórmulas suportam uma versão mais antiga do Java, prende-se com o facto de a sua implementação recorrer a uma biblioteca que apenas suporta essa versão. Uma parte do cálculo do PHD recorre a ela, ao passo que todo o cálculo do SRES é realizado através da mesma. Isso será detalhado nos subcapítulos 4.3 e 4.4.

O diagrama representado na Figura 7 apresenta os principais componentes que compõem o *Readability Checker*.

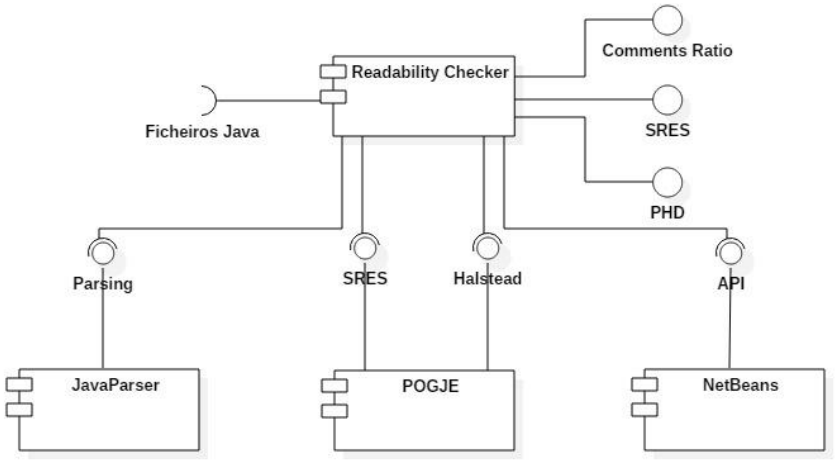

Figura 7 – Diagrama de Componentes - Readability Checker

4.1.1 Principais Tecnologias e Ferramentas Utilizadas

A implementação do *Readability Checker* foi sustentada por um conjunto de tecnologias que permitiu chegar ao resultado descrito ao longo de todo este capítulo. As principais tecnologias utilizadas para a implementação do *plugin* estão explicitadas nos seguintes subcapítulos.

4.1.1.1 NetBeans

Sendo o *Readability Checker* um *plugin* desenvolvido para o NetBeans, a solução óbvia foi utilizar este IDE para a codificação da ferramenta. O *plugin* foi desenvolvido recorrendo à versão 8.2 do NetBeans e a utilização do mesmo foi testada nessa versão do IDE.

Aquando da realização deste trabalho foi lançado o Apache NetBeans 9.0, sendo esta uma nova versão do IDE, agora gerida pela *Apache Software Foundation*. O *plugin* desenvolvido carece ainda de testes nesta nova versão do NetBeans.

4.1.1.2 Maven

O Maven, ou Apache Maven, é uma ferramenta cujo principal propósito é o de tornar o processo de *build* mais rápido e simplificado. Além disso, uma grande vantagem que o Maven oferece é o gerenciador de dependências. No caso da ocorrência de dependências transitivas em alguma biblioteca utilizada, o próprio Maven encarrega-se de gerenciar e descarregar as bibliotecas necessárias.

Várias tecnologias, como é o caso do JavaParser, recorrem ao Maven e os seus desenvolvedores aconselham a adoção desta ferramenta (GitHub Inc., 2018).

4.1.1.3 JavaParser

O desenvolvimento deste *plugin* obrigou à necessidade de analisar o código-fonte de aplicações Java, de forma a ser possível perceber a estrutura gramatical do código em questão. A isto, dá-se o nome de análise sintática, mais conhecido pelo termo em inglês *parsing*.

Dois diferentes caminhos podiam ser tomados para efetuar o *parsing* do código. A primeira passava por utilizar um *parser generator*, como por exemplo o *ANother Tool for Language Recognition* (ANTLR[10]). A segunda passava por utilizar uma biblioteca que oferecesse uma API para analisar e tratar o código Java, tornando assim desnecessário o contacto direto com o *parser generator*. Ainda que a utilização deste último permitisse alcançar os objetivos propostos para o *Readability Checker*, a escolha recaiu sobre a utilização de uma biblioteca, já que esta oferece uma curva de aprendizagem menor, dada a sua utilização ser mais direta. Existindo várias bibliotecas que se destinam a efetuar o *parsing* de código Java, a escolhida para o desenvolvimento do *Readability Checker* foi o JavaParser.

Tendo o seu desenvolvimento iniciado em 2011, e tendo crescido a partir do *parser generator* JavaCC, o JavaParser permite a interação com código Java em forma de objetos num ambiente de desenvolvimento. Esta representação de objetos intitula-se *Abstract Syntax Tree* (AST), ou, em português, Árvore Sintática Abstrata. Além de oferecer mecanismos para navegar a árvore sintática, esta biblioteca também possui a habilidade de manipular a estrutura subjacente do código-fonte. (Smith, Bruggen and Tomassetti, 2018)

O JavaParser oferece várias vantagens. A principal é o facto de os seus sete anos de desenvolvimento lhe terem permitido atingir um nível de maturidade considerável. Para além disso, sendo um software de código aberto, a comunidade de contribuidores é vasta e o código é melhorado como novos *commits* quase diariamente. Isto leva a outro facto importante sobre o JavaParser. Enquanto houver uma comunidade de desenvolvedores que continue a desenvolver esta biblioteca, à medida que novas versões do Java vão sendo lançadas, o seu mecanismo de *parsing* vai sendo atualizado, de forma a contemplar as novas características adicionadas à linguagem. Isto permite uma abstração dessa parte lógica. No caso de se utilizar, por exemplo, o ANTLR, a cada nova versão do Java lançada, ter-se-ia de

[10] *Parser generator* que serve para construir ferramentas de processamento de linguagens formais, como por exemplo tradutores e compiladores (Abbas, 2010).

atualizar os ficheiros gerados por este *parser generator* na ferramenta a implementar. A versão atual do JavaParser suporta a versão 11 do Java – a última versão lançada até à data.

Vale a pena ressalvar que a utilização do JavaParser é feita unicamente nos casos em que é o *Readability Checker* a encarregar-se de determinar o valor de uma variável. Em certos casos mais específicos, explicitados nos capítulos subsequentes, foram utilizadas bibliotecas para calcular o valor de determinadas variáveis, bibliotecas essas que se encarregam de efetuar o *parsing* do código da forma como os seus desenvolvedores melhor entenderam. Nesses casos, o JavaParser não tem qualquer intervenção.

4.1.1.4 POGJE

POGJE é a ferramenta de legibilidade descrita em 2.2.4.2. Ela é utilizada no *Readability Checker* para determinar as variáveis necessárias para o cálculo do SRES e para obter o volume de Halstead necessário para o cálculo da fórmula PHD.

4.1.2 Funcionamento e Fluxo de Trabalho

Sendo o *Readability Checker* um *plugin* destinado para o IDE NetBeans, depois de instalado, ele é apresentado através de um ícone na barra superior do mesmo. Após clicar nesse ícone, a IU demonstrada na Figura 8 é apresentada - mais imagens da IU do *Readability Checker* podem ser consultadas na secção A.1 dos anexos.

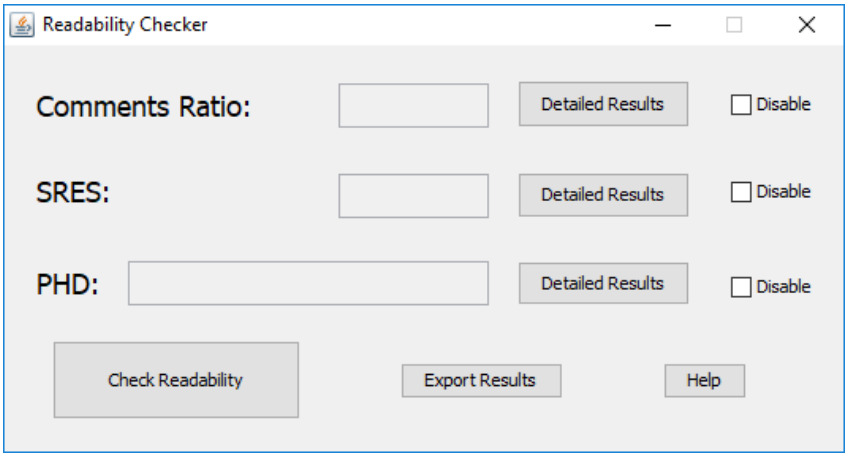

Figura 8 – IU do Readability Checker

Ao clicar no botão *Check Readability*, em primeiro lugar, a aplicação vai verificar qual é o ficheiro Java aberto e a que projeto este pertence. Se estiver aberto algum ficheiro, a aplicação vai carregar todos os ficheiros Java do projeto em causa para avaliação. Se não estiver aberto nenhum ficheiro, nada acontece.

As *checkboxes Disable* servem para desativar a avaliação da legibilidade para qualquer uma das três fórmulas implementadas. Também servem para limpar os campos preenchidos para a respetiva fórmula.

Os botões *Detailed Results* servem para apresentar os resultados detalhados da avaliação da legibilidade para cada uma das fórmulas. Mais detalhes sobre o que é apresentado nestes resultados para cada uma das fórmulas podem ser verificados nos subcapítulos 4.2.3, 4.3.5 e 4.4.5.

O botão *Export Results* serve para guardar os resultados obtidos pelo *Readability Checker*. Tendo sido feita previamente a avaliação da legibilidade de um projeto, ao clicar neste botão, o *plugin* vai guardar num ficheiro de texto os detalhes sobre a legibilidade do projeto e dos ficheiros e métodos que o compõe, mediante as fórmulas que foram aplicadas. No caso de a verificação de legibilidade ainda não tiver sido aplicada, é apresentada uma mensagem de erro e o ficheiro não é criado.

Por fim, o botão *Help* apresenta uma janela com breves considerações sobre cada uma das fórmulas, como por exemplo, como é efetuado o cálculo, ou os valores de legibilidade de referência.

A Figura 9 apresenta uma visão geral do fluxo de trabalho do *Readability Checker*.

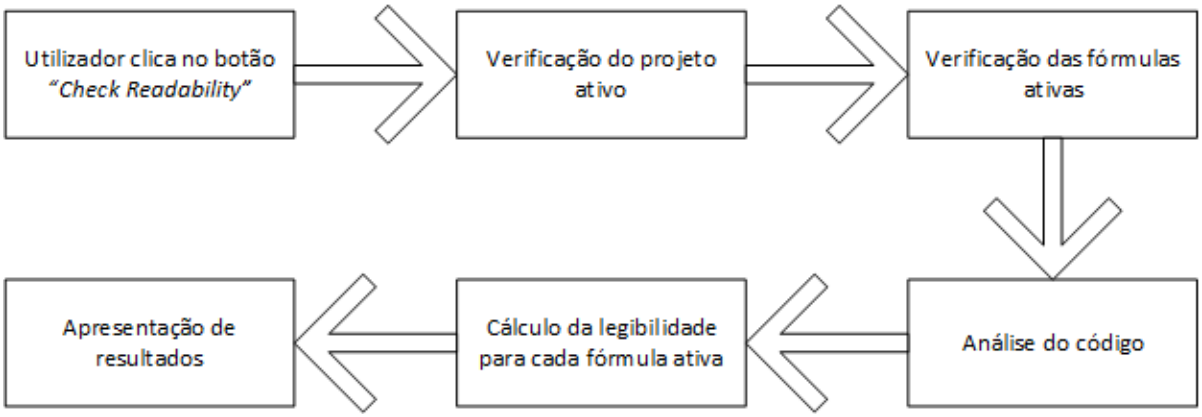

Figura 9 – Fluxo de trabalho do Readability Checker

4.1.3 Estrutura do Código e Princípios de Desenho de Software Utilizados

Este *plugin* foi desenvolvido a pensar na inclusão inicial de três fórmulas de legibilidade, no entanto, foi desenhado de forma a possibilitar a adição futura de novas fórmulas. Como tal, o código foi estruturado de modo a que a implementação das várias fórmulas de legibilidade as tornasse independentes umas das outras.

O projeto contém cinco *packages* distintos:

- *org.myorg.readabilitychecker.codeabstractionlevels*
  Este primeiro *package* contém os objetos representativos das várias partes do código que as fórmulas permitem avaliar. Até ao momento possui duas classes, que representam métodos (avaliados pelas fórmulas *Comments Ratio* e PHD) e ficheiros Java (avaliados pelas fórmulas *Comments Ratio* e SRES).

- *org.myorg.readabilitychecker.formulas.logic*
  Este *package* contém a implementação da lógica para cada uma das fórmulas implementadas. No momento, para cada fórmula há uma classe que implementa toda a lógica necessária para o *parsing* das várias variáveis envolvidas. Estas também são as classes responsáveis por efetuar todos os cálculos necessários para cada uma das fórmulas.

- *org.myorg.readabilitychecker.formulas.objects*
  Cada fórmula possui características e propriedades únicas. Como tal, de forma a diferenciar cada uma delas, todas as fórmulas têm uma classe distinta para a instanciação de objetos do tipo de cada uma delas. Cada classe neste *package* representa uma fórmula e em cada uma delas estão codificados os seus atributos próprios.

- *org.myorg.readabilitychecker.logic*
  Neste *package* estão incluídos os ficheiros que efetuam operações lógicas dentro da aplicação, mas que não estão diretamente relacionadas com o cálculo das fórmulas implementadas. Um exemplo é a classe *MainProjectManager.java* que efetua operações sobre os projetos que estão abertos no IDE.

- *org.myorg.readabilitychecker.main*
  Este *package* contém unicamente o ficheiro que é corrido ao iniciar a aplicação. Esse é o ficheiro encarregue de apresentar a janela principal do *plugin* e invocar as várias funcionalidades da ferramenta.

Desta forma, sempre que uma nova fórmula tiver que ser adicionada ao projeto, é suficiente criar no *package org.myorg.readabilitychecker.formulas.objects* a classe necessária para instanciar objetos dessa mesma fórmula, criar no *package org.myorg.readabilitychecker.formulas.logic* a classe necessária para efetuar a lógica e adicionar à IU do *Readability Checker* os campos necessários para a apresentação de resultados da nova fórmula.

No que toca aos padrões de desenho de software, sabe-se que eles trazem enormes vantagens a projetos de software de grande escala. Duas delas são o facto de oferecerem formas de resolver problemas, recorrendo a soluções provadas eficazes, e o facto de tornar a comunicação entre os *designers* de software consideravelmente mais eficiente (Maioriello, 2002).

Para o desenvolvimento deste *plugin* foi ponderada a adoção de algum padrão de desenho de software, no entanto, dada a dimensão relativamente pequena do mesmo, optou-se por não seguir nenhum padrão. Ao invés, decidiu-se seguir dois princípios de desenho de software intitulados *Don't Repeat Yourself* (DRY) e *Keep It Simple, Stupid* (KISS).

DRY é um princípio básico do desenvolvimento de software que se foca na diminuição da repetição de informação (Baghel, 2018). De forma a evitar a violação deste princípio,

procurou-se dividir o sistema em pedaços independentes, separando o código em pequenas unidades reutilizáveis. Isto torna o sistema mais fácil de ser mantido, reduz a quantidade de tempo de desenvolvimento e também reduz a probabilidade da ocorrência de *bugs* no software.

Por seu lado, KISS é um princípio que procura induzir os desenvolvedores a escrever código simples e claro, tornando-o fácil de perceber. Segundo este princípio, cada método deve resolver apenas um pequeno problema e não demasiados casos de uso.

Foram seguidos estes dois princípios, porque mesmo não utilizando um padrão de desenho de software, tal permite desenvolver uma ferramenta modular, sem duplicação desnecessária de código e facilita a alteração e adição de novas funcionalidades.

4.1.4 Documentação

A documentação é essencial no processo de desenvolvimento de software. Ela permite aos desenvolvedores manter o controlo de todos os aspetos da aplicação e melhora a qualidade do software (Trica, 2014). Uma boa documentação proporciona um desenvolvimento de software mais fluído e facilita a manutenção do mesmo. Ela mostra-se tão importante que foi, como referido anteriormente, incluído no modelo de manutenibilidade proposto por Aggarwal, Singh e Chhabra, em 2002 (Aggarwal, Singh and Chhabra, 2002).

Javadoc é um programa incluído no *kit* de desenvolvimento Java, vulgo JDK, que permite analisar um código em desenvolvimento, procurando por comentários próprios, e possibilitando, assim, a criação de documentação em formato HTML (Fallahi, 1999). Esses comentários, trivialmente chamados de comentários Javadoc, são específicos para documentação e a sua identificação começa por /** e termina por */. São estes identificadores que o Javadoc procura quando analisa o código. Entre eles, podem ser utilizadas *tags* que definem especificações do código. Por exemplo, a *tag @author* serve para indicar o autor (ou autores) de uma classe, ao passo que a *tag @return* indica o que determinado método deve retornar. Várias outras *tags* podem ser incluídas mediante a necessidade específica. Também podem ser adicionadas *tags* HTML, possibilitando assim a formatação do texto que será apresentado nas páginas geradas pelo Javadoc.

De modo a seguir as boas práticas da engenharia de software, *Readability Checker* contém este tipo de comentários. Todas as classes estão documentadas com um comentário Javadoc que indica o seu propósito geral. Os métodos de cada classe também contêm este tipo de comentários. De um modo geral, nos comentários Javadoc efetuados em métodos, é feita uma descrição do propósito do mesmo e são incluídas as *tags* que se mostram necessárias. A título de exemplo, é apresentado <que aplica a fórmula do PHD com o respetivo comentário Javadoc.

```java
/**
 * Applies the PHD formula.
 *
 * @param volume the Halstead's volume of the code.
 * @param lines the number of lines of the code.
 * @param entropy the Shannon's entropy value of the code.
 * @return the PHD readability value.
 */
public double calculatePhd(double volume, int lines, double entropy) {
    double regressionVar = Y_INTERCEPT + (VOLUME_COEF * volume) + (LINES_COEF *
    lines) + (ENTROPY_COEF * entropy);

    return 1 / (1 + Math.exp(-regressionVar));
}
```

Código 1 – Método que aplica a fórmula PHD e respetivo comentário Javadoc

Algumas imagens do Javadoc gerado para o *Readability Checker* podem ser consultadas na secção A.2 dos anexos.

4.1.5 Testes

A fase de testes é imprescindível em qualquer projeto de software que apresente rigor no seu desenvolvimento. Este é um processo que tem em vista verificar se os resultados obtidos pelo sistema e/ou os seus componentes correspondem ao esperado (Guru99, 2018b). Ainda que não permita garantir a total inexistência de *bugs*, esta fase serve para evitar que eles ocorram.

São vários os tipos de testes que podem ser efetuados a uma aplicação. O próprio processo de desenvolvimento pode seguir metodologias baseadas em testes. Os tipos de testes recaem em duas categorias: testes funcionais e testes não funcionais. Os primeiros pretendem verificar se as funções do software operam em conformidade com os requisitos especificados, enquanto que os segundos servem para avaliar aspetos não funcionais da aplicação, como por exemplo performance, usabilidade, ou segurança (Guru99, 2018a). O processo de desenvolvimento e o tipo de testes a efetuar devem ser ponderados mediante a aplicação e a equipa em causa.

Como já foi referido, optou-se por, não utilizando um padrão de desenho de software, desenvolver o código de forma a que os diferentes componentes do mesmo fossem o mais modulares possível, promovendo a reutilização do código. Dessa forma, o *plugin* desenvolvido é constituído por vários módulos que funcionam em conjunto. Tendo isso em conta, optou-se por recorrer a uma metodologia de testes funcionais que passou pela realização de testes unitários aos vários módulos da aplicação desenvolvida.

Testes unitários são testes onde as mais pequenas partes testáveis de um software, chamadas de unidades, são individualmente e independentemente escrutinadas, de modo a garantir que o seu funcionamento está de acordo com os parâmetros estipulados (Rouse, 2017). Estes testes ajudam também a garantir que futuras adições ou alterações efetuadas ao código não comprometerão o normal funcionamento dos módulos já implementados e, consequentemente, da aplicação como um todo. Para garantir que as alterações realizadas a

um módulo não resultaram em erros de funcionamento, basta para tal correr o conjunto de testes que o testam e, caso haja algum problema, esses mesmos testes mostram o que há de errado com o código.

4.1.5.1 Abordagem Tomada

Recorreu-se ao JUnit para a realização dos referidos testes. JUnit é uma *framework* de código aberto desenhada com o propósito de permitir aos desenvolvedores escrever e correr testes em aplicações Java.

Em adição, foram utilizadas as *frameworks* Mockito e PowerMock para possibilitar o *mocking* de objetos. Por vezes, um objeto a ser testado pode ter dependências sobre outros objetos mais complexos. De forma a isolar o comportamento do objeto em causa, desses objetos mais complexos, é prática comum recorrer à criação de *mocks*[11]. Esta prática mostra-se útil quando não é viável incorporar os objetos reais nos testes. O Mockito serve para criar estes objetos que simulam o comportamento de outros objetos, já o PowerMock tem a capacidade de testar, por exemplo, invocações a métodos estáticos ou código escrito por terceiros que não pode ser alterado.

Uma vez que grande parte da lógica está implementada nos *packages* *org.myorg.readabilitychecker.logic* e *org.myorg.readabilitychecker.formulas.logic*, foram escritos testes unitários que permitem testar todas as classes destes dois *packages* e respetivos métodos.

A abordagem tomada para cada módulo dependeu da finalidade do mesmo e da forma como foi feita a sua implementação. De um modo geral, para os métodos que recebem argumentos nos seus parâmetros, foram testados vários tipos de *input* para cada um destes. Por exemplo, se um método recebe uma lista como parâmetro, é testado o comportamento desse método no caso de a lista recebida cumprir com os parâmetros estabelecidos (conter objetos válidos), no caso de estar vazia e no caso de não ter qualquer espaço de memória alocado (vulgarmente referido como estar a *null*). Também foi testado o comportamento das diferentes unidades, no caso de os objetos em causa conterem algum tipo de informação errada, por exemplo, um valor que tenha de ser sempre maior que zero, ser negativo.

O *mocking* mostrou-se particularmente útil para testar funcionalidades que recorrem à API do NetBeans. De forma a analisar o projeto que está ativo, o *Readability Checker* verifica qual é o ficheiro que está a ser apresentado no IDE e carrega todos os ficheiros Java pertencentes ao mesmo projeto. Para isso, o *plugin* recorre-se da API do NetBeans. Esta API possui, entre outras, a capacidade de retornar objetos com informação dos projetos que estão abertos no IDE. Como o JUnit, por si só, não permite testar isso diretamente, e esses objetos são demasiado complexos para serem inicializados pelo desenvolvedor, optou-se por fazer *mocking* aos mesmos e, em seguida, testar o funcionamento da aplicação mediante a informação que estes retornam. O Mockito permite forçar objetos criados através de *mocking*

[11] Objetos que simulam o comportamento de outros objetos reais, de forma controlada.

a retornar um valor predefinido no caso de ser invocado um método através dos mesmos. Tome-se o seguinte excerto de código como exemplo:

```
Mockito.when(projectObjMocked.getProjectDirectory()).thenReturn(someFileObject);
```

Código 2 – Exemplo de utilização do Mockito

Este excerto obriga o objeto *projectObjMocked* (criado por *mocking*, recorrendo ao Mockito) a retornar o objeto *someFileOnject* previamente inicializado (por *mocking* ou não). No lugar de *someFileObject*, pode ser colocado qualquer objeto do mesmo tipo, manipulado da forma que melhor convém para o teste em causa. Sendo assim, tornou-se possível forçar diversos objetos a retornar diferentes valores para os métodos que eles oferecem, e verificar o comportamento da aplicação mediante os vários tipos de retorno obtidos.

4.1.5.2 Nomenclatura

No momento, a aplicação desenvolvida não tem uma dimensão considerável, no entanto, futuramente, com a adição de novas funcionalidades, o número de componentes naturalmente aumentará. Desse modo, novos testes terão de ser adicionados. Quando uma aplicação contém muitos testes e um deles falha, o desenvolvedor pode levar demasiado tempo a encontrar a causa do erro, no caso de os nomes dados aos testes não serem explicativos da situação em teste. Num caso hipotético, se um teste chamado *testGetBytesFromMethod* falhar, não é possível saber imediatamente o que correu mal neste cálculo. Não é possível saber as condições que levaram a que o teste falhasse. Um outro problema que este tipo de nomenclatura apresenta, é o facto de conter o nome do próprio método. Se no futuro, o nome do método for alterado, os nomes de todos os testes sobre esse método terão também de sofrer alterações. Para evitar este tipo de problemas, foram sugeridas ao longo dos anos várias convenções de nomenclatura para testes. De um modo geral, estas convenções sugerem incluir no nome dos testes a circunstância que está a ser testada e o resultado esperado.

A nomenclatura utilizada na realização dos testes do *Readability Checker* foi a *Should_ExpectedBehavior_When_StateUnderTest*. Esta nomenclatura descreve sucintamente o que deve acontecer e em que situação deve ocorrer. Por exemplo, o nome *Should_ThrowException_When_MethodObjIsNull* indica que o teste em causa verifica se uma exceção é lançada, no caso de o objeto do tipo *Method* recebido por parâmetro não estar alocado em memória. Deste modo, espera-se que no futuro, quando a aplicação tiver um número significativamente maior de componentes, o tempo perdido a analisar testes que falham após alterações efetuadas ao código seja consideravelmente reduzido.

4.2 Comments Ratio

Esta fórmula tem como propósito relacionar o número de linhas de código com o número de linhas com comentários, de forma a avaliar a legibilidade do software.

A implementação desta fórmula foi pensada de forma a ser possível aplicá-la a métodos ou a ficheiros Java. Para efetuar o cálculo da legibilidade, o *Comments Ratio* necessita de duas variáveis que podem ser facilmente retiradas do código. A abordagem para o levantamento destas duas variáveis é ligeiramente distinta para métodos e para ficheiros Java.

Nos próximos subcapítulos é detalhada implementação desta fórmula no *Readability Checker*. Em 4.2.1 e 4.2.2 são especificadas as considerações e pressupostos tomados sobre cada uma das duas variáveis que esta fórmula avalia e a forma como foi feita a implementação do *parsing* das mesmas.

Em 4.2.3 são feitas algumas observações sobre como é efetuado o cálculo da fórmula e por fim, em 4.2.4, é apresentado um exemplo prático da aplicação desta fórmula num excerto de código.

4.2.1 Número de Linhas de Código (LOC)

A definição concreta de LOC (ou SLOC, do inglês *Source Lines of Code*) depende, em grande parte, da fonte onde é retirada, no entanto, de um modo geral, existem dois tipos de LOC (ProjectCodeMeter, 2018):

- LOC físicas, que são a contagem de linhas no código do programa, incluindo comentários e linhas em branco.
- LOC lógicas, que são, normalmente, o número de declarações.

No caso desta fórmula, Aggarwal e a sua equipa referem-se às LOC como sendo "o número total de linhas de código (incluindo comentários)", o que leva a crer que eles se estão a referir às LOC físicas.

Tendo em conta a definição de LOC físicas apresentada anteriormente, considerou-se para a implementação desta fórmula que, no caso de um ficheiro Java, o seu LOC é o número total de linhas que o ficheiro tem. Alguns IDE's, como é o exemplo do NetBeans, ao efetuarem a formatação automática do código, adicionam uma linha em branco depois da última linha com código. Esta não é considerada para o cálculo do *Comments Ratio*.

Antes de abordar a contagem do número de linhas de um método, importa primeiro perceber que, para o JavaParser, os comentários que aparecem imediatamente antes da declaração de um método são considerados como comentários associados ao mesmo. Normalmente este é um comentário Javadoc, mas pode também ser um comentário de linha única ou um comentário em bloco. Mais detalhes sobre esta atribuição podem ser encontrados no endereço em rodapé[12].

[12] https://github.com/javaparser/javaparser/wiki/Comments-%28Attribution%29

Uma questão que surgiu aquando da implementação desta fórmula foi se os comentários associados ao método deviam ser considerados como parte integrante do mesmo ou não. Aggarwal e os restantes investigadores não fazem referência direta a isto, pelo que surgiu aqui espaço para diferentes interpretações.

Como referido em 3.1.1, o modelo de manutenibilidade onde é proposto o *Comments Ratio* apresenta a qualidade da documentação como um dos três aspetos a ter em conta e o Javadoc faz parte da documentação do software. Contudo, no código-fonte, este tipo de documentação aparece em forma de comentário e estando ele associado ao método, tal significa que pretende esclarecer o que este faz. A acrescentar a isto, o que estes investigadores avaliam na qualidade da documentação é a legibilidade do texto. Para isso, eles usam a fórmula de legibilidade de texto *Gunning's Fog Index* (Aggarwal, Singh and Chhabra, 2002). No caso deste trabalho, a legibilidade do texto do Javadoc não é considerada, mas sim o número de linhas que cada comentário deste tipo apresenta, que é a métrica em que os investigadores se baseiam.

Sendo assim, optou-se por considerar os comentários associados a cada método como parte integrante dos mesmos, o que faz com que o número total de linhas de um método seja a soma entre o número de linhas do corpo do método e o número de linhas do comentário associado.

Para um ficheiro Java, o número de linhas de código pode ser conseguido determinando o número da última linha do mesmo. Isto consegue-se determinando a posição final da *CompilationUnit*[13], recorrendo ao método *getEnd* proveniente da interface *NodeWithRange*.

Já no caso de métodos Java, sendo os mesmos representados por um *Node*, o número de linhas de código é conseguido subtraindo a sua posição final à sua posição inicial (*getBegin*) e somando um ao resultado dessa subtração. A este valor é ainda somado o número de linhas que o comentário associado ao método contém. Como o comentário associado é do tipo *Comment*, e este estende a classe *Node*, o número de linhas que o comentário associado tem é conseguido da mesma forma que o número de linhas do método.

De modo a apresentar uma visão mais concreta sobre a forma como é efetuado o *parsing* desta variável, na Figura 24 incluída na secção A.3 dos anexos, está representado o diagrama de sequência que demonstra a obtenção do número de linhas de código e do número de linhas com comentários (LOM) para os ficheiros Java de um projeto.

Em suma, estas foram as considerações tomadas para a atribuição dos valores do LOC:

- O número de linhas de um ficheiro Java é igual ao número da última linha que contenha algum elemento de código (uma instrução, o final de uma declaração, ou um comentário);

[13] É a raíz da AST. É um objeto que, por norma, representa um ficheiro Java analisado pelo JavaParser.

- O número de linhas de um método é a contagem de linhas desde o início até ao final da declaração do mesmo, somando o número de linhas que o comentário associado tem (normalmente, este é um comentário Javadoc);
- Linhas em branco são consideradas para o cálculo das LOC e das LOM para ficheiros Java e para métodos;
- Linhas com comentários Javadoc são consideradas no cálculo das LOC para ficheiros Java;
- A linha em branco adicionada automaticamente pela formatação automática de alguns IDE's não é considerada para os cálculos do *Comments Ratio*.

4.2.2 Número de Linhas com Comentários (LOM)

Aggarwal e os seus colegas definem LOM como sendo o total de linhas com comentários no código (Aggarwal, Singh and Chhabra, 2002). Existem três tipos de comentários que a linguagem Java suporta (Walrath and Campione, 1997) e todos eles são considerados na implementação da fórmula no *Readability Checker*:

- Comentários de linha única (iniciados por // e terminados no final da linha);
- Comentários de bloco (iniciados por /* e terminados por */);
- Comentários Javadoc (iniciados por /** e terminados por */).

Sendo assim, o LOM de um ficheiro Java é a soma de todas as linhas com comentários que o ficheiro contém. No caso dos métodos, o valor de LOM é a soma entre o número de linhas com comentários dentro do corpo do método e o número de linhas que o comentário associado ao método tem.

O JavaParser permite obter todos os comentários contidos numa *CompilationUnit* através do método *getAllContainedComments*. Este método retorna uma lista com elementos do tipo *Comment*. Como foi mencionado em 4.2.1, a classe *Comment* é uma extensão de *Node*, pelo que o número de linhas que determinado comentário apresenta é obtido da mesma forma que qualquer outro elemento *Node*: subtraindo o número da linha em que ele termina ao número da linha em que ele começa, e adicionando um.

Como foi referido no subcapítulo anterior, o diagrama de sequência que apresenta a obtenção do LOC e do LOM para ficheiros Java está representado na Figura 24, incluída na secção A.3 dos anexos.

Um sumário com as considerações tomadas para o cálculo do LOM é apresentado de seguida:

- São considerados todos os três tipos de comentários (comentários de linha única, comentários em bloco e comentários Javadoc);

- Linhas que contenham tanto declarações como comentários são consideradas como linhas com comentários.

4.2.3 Considerações Sobre o Cálculo da Fórmula e Apresentação de Resultados

O *Readability Checker* apresenta os resultados do cálculo desta fórmula de duas formas distintas. Na janela principal apresenta o valor de legibilidade do projeto. Clicando no botão *Detailed Results* é aberta uma janela que mostra a legibilidade de todos os ficheiros Java e respetivos métodos que os formam. Nesta janela, são apresentados também os valores de cada uma das variáveis para cada método, ou ficheiro envolvidos nos cálculos.

O valor de legibilidade do projeto é a média do valor de legibilidade de todos os ficheiros que o formam. Se um ficheiro não tiver qualquer comentário, a fórmula não lhe é aplicada, uma vez que isso levaria a uma divisão por zero e a um consequente resultado indefinido. Para além disso, esse ficheiro é desconsiderado para o cálculo da média da legibilidade do projeto. Pelo mesmo motivo, caso um método não contenha comentários, o cálculo da legibilidade não lhe é aplicado.

Aggarwal e a sua equipa consideraram, como referido em 3.2.1.1, que valores de legibilidade entre zero e cinco são bons, entre cinco e oito são médios e superiores a oito são maus, contudo, como foi demonstrado em 3.3.3, estes valores não devem ser levados à risca. Ao invés, uma visão crítica deve ser tomada sobre os valores de legibilidade adequados para cada projeto.

4.2.4 Exemplo

O seguinte excerto de código serve para possibilitar um melhor entendimento sobre como funciona a atribuição dos valores a cada variável:

```
package example;

public class Example {

    // random comment

    /**
     * Sums the values of an array of type int.
     *
     * @param values array of type int.
     * @return the sum of all the values in <code>values</code>.
     */
    public int sumIntArrayValues(int[] values) {
        int returnValue = 0; // return variable initialization

        /*
         * Loop the array and sum values
         */
        for (int i = 0; i < values.length; i++) {
            returnValue += values[i];
        }
```

```
            // return sum
            return returnValue;
        }
    }
```

Código 3 – Declaração de uma classe Java para demonstrar o funcionamento do Comments Ratio

Este exemplo contém declarações lógicas e comentários de todos os três tipos mencionados anteriormente. A Tabela 5 sumariza os valores do LOC e do LOM para o código do ficheiro Java apresentada e para o método *sumIntArrayValues*.

Tabela 5 – Sumário dos valores envolvidos no cálculo do código apresentado

Variável	**Ficheiro**	**Método**
LOC	26	19
LOM	12	11
Comments Ratio	2.17	1.73

O seguinte diagrama de atividade procura clarificar o fluxo de trabalho envolvido no funcionamento desta fórmula:

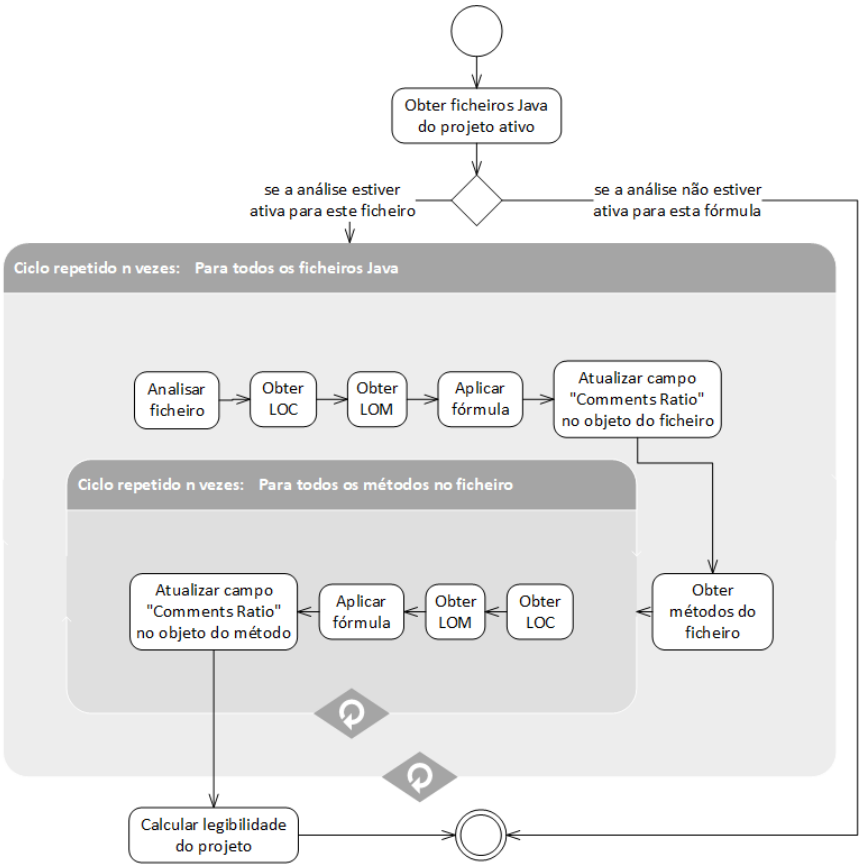

Figura 10 – Diagrama de Atividade - Comments Ratio

4.3 SRES

Como foi mencionado anteriormente, das três fórmulas integradas no *Readability Checker*, o *Software Readability Ease Score* é a única que não foi implementada de raiz. Em 2.2.4.2 foi referido que Nadeem Abbas desenvolveu uma aplicação intitulada POGJE, onde é implementada a lógica da métrica de legibilidade proposta por Börstler, Caspersen e Nordström, em 2007, de forma a possibilitar a avaliação da legibilidade de ficheiros Java. Essa implementação foi utilizada no *Readability Checker*. A sua integração vai ser detalhada ao longo deste capítulo.

Tal como acontece com a fórmula *Comments Ratio*, o SRES também necessita de apenas duas variáveis para efetuar o cálculo da legibilidade, no entanto, essas duas variáveis não são tão diretas e objetivas. Isto porque esta fórmula de legibilidade procura relacionar características da semântica do texto em linguagem natural com características semânticas de linguagem de código. Desse modo, os investigadores ponderaram vários fatores sobre o que considerar para a inclusão de cada uma das duas variáveis que esta fórmula contempla.

Nos próximos subcapítulos é aprofundada a integração desta fórmula no *Readability Checker*. Numa primeira instância é feita uma análise à implementação do POGJE e é explicada a sua integração com o *plugin* desenvolvido com este trabalho.

De seguida, em 4.3.2, é feita uma abordagem às características do código que são recolhidas para calcular as duas variáveis necessárias para esta fórmula.

Em 4.3.2 e 4.3.3 é detalhada a forma como são calculadas as variáveis ASL e AWL, necessárias para o cálculo do SRES.

Em 4.3.5 são apresentadas algumas considerações que foram tomadas para o cálculo desta fórmula, aquando da integração do POGJE com o *plugin* desenvolvido e, por fim, em 4.3.6, é apresentado o exemplo da declaração de uma classe Java. Essa declaração é analisada, de forma a mostrar como o POGJE interpreta o código e o decompõe nas várias variáveis envolvidas nos cálculos do SRES.

4.3.1 Implementação do POGJE e Integração com o Readability Checker

Toda a implementação do POGJE está detalhada na tese intitulada *Properties of "Good" Java Examples*, escrita por Nadeem Abbas (Abbas, 2010). De forma breve, o POGJE recorre à utilização do ANTLR para efetuar o *parsing* do código. Este *parser generator* gera, recorrendo a uma gramática da linguagem Java, os seguintes três componentes necessários para efetuar a análise de código: um *Java Lexer*, um *Parser* e um *Tree Parser*.

Uma limitação que esta implementação apresenta, é o facto de apenas suportar totalmente a versão 1.5 do Java, quando à data do desenvolvimento deste trabalho, a versão mais atualizada ser a 1.11 (mais conhecida por Java 11). Nesse caso, podem surgir alguns problemas no *parsing* do código, caso este possua características que foram adicionadas em

versões mais recentes do Java (p.e. a utilização de *lambda expressions*, introduzidas na versão 1.7 do Java).

Foi ponderada uma solução que passaria por descompilar o ficheiro JAR do POGJE, utilizar o ANTLR em conjunto com uma gramática do Java 11 para gerar novos ficheiros *JavaLexer.java*, *JavaParser.java* e *JavaTreeParser.java*, substituir os ficheiros antigos pelos atualizados e voltar a compilar o código. No entanto, a grande maioria da lógica do POGJE está diretamente codificada no *JavaTreeParser.java*, pelo que isso obrigaria a uma alteração completa deste ficheiro. Sendo ele um ficheiro com mais de 10,000 linhas de código, tais alterações poderiam não só originar *bugs* na aplicação, mas também desfasamentos nos resultados dos cálculos.

Tendo isto em conta, a opção que se mostrou mais viável foi a de não efetuar qualquer alteração e utilizar a biblioteca da forma como foi originalmente implementada. Deste modo, ainda que num reduzido número de casos não seja possível efetuar o *parsing* do código, os resultados de legibilidade obtidos serão sempre fiéis às ideias colocadas em prática por Abbas e Börstler.

O diagrama de componentes ilustrado na Figura 11 representa os elementos principais que interferem no funcionamento do POGJE.

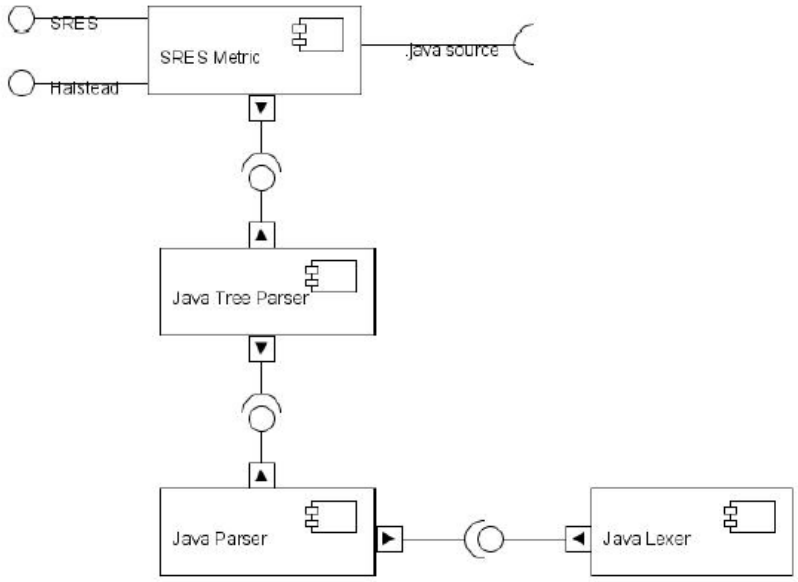

Figura 11 – Diagrama de Componentes - POGJE (retirado de (Abbas, 2010))

O ficheiro JAR desta aplicação está disponível *online*[14]. Isto possibilitou a sua fácil integração com o *Readability Checker*, já que ele pôde ser adicionado ao projeto como uma dependência, permitindo a sua utilização de modo semelhante ao de uma biblioteca Java.

[14] https://www.bth.se/wp-content/uploads/2018/03/Pogje.zip

Ao nível da implementação, o *Readability Checker* utiliza o método *countMetrics* da classe *ParserInterface*, pertencente ao POGJE, para efetuar o *parsing* do código. Para além do *parsing*, e recorrendo a outros componentes da aplicação, este método encarrega-se de efetuar a contagem das três variáveis necessárias para calcular o ASL e o AWL: contagem de palavras Java, contagem de frases Java, e comprimento das palavras Java. Com estas variáveis determinadas, é então possível calcular o resultado do SRES, recorrendo à fórmula apresentada na equação 37.

A Figura 25, incluída na secção A.4 dos anexos, apresenta, de forma resumida, a interação entre o *Readability Checker* e o POGJE, na obtenção das variáveis necessárias para o cálculo do SRES.

4.3.2 Palavras, Frases e Sílabas

A fórmula FRES utiliza, como referido em 2.2.2.1, o comprimento médio das frases, ou ASL, e a média de sílabas por palavra, ou ASW, para avaliar a legibilidade de texto. Para tal, três variáveis são necessárias para possibilitar a realização desse cálculo. São elas o número de palavras, o número de frases e o número de sílabas. Dessa forma, recorrendo às três variáveis referidas, o ASL e o ASW são calculados através das fórmulas representadas pelas equações 50 e 51, respetivamente.

$$ASL = \frac{Número\ de\ Palavras}{Número\ de\ Frases} \tag{50}$$

$$ASW = \frac{Número\ de\ Sílabas}{Número\ de\ Palavras} \tag{51}$$

Por seu turno, o SRES utiliza uma abordagem semelhante, mas com algumas adaptações, dado o seu foco ser a avaliação de software e não de texto. O SRES também utiliza o ASL como uma das variáveis, porém o ASW é representado nesta fórmula pelo AWL.

Nos subcapítulos 4.3.3 e 4.3.4, estão detalhados os cálculos dessas duas variáveis e a sua implementação no *Readability Checker*, no entanto, e de forma a tornar mais percetível as semelhanças entre as duas fórmulas, a Tabela 6 sumariza as variáveis envolvidas no FRES e as respetivas equivalências no cálculo do SRES.

Tabela 6 – Variáveis utilizadas pelo FRES e respetivas interpretações do SRES

FRES	**SRES**
Comprimento Médio das Frases (ASL)	Comprimento Médio das Frases (ASL)
Média de Sílabas por Palavra (ASW)	Comprimento Médio das Palavras (AWL)
Palavras	Lexemas

FRES	SRES
Frases	Declarações
Sílabas por Palavra	Comprimento das Palavras

4.3.3 Comprimento Médio das Frases (ASL)

Como referido em 4.3.2, o ASL é calculado pelo FRES através da divisão entre o número de palavras e o número de frases. Para esta variável, o SRES utiliza a mesma abordagem, ou seja, o cálculo do comprimento médio das frases para esta fórmula é baseado na contagem de palavras e frases Java.

Palavras são para o SRES interpretadas através de lexemas, mais especificamente, identificadores, palavras-chave e outros *tokens* léxicos. Cada um destes elementos, separados por um espaço, são considerados como sendo uma palavra. A contagem de palavras é então a soma de todas as palavras que o código contém. (Abbas, 2010)

Já as frases são para o SRES, em traços gerais, delimitadas por ponto e vírgula ';' e por chavetas '{}'. Dito por outras palavras, o SRES interpreta declarações ou blocos (conjuntos de declarações) Java como sendo frases. A interpretação desta característica é um pouco menos direta que a interpretação das palavras. Algumas exceções foram tidas em conta por Abbas e Börstler, de forma a cobrir cenários mais ambíguos, por exemplo o caso da ocorrência de blocos de código imbricados, ou acesso a membros da classe. Todas essas exceções estão detalhadas na tese escrita por Nadeem Abbas. (Abbas, 2010)

Posto isto, o cálculo do ASL traduz-se pela seguinte fórmula:

$$ASL = \frac{Número\ de\ Palavras}{Número\ de\ Frases} \tag{52}$$

4.3.4 Comprimento Médio das Palavras (AWL)

A segunda variável que o FRES utiliza para o cálculo da legibilidade de texto é o número médio de sílabas por palavra, ou ASW. No caso do SRES, e porque esta fórmula interpreta o comprimento das palavras como sendo sílabas, a segunda variável que é avaliada é média do comprimento das palavras, ou AWL. Para tal, o SRES também recorre ao número de palavras Java, porém, ao contrário do que acontece com o FRES, esta fórmula utiliza o comprimento das palavras como substituto da variável "sílabas por palavra", como mostra a Tabela 6.

O comprimento de palavras é, então, a soma, em número de caracteres, do comprimento de todas as palavras do código, ao passo que o número de palavras é, como foi dito no subcapítulo anterior, a soma de todas as palavras contidas no código. (Abbas, 2010)

Posto isto, a fórmula para o cálculo do AWL é a seguinte:

$$AWL = \frac{Comprimento\ das\ Palavras}{Número\ de\ Palavras} \quad (53)$$

4.3.5 Considerações Sobre o Cálculo da Fórmula e Apresentação de Resultados

Tal como acontece com o *Comments Ratio*, o *Readability Checker* apresenta os resultados desta fórmula de duas formas distintas. Na janela principal apresenta o valor de legibilidade do projeto em causa e na janela dos resultados detalhados são apresentados os resultados da legibilidade de cada ficheiro, juntamente com os respetivos valores do ASL e do AWL.

O valor da legibilidade do projeto é a média do valor de legibilidade de todos os ficheiros pertencentes ao mesmo. Caso algum ficheiro tenha o valor do ASL ou do AWL igual a zero, significa que ele não tem palavras ou frases Java (i.e., ocorreu algum problema durante *parging* do ficheiro), sendo esse mesmo ficheiro desconsiderado para o cálculo da média.

De um modo geral, quanto menor for o valor do ASL e do AWL, mais facilmente o código vai ser lido e compreendido, uma vez que o tamanho médio das frases e das palavras é menor (Abbas, 2010). Sendo assim, quanto menor for o valor do SRES, mais legível será o código. Na publicação de 2007, onde Börstler e a sua equipa introduzem pela primeira vez o conceito do SRES, eles indicam que o valor de limite de referência para o resultado de legibilidade é de 7, e pode apresentar um desvio de ±2 (Börstler, Caspersen and Nordström, 2007). Já na tese escrita por Abbas, ele atribui um valor limite para o ASL de cinco e um valor limite para o AWL de seis (Abbas, 2010). Como a ferramenta que ele desenvolveu, apesar de calcular estas duas variáveis, não aplica a fórmula SRES, ele não refere qualquer valor limiar para o resultado do cálculo, no entanto, ao calcular o SRES recorrendo aos valores do ASL e do AWL que ele apresenta ($\boldsymbol{SRES = 5 - 0.1 \times 6}$), obtém-se o valor 4.4.Fica assim inconclusivo o valor de referência exato, mas procurando um meio termo entre estes dois valores, pode-se pressupor que este rodará o valor de legibilidade 6. Tal valor deduz-se através do resultado intermédio entre os dois valores limite apresentados: $\left(\frac{\mathbf{7-4.4}}{\mathbf{2}}\right) + \mathbf{4.4} = \mathbf{5.7} \approx \mathbf{6}$.

4.3.6 Exemplo

Para uma melhor visualização sobre a forma como é efetuada a interpretação do código pelo POGJE, o Código 4 será analisado e decomposto nas várias variáveis envolvidas nos cálculos.

```
package demosres;

public class DemoSRES {
    public static void main(String[] args) {
        String str = “Hello, world!”;
    }
}
```

Código 4 – Declaração de uma classe Java para demonstrar o funcionamento do SRES

De forma a facilitar a explicação e a compreensão, o código apresentado é uma simples declaração de uma classe Java, com declaração do método *main* e declaração e inicialização de uma variável do tipo *String*.

4.3.6.1 Frases

Ao nível das frases, e como Nadeem Abbas explica na sua tese, elas são delimitadas por ponto e vírgula e por chavetas. Sendo assim, verifica-se que a classe apresentada contém quatro frases, sendo elas apresentadas de seguida:

1.
```
package demosres;
```

2.
```
public class DemoSRES {
    public static void main(String[] args) {
        String str = “Hello, world!”;
    }
}
```

3.
```
public static void main(String[] args) {
    String str = “Hello, world!”;
}
```

4.
```
String str = “Hello, world!”;
```

4.3.6.2 Palavras

Por seu turno, são detetadas 16 palavras com um comprimento total de 91 caracteres pelo POGJE. A listagem dessas palavras é apresentada na Tabela 7, juntamente com o respetivo comprimento.

Tabela 7 – Palavras contidas no excerto de código apresentado e respetivo comprimento

Palavra	**Comprimento**
package	7
demosres	8
public	6
class	5
DemoSRES	8
public	6
static	6
void	4

Palavra	Comprimento
main	4
String	6
[]	2
args	4
String	6
str	3
=	1
“Hello, world!”	15

4.3.6.3 Cálculo do AWL, do ASL e do SRES

Aplicando as fórmulas que foram introduzidas anteriormente, é então possível calcular o resultado do ASL da forma $\boldsymbol{ASL} = \frac{\mathbf{16}}{\mathbf{4}} = \mathbf{4}$ e do AWL da forma $\boldsymbol{AWL} = \frac{\mathbf{91}}{\mathbf{16}} = \mathbf{5.69}$. Aplicando a fórmula de legibilidade em causa, obtém-se o seguinte resultado de legibilidade: $\boldsymbol{SRES} = \mathbf{4} - \mathbf{0.1} \times \mathbf{5.69} = \mathbf{3.43}$.

4.4 PHD

A abordagem para o cálculo desta fórmula é diferente da tomada para o cálculo do *Comments Ratio* e do SRES. Como referido em 3.1.2, PHD é uma fórmula que foi baseada nas métricas desenvolvidas por Buse e Weimer. Estas métricas foram pensadas para serem aplicadas apenas em pequenos excertos de código. Os excertos que estes dois investigadores utilizaram para recolher as classificações atribuídas por desenvolvedores, variaram o seu tamanho entre as 4 e as 11 linhas de código (Posnett, Hindle and Devanbu, 2011). Tendo isto em mente, dois caminhos podiam ser tomados para a implementação da fórmula. O primeiro seria dividir o código em pequenos excertos, aplicar a fórmula a cada um deles e calcular a média. O segundo seria restringir a aplicação avaliar métodos com um máximo de linhas predefinido.

Nos próximos subcapítulos é detalhada a implementação desta fórmula no *Readability Checker* e são mencionadas todas as decisões tomadas.

Em primeiro lugar, será justificada a abordagem foi seguida para a forma como é efetuada a avaliação do código, de modo a ultrapassar o problema de tamanho apresentado.

De seguida, são abordadas as três variáveis que compõem o cálculo desta fórmula. Cada uma dessas variáveis é explicada em detalhe. São referidos e justificados todos os pressupostos e considerações tomados para cada uma delas e também é referido o modo como foi efetuada a sua implementação no *Readability Checker*.

De seguida, são referidas algumas considerações que foram tomadas no cálculo desta fórmula e para finalizar, é apresentado um exemplo que demonstra como são efetuados todos os cálculos intermédios necessários para o cálculo final da legibilidade.

4.4.1 Escolha da Abordagem Tomada

Como foi mencionado em 4.4, devido à forma como esta fórmula foi pensada, seria errado aplicá-la a excertos de código com muito mais de 11 linhas. Isto faria com que, dependendo do tamanho do código, houvesse um grande número de operadores e operandos e, consequentemente, o valor do volume seria demasiado elevado para ser usado com os coeficientes sugeridos pelos autores, levando a um resultado de legibilidade erróneo.

Desse modo, duas soluções poderiam ser tomadas. A primeira passaria por dividir os ficheiros Java em blocos com n linhas de código ($\mathbf{4} \leq \boldsymbol{n} \leq \mathbf{11}$), aplicar a fórmula a cada um desses excertos e, por fim, calcular a média dos resultados obtidos. A segunda solução seria limitar a aplicação a avaliar apenas métodos com um máximo de 11 linhas de código, já que este foi o número máximo de linhas utilizado por Buse e Weimer.

Por vários motivos, optou-se por não dividir o código em pequenos excertos, sendo o principal a complexidade de efetuar o *parsing* do código dessa forma. O JavaParser, por exemplo, não permite efetuar o *parsing* de excertos de código incompletos. Tome-se como exemplo o seguinte excerto de código:

```
int[] arr = {
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9,
    10
};
```

Código 5 – Declaração e inicialização de um array em múltiplas linhas de código

Este excerto de código apresenta a declaração e inicialização de um *array* de inteiros em 12 linhas de código diferentes. *Parsers*, como é o caso do JavaParser, interpretam este excerto de código como uma declaração única, independentemente de o *array* estar inicializado apenas numa linha de código ou em várias linhas. Dividir este excerto em dois e forçar o JavaParser a analisar ambas as partes levaria a um erro de *parsing*.

Mesmo que o JavaParser permitisse analisar excertos de código incompletos, vários pressupostos teriam de ser tomados. Um simples exemplo é o facto de um conjunto de chavetas '{}' ser considerado como um operador único segundo algumas estratégias de contagem (Abbas, 2010). No caso de uma classe que tenha o início da sua definição na linha 3 e o final na linha 20, dividindo o código em excertos de 11 linhas, a chaveta inicial e a chaveta

final seriam incluídas em excertos diferentes, logo, ter-se-ia de tomar um pressuposto sobre como interpretar conjuntos de chavetas que estejam em excertos diferentes. Como foi dito, este é apenas um exemplo simples. Conforme a complexidade do código e a ambiguidade dos casos aumentam, novos pressupostos teriam de ser tomados.

A implementação de uma ferramenta capaz de efetuar o *parsing* de excertos de código Java incompletos mostrou-se como uma possível opção, contudo o *parsing* de código incompleto não apresenta uma solução direta e concreta. Demasiadas ambiguidades surgem com este tipo de análise. Tendo acesso apenas a uma pequena parte do código-fonte, nem sempre é possível determinar com exatidão o papel que determinado *token* tem no código como um todo, caso não exista uma contextualização sobre o restante código em causa. Por exemplo, imaginando que se dividiria o Código 5 em duas partes, seria extremamente difícil para um *parser*, senão impossível, saber o significado da segunda parte do código sem ter nenhum contexto do restante. Esta segunda metade do código pode, por exemplo, fazer parte de um comentário e não de uma inicialização de um *array*. Desse modo, um estudo aprofundado sobre como deveria ser efetuado um *parsing* deste tipo teria de ser realizado, estudo esse que, além de demasiado exaustivo, levaria o foco deste trabalho para outro campo que não o da legibilidade de software. Como tal, essa solução foi descartada.

Uma outra opção seria a de utilizar uma biblioteca que já fizesse esse tipo de *parsing*. Barthélémy Dagenais e Laurie Hendren publicaram um artigo em 2008, intitulado *"Enabling Static Analysis for Partial Java Programs"*. Nele, os investigadores abordam o problema do *parsing* de programas incompletos. Com este estudo, desenvolvem uma biblioteca intitulada *Partial Program Analysis* (PPA) para o IDE Eclipse, capaz de efetuar o *parsing* de programas Java incompletos. Segundo os testes realizados, o PPA obteve uma percentagem de 91.2% de factos corretos ao analisar uma classe de cada vez e apenas 2.7% de factos erróneos (Dagenais and Hendren, 2008).

Contudo, a adição desta biblioteca ao *Readability Checker* mostrou-se inviável, já que este é um *plugin* direcionado para o IDE NetBeans, e o PPA é uma extensão do compilador Java do Eclipse, sendo a sua implementação profundamente dependente dos componentes desse mesmo compilador, e não a ideia geral.

Um último exemplo da limitação de dividir o código em pequenos excertos é o caso de se dividir o código a cada $\boldsymbol{x}$ linhas e surgir uma classe com $\boldsymbol{x + 1}$ linhas. Isto fará com que o código seja dividido num bloco com $\boldsymbol{x}$ linhas e noutro com apenas uma linha. Se a linha $\boldsymbol{x + 1}$ contiver apenas a chaveta que termina o corpo da classe, isto quererá dizer que o segundo excerto de código avaliado terá um operador e zero operandos. Sendo o cálculo do volume efetuado através da fórmula $\boldsymbol{V = N \log_2 n}$, o resultado deste cálculo seria impossível, já que o logaritmando, no caso, $\boldsymbol{n}$, deve sempre ser um número maior que zero.

Verifica-se então que dividir o código em pequenos excertos e calcular a média dos valores de legibilidade, para além de implicar um estudo demasiado extensivo, que não foca no âmbito deste trabalho, poderia levar a resultados não condizentes com a realidade. Dessa forma, a

solução que se optou por seguir foi a de limitar a avaliação de métodos com um máximo de 11 linhas de código para esta fórmula.

4.4.2 Volume

Num artigo publicado em 1977, Maurice Halstead desenvolveu uma teoria que tem como finalidade dar uma medida objetiva sobre a complexidade do software (Hamer and Frewin, 1982). Com este estudo, Halstead conseguiu chegar à teorização de várias fórmulas que pretendem avaliar determinados aspetos da complexidade do código. Um desses aspetos é o cálculo do volume.

O volume tem como finalidade avaliar os conteúdos de informação do programa (Verifysoft Technology GmbH, 2017). Dito de outra forma, representa o tamanho, em *bits*, de espaço necessário para o guardar (Virmani, 2017).

Como se pode verificar pela fórmula 10, o volume é calculado através do comprimento e do vocabulário do programa, sendo para isso necessário obter o número total e único de operadores e de operandos. A implementação do *parsing* destas variáveis do código seria uma opção, no entanto, e de forma a "não reinventar a roda", procurou-se utilizar uma implementação levada a cabo por terceiros. Como tal, procurou-se investigar algumas implementações das métricas de Halstead em Java, de uma forma que permitisse a sua integração com o *Readability Checker*. Esta investigação permitiu concluir que existem algumas implementações destas métricas destinadas a esta linguagem, estando elas disponibilizadas em repositórios de código, contudo, estas implementações carecem de testes que comprovem o quão corretos os resultados obtidos são. Apenas uma das implementações das métricas de Halstead levantadas por esta investigação apresenta, não só uma completa estratégia para a contagem das diferentes variáveis necessárias para esta métrica, mas também uma comparação com uma outra ferramenta que efetua os cálculos das métricas de Halstead (Abbas, 2010), ferramenta esta que é comercial. Essa implementação é a do POGJE, ferramenta que já é utilizada no *Readability Checker* para calcular o SRES. Sendo assim, esta foi a ferramenta utilizada para efetuar o cálculo do volume de Halstead no *plugin* desenvolvido.

Importa referir que não existe um *standard* definido que defina concretamente o que são operadores e operandos Java, no âmbito do cálculo das métricas de Halstead (Nandy, 2007). Sendo assim, diferentes abordagens podem ser tomadas, mediante o entendimento do desenvolvedor em causa. Detalhes sobre as considerações tomadas no POGJE para o *parsing* dos operadores e operandos estão especificados em (Abbas, 2010).

Como foi referido em 4.3.1, o POGJE recorre ao método *countMetrics* para efetuar o *parsing* do código e para fazer a contagem das variáveis envolvidas no cálculo do SRES e das métricas de Halstead. Esse método recebe uma variável do tipo *String* que contém o caminho para o ficheiro Java a ser testado. Já dentro do método, esse caminho é utilizado para instanciar um objeto do tipo *java.io.FileInputStream* e assim poder ser instanciado um objeto do tipo

ANTLRInputStream para ser feita a análise do código. Como no caso do PHD a avaliação recai sobre métodos Java e não sobre ficheiros, a utilização deste método não poderia ser feita exatamente da mesma forma. Enviar o código ao invés do caminho para um ficheiro não era uma solução viável, uma vez que não é possível criar o objeto *ANTLRInputStream* recorrendo a uma *String* com código Java. Desse modo, a solução passou por fazer uma pequena alteração no método *countMetrics*. Ao invés de utilizar esse método diretamente do POGJE, foi feita uma cópia do mesmo para o código do *Readability Checker*. A alteração efetuada ao *countMetrics* passou por criar um objeto do tipo *java.io.ByteArrayInputStream* instanciado utilizando uma *String* com código Java e assim poder utilizar esse objeto para instanciar o objeto *ANTLRInputStream*. Desse modo, ao invés de receber uma *String* com o caminho para um ficheiro Java, o método recebe, de igual forma, uma *String*, mas com o código a ser avaliado.

Sendo assim, o *Readability Checker* envia a declaração de um método para o *countMetrics* adaptado, e este, recorrendo aos componentes do POGJE, encarrega-se de a analisar em busca dos operadores e dos operandos. O próprio POGJE, recorrendo-se dos resultados obtidos por essa análise, efetua o cálculo do comprimento e do vocabulário do método. Utilizando esses valores, o *Readability Checker* aplica a fórmula, de forma a obter o volume do método em causa.

O diagrama de sequência representado na Figura 26, incluída na secção A.5 dos anexos, apresenta uma visão genérica da forma como é efetuado o cálculo do volume. Importa notar que neste diagrama não estão incluídos todos os passos necessários para este caso. Nele, apenas estão contempladas as alterações efetuadas ao método *countMetrics*, e a interação com o POGJE.

4.4.3 Linhas

Para esta fórmula os investigadores não definem concretamente as considerações tomadas para esta variável, dizem apenas que usam linhas de excertos de código para efetuar os cálculos. Contudo, eles referem que alguns dos excertos utilizados contêm comentários, pelo que tal leva a crer que eles recorrem às LOC físicas para avaliar o tamanho do código. Em 4.2.1 é apresentada a definição de LOC e os diferentes tipos de LOC existentes.

O número de linhas de código que o método contém é obtido de modo semelhante à forma como o *Comments Ratio* o faz, no entanto, ao contrário do que acontece com a implementação dessa fórmula, o comentário Javadoc associado ao método não é considerado como parte integrante do mesmo. Isto, porque apesar de esta fórmula considerar a utilização de linhas com comentários, ela não os utiliza como uma variável para o cálculo da legibilidade. Além disso, esta fórmula está limitada a um máximo de 11 linhas de código, pelo que incluir os comentários Javadoc, que para esta fórmula não inferem nada em concreto sobre a legibilidade do código, apenas limitaria ainda mais a utilização da mesma.

Sendo assim, o número de linhas do método é conseguido subtraindo a linha em que o método termina à linha em que ele é declarado, somando então um a esse resultado.

Apresenta-se de seguida um sumário das considerações tomadas para a obtenção do número de linhas de código no cálculo desta fórmula:

- O número de linhas de um método é a contagem de linhas desde o início até ao final da declaração do mesmo;
- Linhas com comentários dentro do corpo do método são consideradas;
- Linhas em branco dentro do corpo do método são consideradas;
- Ao contrário do *Comments Ratio*, comentários Javadoc não são considerados para o cálculo do número de linhas do método.

4.4.4 Entropia

A entropia é, na sua génese, um conceito da física, no entanto ela também pode ser aplicada no contexto da informação. Nos seguintes subcapítulos é apresentada uma definição de entropia para este contexto, juntamente com uma demonstração prática, e é detalhada a forma como foi implementado o cálculo da entropia no *Readability Checker*.

4.4.4.1 Definição de Entropia

No contexto em que este trabalho se insere, a entropia pode ser definida como sendo a complexidade, o grau de desordem, ou a quantidade de informação num conjunto de dados (Posnett, Hindle and Devanbu, 2011). Este conceito foi introduzido em 1948 pelo matemático americano Claude Shannon, no seu artigo *"A Mathematical Theory of Communication"* (Shannon, 1948).

Para perceber o conceito de entropia de informação, apresenta-se um exemplo retirado de (Serrano, 2017).

Imagine-se um caso onde existem três baldes com bolas vermelhas e azuis:

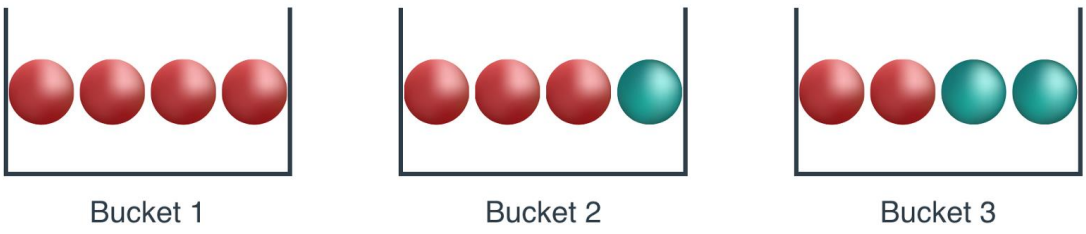

Figura 12 – Os baldes (retirado de (Serrano, 2017))

No caso de se retirar uma bola de cada balde, sabe-se que há 100% de probabilidade de tirar uma bola vermelha no balde um, 75% de probabilidade no balde dois e 50% de probabilidade no balde três. Por associação de ideias, é justo dizer que de entre todos os baldes, o balde número um dá a maior quantidade de conhecimento sobre qual bola se irá retirar e o número três, a menor quantidade de conhecimento. Entropia é, de certa forma, o oposto de conhecimento. Sendo assim, o balde um é o que tem menor quantidade de entropia e o balde três é o que tem maior quantidade de entropia.

Aplicando este conhecimento, foi então teorizada a fórmula 13 para que fosse possível calcular a entropia de um conjunto de dados. De modo a permitir a compreensão da lógica desta fórmula, faz-se em seguida referência a um exemplo retirado de (Vajapeyam, 2014), onde são apresentadas três probabilidades $\boldsymbol{p(x)}$:

- $\boldsymbol{p(a) = 0.5 = \frac{2}{4}}$
- $\boldsymbol{p(b) = 0.25 = \frac{1}{4}}$
- $\boldsymbol{p(c) = 0.25 = \frac{1}{4}}$

Aqui, $\boldsymbol{p(a)}$ é escrito no formato $\boldsymbol{\frac{1}{4}}$ e não $\boldsymbol{\frac{1}{2}}$, de forma a todas as três probabilidades terem o mesmo denominador.

Em $\boldsymbol{p(c) = \frac{1}{4}}$, o numerador um e o denominador quatro indicam que o valor 'c' é tido uma a cada quatro vezes, o que significa que podem existir até três outros valores que a variável pode tomar em outras alturas. Existem então quatro valores e será necessário o seguinte número de *bits* para os representar a todos: $\boldsymbol{\log_2(4) = \log_2 \frac{1}{p} bits = 2\ bits}$. Prova-se então que 'c' precisará de dois *bits* para ser representado de forma diferente dos outros valores e, dado que tem a mesma probabilidade, 'b' também necessitará de dois *bits* para ser representado.

No caso do valor 'a', entende-se que ele pode ser intercetado duas vezes a cada quatro tentativas. Por outras palavras, de quatro valores que a variável pode tomar, dois vão ser o valor 'a', um vai ser 'b' e outro vai ser 'c'. Para representar os dois valores 'a' de entre os quatro valores, não são necessárias duas representações diferentes, já que dois 'a' são os mesmos. Sendo assim, do conjunto total de *bits* pode ser subtraído $\boldsymbol{\log_2(2)}$ *bits*, já que, em princípio, é suposto esses $\boldsymbol{\log_2(2)}$ *bits* servirem para diferenciar os dois valores 'a' que são, na verdade, idênticos. Daqui conclui-se que para representar 'a' se tem $\boldsymbol{\log_2(4) - \log_2(2) = 1\ bit}$.

Para determinar a quantidade de armazenamento geral necessária para a variável, adicionam-se os valores determinados anteriormente para cada variável individual em proporção às suas frequências de ocorrência:

- No caso de 'a': $\boldsymbol{0.5 * 1\ bit}$

- No caso de 'b': $\mathbf{0.25 * 2}\ \boldsymbol{bits}$
- No caso de 'c': $\mathbf{0.25 * 2}\ \boldsymbol{bits}$

No total, fica $\mathbf{0.5 * 1 + 0.25 * 2 + 0.25 * 2 = 1.5}\ \boldsymbol{bits}$. Este é o resultado da aplicação da equação aos valores referidos.

Assim, mostra-se que entropia é uma medida direta do número de *bits* necessário para guardar a informação numa variável, tornando-a assim numa medida direta da "quantidade de informação" contida numa variável (Vajapeyam, 2014).

4.4.4.2 Implementação do Cálculo da Entropia

Para esta fórmula, Posnett e os restantes investigadores recorreram à contagem de termos para efetuar o cálculo da entropia. Termos podem ser *tokens* ou *bytes*. Como foi referido anteriormente, no caso do PHD os investigadores recorreram a ambos (Posnett, Hindle and Devanbu, 2011).

Neste contexto, um *byte* simboliza um caráter contido no código. Na área da informática, um caráter representa qualquer letra, número, espaço, sinal de pontuação, ou símbolo que possa ser escrito num computador, sendo que a lista completa de caracteres interpretados por um computador é definida pelo *American Standard Code for Information Interchange* (ASCII[15]) (Sharpened Productions, 2018). Ainda que os autores não façam referência a isso, esta definição indica que espaços e caracteres especiais, como por exemplo o caráter de nova linha ('\n'), devem ser considerados para os cálculos.

Ao nível da implementação do cálculo da entropia, ela apresenta dois pontos principais: o *parsing* dos termos e a aplicação da fórmula do cálculo da entropia.

O *parsing* dos termos é dividido em dois métodos que se encarregam respetivamente dos *tokens* e dos *bytes*.

O *parsing* dos *tokens* do código é efetuado recorrendo ao JavaParser. Esta biblioteca utiliza a classe *JavaToken* para identificar um *token* Java. Desta forma, recorrendo ao método *getTokenRange* que o JavaParser oferece, todos os *tokens* identificados por ele são colocados numa lista, que é retornada para o método que trata os termos.

Importa referir que, para este *parsing*, comentários foram considerados como sendo *tokens*, já caracteres de nova linha e espaços não foram considerados como tal (Pattis, 2004).

No que toca ao *parsing* de *bytes*, o objeto do tipo *Method*, representativo de um método, é enviado para o método que se encarrega de, recorrendo ao JavaParser, obter a declaração do método em causa e de, posteriormente, adicionar cada caráter do código a uma lista. Essa lista é então retornada ao método responsável pelo tratamento dos termos.

[15] Em português, Código Padrão Americano para o Intercâmbio de Informação, é um *standard* para a codificação de texto em computadores.

O método encarregue de tratar dos termos junta as duas listas recolhidas, normalizando-as para o tipo de dados *String*. Esta normalização mostra-se necessária, já que o cálculo da entropia deve ser aplicado à lista com todos os termos, sendo que *tokens* apenas podem ser representados pelo tipo de dados *String*. Esta nova lista normalizada é então enviada para um método que calcula a entropia de informação contida numa lista, utilizando a lógica da fórmula proposta por Claude Shannon. Esse método retorna um valor do tipo *double* que representa a entropia do método em causa.

A Figura 27, incluída na secção A.5 dos anexos, apresenta o diagrama de sequência simplificado, representativo da obtenção da entropia para os vários métodos de um ficheiro.

4.4.5 Considerações Sobre o Cálculo da Fórmula e Apresentação de Resultados

As outras duas fórmulas incluídas no *Readability Checker* permitem avaliar o código independentemente do seu tamanho, esta fórmula não. Dado que o PHD apenas avalia métodos com um tamanho máximo de 11 linhas de código, não é possível avaliar a legibilidade de uma classe, nem de projetos. Por esse motivo, e de modo a tornar mais prática a utilização desta fórmula, quando o utilizador clica no botão *Check Readability* do *Readability Checker*, o PHD não avalia todos os métodos com o máximo de linhas de código predefinido do projeto, mas apenas da classe que está aberta nesse momento.

Sendo assim, no ecrã principal do *Readability Checker*, ao invés de ser apresentado o resultado de legibilidade do projeto obtido por esta fórmula, é apresentado o número de métodos da classe em causa que foram avaliados pelo PHD. Selecionando a opção *Detailed Results* é aberta uma janela onde são apresentados os detalhes sobre cada método, nomeadamente o volume, o número de linhas, o valor da entropia e o respetivo resultado de legibilidade segundo o PHD.

Os autores da fórmula não referem um valor do PHD que seja a fronteira entre um resultado bom e um resultado mau de legibilidade, no entanto, devido à aplicação da fórmula logística, os resultados obtidos por esta fórmula variam entre zero e um. Quanto mais próximo de um for o resultado do PHD, mais legível é o código em causa.

4.4.6 Exemplo

De forma a apresentar uma visão mais concreta sobre o modo como é efetuado o cálculo da legibilidade através desta fórmula, neste subcapítulo são apresentados todos os cálculos envolventes para o método que se apresenta de seguida:

```
public static void exemplo() {
    String str = “Exemplo”;
    // Apresentar variável na consola
    System.out.println(str);
}
```

Código 6 – Declaração de um método Java para demonstrar o funcionamento do PHD

4.4.6.1 Volume

O cálculo do volume compreende duas variáveis: operadores e operandos. De forma a facilitar a visualização dos operandos e operadores, a Tabela 8 sumariza estas duas variáveis e o número de ocorrências de cada uma no código.

Tabela 8 – Lista de operadores e operandos contidos no código apresentado

Valor	**Operador**	**Operando**	**Ocorrências**
public	✓		1
static	✓		1
void	✓		1
exemplo		✓	1
{}	✓		1
String	✓		1
str		✓	2
=	✓		1
“Exemplo”		✓	1
;	✓		2
System.out.println(str)	✓		1
.	✓		2

Tendo em conta os resultados apresentados na Tabela 8, concluem-se assim os seguintes quatro resultados necessários para o cálculo do volume:

- $N1 = 11$
- $n1 = 9$
- $N2 = 4$
- $n2 = 3$

Com estes valores, é então possível calcular o comprimento do programa, o vocabulário e o volume, recorrendo respetivamente às fórmulas 8, 9 e 10:

- Comprimento

$$N = N1 + N2 = 11 + 4 = 15 \quad (54)$$

- Vocabulário

$$n = n1 + n2 = 9 + 3 = 12 \quad (55)$$

- Volume

$$V = N \log_2 n = 15 \log_2 12 = 53.77 \quad (56)$$

Conclui-se, então, que o valor do volume do código apresentado é 53.77.

4.4.6.2 Linhas

Esta é a variável mais direta que esta fórmula apresenta. Tendo em mente as considerações tomadas em 4.4.3 sobre o que esta fórmula considera como linhas de código, determina-se então que o Código 6 tem cinco linhas de código.

4.4.6.3 Entropia

Referiu-se em 4.4.4 que o cálculo da entropia contempla *tokens* e *bytes* que determinado excerto de código contém. A Figura 13 e a Figura 14 listam, respetivamente, os *tokens* e os *bytes* contidos no Código 6.

public	static	void
exemplo	(	)
{	String	str
=	“Exemplo”	;
// Apresentar variável na consola	System	.
out	.	println
(	str	)
;	}	

Figura 13 – Listagem dos tokens contidos no Código 6

p	u	b	l	i	c		s	t	a	t	i	c		v	o	i	d	
e	x	e	m	p	l	o	(	)		{	\r	\n					S	t
r	i	n	g		s	t	r		=		“	E	x	e	m	p	l	o
“	;	\r	\n					/	/		A	p	r	e	s	e	n	t
a	r		v	a	r	i	á	v	e	l		n	a		c	o	n	s
o	l	a	\r	\n					S	y	s	t	e	m	.	o	u	t
.	p	r	i	n	t	l	n	(	s	t	r	)	;	\r	\n	}		

Figura 14 – Listagem dos bytes contidos no Código 6

De notar que os espaços vazios na Figura 14 indicam que o caráter em causa é um espaço.

A lista de termos é a união de ambos os valores incluídos na Figura 13 e na Figura 14. Utilizando estes mesmos valores para aplicar a fórmula da entropia, recorrendo para isso ao método *calculateShannonEntropy* do *Readability Checker*, verifica-se então que o valor da entropia para o código apresentado é 4.98.

4.4.6.4 Cálculo do PHD

Tendo todas as três variáveis calculadas, é então possível aplicar o PHD. Dado que o resultado desta fórmula é obtido através do cálculo da função logística, é, em primeiro lugar, necessário calcular o valor da variável de regressão, que neste caso é representado pelo identificador z. É neste cálculo que são utilizadas as variáveis calculadas anteriormente.

$$z = 8.87 - 0.033 * 53.77 + 0.40 * 5 - 1.5 * 4.98 = 1.63 \quad (57)$$

Tendo a variável de regressão calculada, é então possível obter o valor de legibilidade. Para isso, recorre-se à utilização da fórmula 16.

$$\frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-1.63}} = 0.84 \quad (58)$$

Verifica-se com isto que, segundo a fórmula PHD, o valor de legibilidade do Código 6 é 0.89.

De forma a tornar mais percetível o fluxo de eventos para o cálculo desta fórmula, é apresentado na Figura 15 o diagrama de atividade com os eventos principais para este caso.

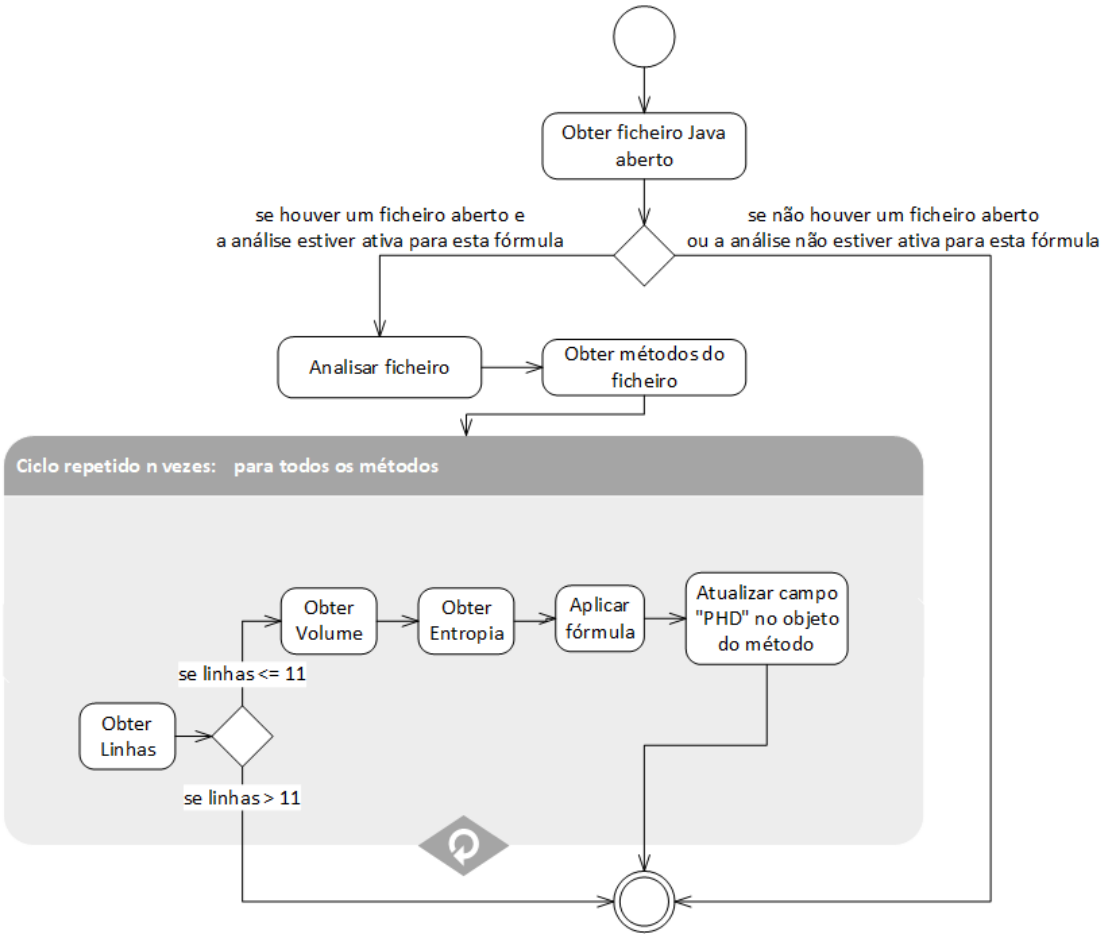

Figura 15 – Diagrama de Atividade - PHD

4.5 Avaliação da Solução

Depois de efetivada a sua implementação, mostrou-se necessária a utilização e avaliação do *plugin* por parte de outros desenvolvedores, de forma a perceber se os requisitos propostos por este trabalho foram atendidos.

Nos próximos subcapítulos é feita a avaliação da solução criada. Em 4.5.1 são apresentadas as hipóteses de avaliação, em 4.5.2 é detalhada a metodologia de avaliação que se mostrou pertinente para o caso, em 4.5.3 são apresentados os critérios de aceitação definidos para procurar tornar a avaliação da solução mais fidedigna e por fim, em 4.5.4 são analisados os resultados obtidos com a avaliação efetuada.

4.5.1 Hipóteses de Avaliação

Um dos pontos essenciais para a avaliação da solução é perceber quais as hipóteses que devem ser testadas, de forma a suportar os resultados do trabalho efetuado.

Como tem vindo a ser referido ao longo deste documento, este trabalho consistiu em desenvolver um *plugin* capaz de analisar o código escrito por um desenvolvedor e estimar a legibilidade do mesmo. Deste modo, consideram-se as seguintes hipóteses de avaliação para este caso:

- O *plugin* contribui para um maior cuidado por parte dos desenvolvedores em relação à legibilidade do código que escrevem.
- O *plugin* pode ser utilizado por iniciantes ao desenvolvimento de software (p.e. alunos do primeiro ano de Engenharia Informática), de modo a fomentar um maior cuidado em relação à legibilidade e consequente qualidade do código que estes escrevem, com vista a torná-los melhores profissionais.

4.5.2 Metodologia de Avaliação

Para responder de forma concreta às hipóteses de avaliação apresentadas em 4.5.1, seria necessário um estudo com uma duração considerável, de modo a perceber se a utilização prolongada do *plugin* desenvolvido teria um impacto considerável na melhoria da qualidade do código escrito por desenvolvedores e por iniciantes à programação.

Como o tempo disponível para testar a solução não permite a realização de um teste tão exaustivo, optou-se por testar o *plugin* junto de estudantes e graduados na área da informática e, posteriormente, foi realizado um inquérito a estes, de modo a auferir se a ferramenta testada é capaz de responder às hipóteses em teste.

Para tal, contactaram-se atuais e antigos estudantes da licenciatura e mestrado em Engenharia Informática do ISEP e da Faculdade de Engenharia da Universidade do Porto (FEUP). Este estudo englobou alunos segundo ao quinto ano (equivalente ao segundo ano de mestrado), bem como profissionais já graduados. Este conjunto de pessoas foi convidado a testar o *plugin* desenvolvido durante um período de cerca de uma semana em projetos em que estão a trabalhar ou já trabalharam. De modo a facilitar o processo, foi publicada uma

página[16] na plataforma GitHub, onde foi disponibilizado o *plugin*, juntamente com as instruções de instalação e algumas informações sobre as fórmulas implementadas. Além disso, foi criado um questionário na plataforma *Google Forms*, onde foram colocadas questões relativas ao conceito de legibilidade de software e à utilização do *Readability Checker*. Por um lado, estas questões servem para ajudar a perceber a noção que a comunidade tem em relação à legibilidade de software e o papel, do ponto de vista dos inquiridos, que as instituições de educação citadas têm no ensino desta característica da qualidade de software aos alunos. Por outro lado, serve para tentar perceber se a ferramenta desenvolvida tem impacto na legibilidade do software quando utilizada durante o desenvolvimento de um projeto e se, consequentemente, cumpre as hipóteses de avaliação especificadas em 4.5.1. As questões apresentadas no questionário em causa podem ser consultadas na secção A.6 dos anexos, juntamente com as respetivas opções de resposta.

Com o questionário aplicado, foi então possível obter algumas conclusões sobre o impacto que a utilização do *Readability Checker* pode ter, tanto no ensino, como no processo de desenvolvimento de software. Além disso, serviu também para perceber o que pode ser alterado ou adicionado, de forma a melhorar a ferramenta.

4.5.3 Critérios de Aceitação

Antes de iniciar a análise dos resultados obtidos com o questionário aplicado, e de forma a filtrar possíveis questionários inválidos, foram definidos alguns critérios de aceitação. Ao efetuar esta filtragem, procura-se tornar os resultados obtidos mais fidedignos.

Antes de apresentar os critérios de aceitação, importa perceber a forma como está estruturado o questionário. Como foi referido em 4.5.2, as questões realizadas podem ser consultadas na secção A.6 dos anexos, no entanto, de um modo geral, o questionário é dividido em três secções.

A primeira procura obter duas informações em relação ao inquirido: a instituição de ensino superior que este frequenta ou frequentou e o ano em que se encontra. Ambos os campos são de preenchimento obrigatório, sendo o primeiro de resposta curta, inserida pelo próprio.

A segunda secção pretende, como já foi referido, perceber o nível de conhecimento do inquirido em relação à legibilidade de código-fonte e a influência que esta tem no desenvolvimento de software, do ponto de vista do respetivo. As respostas às questões efetuadas nesta secção são de escala linear, com opções de resposta de um a cinco, sendo um o valor mínimo e cinco o valor máximo. Todas as questões incluídas nesta secção são de resposta obrigatória.

Por fim, a terceira secção pretende perceber a opinião do conjunto de pessoas que testou o *plugin* em relação ao mesmo. Esta secção apresenta questões relacionadas com o *Readability*

[16] https://cdtpinto.github.io/projects/readabilitychecker

Checker e com a sua utilização e funcionamento. À exceção da última questão, as respostas para as questões apresentadas nesta secção são de escala linear de um a cinco, sendo a sua interpretação efetuada do mesmo modo que na segunda secção. A última questão pretende incentivar os inquiridos a darem sugestões e fazerem críticas à ferramenta em teste. Esta é a única questão desta secção cuja resposta é facultativa.

Visto que foi possível definir as questões cuja resposta era obrigatória para que fosse possível submeter o questionário, não se mostrou necessário excluir submissões com respostas em falta. Sendo assim, o primeiro critério de aceitação definido foi a exclusão de questionários cujas respostas dadas a questões cuja resposta era de escala linear fossem todas com o valor mínimo, ou com o valor máximo. Isto pretende filtrar questionários que procurem inflacionar positiva ou negativamente os resultados. O segundo e último critério de aceitação definido foi a exclusão de questionários cujas questões que permitiam a inserção de texto por parte do inquirido apresentassem respostas descontextualizadas.

Ao analisar individualmente os questionários obtidos, percebeu-se que nenhum destes violou os critérios definidos anteriormente, pelo que todos foram aceites e considerados para a análise de resultados.

4.5.4 Análise de Resultados

Neste subcapítulo é feita a análise aos resultados obtidos com o questionário realizado. Procura-se com esta análise, não só saber se o *Readability Checker* responde às hipóteses de avaliação enunciadas em 4.5.1, mas também perceber se, de um modo geral, o *plugin* pode ser uma contribuição para as boas práticas da engenharia de software.

Foram contactadas cerca de 30 pessoas para testar o *plugin* desenvolvido com este trabalho. No total, receberam-se 23 respostas ao questionário realizado, tendo sido todas elas consideradas para a análise de resultados, já que, como se verificou em 4.5.3, nenhum questionário se mostrou inválido.

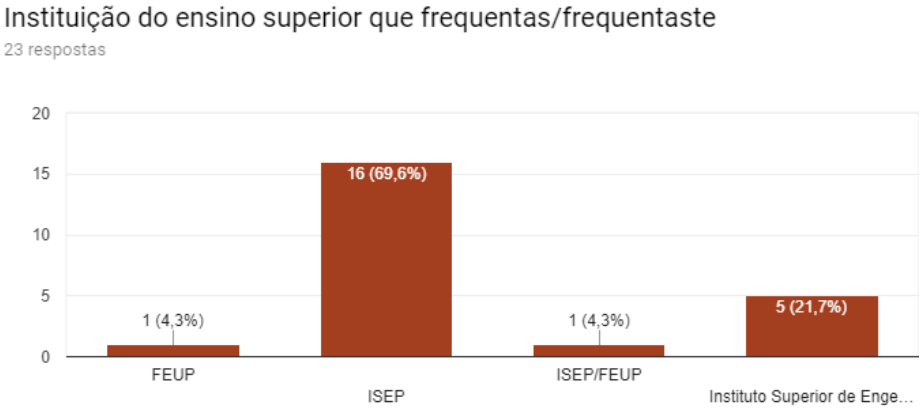

Gráfico 7 – Gráfico de barras com as instituições frequentadas pelos inquiridos

Como se pode confirmar, analisando o Gráfico 7, dos 23 inquiridos, 22 são atuais ou antigos alunos do ISEP e dois da FEUP, sendo que um destes frequentou as duas instituições de ensino. Percentualmente, tal traduz-se em 95,5% dos inquiridos como atuais ou antigos alunos o ISEP e 4,5% como atuais ou antigos alunos da FEUP (foi desconsiderada para este cálculo a resposta que apresenta ambas as instituições de ensino).

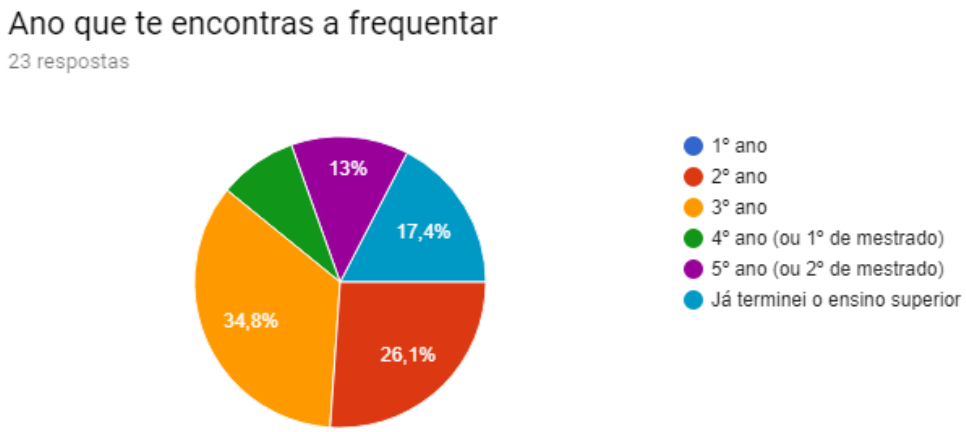

Gráfico 8 – Gráfico circular com a situação académica dos inquiridos

O Gráfico 8 procura dar uma estimativa global sobre o grau de formação de cada um dos inquiridos, através da situação académica em que estes se encontram.

Verifica-se através deste gráfico que a maior parte dos inquiridos se encontra na licenciatura (seis alunos do segundo ano e oito do terceiro). Foi possível também obter duas respostas de alunos do primeiro ano de mestrado e três de alunos do segundo ano. Quatro dos questionários recebidos referem-se a profissionais já graduados. Aqui não houve distinção entre o grau de formação destes inquiridos, podendo eles serem licenciados ou mestres.

Importa notar que o motivo de o conjunto de inquiridos não incluir formandos do primeiro ano de licenciatura se deve ao facto de que o teste ao *plugin* e o respetivo inquérito foram realizados no início do ano letivo, pelo que por essa altura, a maioria dos alunos ainda não tem conhecimento suficiente ao nível de programação que lhes permita testar a ferramenta em causa.

A primeira questão incluída na segunda secção do questionário pretende perceber o quão familiarizados estão os inquiridos com o conceito de legibilidade de software. As respostas obtidas, sumarizadas no Gráfico 9, mostram que apenas duas pessoas (8,7% da amostra) se sentem pouco familiarizadas com este conceito. A maioria dos inquiridos (56,5%) sente-se, familiarizado ou muito familiarizado com o conceito, sendo que os restantes (34,8%) consideram ter um conhecimento suficiente no assunto.

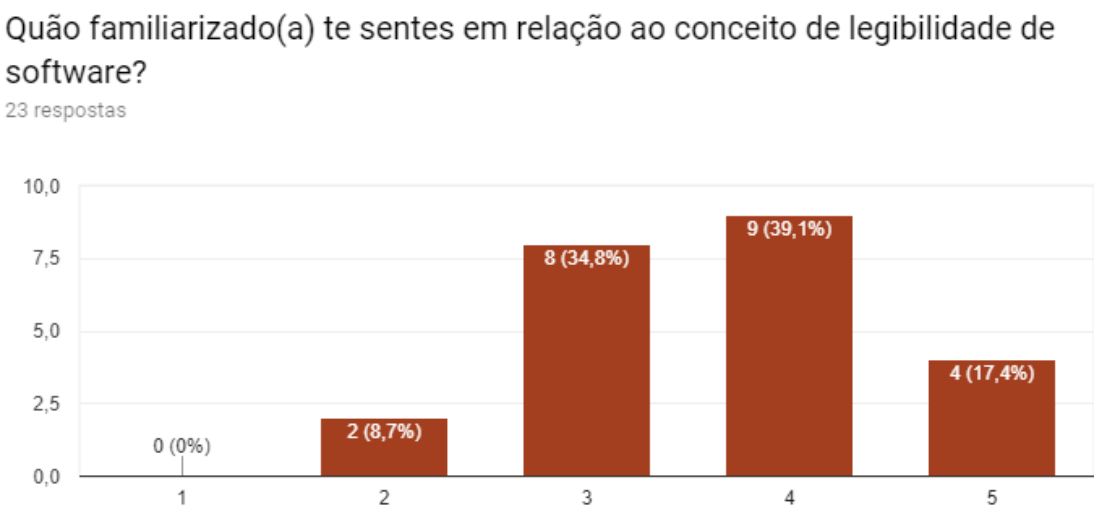

Gráfico 9 – Gráfico de barras com o nível de familiaridade dos inquiridos com a legibilidade de software

A questão seguinte procurou perceber o nível de cuidado que os inquiridos consideram ter quando escrevem código.

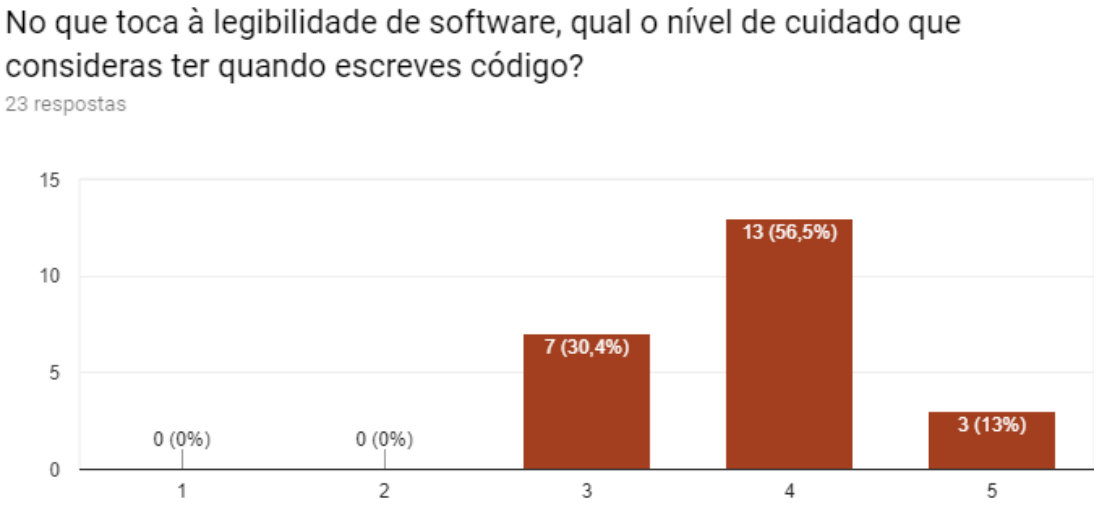

Gráfico 10 – Gráfico de barras com o nível de cuidado dos inquiridos ao nível da escrita de código legível

No Gráfico 10 estão sumarizadas as respostas obtidas. Com a sua análise, percebe-se que todos os inquiridos consideram ter um mínimo de cuidado quando escrevem código. Nenhum inquirido considerou ter pouco cuidado com a legibilidade do código que escreve.

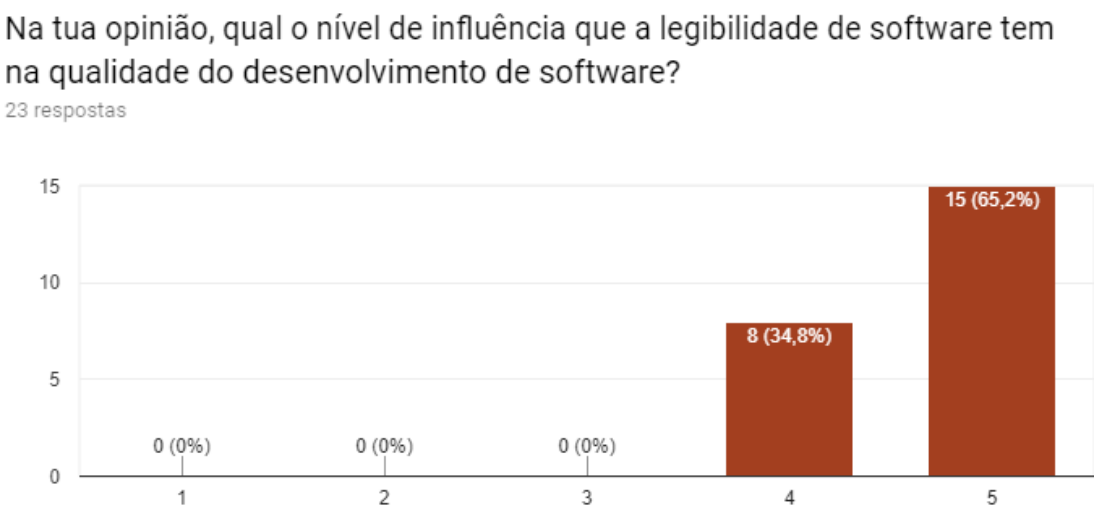

Gráfico 11 – Gráfico de barras com a influência da legibilidade no desenvolvimento de software segundo os inquiridos

O Gráfico 11 sumariza a opinião dos inquiridos em relação à influência que a legibilidade de código-fonte tem na qualidade do desenvolvimento de software. Os resultados obtidos são bastante conclusivos. Toda a amostra considera que a legibilidade influencia a qualidade geral do software, sendo que a maioria (65,2%) considerou que ela tem muita influência.

Estes resultados mostram-se condizentes com estudos já realizados por diferentes investigadores, como foi sendo referido ao longo deste documento, que a legibilidade de software influencia diretamente a qualidade do desenvolvimento de software.

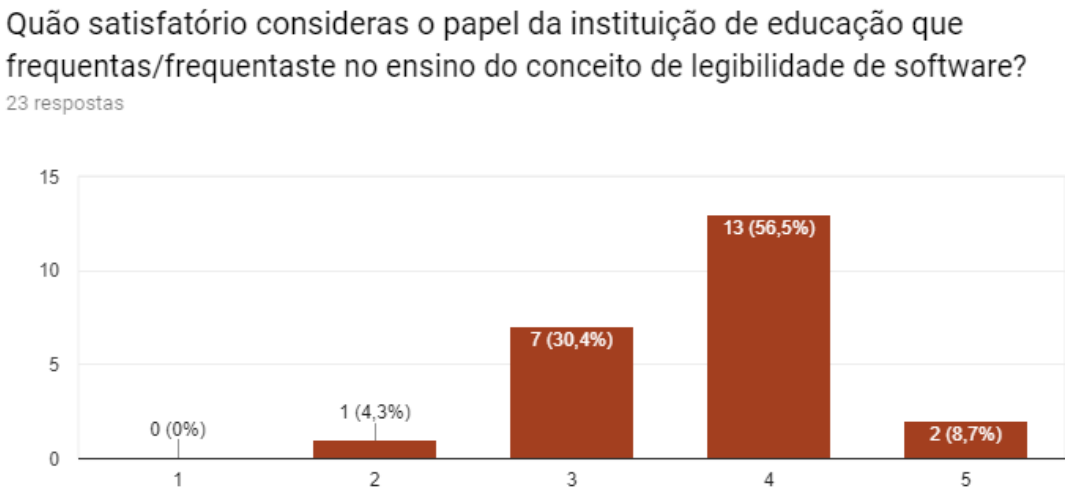

Gráfico 12 – Gráfico de barras com a opinião dos inquiridos sobre o ensino de noções de legibilidade

A última pergunta da segunda secção do questionário procura avaliar o nível de satisfação dos inquiridos em relação ao papel das instituições de ensino mencionadas no ensino do conceito de legibilidade de software. O Gráfico 12, que sumariza as respostas obtidas a essa questão, permite perceber que que as instituições têm um papel positivo nesse sentido. Apenas um dos inquiridos considerou que a instituição de ensino que frequenta ou frequentou não tem um papel suficientemente satisfatório. 30,4% dos inquiridos considera razoável, ao passo que maioria (56,5%) considera satisfatório o papel das respetivas instituições de ensino. Duas pessoas consideram muito satisfatório o papel das respetivas instituições de ensino.

Isto demonstra que as instituições de ensino referidas, com principal destaque para o ISEP, já que esta representa 95,5% da amostra, têm um papel satisfatório no ensino do conceito de legibilidade de software. Contudo, os 34,7% que representam um nível de satisfação razoável ou baixo, mostram que há ainda uma margem de possível progresso.

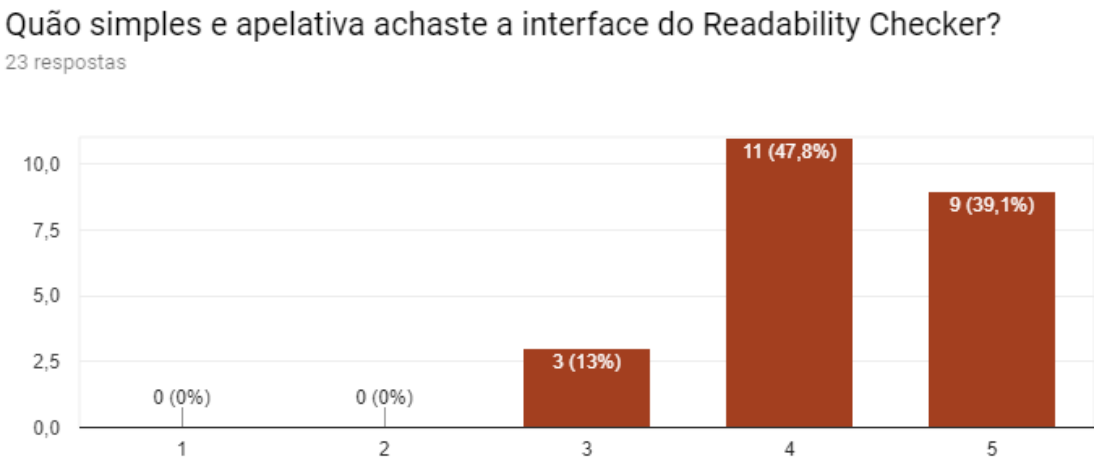

Gráfico 13 – Gráfico de barras para a avaliação da interface do Readability Checker

A primeira questão apresentada na terceira secção do questionário pretende perceber se a IU do *Readability Checker* se mostrou do agrado dos inquiridos. As respostas obtidas mostram que o *plugin* foi do agrado da grande maioria da amostra (86,9%). Apenas 13% do conjunto de pessoas que testou o *plugin* considerou a qualidade da IU aceitável, mas insuficientemente simples e apelativa.

Tal talvez se deva talvez ao facto de não ser imediatamente possível perceber os valores de legibilidade que são considerados bons ou maus. Alguém que nunca tenha tido contacto com o *plugin*, ao utilizá-lo pela primeira vez, vai obter os valores de legibilidade para o projeto em causa, mas não tem um *feedback* imediato sobre o significado dos resultados obtidos. Essa foi, inclusive, uma das sugestões feitas por um dos inquiridos.

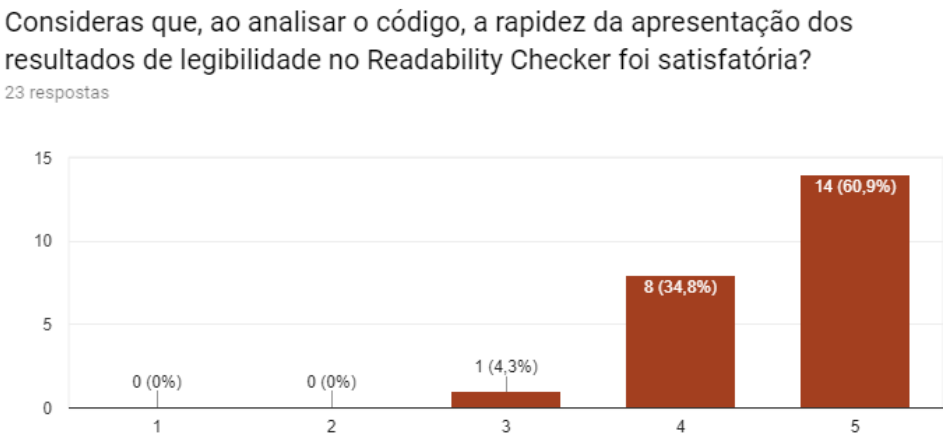

Gráfico 14 – Gráfico de barras para avaliar a eficiência de análise do código

O Gráfico 14 sumariza a opinião dos inquiridos no quesito da rapidez de análise do código e apresentação de resultados por parte do *plugin*. Esta questão procura perceber se o *Readability Checker* apresenta um nível de eficiência satisfatório.

Pelo que se pode apurar com a análise do gráfico, o *plugin* apresentou uma rapidez muito satisfatória na opinião da maioria dos inquiridos (60,9%). 34,8% destes acharam a rapidez de análise e apresentação de resultados satisfatória, ao passo apenas para uma pessoa (4,3%), o *plugin* apresentou uma rapidez aceitável.

Não tendo havido nenhum inquirido a considerar a rapidez de funcionamento do *Readability Checker* insatisfatória, conclui-se que a ferramenta desenvolvida se mostra fiável neste quesito.

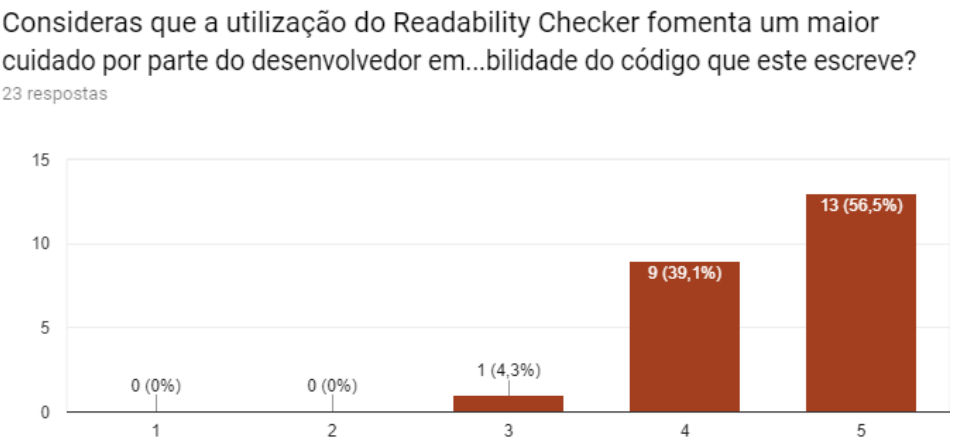

Gráfico 15 – Gráfico de barras para avaliação da influência do uso do Readability Checker no desenvolvimento de software

A primeira hipótese de avaliação que foi apresentada em 4.5.1 prende-se com a influência que a utilização do *Readability Checker* durante o processo de desenvolvimento de software tem na legibilidade do código que um desenvolvedor escreve. Esta foi uma das questões explicitamente colocadas aos inquiridos e o Gráfico 15 sumariza os resultados obtidos.

Analisando o gráfico, percebe-se que mais de metade da amostra concorda totalmente que o desenvolvimento de software, quando apoiado na utilização da ferramenta desenvolvida, fomenta um maior cuidado por parte dos desenvolvedores quanto à legibilidade do código que estes escrevem. Reforçando ainda mais esta ideia, 39,1% dos inquiridos concordou com essa afirmação, sendo que apenas uma pessoa (4,3%) não concordou nem discordou.

Não havendo respostas negativas, assume-se que o *Readability Checker* consegue responder à primeira hipótese de avaliação apresentada, mostrando-se assim capaz de influenciar positivamente o cuidado que os desenvolvedores têm em relação à legibilidade do código por si desenvolvido.

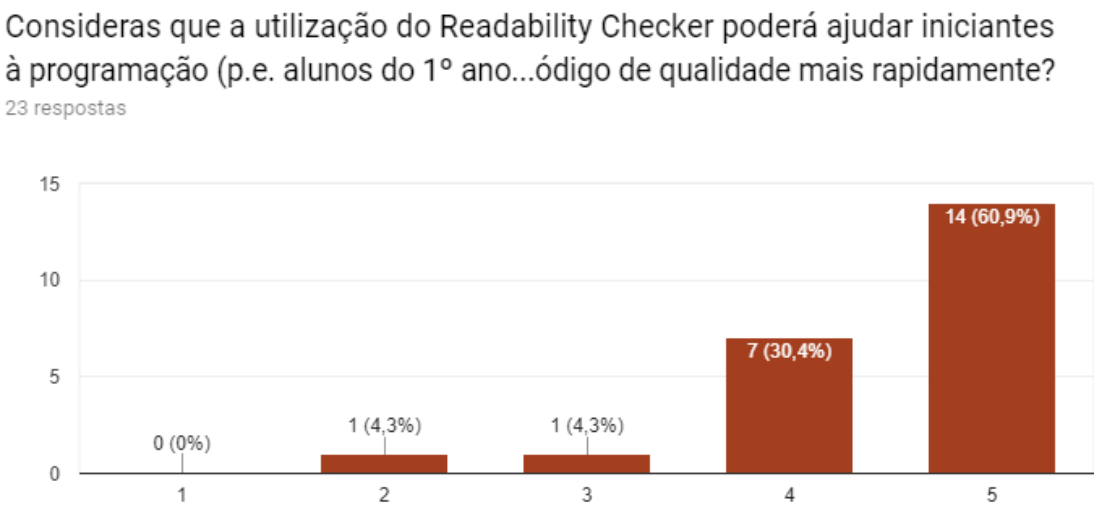

Gráfico 16 – Gráfico de barras para avaliação da influência do uso do Readability Checker no desenvolvimento de novos profissionais

A segunda hipótese de avaliação procura perceber se a utilização do *Readability Checker* por parte de iniciantes à programação, como é o caso de alunos do primeiro ano da Licenciatura em Engenharia Informática, pode ajudar a que estes sejam capazes de desenvolver código com um bom grau de legibilidade mais rapidamente.

Apenas uma pessoa discordou dessa afirmação e outra não concordou nem discordou. A grande maioria dos inquiridos (91,3%) concordou que o *Readability Checker* pode influenciar positivamente na aprendizagem de boas práticas de escrita de código legível por parte de iniciantes no desenvolvimento de software.

Desta forma, verifica-se que o *plugin* desenvolvido com este trabalho se mostra capaz de responder à segunda hipótese de avaliação, podendo a sua utilização ser um benefício no desenvolvimento das capacidades de escrita de código legível em iniciantes no desenvolvimento de software.

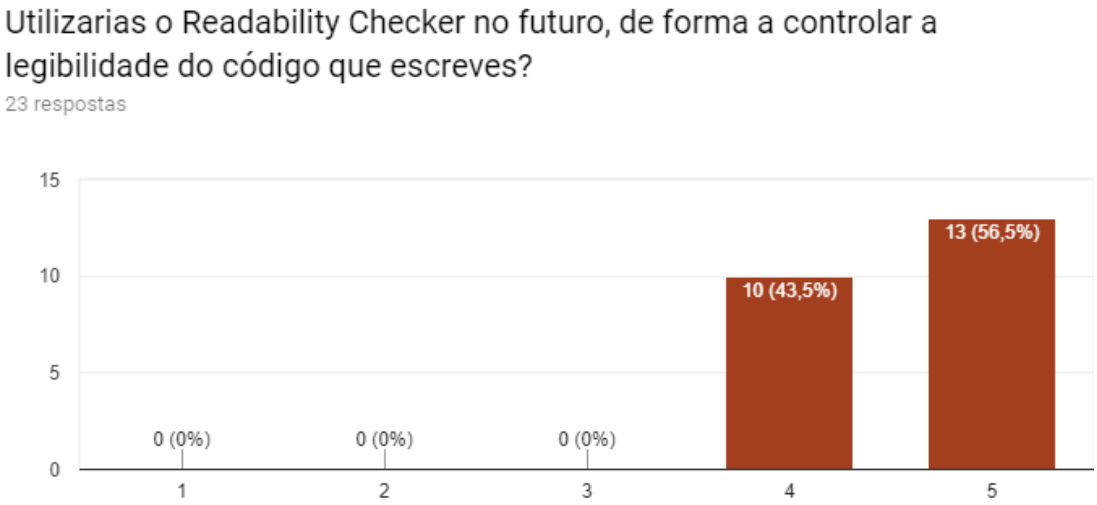

Gráfico 17 – Gráfico de barras para auferir a possível utilização futura do plugin pelos inquiridos

A última questão de resposta obrigatória do questionário procurou perceber se o *Readability Checker* foi do agrado geral dos inquiridos e se estes consideram a sua utilização futura como auxiliar no processo de desenvolvimento de software.

Os resultados obtidos foram bastante conclusivos. Sendo que 56,5% dos inquiridos considera muito provável e os restantes 43,5% considera provável a futura utilização do *plugin*, verifica-se assim que este correspondeu com as expectativas da totalidade da amostra.

Deste modo, é possível concluir que a utilização do *Readability Checker* pode trazer vantagens, tanto na legibilidade do software escrito por desenvolvedores, como também na formação de novos profissionais, mostrando-se assim, como uma contribuição para as boas práticas da engenharia de software.

A última questão apresentada no questionário era de resposta opcional e incentivava os inquiridos a apontar críticas e a dar sugestões para o desenvolvimento futuro da ferramenta. Conseguiu-se um total de cinco respostas a esta questão.

Uma das respostas mais pertinentes foi a sugestão da possibilidade de escolher o destino de exportação do ficheiro com os resultados de legibilidade avaliados. Mostrando-se esta uma funcionalidade interessante, foi, entretanto, implementada na ferramenta. Outra sugestão foi comentar os valores de legibilidade de acordo com o que é considerado código mais ou menos legível. Esta ideia está a ser considerada, no entanto ainda não foi implementada, já

que esses valores podem ser consultados recorrendo ao botão *Help* do *plugin*. Um dos inquiridos mencionou que a estética das janelas pode ser melhorada. Este é um dos pontos que eventualmente será abordado, já que as janelas apresentadas são, de facto minimalistas. Contudo, nesta primeira versão, o que se procurou foi essencialmente funcionalidade, tendo sido a parte estética relegada para segundo plano. Outra sugestão feita pelos inquiridos, foi a de lançar versões do *Readability Checker* para outros IDE's. Este é um dos principais objetivos para a evolução futura do *plugin*, já que, além de contribuir significativamente para o aumento do público-alvo, a sua implementação para outros ambientes de desenvolvimento não se mostra difícil, dado o modo como foi estruturado o código da ferramenta. A parte lógica da mesma pode ser facilmente adicionada a outra ferramenta, sendo apenas necessário desenvolver a IU desta. A última sugestão feita sugere o desenvolvimento de um método de avaliação sem a necessidade de guardar o código e premir o botão que ativa a análise do mesmo. Uma espécie de avaliação em tempo real, com apresentação e atualização de resultados contínuas. O principal foco do *Readability Checker* é o de permitir a análise da legibilidade do código em tempo de desenvolvimento, sem que o utilizador seja obrigado a sair do IDE e a lançar uma aplicação externa a este para o efeito. Esse quesito foi cumprido, contudo, esta é uma sugestão interessante que pode vir a ser analisada no futuro.

5 Conclusões e Trabalho Futuro

Findo o trabalho, mostra-se pertinente perceber o que foi alcançado e identificar oportunidades de evolução para o mesmo.

Neste último capítulo são apresentadas as conclusões obtidas com o trabalho realizado e são mencionados os aspetos que serão abordados para o desenvolvimento futuro.

5.1 Conclusões

Com este trabalho foram especificadas e analisadas as várias fórmulas de legibilidade de software propostas e foi desenvolvido um *plugin* para o IDE NetBeans, intitulado *Readability Checker*. Este *plugin* permite avaliar a legibilidade de código-fonte recorrendo a três fórmulas de legibilidade distintas.

A análise efetuada às fórmulas de legibilidade permitiu, não só perceber o tipo de abordagem tomada por cada uma, mas também decidir quais as que seriam de maior interesse implementar em primeiro lugar no *plugin*.

Com a análise realizada, foi também possível comprovar que uma das fórmulas propostas não permite obter resultados de legibilidade fidedignos, pelo que a sua implementação na ferramenta de legibilidade desenvolvida foi descartada.

Estando integrado no próprio IDE, o *Readability Checker* diferencia-se das demais ferramentas que encaixam na mesma categoria, já que isso permite a avaliação do código em tempo de desenvolvimento.

A ferramenta foi testada por alunos da licenciatura e do mestrado em Engenharia Informática e por profissionais graduados. Estes responderam então a um questionário cuja análise permitiu comprovar os benefícios que a utilização do *Readability Checker* pode trazer, tanto a nível profissional, como a nível académico.

5.2 Trabalho Futuro

Ainda que os objetivos propostos tenham sido alcançados, este é um trabalho que merece ser continuado. No que toca ao rumo a tomar a partir daqui, há vários aspetos que serão abordados.

O primeiro ponto será, naturalmente, a avaliação dos benefícios que trará a implementação das duas fórmulas de legibilidade que foram descartadas para esta primeira versão do *plugin* e posterior efetivação, caso tal se mostre pertinente.

O segundo tópico a ser avaliado será a integração do *Readability Checker* noutros IDE's. Durante a implementação de uma das fórmulas, mostrou-se pertinente ter possibilidade de avaliar excertos de código incompletos. Tal implementação verificou-se inviável, dado o grande trabalho de estudo e implementação que tal acarretaria, e o desvio que tal iria levar do foco deste trabalho. Uma investigação sobre métodos de análise de código incompleto permitiu chegar à conclusão de que não havia até à data nenhuma ferramenta que pudesse ser utilizada no NetBeans para o efeito, contudo, mostrou que havia uma destinada para o Eclipse. Deste modo, será ponderada a implementação do *Readability Checker* como um *plugin* também para esse IDE. Isso permitirá melhorar a implementação da fórmula de legibilidade PHD e permitir que ela não fique limitada a métodos com um máximo de linhas de código.

Um outro ponto que poderá ser analisado no futuro é a possível implementação de mecanismos que permitam aplicar as fórmulas implementadas a outras linguagens de programação para além do Java, caso estas o permitam.

Serão também acompanhados os avanços no estudo da legibilidade de software e analisadas novas propostas de fórmulas de legibilidade que eventualmente surgirão, fruto desses avanços.

Referências

Abbas, N., 2010. *Properties of 'Good' Java Examples*. [online] Umeå University. Available at: <http://www8.cs.umu.se/education/examina/Rapporter/NadeemAbbas_v2.pdf>.

Aggarwal, K.K., Singh, Y. and Chhabra, J.K., 2002. An Integrated Measure of Software Maintainability. In: *Annual Reliability and Maintainability Symposium*. [online] Seattle, WA, USA, USA: IEEE, pp.235–241. Available at: <https://ieeexplore.ieee.org/document/981648/>.

Baghel, A.S., 2018. *Software Design Principles DRY and KISS*. [online] Available at: <https://dzone.com/articles/software-design-principles-dry-and-kiss> [Accessed 9 Aug. 2018].

Batool, A., Rehman, M.H. ur, Khan, A. and Azeem, A., 2015. Impact and Comparison of Programming Constructs on JAVA and C# Source Code Readability. *International Journal of Software Engineering and Its Applications*, [online] 9(11), pp.79–90. Available at: <http://dx.doi.org/10.14257/ijseia.2015.9.11.07>.

Boehm, B. and Basili, V.R., 2001. Software Defect Reduction Top 10 List. *Computer*, [online] 34(1), pp.135–137. Available at: <http://dx.doi.org/10.1109/2.962984>.

Börstler, J., Caspersen, M.E. and Nordström, M., 2007. *Beauty and the Beast - Toward a Measurement Framework for Example Program Quality*. [online] Available at: <https://www8.cs.umu.se/research/reports/show.cgi?year=2007&nr=023>.

Börstler, J., Caspersen, M.E. and Nordström, M., 2015. Beauty and the Beast: on the readability of object-oriented example programs. *Software Quality Journal*, [online] 24, pp.1–16. Available at: <https://www.researchgate.net/publication/273504596/download>.

Burazin, I., 2015. *Most Popular Desktop IDEs & Code Editors in 2014*. [online] Available at: <https://blog.codeanywhere.com/most-popular-ides-code-editors/> [Accessed 19 Aug. 2018].

Buse, R.P.L. and Weimer, W.R., 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, [online] 36(4), pp.546–558. Available at: <http://dx.doi.org/10.1109/TSE.2009.70>.

Butler, S., 2009. The Effect of Identifier Naming on Source Code Readability and Quality. In: *Proceedings*

of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium. [online] Amsterdam, The Netherlands: ACM, pp.33–34. Available at: <http://doi.acm.org/10.1145/1595782.1595796>.

Butler, S., Wermelinger, M., Yu, Y. and Sharp, H., 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. [online] Washington, DC, USA: IEEE Computer Society, pp.31–35. Available at: <http://dx.doi.org/10.1109/WCRE.2009.50>.

Butler, S., Wermelinger, M., Yu, Y. and Sharp, H., 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. [online] Washington, DC, USA: IEEE Computer Society, pp.156–165. Available at: <http://dx.doi.org/10.1109/CSMR.2010.27>.

Chisnall, D., 2006. *What Makes a Good Programming Language?* [online] Available at: <http://www.informit.com/articles/article.aspx?p=661370&seqNum=4> [Accessed 5 Apr. 2018].

Colmer, R., 2018. *The Flesch Reading Ease and Flesch-Kincaid Grade Level*. [online] Available at: <https://readable.io/blog/the-flesch-reading-ease-and-flesch-kincaid-grade-level/> [Accessed 13 Aug. 2018].

Computer Hope, 2017. *What is an Operand?* [online] Available at: <https://www.computerhope.com/jargon/o/operand.htm> [Accessed 17 Aug. 2018].

Dagenais, B. and Hendren, L., 2008. Enabling Static Analysis for Partial Java Programs. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. [online] Nashville, TN, USA: ACM, pp.313–328. Available at: <http://doi.acm.org/10.1145/1449764.1449790>.

Deimel Jr., L.E., 1985. The Uses of Program Reading. *SIGCSE Bulletin*, [online] 17(2), pp.5–14. Available at: <http://doi.acm.org/10.1145/382204.382524>.

Dibble II, C. and Gestwicki, P., 2014. Refactoring Code to Increase Readability and Maintainability: A Case Study. *Journal of Computing Sciences in Colleges*, [online] 30(1), pp.41–51. Available at: <https://dl.acm.org/citation.cfm?id=2667378>.

DuBay, W.H., 2004. *The Principles of Readability*. [online] Costa Mesa, California. Available at: <http://www.impact-information.com/impactinfo/readability02.pdf>.

Elshoff, J.L. and Marcotty, M., 1982. Improving Computer Program Readability to Aid Modification. *Communications of the ACM*, [online] 25(8), pp.512–521. Available at: <http://doi.acm.org/10.1145/358589.358596>.

Fallahi, B., 1999. *What Is Javadoc?* [online] Available at: <http://www.devx.com/tips/Tip/13579> [Accessed 10 Aug. 2018].

GitHub Inc., 2018. *JavaParser*. [online] Available at: <https://github.com/javaparser/javaparser> [Accessed 13 Oct. 2018].

Guru99, 2018a. *Functional Testing Vs Non-Functional Testing: What's the Difference?* [online] Available at: <https://www.guru99.com/functional-testing-vs-non-functional-testing.html> [Accessed 28 Aug. 2018].

Guru99, 2018b. *What is Software Testing? Introduction, Basics & Importance*. [online] Available at: <https://www.guru99.com/software-testing-introduction-importance.html> [Accessed 27 Aug. 2018].

Hamer, P.G. and Frewin, G.D., 1982. M.H. Halstead's Software Science - A Critical Examination. In: *Proceedings of the 6th International Conference on Software Engineering*. [online] Tokyo, Japan: IEEE Computer Society Press, pp.197–206. Available at: <http://dl.acm.org/citation.cfm?id=800254.807762>.

Haneef, N.J., 1998. Software Documentation and Readability: A Proposed Process Improvement. *ACM SIGSOFT Software Engineering Notes*, [online] 23(3), pp.75–77. Available at: <http://doi.acm.org/10.1145/279437.279470>.

Iacovelli, D., 2018. *Make your code explain itself. If it has too many comments, you're doing it wrong!* [online] Available at: <https://medium.com/@douglas.iacovelli/make-your-code-explain-itself-if-it-has-too-many-comments-youre-doing-it-wrong-c856a0df6a65> [Accessed 20 Aug. 2018].

Jeremy, 2013. *8 Controversial Ways to Improve Business Efficiency*. [online] Available at: <https://www.brandwatch.com/blog/8-controversial-ways-to-improve-business-efficiency-guest-post/> [Accessed 22 Feb. 2018].

Kern, R.P., 1980. *Usefulness of Readability Formulas for Achieving Army Readability Objectives: Research and State-of-the-Art Applied to the Army's Problem*. [online] Available at: <https://www.researchgate.net/publication/235076531_Usefulness_of_Readability_Formulas_for_Achieving_Army_Readability_Objectives_Research_and_State-of-the-Art-Applied_to_the_Army's_Problem>.

Klare, G.R., 2000. Readable Computer Documentation. *ACM Journal of Computer Documentation*, [online] 24(3), pp.148–168. Available at: <http://doi.acm.org/10.1145/344599.344645>.

Knight, J.C. and Myers, E.A., 1991. Phased Inspections and Their Implementation. *ACM SIGSOFT Software Engineering Notes*, [online] 16(3), pp.29–35. Available at: <http://doi.acm.org/10.1145/127099.127101>.

Kondru, J., 2006. *Using Part of Speech Structure of Text In the Prediction of It's Readability*. [online] The University of Texas at Arlington. Available at: <https://rc.library.uta.edu/uta-ir/bitstream/handle/10106/178/umi-uta-1575.pdf?sequence=1&isAllowed=y>.

Limited Liability Company, 2018a. *Negative Correlation*. [online] Available at: <https://www.investopedia.com/terms/n/negative-correlation.asp> [Accessed 31 Mar. 2018].

Limited Liability Company, 2018b. *Positive Correlation*. [online] Available at: <https://www.investopedia.com/terms/p/positive-correlation.asp> [Accessed 31 Mar. 2018].

Liu, Y., Sun, X. and Duan, Y., 2015. Analyzing program readability based on WordNet. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. [online] Nanjing, China: ACM, pp.1–2. Available at: <http://doi.acm.org/10.1145/2745802.2745837>.

LoveToKnow Corp., 2018a. *Negative Correlation Examples*. [online] Available at: <http://examples.yourdictionary.com/negative-correlation-examples.html> [Accessed 31 Mar. 2018].

LoveToKnow Corp., 2018b. *Positive Correlation Examples*. [online] Available at: <http://examples.yourdictionary.com/positive-correlation-examples.html> [Accessed 31 Mar. 2018].

Lund Research Ltd., 2018. *Spearman's Rank-Order Correlation*. [online] Available at: <https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php> [Accessed 12 Oct. 2018].

Maioriello, J., 2002. *What Are Design Patterns and Do I Need Them?* [online] Available at: <https://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm> [Accessed 9 Aug. 2018].

McClure, G.M., 1987. Readability Formulas: Useful or Useless? *IEEE Transactions on Professional Communication*, [online] PC-30(1), pp.12–15. Available at: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6449109>.

My Byline Media, 2018a. *Advantages and Disadvantages of Using Readability Formulas*. [online] Available at: <http://www.readabilityformulas.com/articles/advantages-and-disadvantages-of-readability-formulas.php> [Accessed 18 Feb. 2018].

My Byline Media, 2018b. *History and Development of Readability Formulas*. [online] Available at: <http://www.readabilityformulas.com/articles/history-and-development-of-readability-formulas.php> [Accessed 16 Sep. 2018].

My Byline Media, 2018c. *The Automated Readability Index (ARI)*. [online] Available at: <http://www.readabilityformulas.com/automated-readability-index.php> [Accessed 16 Sep. 2018].

My Byline Media, 2018d. *The Coleman-Liau Readability Formula (also known as The Coleman-Liau Index)*. [online] Available at: <http://www.readabilityformulas.com/coleman-liau-readability-formula.php> [Accessed 24 Feb. 2018].

My Byline Media, 2018e. *The Flesch Reading Ease Readability Formula*. [online] Available at: <http://www.readabilityformulas.com/flesch-reading-ease-readability-formula.php> [Accessed 16 Sep. 2018].

My Byline Media, 2018f. *The SMOG Readability Formula, a Simple Measure of Gobbledygook*. [online] Available at: <http://www.readabilityformulas.com/smog-readability-formula.php> [Accessed 16 Sep. 2018].

Namani, R. and Kumar, J., 2012. A New Metric for Code Readability. *IOSR Journal of Computer Engineering*, [online] 6(6), pp.44–48. Available at: <https://www.researchgate.net/publication/272719950_A_New_Metric_for_Code_Readability>.

Nandy, I., 2007. *Halstead's Operators and Operands in C, C++, JAVA*. [online] Available at: <https://www.scribd.com/doc/99533/Halstead-s-Operators-and-Operands-in-C-C-JAVA-by-Indranil-Nandy#scribd> [Accessed 15 Jul. 2018].

Nicola, S., Ferreira, E.P. and Ferreira, J.J.P., 2012. Conceptual Model for Decomposing the Value for the Customer. *3rd Industrial Engineering and Management Symposium*, [online] pp.41–43. Available at: <http://recipp.ipp.pt/handle/10400.22/1248>.

Oak, M., 2014. *Readability and Maintainability of code*. [online] Available at: <https://madhuraoakblog.wordpress.com/2014/04/30/readability-and-maintainability-of-code/> [Accessed 22 Feb. 2018].

Pattis, R.E., 2004. *Tokens in Java Programs*. [online] Available at: <https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/tokens/lecture.html#Tokens> [Accessed 15 Sep. 2018].

Posnett, D., Hindle, A. and Devanbu, P., 2011. A Simpler Model of Software Readability. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. [online] Waikiki, Honolulu, HI, USA: ACM, pp.73–82. Available at: <http://doi.acm.org/10.1145/1985441.1985454>.

Princeton University, 2018. *What is WordNet?* [online] Available at: <https://wordnet.princeton.edu/> [Accessed 18 Feb. 2018].

ProjectCodeMeter, 2018. *Source lines of code*. [online] Available at: <http://www.projectcodemeter.com/cost_estimation/help/GL_sloc.htm> [Accessed 15 Jul. 2018].

Rocha, C., 2009. *Legibilidade, 'leiturabilidade' e «agradabilidade de leitura»*. [online] Available at: <https://ciberduvidas.iscte-iul.pt/consultorio/perguntas/legibilidade-leiturabilidade-e-agradabilidade-de-leitura/26814> [Accessed 10 Sep. 2018].

Rouse, M., 2017. *What is unit testing?* [online] Available at: <https://searchsoftwarequality.techtarget.com/definition/unit-testing> [Accessed 28 Aug. 2018].

Rugaber, S., 2000. The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering*, [online] 9(1–4), pp.143–192. Available at: <https://doi.org/10.1023/A:1018976708691>.

Sampaio, I., 2017. *Boas Práticas na Programação Orientada a Objectos a Adoptar Pelos Alunos de Informática do Ensino Superior*. UTAD.

Sampaio, I.B. and Barbosa, L., 2016. Software readability practices and the importance of their teaching. In: *2016 7th International Conference on Information and Communication Systems (ICICS)*. [online] Irbid, Jordan: IEEE, pp.304–309. Available at: <http://hdl.handle.net/10400.22/10095>.

Serrano, L., 2017. *Shannon Entropy, Information Gain, and Picking Balls from Buckets*. [online] Available at: <https://medium.com/udacity/shannon-entropy-information-gain-and-picking-balls-from-buckets-5810d35d54b4> [Accessed 7 Jul. 2018].

Shannon, C.E., 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal*, [online] 27(3), pp.379–423. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x>.

Sharpened Productions, 2018. *Character Definition*. [online] Available at: <https://techterms.com/definition/character> [Accessed 27 Aug. 2018].

Singh, B. and Gautam, S., 2016. The Impact of Software Development Process on Software Quality: A Review. In: *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*. [online] Tehri, India: IEEE, pp.666–672. Available at: <http://ieeexplore.ieee.org/document/8082729/>.

Smith, N., Bruggen, D. Van and Tomassetti, F., 2018. *JavaParser: Visited*. [online] Leanpub. Available at: <https://leanpub.com/javaparservisited>.

Sprunck, M., 2012. *Findbugs - Static Code Analysis of Java*. [online] Available at: <http://www.methodsandtools.com/tools/findbugs.php> [Accessed 16 Sep. 2018].

Tashtoush, Y., Odat, Z., Alsmadi, I. and Yatim, M., 2013. Impact of Programming Features on Code Readability. *International Journal of Software Engineering and Its Applications*, [online] 7(6), pp.441–458. Available at: <http://dx.doi.org/10.14257/ijseia.2013.7.6.38>.

Techopedia Inc., 2018. *What is an Operator?* [online] Available at: <https://www.techopedia.com/definition/3485/operator-programming> [Accessed 17 Aug. 2018].

Thakur, D., 2018. *Java Tokens - What is Java Tokens?* [online] Available at: <http://ecomputernotes.com/java/what-is-java-language/what-is-java-tokens> [Accessed 6 Jul. 2018].

Trica, A., 2014. *The Importance of Documentation in Software Development*. [online] Available at: <https://learn.filtered.com/blog/the-importance-of-documentation-in-software-development> [Accessed 10 Aug. 2018].

Vajapeyam, S., 2014. Understanding Shannon's Entropy Metric for Information. [online] pp.1–6. Available at: <https://www.researchgate.net/publication/262189154_Understanding_Shannon's_Entropy_metric_for_Information>.

Verifysoft Technology GmbH, 2017. *Measurement of Halstead Metrics with Testwell CMT++ and CMTJava (Complexity Measures Tool)*. [online] Available at: <https://www.verifysoft.com/en_halstead_metrics.html> [Accessed 17 Jul. 2018].

Virmani, S., 2017. *Software Engineering | Halstead's Software Metrics*. [online] Available at: <https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/> [Accessed 17 Jul. 2018].

Walrath, K. and Campione, M., 1997. *Comments in Java Code*. [online] Available at: <http://journals.ecs.soton.ac.uk/java/tutorial/getStarted/application/comments.html> [Accessed 14 Jul. 2018].

Wang, X., Pollock, L. and Vijay-Shanker, K., 2011. Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability. In: *Proceedings of the 2011 18th Working Conference on Reverse Engineering*. [online] Limerick, Ireland: IEEE Computer Society, pp.35–44. Available at: <https://doi.org/10.1109/WCRE.2011.15>.

Xu, W., Xu, D. and Deng, L., 2017. Measurement of Source Code Readability Using Word Concreteness and Memory Retention of Variable Names. In: *2017 IEEE 41st Annual Computer Software and*

Applications Conference (COMPSAC). [online] Turin, Italy: IEEE, pp.33–38. Available at: <https://ieeexplore.ieee.org/document/8029587/>.

Zamanian, M. and Heydari, P., 2012. Readability of Texts: State of the Art. *Theory and Practice in Language Studies*, [online] 2(1), pp.43–53. Available at: <http://www.academypublication.com/issues/past/tpls/vol02/01/06.pdf>.

Anexos

A.1 – Imagens do Readability Checker

Para demonstrar o seu funcionamento, nesta secção são apresentadas algumas imagens da utilização do *Readability Checker*.

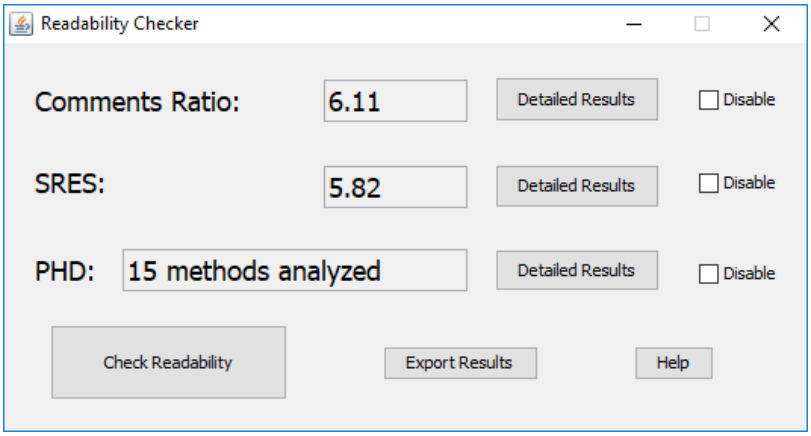

Figura 16 – Apresentação de resultados no Readability Checker

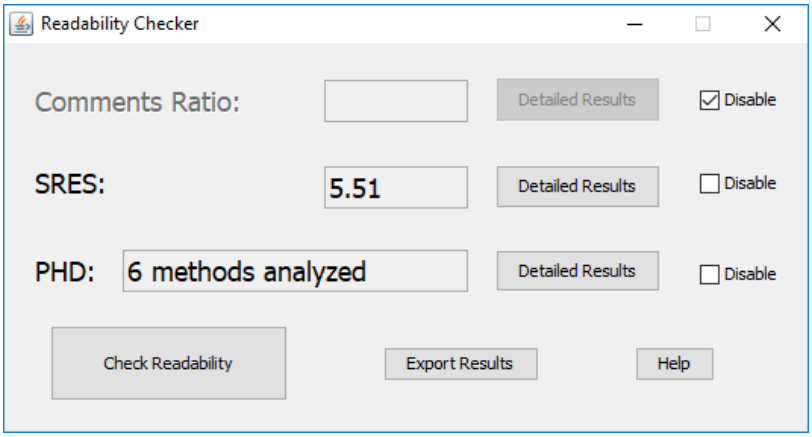

Figura 17 – Apresentação de resultados no Readability Checker com uma fórmula desativada

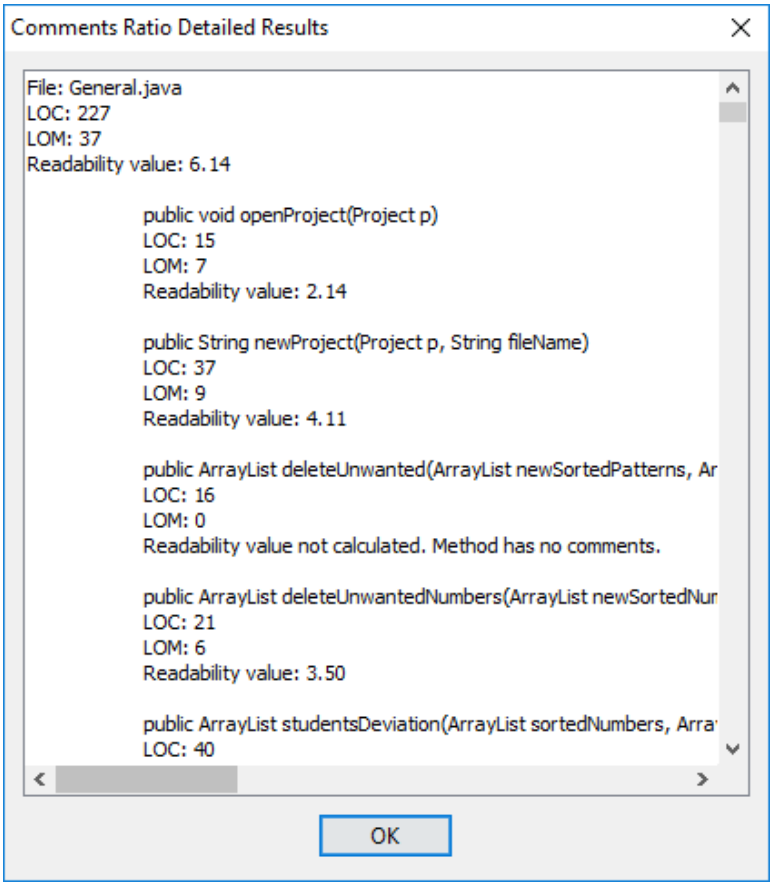

Figura 18 – Apresentação dos resultados detalhados para a fórmula Comments Ratio no Readability Checker

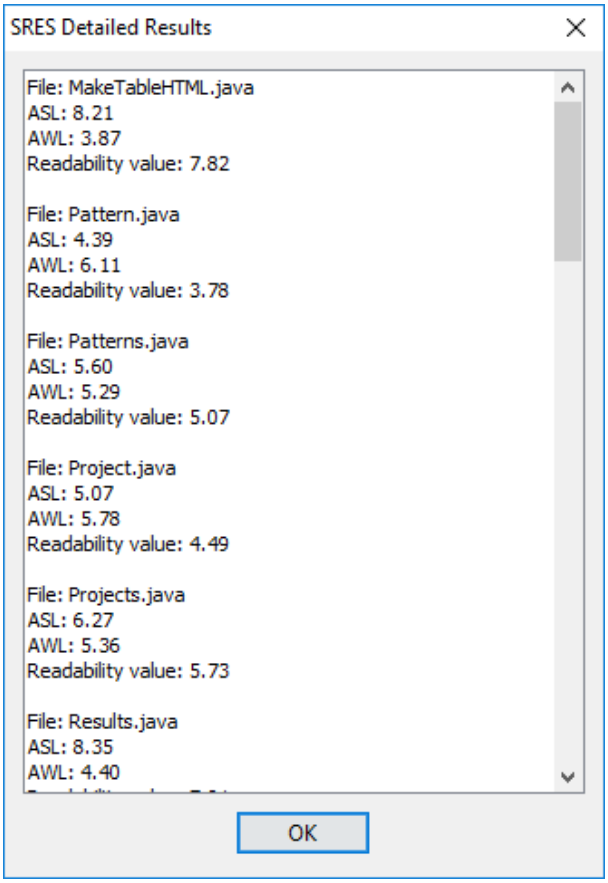

Figura 19 – Apresentação dos resultados detalhados para a fórmula SRES no Readability Checker

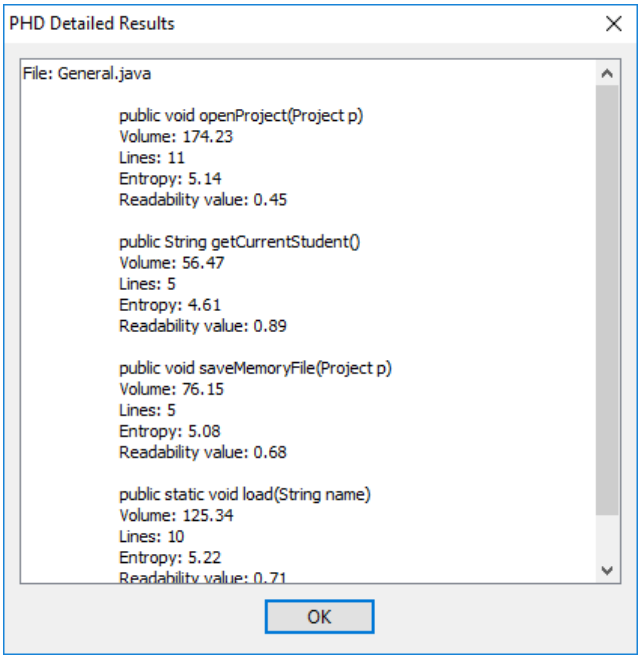

Figura 20 – Apresentação dos resultados detalhados para a fórmula PHD no Readability Checker

A.2 – Javadoc Readability Checker

Nesta secção são apresentadas algumas imagens retiradas do Javadoc gerado para o *Readability Checker*.

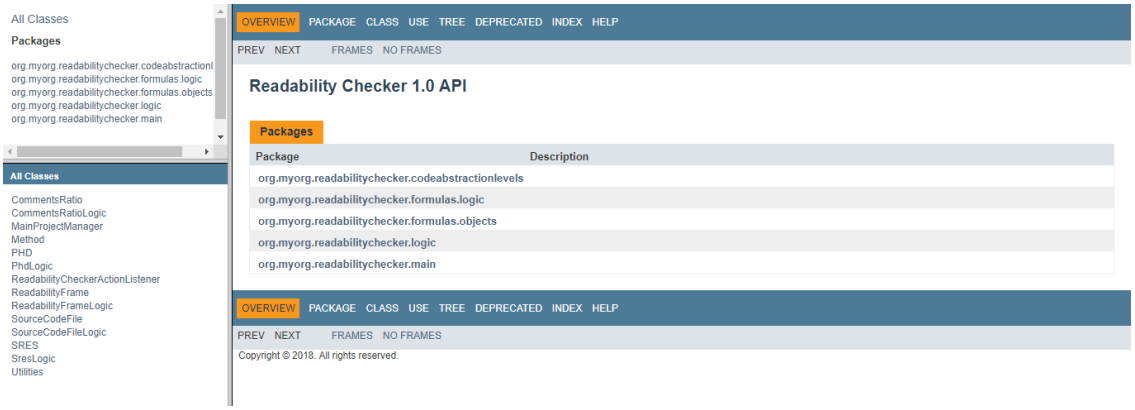

Figura 21 – Readability Checker - Página inicial do Javadoc

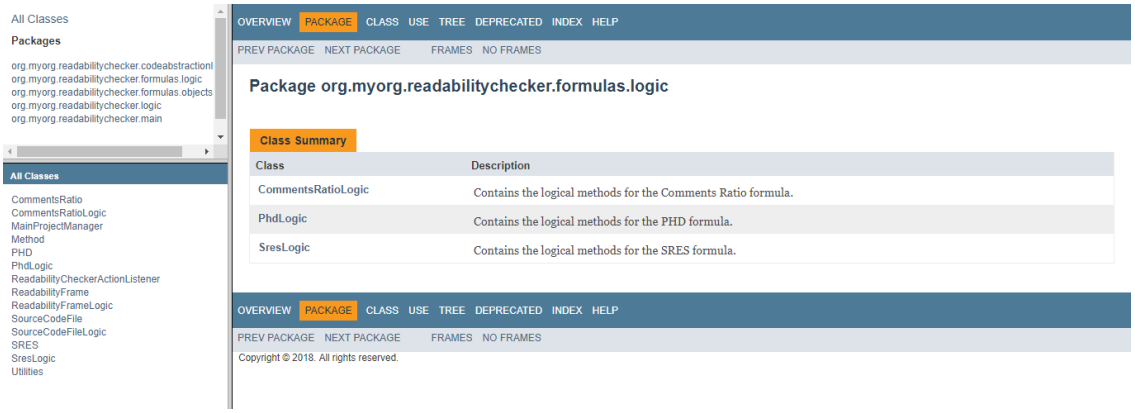

Figura 22 – Readability Checker - Javadoc (Classes)

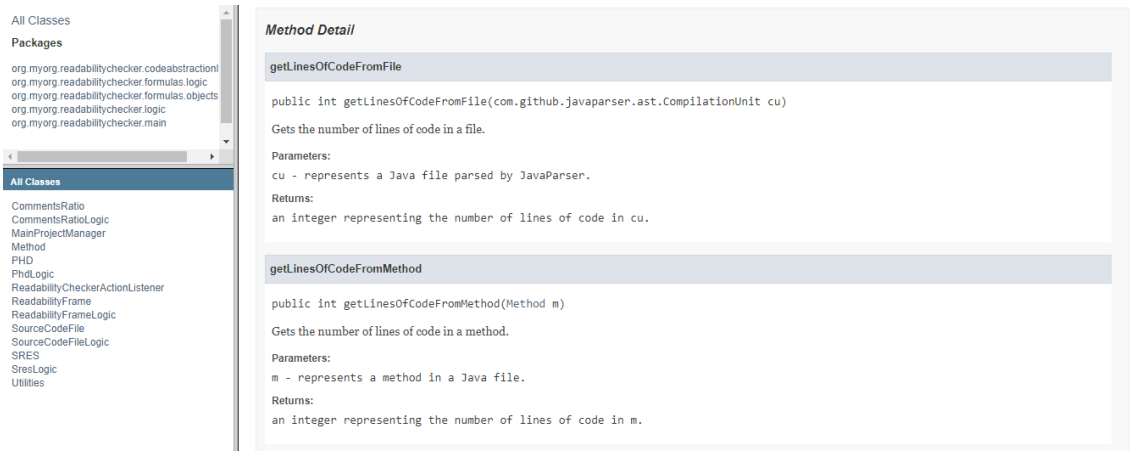

Figura 23 – Readability Checker - Javadoc (Métodos)

A.3 – Diagramas Comments Ratio

Esta secção serve para apresentar os diagramas que foram desenvolvidos para clarificar a forma como foi implementada a fórmula *Comments Ratio*, mas cujas dimensões não se mostraram adequadas para que estes possam ser apresentados no corpo do documento.

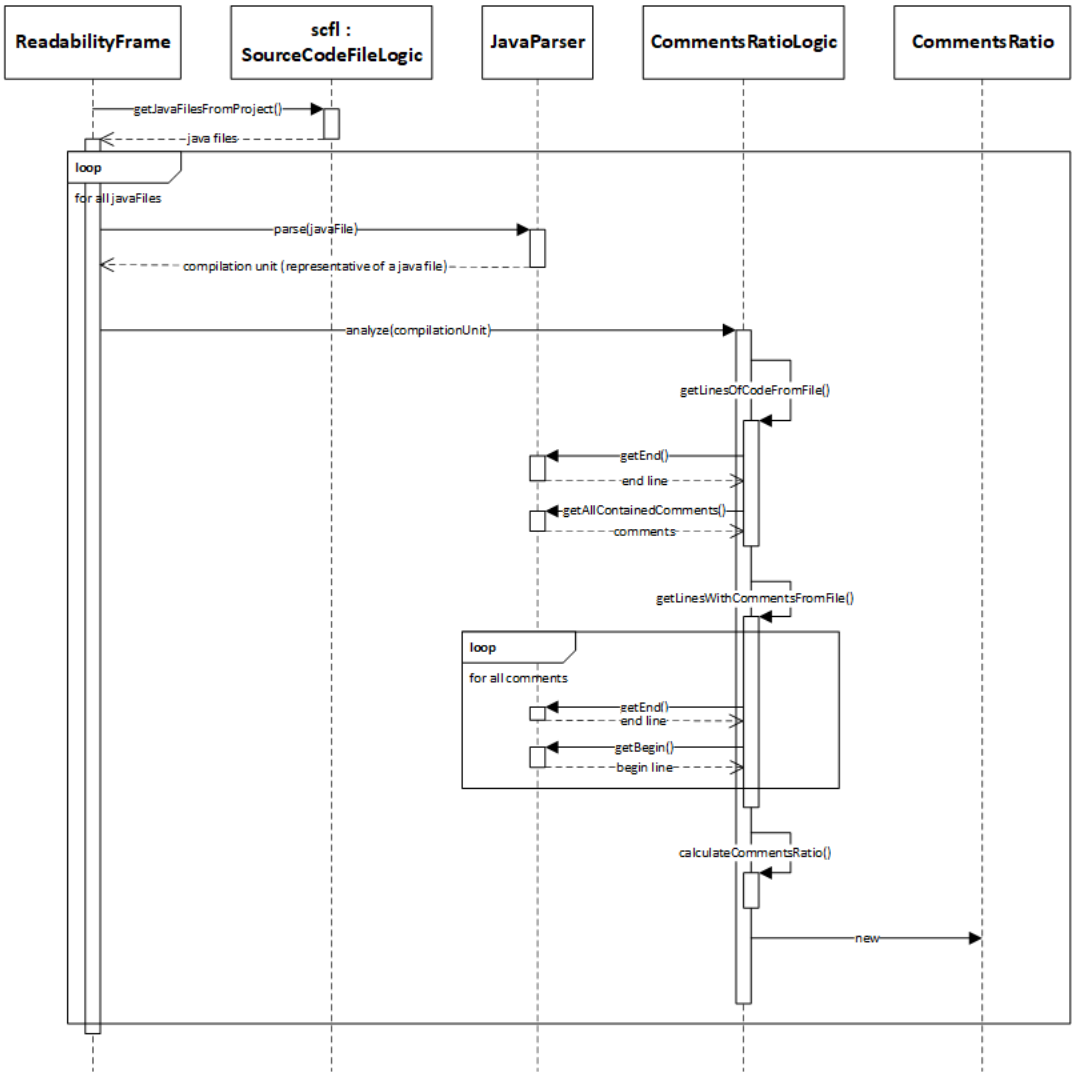

Figura 24 – Diagrama de Sequência: obtenção do LOC e do LOM para ficheiros

A.4 – Diagramas SRES

Nesta secção são apresentados os diagramas que se mostraram pertinentes desenvolver, de forma a tornar mais clara a implementação da fórmula SRES, mas cujas dimensões são desadequadas para que os mesmos possam ser apresentados no corpo da tese.

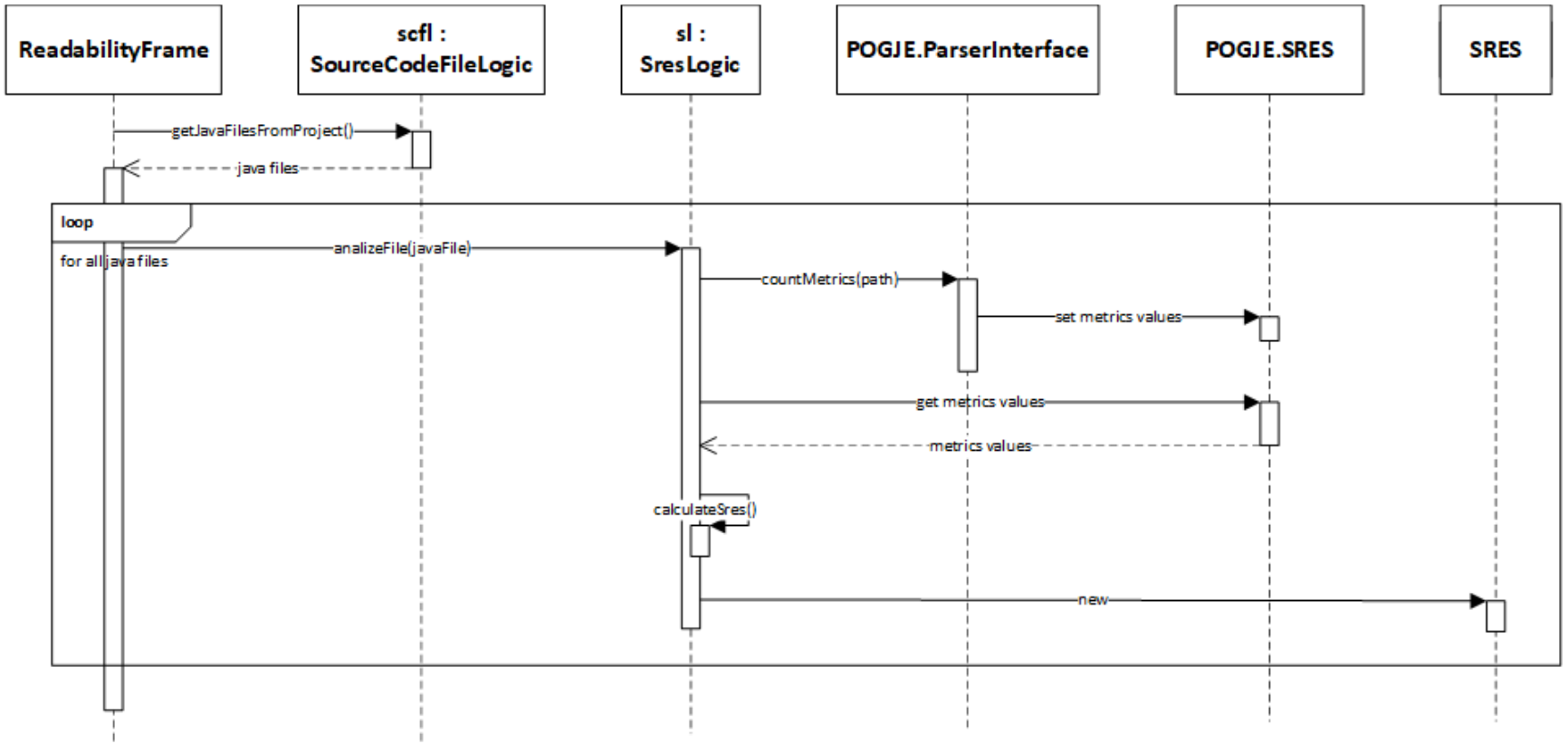

Figura 25 – Diagrama de Sequência: interação entre o Readability Checker e o POGJE

A.5 – Diagramas PHD

Nesta secção estão apresentados os diagramas que se consideraram apropriados apresentar, de modo a clarificar a implementação da fórmula PHD, mas cujas dimensões desproporcionais desencorajaram a sua utilização no corpo deste documento.

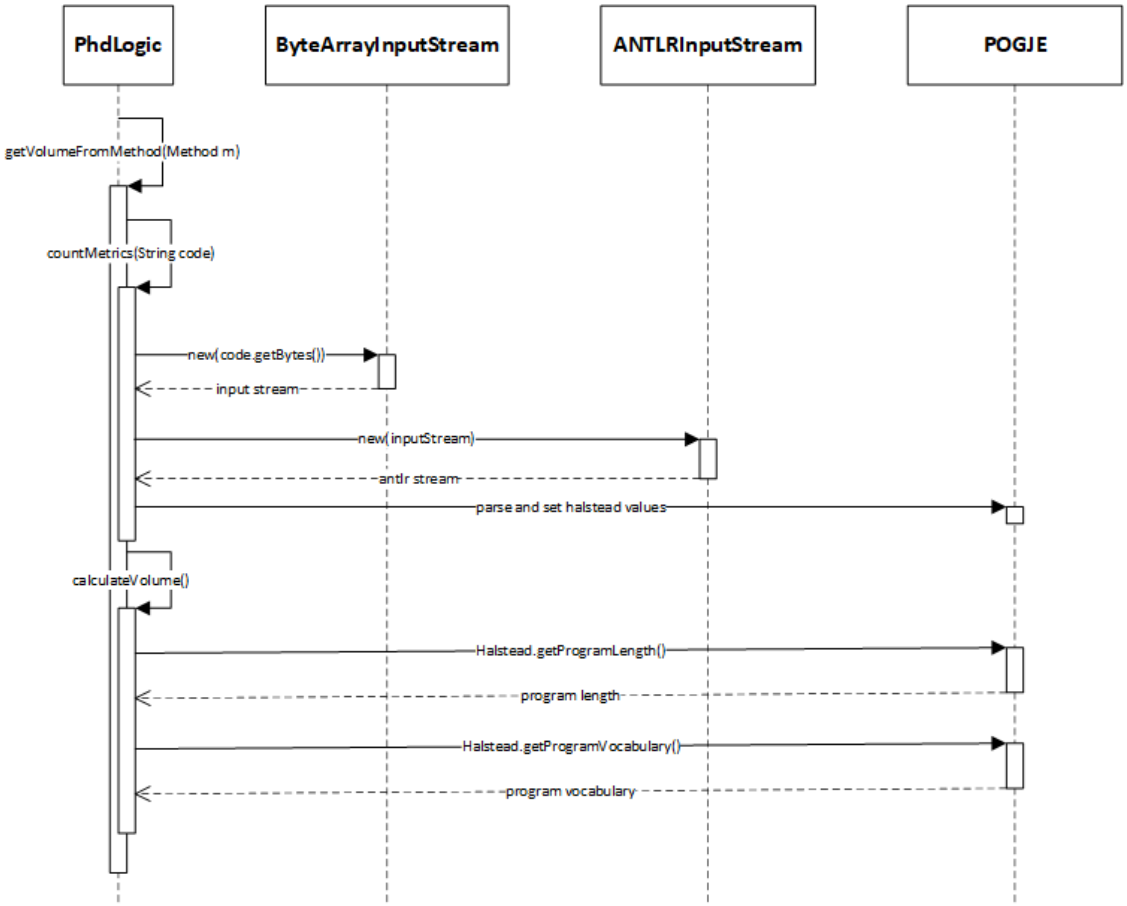

Figura 26 – Diagrama de Sequência: obtenção do volume

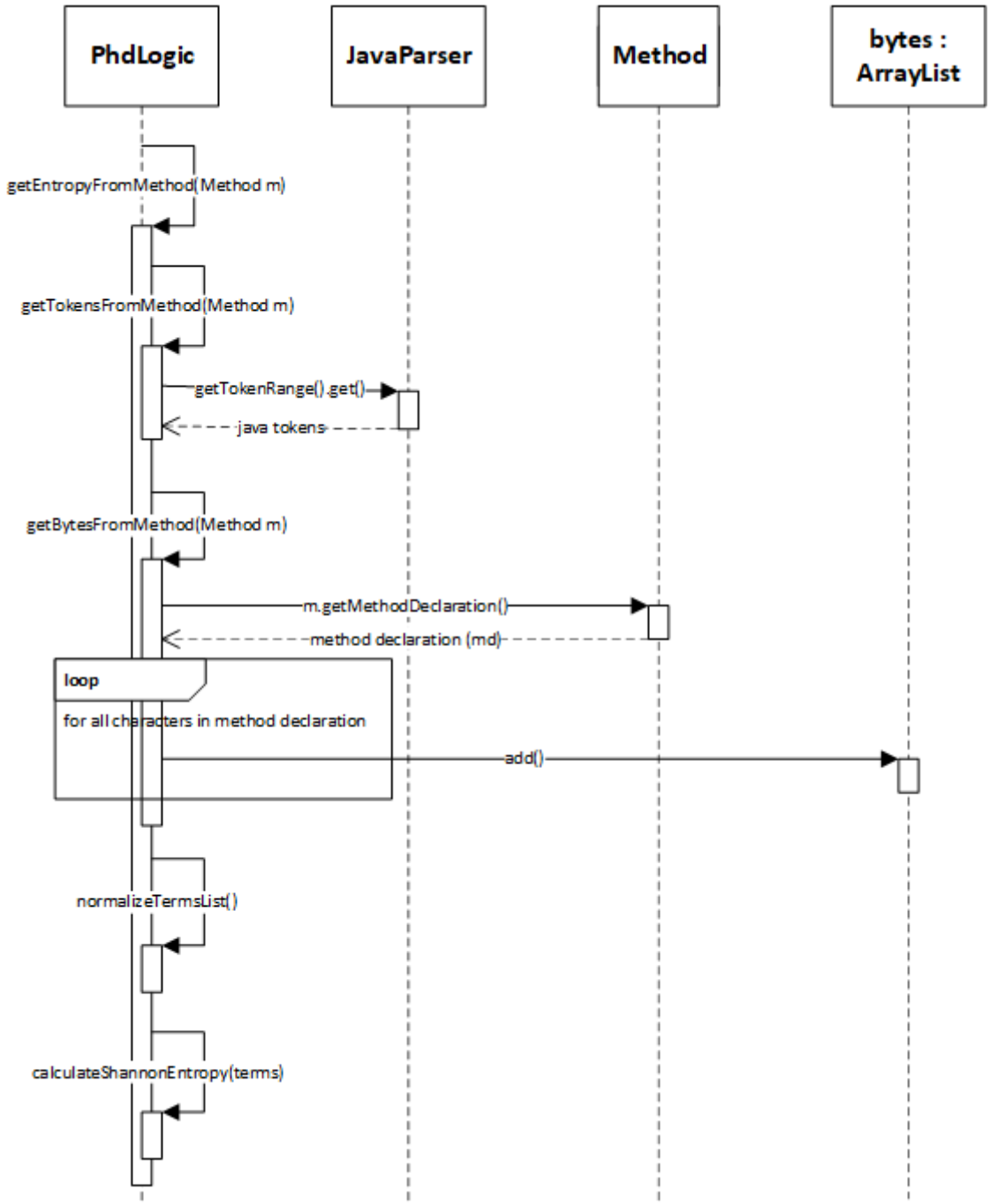

Figura 27 – Diagrama de Sequência: obtenção da entropia

A.6 – Questionário

Nesta secção estão apresentadas as questões efetuadas no questionário realizado no âmbito deste trabalho.

Alguns dados

Daryl Posnett, Abram Hindle e Prem Devanbu definiram legibilidade de software como sendo a propriedade que influencia o quão facilmente um excerto de código consegue ser lido e compreendido.
O Readability Checker pretende notificar o desenvolvedor sobre a legibilidade do código que este escreve, em tempo de desenvolvimento.
O presente questionário serve para auferir algumas informações sobre a utilização do plugin.

*Obrigatório

1. **Instituição do ensino superior que frequentas/frequentaste** *

2. **Ano que te encontras a frequentar** *
   Marcar apenas uma oval.
   - ◯ 1º ano
   - ◯ 2º ano
   - ◯ 3º ano
   - ◯ 4º ano (ou 1º de mestrado)
   - ◯ 5º ano (ou 2º de mestrado)
   - ◯ Já terminei o ensino superior

Figura 28 – Primeira parte do questionário

Legibilidade de software

Esta secção serve para obter alguns dados estatísticos sobre a noção legibilidade de software no seio da comunidade de participantes neste teste.

3. **Quão familiarizado(a) te sentes em relação ao conceito de legibilidade de software?** *
   Marcar apenas uma oval.

	1	2	3	4	5	
Nada familiarizado(a)	◯	◯	◯	◯	◯	Muito familiarizado(a)

4. **No que toca à legibilidade de software, qual o nível de cuidado que consideras ter quando escreves código?** *
   Marcar apenas uma oval.

	1	2	3	4	5	
Nenhum cuidado	◯	◯	◯	◯	◯	Muito cuidado

Figura 29 – Segunda parte do questionário

5. **Na tua opinião, qual o nível de influência que a legibilidade de software tem na qualidade do desenvolvimento de software?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Nenhuma influência	◯	◯	◯	◯	◯	Muita influência

6. **Consideras que o ensino de noções de legibilidade de software é algo que a instituição de ensino que frequentas/frequentaste procura dar aos seus alunos?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

Figura 30 – Terceira parte do questionário

Readability Checker

Esta secção serve para tentar perceber se o Readability Checker pode contribuir para uma melhoria na qualidade do desenvolvimento de software.

7. **Quão simples e apelativa achaste a interface do Readability Checker?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Nada	◯	◯	◯	◯	◯	Muito

8. **Consideras que, ao analisar o código, a rapidez da apresentação dos resultados de legibilidade no Readability Checker foi satisfatória?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Nada satisfatória	◯	◯	◯	◯	◯	Muito satisfatória

9. **Consideras que a utilização do Readability Checker fomenta um maior cuidado por parte do desenvolvedor em relação à legibilidade do código que este escreve?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

10. **Consideras que a utilização do Readability Checker poderá ajudar iniciantes à programação (p.e. alunos do 1° ano) a serem capazes de escrever código de qualidade mais rapidamente?** *
Marcar apenas uma oval.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

Figura 31 – Quarta parte do questionário

11. **Utilizarias o Readability Checker no futuro, de forma a controlar a legibilidade do código que escreves?** *

Marcar apenas uma oval.

	1	2	3	4	5	
Nada provável	◯	◯	◯	◯	◯	Muito provável

12. **O que achas que poderia ser melhorado no Readability Checker? Críticas e sugestões são bem-vindas.**

Figura 32 – Quinta parte do questionário