Universidade de Trás-os-Montes e Alto Douro

Boas Práticas na Programação Orientada a Objectos a Adoptar pelos Alunos de Informática do Ensino Superior

Tese de Doutoramento em

Informática

Adelaide Isabel dos Santos Vieira Braga Sampaio

Orientação de

Doutor Luís Filipe Leite Barbosa

Vila Real, 2017

Universidade de Trás-os-Montes e Alto Douro

Boas Práticas na Programação Orientada a Objectos a Adoptar pelos Alunos de Informática do Ensino Superior

Tese de Doutoramento em

Informática

Adelaide Isabel dos Santos Vieira Braga Sampaio

Orientação de:

Doutor Luís Filipe Leite Barbosa

Composição do Júri:

Doutor José Boaventura Ribeiro da Cunha

Doutor Manuel José Cabral dos Santos Reis

Doutor Luís Filipe Leite Barbosa

Doutor Vitor Manuel Basto Fernandes

Doutora Micaela Gonçalves Pinto Dinis Esteves

Vila Real, 2017

ao Betó,

à Inês, ao Quico, ao Luís,

à Pipas, à Cli, à Silly, à Sparky

e ao Roy

Agradecimentos

Gostaria de agradecer a todos que, de alguma maneira, contribuíram para a realização deste trabalho de doutoramento.

Em primeiro lugar cabe-me agradecer ao meu orientador, o Professor Luís Barbosa, pela sua paciência e disponibilidade para a realização deste trabalho de grande dimensão temporal, mas muito gratificante.

Agradeço igualmente ao Instituto Politécnico do Porto e à minha escola, Instituto Superior de Engenharia do Porto, todas as ajudas concedidas ao longo do trabalho. Fica também aqui manifestado o meu obrigada a todos os colegas, e, à Direcção do Departamento de Engenharia Informática, em especial ao Professor Luiz Faria, pelo seu apoio e facilidades concedidas. Só desta forma foi possível realizar este trabalho.

Agradeço à Professora Barbara Kitchenham da Universidade de Keel pela sua disponibilidade em responder a todas as dúvidas sobre estudos empíricos.

Quero ainda agradecer à Universidade de Trás-os-Montes e Alto Douro, à sua Escola de Ciências e Tecnologia, e particularmente ao seu Departamento de Engenharias, por esta oportunidade.

Para o final, fica o meu agradecimento àqueles sem os quais nada disto faria sentido, a minha família. Obrigada pela dedicação e estímulo permanentes. Obrigada por me aturarem! Desculpem pelo tempo que não me foi possível passar convosco e pelas preocupações que vos causei. Obrigada Betó, Inês, Quico e Luís!

Resumo

O trabalho descrito nesta tese consistiu no estudo de boas práticas na programação orientada a objectos a adoptar pelos alunos de informática dos cursos de ensino superior, de modo a melhorar a legibilidade dos seus programas. A leitura é uma competência essencial, a qual pode ter impacto significativo no percurso de um indivíduo desde criança até à idade adulta bem como no desenvolvimento da sua carreira futura. O código-fonte, apesar da sua especificidade, é um texto para ser lido, o que torna relevante o desenvolvimento da competência de leitura de código nos cursos de informática.

Nesse sentido, inicialmente foi realizado um inquérito através de um questionário destinado a docentes de programação orientada a objectos com um conjunto preliminar de práticas, com o intuito, de avaliar a importância do ensino de boas práticas aos alunos de informática do ensino superior em Portugal. Os resultados obtidos do questionário são aqui apresentados.

Através da revisão da literatura foi possível obter uma vasta lista de práticas a seguir, e algumas a evitar de modo a aumentar a legibilidade do código-fonte. A revisão da literatura debroçou-se sobre duas perspectivas. Uma primeira na indústria para ter conhecimento da opinião dos profissionais. A segunda perspectiva foi baseada na literatura com estudos empíricos de práticas de legibilidade.

Foram realizados dois estudos experimentais em sala de aula. Para isso, dada a vasta lista de práticas, seleccionaram-se duas. Os resultados dos dois estudos não foram estatisticamente significativos.

Por decisão pessoal, a autora do texto não escreve segundo o novo Acordo Ortográfico.

Palavras-chave: Práticas, legibilidade, código-fonte, aluno, ensino, POO.

Abstract

The work described in this thesis consisted in the study of good practices in the object-oriented programming to be adopted by computer students of higher education courses, in order to improve the readability of their programs. Reading is an essential skill, which can have a significant impact on an individual's journey from child to adulthood as well as on the development of his or her future career. The source code, despite its specificity, is a text to be read, which makes relevant the development of the competence to read code in computer courses.

In this sense, a survey was initially carried out through a questionnaire for object - oriented programming teachers with a preliminary set of practices with the purpose of evaluating the importance of teaching good practices to students of higher education in Portugal. The results obtained from the questionnaire are presented here.

Through the literature review it was possible to obtain a vast list of practices to follow, and some to avoid in order to increase the readability of the source code. The literature review was debated on two perspectives. A first in the industry to have knowledge of the opinion of professionals. The second perspective was based on the literature with empirical studies of legibility practices.

Two experiments were carried out in the classroom. For this, given the vast list of practices, two were selected. The results of the two experiments were not statistically significant.

By personal decision, the author of the text does not write according to the new Orthographic Agreement.

Palavras-chave: Practices, readability, source-code, student, teaching, OOP.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Definições

Lista de abreviaturas

ACM - Association for Computing Machinery

ALICE - Alliance for Logistics Innovation Through Collaboration in Europe

ANOVA - ANalysis of VAriance

API - Application Programming Interface

APROG - Algoritmia e Programação

AR - Argumentos Repetidos

ARI - Automated Readability Index

BC - Bad Comment

CFS - Cognitive Functional Size

CC-R - Cognitive-Components Resource model)

CE - Central Executive

CHASE - Cooperative and Human Aspects of Software Engineering

CHI - Conference on Human Factors in Computing Systems

CMNTS - Conteúdo dos comentários adequado

CNTRL - Flow Control

CQS - Princípio *Command-Query Separation*

DOQ - Qualidade da documentação

DP - Proposicional Density

ECIM - Engenharia da Computação e Instrumentação Médica

EIM - Engenharia de Instrumentação e Metrologia

IEEE - Institute of Electrical and Electronics Engineers

ESE - Empirical Software Engineering

ESEM - Empirical Software Engineering and Measurement

FE - Feature Envy

FSE - Foundations of Software Engineering

GC - God Class

GC - Good Comment

GM - God Method

HSD - Honestly Significant Difference

IBM - International Business Machines

ICPC - International Conference on Program Comprehension

ICSE - International Conference on Software Engineering

ICSM - International Conference on Software Maintenance

IDD - IDentifier Dictionary

IDE - Integrated Development Environment

IDNTF - Identifier adequation

INDNT - adequated Indentation

ISEP - Instituto Superior de Engenharia do Porto

ISP - Interface Segregation Principle

JSEP - Journal of Software: Evolution and Process

LOC - Lines Of Code

LL - Length Of Line

LMC - Long Message Chain

LTM - Long-Term Memory

NA - Number of Arguments

NALS - National Adult Literacy Surveys

NAEP - National Assessment of Educational Progress

NC - No Comment

NLAS - New Line After Semicolon

NHST - Null Hypothesis Statistical Test

NOBL - Number Of Blank Lines

NOCL - Number Of Comment Lines

NSL - Número de linhas com instruções

OCDE - Organização para a Cooperação e Desenvolvimento Económico

ORF - Oral Reading Fluency

POO - Programação Orientada a Objectos

PRCDR - Adequated Procedure

PROGR - Programação

PPROG - Paradigmas da Programação

PSK - Powers-Sumner-Kearl

RB - Refused Bequest

RSC - Legibilidade do código-fonte

SMS - Systematic Mapping Study

SMOG - Simple Measure Of Gobbledygook

SRES - Software Readability Ease Score

SS - Shotgun Surgery

STM - Short-Term Memory

STRCT - Adequated Structure

STYLE - Utilização de características estilísticas úteis ao programa

TOSEM - Transactions on Software Engineering and Methodology

TSE - Transactions on Software Engineering

UTAD - Universidade de Trás-os-Montes e Alto-Douro

UOS - Qualidade da documentação

VAR - Comprimento médio normalizado das variáveis

VLHCC - Symposium on Visual Languages and Human-Centric Computing

WCRE - Working Conference on Reverse Engineering

WM - Working Memory

YALS - Young Adult Literacy Survey

Capítulo 1

Introdução

Sabe-se que a legibilidade do código-fonte constitui uma preocupação no desenvolvimento de software e o seu interesse deve-se em parte à sua relação com a manutenção do software. Também é sabido que uma grande percentagem dos custos do ciclo de vida do software dizem respeito à sua manutenção. Neste capítulo é apresentada a motivação para a realização do presente trabalho e são também definidos os seus objectivos e o seu âmbito.

1.1 Motivação

A legibilidade do código-fonte constitui uma preocupação no desenvolvimento de software desde há muito. Vários autores, como Antony Hoare em 1973 [122], Martin Hopkins [125] e Terry Baker [16], ambos em 1972, referem que a legibilidade é um dos atributos mais importantes do software. É interessante notar que nessa época não era comum um programador ler o código de outros, a não ser quando o autor do código tinha abandonado a organização, ou tinha sido promovido [301]. Como conta Yourdon, um funcionário terá dito [301, p.81] "Reading someone else's program is like reading their personal mail."

Os custos de manutenção são o factor dominante dos custos do software [126]. Através de vários estudos foi estimado que 50% dos custos do ciclo de vida do software dizem respeito à sua manutenção [170, 199]. Mas há autores que referem valores superiores [211]. Barry Bohem [29] refere 70% como um valor típico para a percentagem do custo consumido pela manutenção. Em [279] é indicado um valor de 60%. Já em [240] é referido que o custo de manutenção pode chegar a 90% do custo total.

O software que não se compreende é necessariamente difícil ou impossível de modificar. A compreensão de código é considerada uma actividade transversal à manutenção de programas [206]. Estima-se que a actividade de compreensão do código possa representar até 50% do esforço de manutenção do software [181]. Mas segundo [112], a compreensão de programas representa mais de metade do tempo consumido na manutenção do software, ou seja, o desenvolvedor passa mais de metade do seu tempo a tentar compreender o software. Portanto, a compreensão de programas é uma actividade essencial na evolução e manutenção do software.

A compreensão dos programas é uma área de estudo importante não só para a manutenção do software, mas também para a análise dos programas, para o *refactoring* e para a reengenharia [216]. Devido à sua importância, tem sido estudada intensamente apesar dos muitos problemas que ainda continuam por resolver [216]. Por exemplo, é consensual que a qualidade dos identificadores tem um efeito profundo na compreensão dos programas [126] e por isso a escolha dos identificadores afectará os custos do software. Mas simultaneamente, não existe um consenso absoluto acerca do formato dos identificadores.

O interesse da legibilidade deve-se em parte à sua relação com a manutenção do software [73]. A leitura do código é o primeiro passo na manutenção do software [19]. A leitura de código é uma actividade central da manutenção [218] e a legibilidade é um factor vital para a compreensão e consequentemente para a manutenção do software [126, 57]. Assim se compreende que o custo elevado da manutenção do software esteja ligado à dificuldade de leitura e compreensão do código. Na realidade, o desenvolvimento também requer a leitura de código, ou a reutilização e, portanto, pode-se considerar a legibilidade como factor transversal à generalidade do ciclo de vida do software.

Já foi afirmado que os programas não servem unicamente para conseguir que um computador realize determinadas operações [2, p.xxii] "Thus, programs must be written for people to read, and only incidentally for machines to execute." Nessa perspectiva a legibilidade será uma qualidade essencial do software, e a legibilidade representa uma grande parte do esforço da manutenção do software [19].

A legibilidade é um atributo relacionado com outros, como, por exemplo, a robustez, a modificabilidade, a complexidade, a compreensibilidade e a facilidade de reutilização [19]. Código mais legível é mais facilmente mantido [273, 281] por ser menos complexo,

mais robusto, mais compreensível, mais facilmente reutilizável e modificável.

Código-fonte que pode ser lido mais facilmente pelo seu autor ou por outros desenvolvedores permite reduzir o seu custo de manutenção, mas eventualmente, com o custo de um maior esforço para o tornar legível [17]. O esforço tem a ver com conseguir produzir componentes mais facilmente reutilizáveis, mais genéricos, mais bem testados, ou seja, prontos para serem utilizados noutras aplicações informáticas. No entanto, devidamente preparado, tal esforço será reduzido e compensado posteriormente em manutenção futura pela redução do tempo de leitura [85, 57]. Claro que é expectável que a dificuldade de leitura seja maior quando se trata de código escrito por outros.

Em [85] é proposta a introdução de um passo específico para melhoria da legibilidade na manutenção de programas. Ou seja, sempre que houver necessidade de modificar um programa existente, a sua legibilidade deveria ser avaliada e, no caso de ser considerado pouco legível, ele deveria ser tornado mais legível. Este novo passo deveria acontecer no início do ciclo de modificação. Esses autores recomendam várias alterações possíveis para tornar um programa mais legível.

Num estudo por inquérito na Microsoft, mais de 90% dos 111 engenheiros inquiridos, de uma lista de 17 indicadores possíveis para tomadas de decisão acerca da engenharia do processo de desenvolvimento de software, responderam que utilizariam a legibilidade como indicador nas suas decisões [40]. Entre todos os indicadores, o da legibilidade foi o terceiro mais requisitado. Em [160] é referido que mais de 40% do esforço de manutenção é passado na leitura do código. Actualmente, muitos desenvolvedores passam a maior parte do tempo a manter e fazer evoluir um programa e não a produzir novos programas [19]. A legibilidade é também muito pertinente no âmbito do código aberto (*open-source*) onde surge a necessidade de ler o código escrito por outros programadores, tanto por elementos do respectivo projecto, como por aqueles que pretendem utilizar esse código e por isso, necessitam lê-lo. Nesse sentido, é de supor que o esforço reservado à leitura seja actualmente acima dos 40% indicados antes.

No desenvolvimento de software é importante ter em atenção que futuramente esses sistemas poderão ser sujeitos a modificações. Se um sistema é criado com a preocupação de que no futuro irá ser sujeito a modificações, então é expectável que possa ser menos problemático proceder-se posteriormente às intervenções necessárias ao sistema do que se

este não tivesse sido criado com tal preocupação [256]. Aliás, a necessidade de modificações é quase inevitável. Essas modificações sobre o software são muitas vezes motivadas por alterações aos requisitos iniciais que estiveram subjacentes à criação desse software, ou adaptação ao meio, ou para correcção de defeitos que o software apresente durante a sua operação [168]. Em resultado dessas intervenções sucessivas, a estrutura do código-fonte de um programa pode degradar-se [168, 169, 81]. Naturalmente que poderão ser tomadas medidas que contrariem essa tendência, a começar logo no seu desenvolvimento.

Portanto, um objectivo muito importante no desenvolvimento de um programa é a identificação da estrutura do programa que facilite a modificação desse mesmo programa sempre que de tal haja necessidade [89, 254]. Existem algumas formas de perceber se o código irá trazer, mais cedo ou mais tarde, problemas de manutenção. Martin Fowler definiu o termo "*code smell*" como termo indicador de código deficiente, isto é, código que irá trazer problemas quando se lhe pretenderem realizar modificações [97].

Quanto mais simples for um programa para a mesma tarefa, mais fácil será a identificação dos elementos essenciais à sua compreensão, ou seja, a sua leitura. Um programa com estas características torna-se mais compreensível por requerer uma menor sobrecarga cognitiva [33]. Portanto, pode-se dizer que quanto mais legível for o código, mais rapidamente e com mais segurança um programador poderá obter informação crítica sobre um programa lendo o seu código. De acordo com o conceito de "*deslocated plan*" partes do software que estão conceptualmente relacionadas encontram-se fisicamente separadas no programa [293]. No contexto da manutenção, um programa é considerado **legível** se a informação necessária para a sua manutenção é encontrada facilmente lendo apenas o código [273].

As características de cada programa influenciam a capacidade das pessoas os lerem e compreenderem. As características individuais de cada pessoa influenciam a sua capacidade de leitura e compreensão de programas [266]. É inquestionável a importância de os alunos serem capazes de produzir código de qualidade. Ao mesmo tempo, sabe-se que apesar de nem todas as pessoas terem as mesmas capacidades para a concepção de software, um bom ensino, ferramentas adequadas e a utilização de exemplos têm um papel dominante [295]. Desse modo, a capacidade dos profissionais em produzir código de qualidade deve começar a ser desenvolvida logo durante a sua formação.

A legibilidade do código, a facilidade de modificação do código, a coesão do código e o seu baixo acoplamento, são características de qualidade consideradas essenciais no desenvolvimento de software [271]. No trabalho presente a qualidade cinge-se à legibilidade.

A leitura é uma competência essencial, a qual pode ter impacto significativo no percurso de um indivíduo desde criança até à idade adulta bem como no desenvolvimento da sua carreira futura, daí ser imperativo encorajar as crianças a aprender a ler [71]. O código-fonte, apesar da sua especificidade, é um texto para ser lido, o que torna relevante o desenvolvimento da competência de leitura de código nos cursos de informática. Já foi observado que uma das competências mais negligenciadas na programação é a capacidade de ler programas [69].

Para além disso, o código escrito pelos alunos tem frequentemente de ser lido por outras pessoas, o que por si só seria suficiente para que a legibilidade fosse um factor a ter em conta nos cursos de informática. As outras pessoas poderão ser os seus colegas de grupo, os seus docentes, ou eventualmente profissionais de organizações para o caso de alunos que realizam estágios, habitualmente como projecto final de curso. A avaliação dos trabalhos de programação dos alunos pode considerar, explicita ou implicitamente, a legibilidade do código apresentado nesses trabalhos. Por experiência própria e também tanto quanto se pode apurar junto de colegas, considerar a legibilidade do código como um factor de avaliação dos trabalhos é uma prática comum em cursos de engenharia informática.

A qualidade do código de um aluno pode ser vista sobre três perspectivas:

- a do professor que tem necessidade de analisar o código;
- a da engenharia de software (que define o processo para o desenvolvimento do produto e os critérios que este deve cumprir);
- a do aluno que desenvolve o produto.

Não é certo que estas perspectivas sejam coincidentes. Em princípio, a perspectiva do professor será um subconjunto da engenharia de software. A dos alunos será até certo ponto o resultado das duas anteriores. Para que os alunos produzam código de qualidade será necessário ensinar como o fazer. Para tal, é necessário saber que características deve

possuir o código para que possa ser considerado de qualidade e como fazer para se atingir a qualidade pretendida.

A autora na sua experiência de docente tem verificado frequentemente uma grande variabilidade na legibilidade de programas diferentes, de diferentes alunos e, por vezes, também no mesmo programa, mesmo quando escrito por um único aluno. A existência de linhas de orientação sobre a legibilidade que possam ser seguidas pelos alunos contribuiria certamente para melhorar e uniformizar a legibilidade do código dos alunos. Estas linhas de orientação podem assumir a forma de boas práticas a seguir. A não adesão a linhas de orientação é um dos dois factores principais para o surgimento de problemas na manutenção [49]. O outro é a documentação. A existência de linhas de orientação permitiria também um melhor alinhamento das três vistas anteriores.

Apesar da importância da legibilidade, e tanto quanto foi possível pesquisar, ainda há muito por fazer. Por exemplo, não foi possível encontrar uma lista de boas práticas de legibilidade destinada ao ensino da programação Orientada a Objectos (POO) e devidamente testada através de estudos empíricos.

A legibilidade é uma qualidade do software e, dada a sua importância, será de introduzir o seu ensino nos cursos de informática, tendo como enfoque como conseguir produzir código mais legível. Esta posição não é nova (e.g. [33]).

Tem sido referido por diversos autores [39, 114] que código de qualidade estará correlacionado com código legível. Como se procurou mostrar, a legibilidade tem um grande impacto no custo do software. Portanto, a importância da legibilidade do código é indiscutível e dada a sua importância e necessidade para a manutenção do software, a qual é em geral, inevitável [159, 23].

A inteligibilidade é a qualidade do que pode ser compreendido e é sinónimo de compreensível. Tal como acontece com qualquer texto, também o código deverá ser inteligível. A legibilidade no software tal como é tratada na literatura sobre legibilidade do software (e.g. [85, 39, 5]) não trata apenas da leitura do código, mas também da qualidade de este poder ser compreendido. Sendo assim, a legibilidade pode ser entendida como inteligibilidade. Contudo, aqui usar-se-á sempre o termo legibilidade até por ser a tradução directa do Inglês *readability*.

1.2 Objectivos

Pelo que foi exposto na secção anterior, ou seja, pelas implicações que a legibilidade do código dos alunos assume nos cursos de informática e assumirá futuramente para esses alunos ao nível profissional, torna-se óbvia a importância de os alunos serem capazes de produzir código legível.

Existe um grande variedade de boas práticas para a programação orientada aos objectos. Algumas encontram-se organizadas em conjuntos, como por exemplo, os *bad smells* [97] orientados essencialmente para a POO. No entanto, especificamente para a legibilidade não foi possível encontrar um conjunto bem definido de boas práticas que os alunos devam seguir de modo a produzirem código legível e que seja recomendado de forma mais ou menos consensual nem que tenha sido validado através de estudos. Esta lacuna constitui um problema.

O principal objectivo deste trabalho é contribuir para a definição de práticas que possam compor um conjunto de boas práticas de legibilidade no software, o qual possa ser usado, em particular, pelos alunos de cursos de introdução à POO. A legibilidade é um conceito abstracto que pode ser influenciado por diversos factores, portanto, multidimensional. As várias práticas deverão relacionar-se com esses factores.

Este objectivo é também relevante devido ao impacto que as linhas de orientação, ou a sua falta, têm na manutenção, como referido na secção anterior. O critério para selecção de possíveis boas práticas a incluir nesse conjunto é terem sido validadas, ou seja, terem sido alvo de investigação empírica.

Um segundo objectivo é aumentar o conhecimento existente acerca da situação da legibilidade do software nos vários cursos de engenharia informática, especificamente em disciplinas de introdução à POO e conhecer o interesse existente na legibilidade nesses cursos.

Para atingir o segundo objectivo será elaborado um inquérito aos docentes de diferentes cursos de Informática do ensino superior onde seja leccionado o paradigma da Programação Orientada aos Objectos. Para o primeiro objectivo, as práticas seleccionadas serão validadas através de um estudo experimental em ambiente do ensino superior. Para ambos os objectivos será necessário proceder ao levantamento das práticas de legibilidade

consideradas relevantes por profissionais e investigadores. O ensino desse conjunto de boas práticas pode ser influenciado pela forma e ordem como as práticas serão ensinadas, mas tal problema sai fora do âmbito desta tese.

1.3 Âmbito do Trabalho

A qualidade no software enquadra-se na área da engenharia de software. Sendo a legibilidade uma qualidade também se enquadra nessa área.

Este trabalho insere-se no âmbito das boas práticas de legibilidade na programação orientada a objectos para os alunos do ensino superior de cursos de informática. O objecto de estudo deste trabalho são as práticas de legibilidade.

Neste trabalho, é feita uma separação entre estilos de programação e práticas de programação, nomeadamente de legibilidade, considerando-se que estilos de programação não se enquadram nas práticas de programação. Apesar de não haver uma separação clara entre estilos e práticas, os estilos estão mais fortemente associados a aspectos de gosto pessoal, como os tipográficos, e muitas vezes estéticos, enquanto que as práticas, em grande medida, não o são [27]. Contudo existe alguma sobreposição. Por exemplo, no tabulamento, que se pode considerar como tendo um efeito visual estético, mas também é considerada uma boa prática de legibilidade. Actualmente, as questões de estilo estão em grande parte resolvidas pelos editores de código-fonte. Seja como for, as questões estilísticas não fazem parte do âmbito deste trabalho.

Há outros aspectos que podem ser importantes para a legibilidade mas que saem do âmbito deste trabalho, uma vez que este se orienta para os primeiros anos de aprendizagem orientada a objectos (OO), nomeadamente, a utilização de expressões lambda que permitem um estilo funcional no paradigma OO.

1.4 Organização da Tese

Este documento é composto por seis capítulos. Neste primeiro capítulo é feita a introdução do problema e a motivação para a realização do trabalho.

O segundo capítulo é designado por Teoria da Legibilidade, onde é apresentada a evolução do estudo da legibilidade e a sua divisão em diferentes períodos. São também

descritas as fórmulas de legibilidade, bem como as críticas e limitações às fórmulas. Neste capítulo é também objecto de estudo os conceitos fundamentais da legibilidade, tais como, literacia, compreensão e os factores de legibilidade. Será apresentada a definição de legibilidade adoptada neste trabalho.

O terceiro capítulo intitula-se Revisão da Literatura. Nesta revisão são apresentadas duas perspectivas, uma primeira onde é dada uma visão da indústria e a segunda obtida através de estudos empíricos. Na visão da indústria são descritas as opiniões baseadas na experiência de autores conceituados. Em ambas as perspectivas, a revisão é por ordem cronológica no sentido de se poder ter uma visão da evolução do software, em que muitas vezes as suas conclusões são questionadas em estudos posteriores.

O quarto capítulo é designado por Estudo Inicial. Neste capítulo são apresentados os resultados de um questionário realizado aos docentes que leccionam/leccionaram Programação Orientada a Objectos em cursos de Engenharia Informática do Ensino Superior em Portugal.

O quinto capítulo é designado por Estudo Experimental. Neste capítulo são apresentados os resultados de um estudo empírico realizado com os alunos de Programação Orientada a Objectos da Licenciatura em Engenharia Informática do Instituto Superior de Engenharia do Porto.

No capítulo seis são apresentadas as conclusões do presente trabalho seguidas das referências bibliográficas e dos apêndices.

Capítulo 2

Teoria da Legibilidade

Este capítulo trata do estudo da legibilidade de textos, em geral, a qual tem estado na base do estudo da legibilidade no software. Após uma revisão da literatura da área é apresentada uma breve perspectiva histórica do estudo da legibilidade, algumas críticas às fórmulas de legibilidade e os principais factores que afectam a legibilidade.

2.1 Introdução

Em inglês existem duas palavras, "readability" e "legibility", para a mesma palavra em português, legibilidade. Em inglês a palavra "legibility" refere-se ao aspecto tipográfico dos caracteres, como por exemplo, o seu tipo (de letra), ou se possui ou não serifas. Corresponde ao aspecto visual. A palavra "readability" refere-se às palavras e frases, ou seja, o texto é visto como um todo, estando relacionada com a facilidade de leitura de um texto [79]. Relaciona-se com a forma como o texto está escrito.

Neste trabalho, a legibilidade é a correspondente ao termo em inglês "readability".

A legibilidade de textos tem sido formalmente definida como a soma de todos os elementos no material textual que afectam a compreensão por parte do leitor, a velocidade de leitura e o nível de interesse do material [63]. Estes elementos podem incluir características tais como complexidade da sintaxe da frase. Em adição às características do texto, a legibilidade do texto é também uma função dos próprios leitores: o seu conhecimento de base social e educacional, o interesse e conhecimento técnico, a sua motivação na aprendizagem, bem como outros factores que podem ter um papel crítico de quão um texto é legível para um indivíduo ou população [58].

A percepção é o processamento de informação sensorial e faz parte da cognição humana, a qual também inclui consciência, raciocínio e aprendizagem. 75% de toda a informação do mundo real é captada visualmente, só 13% é captada através da audição e os restantes 12% são captados pelos outros sentidos [75, p.15].

Muitas técnicas de memorização, também chamadas de mnemónicas, exploram a memória visual pela visualização interna [75, p.16].

A informação textual pode ser aumentada através de sugestões visuais, como por exemplo, os sublinhados, a cor, a fonte da letra, o tipo de letra, e a própria orientação. Todos estes componentes realçam partes importantes, ou fazem com que a estrutura seja mais explícita. A arte de apresentação de textos numa forma legível e agradável visualmente é conhecida como tipografia [75, 35].

É possível encontrar diferentes definições para legibilidade. Na Tabela 2.1 são apresentadas algumas dessas definições.

A definição de George Klare [154] centra-se sobre o estilo de escrita como separado de questões, tais como, o índice, a coerência e a organização, associadas frequentemente à edição do texto.

A definição de McLaughlin[1] [184, 79] envolve o texto e o leitor, considerando as características dos leitores nomeadamente as suas competências de leitura, o conhecimento prévio e a motivação.

Gavora [101] refere que a legibilidade é uma das propriedades de texto mais importantes que afecta a sua compreensão, e que a sua compreensão é uma interpretação subjectiva do significado do conteúdo do texto por parte do leitor, uma vez que cada leitor o interpreta de uma forma única.

Zamanian [305] afirma que se assume frequentemente que a comunicação pressupõe compreensão. Esta comunicação também inclui a comunicação escrita. Estudos sobre a legibilidade de textos mostram que um texto de baixa legibilidade é também difícil de compreender [211].

Como se percebe, este conjunto de definições reflete de alguma forma a facilidade de leitura. Mas em geral, também contempla explicitamente a compreensão do texto. O

[1]McLaughlin é o criador de uma fórmula de legibilidade muito conhecida, designada por SMOG.

Tabela 2.1: Definições de Legibilidade.

Definição	**Autor, Ano**	**Referência**
"A soma total (incluindo todas as interacções) de todos os elementos dentro de uma peça de material impresso que afecta o sucesso que os leitores podem ter com esse material. O sucesso é a medida em que compreendem esse material, que o lêm a uma velocidade considerada óptima e que o consideram interessante."	Edgar Dale e Jeanne Chall's, 1949	[63, 79]
"a facilidade de compreensão tem a ver com o estilo de escrita"	George Klare, 1963	[154]
"o grau em que uma dada classe de pessoas considera certa leitura interessante (atraente) e compreensível."	G. Harry McLaughlin, 1969	[184, 79]
"Facilidade com que os materiais escritos podem ser lidos e entendidos. Isso depende de vários factores, incluindo o comprimento médio das frases, o número de novas palavras contidas, bem como a complexidade gramatical do idioma utilizado."	Richards et al., 1992	[120, p.306]
"a facilidade de leitura de palavras e frases é um atributo de clareza"	Gretchen Hargis e colegas (IBM), 1998	[79]
"carácter do que é legível; nitidez."	Dicionário da Porto Editora, 2004	[212]
"julgamento humano de quão fácil um texto é de entender."	Raymond Buse, 2008	[38]
"Apto para ser lido, interessante, agradável e atraente em estilo; e aprazível."	Webster Dictionary, 2000	[264]
"Facilidade com que o texto pode ser lido e entendido."	Peter Gavora, 2012	[101]

acto "mecânico" de ler um texto não é suficiente. Um texto tem de ser compreendido. Portanto, no contexto do presente trabalho, leitura e compreensão são indissociáveis.

Com base nas definições e tendo em atenção as observações anteriores, neste texto adoptou-se a definição de legibilidade de Gavora por explicitar que a legibilidade não envolve apenas o acto, mais ou menos mecânico, de ler, mas também a compreensão do que se lê.

Definição 1 (Legibilidade)**.** *A legibilidade consiste na capacidade de ler e entender os textos.*

2.2 Evolução do Estudo da Legibilidade

Como se verá, o estudo da legibilidade já tem muito tempo. Segundo Edgar Dale (citado por [193]), a legibilidade é um tema muito antigo que existiu desde sempre e afirma [193, p.638]:

"*Readability is as old as the hills and the written stories that have described them.*"

Dois exemplos em [239] atestam bem da antiguidade da preocupação com a linguagem: uma passagem da bíblia (I Coríntios) que refere a facilidade de compreensão das palavras e outro por Irving Lorge, que refere que já em 900 d.C. os Talmudistas faziam algumas contagens de palavras e ideias. A compreensão da linguagem remonta à retórica clássica de Platão e Aristoteles, assim como à análise de vocabulário da bíblia pelos antigos eruditos Hebreus (Lorge1944 citado por [304]). Também já na obra "Retórica" de Aristoteles, existia a preocupação com a melhoria dos textos e derivar regras de comunicação mais efectiva [12].

Segundo [12, p.285] o que torna a noção moderna de legibilidade diferente de outras discussões de comunicação efectiva são duas características específicas: (1) a ênfase quanto à facilidade em compreender um texto; e (2) a ênfase na quantificação. Estas duas características são evidentes em várias fórmulas de legibilidade como se poderá ver mais à frente com a fórmula Flesch (Flesch, 1948)[93], a fórmula Dale-Chall (Dale e Chall, 1948)[63] e o Índice de FOG (Gunning, 1952)[143].

Essas ênfases tornam a legibilidade um conceito particularmente atraente para os educadores. A atracção pelo uso de fórmulas para medir a capacidade de leitura encontra-se

na crença de que, em princípio, podem avaliar objectiva e quantitativamente a dificuldade do material escrito, sem medir as características dos leitores. Além disso, uma fórmula de legibilidade pode devolver uma pontuação numérica, dando ao utilizador a sensação de saber com precisão o nível de dificuldade de um texto [12, p.285].

Os estudos de legibilidade apenas começaram a ser realizados de forma mais sistemática no início do século XX (e.g. [305]). Num relatório de 1964 [88] já são listadas 701 teses de doutoramento associadas ao tema da leitura realizadas entre 1919 e 1961 e que os autores dividiram segundo 34 temas.

O estudo da legibilidade pode ser apresentado segundo diferentes perspectivas. Por exemplo, em [305] podem-se considerar dois grandes períodos no desenvolvimento da área de estudos da legibilidade. Aí, o início do primeiro período situa-se na década de 1920. O segundo período terá tido início na parte final da década de 50 [305].

Em [120], o estudo da legibilidade é dividido em três grandes vertentes, ou grupos de estudos. Estes grupos de estudos são explicados em seguida e são: (i) desenvolvimento das fórmulas da legibilidade; (ii) aplicação prática das fórmulas; (iii) avaliação da utilização das fórmulas.

No primeiro grupo de estudos da legibilidade foram apresentadas diversas fórmulas para medição da facilidade de leitura de textos. Essas fórmulas foram obtidas/desenvolvidas em resultado de terem sido questionados estudantes, bibliotecários, livreiros e professores acerca do que lhes parecia tornar os textos mais legíveis.

O segundo grupo de estudos da legibilidade consiste na aplicação das fórmulas de legibilidade, inclusivé, nos dias de hoje em que são bastante populares. São aplicadas numa grande diversidade de textos, tais como, livros de leitura, documentos governamentais, indústria, entre outros. Alguns exemplos do uso das fórmulas na determinação do nível de legibilidade na área do ensino são os seguintes [20, p.199]:

- Os bibliotecários recomendam livros baseados nos níveis de legibilidade;
- As editoras determinam a dificuldade do material curricular;
- Os investigadores desenvolvem avaliações da leitura e materiais de ensino;
- Os professores diferenciam o ensino.

O terceiro grupo de estudos de legibilidade pretende avaliar os prós e contras da utilização das diferentes fórmulas. São avaliados vários aspectos, tais como, a sua validade, o nível de adequação e a sua discrepância entre os resultados de diferentes fórmulas. Estes estudos têm a vantagem de ajudar os leitores a familiarizarem-se com as fórmulas e com as suas características e a sua aplicabilidade.

Em [79], o estudo da legibilidade pode ser analisado segundo três áreas:

- Estudos sobre a literacia dos adultos;
- Estudos de legibilidade clássica;
- Novos estudos da legibilidade.

Como se pode constatar, as duas últimas áreas são cronologicamente sequenciais. Para além disso, estas duas últimas áreas têm uma certa correspondência com os períodos definidos em [305]. Mais concretamente, a segunda área corresponde aproximadamente ao primeiro período, mas com o início ainda no século XIX, e o início da terceira área corresponde ao início do segundo período, mas sem um final definido e que se poderia estender até à data do artigo.

Nas secções seguintes apresentam-se os dois períodos referidos antes e aborda-se a área dos estudos de literacia pela sua relação próxima com a legibilidade, aproveitando-se para se introduzir uma primeira visão sobre a história do estudo da legibilidade.

2.2.1 Literacia

A literacia define-se como a competência de ler e escrever. A literacia funcional também se relaciona com a capacidade para ler e escrever, mas para além disso inclui a capacidade de actuar com base no nível em que a pessoa possui a primeira [7]. No âmbito deste trabalho a preocupação é com a leitura. Considerando apenas a leitura, a literacia funcional pode ser definida como se segue generalizando-se a definição em [7] relativa à área da saúde e adaptada da OCDE [237].

Definição 2 (Literacia Funcional)**.** *Literacia funcional é a competência de ler e compreender o material escrito e actuar com base neste.*

Utiliza-se aqui o termo competência para tornar claro que envolve aprendizagem, portanto, uma capacidade adquirida, ou desenvolvida, como utilizado em [98, p.11].

A literacia relaciona-se naturalmente com a legibilidade, mas medir o nível de literacia de alguém e o nível de legibilidade de um material escrito pode produzir (mas não tem de produzir) valores não coincidentes [64]. O estudo da literacia recorre frequentemente a fórmulas de legibilidade [203]. A medição da legibilidade indica geralmente o nível de ensino requerido. A literacia funcional pretende medir o que foi aprendido durante os anos de escolaridade do indivíduo e a capacidade de ler e compreender informação nova [7, p.2].

Com base na definição de literacia funcional, pode-se afirmar que a legibilidade será uma parte da literacia e, sendo assim, que a literacia tem um carácter mais abrangente. Também, os níveis de literacia, definidos de acordo com padrões existentes, indicam competências necessárias independentemente da idade, o que não acontece com várias fórmulas de legibilidade que dependem da idade do leitor. Em rigor, sabe-se que os idosos apresentam piores níveis de literacia funcional porque as suas capacidades de leitura também se degradam [15, p.368].

Logo, o nível de legibilidade também deve ter em conta os leitores mais velhos. Seja como for, estas diferenças devem ser interpretadas como meramente indicativas pois sabe-se que existe alguma falta de consenso nas definições que têm sido propostas para literacia funcional [14].

O estudo da literacia aqui abordado tem por base o exposto em [79] relativamente aos Estados Unidos da América. Através destes estudos, descobriu-se que existem grandes diferenças nos adultos em termos de competências de leitura e as suas implicações na sociedade.

Antes dos meados do século XIX, os alunos aprendiam essencialmente a partir de livros que as suas famílias tinham, em particular da própria Bíblia e de cartilhas. Foi apenas em 1847 que surgiu a primeira escola primária, em Boston, com vários livros de acordo com os graus de leitura dos seus alunos [79]. Aí, os professores verificaram que a aprendizagem da leitura era gradual e que melhorava quando os livros se adequavam ao seu nível. De tal forma, que os alunos passaram a ser agrupados de acordo com o seu nível de leitura em cada um dos anos até ao final do ensino secundário.

Ao longo do tempo foram realizados vários estudos sobre literacia [79], tendo sido definidos níveis de literacia.

Em 1926, foram criados testes de leitura normalizados (*Standardized Reading Test*) para medir a capacidade de leitura por parte dos alunos nos Estados Unidos. Através do uso destes testes, os professores definiram objectivos de leitura para as suas turmas. Os testes publicados em 1926 foram revistos posteriormente em 1950, 1961 e 1979. Tornaram-se uma medida importante da capacidade de leitura dos alunos nos Estados Unidos.

No caso dos adultos, o primeiro teste sistemático para avaliação do seu nível de literacia realizou-se em 1917 no exército norte-americano. Desde os anos 50, do século passado, que para entrar nas forças armadas, tem de se obter aproveitamento a um teste de literacia. Através dos testes elaborados pelo exército é possível obter muita informação acerca das aptidões, competências cognitivas e a capacidade de executar uma tarefa da pessoa avaliada. As principais descobertas de investigação militar foram que as medidas de literacia estão muito relacionadas com [20, p.5-6]:

- A medida de inteligência e aptidão para as tarefas;
- O nível de conhecimentos de cada indivíduo;
- O desempenho no trabalho; foram realizados centenas de estudos que mostraram não haver separação entre o nível de literacia e o desempenho dos indivíduos;
- Os programas de literacia no local de trabalho são muito eficazes, e rapidamente, a produzir melhorias significativas na capacidade de leitura sobre temas relacionados com o trabalho;
- Os leitores mais avançados possuem mais conhecimentos e executam bem tarefas em muitos domínios do conhecimento. No caso de leitores com poucas competências ao nível da leitura, estes executam pior os seus trabalhos.

A marinha norte-americana também se interessou pelo problema de compreensão de material técnico por parte do pessoal militar [143, 20]. O interesse surgiu após um estudo, elaborado pela Força Aérea Americana, indicar que o nível de dificuldade de leitura das instruções dos manuais utilizados conduzia a muitos erros: quanto mais difícil de ler era

o material, mais erros eram cometidos. Uma das formas de resolver o problema foi tornar o material mais legível. Para tal usaram as fórmulas de legibilidade como método de estimativa de leitura.

Relativamente à literacia de civis, os testes a adultos começaram em 1937 em Chicago com a realização de dois estudos, que são descritos em seguida, segundo o exposto em [79].

Em 1937, a Universidade de Chicago realizou um inquérito em que foram inquiridos mil adultos com diferentes níveis de educação. Foram medidas as suas competências na leitura de diferentes tipos de textos tais como anúncios de alimentação, anúncios de filmes e listas de telefones, e, ainda usaram testes mais tradicionais de compreensão de parágrafos e vocabulário. O estudo confirmou a existência de uma relação entre as competências de leitura e os anos de educação completados. Mais concretamente, que as competências de leitura aumentam com o aumento da educação. O estudo sugeriu que a educação deve levar as pessoas a lerem mais e que se lerem mais, isso aumentará as suas competências de leitura.

O segundo estudo, foi realizado entre os anos de 1970 e 1971 pela *National Assessment of Educational Progress (NAEP)* e testou jovens estudantes e adultos. Os alunos tinham 9, 13 e 17 anos e os adultos idades dos 26 aos 35. A ambos foi solicitada a execução de 21 tarefas distintas. O resultado do estudo revelou que a literacia aumenta progressivamente ao longo do percurso escolar, ao longo das várias idades, desde crianças, até se tornarem adultos.

Mais tarde, foram realizados outros dois estudos importantes, o primeiro a jovens e o segundo a adultos. O primeiro estudo foi realizado em 1985 e foi intitulado de *Young Adult Literacy Survey (YALS)*, aplicado a jovens dos 17 aos 25 anos, e em 1992 foi realizado o outro estudo para adultos, intitulado de *National Adult Literacy Surveys (NALS)*. Ambos os estudos mediram da mesma forma a literacia. Mediram em três áreas específicas e usaram a mesma escala de pontuação com valores entre 1 e 500 e ambos usaram os níveis de competência definidos pela NAEP em 1985. As áreas de medição foram as seguintes:

- Literacia da prosa - significado dos textos;
- Literacia de documentos - pesquisa de informação, por exemplo, num formulário;

- Literacia quantitativa - tarefas matemáticas e espaciais.

A tabela seguinte, retirada de [79], mostra a correspondência entre os níveis de NAEP e o grau de leitura [79].

Tabela 2.2: Níveis de Literacia.

Nível NAEP	Pontuação Literacia	Grau
I Rudimentar	150	1.5
II Básico	200	3.6
III Intermédio	250	7.2
IV Perito	300	12
V Avançado	350	16+

A investigação acerca da literacia, não só com militares, mas também com civis, mostrou que os programas de literacia no local de trabalho produziam resultados positivos, duradouros e com baixo custo [79, p.9]. Uma análise de 75 anos de estudos de literacia pode ser encontrada em [265].

2.2.2 Primeiro Período

De acordo com [305] o primeiro período de estudos de legibilidade, dita clássica, iniciou-se por volta de 1920. No entanto, [79] situa os estudos mais antigos sobre legibilidade no final do século XIX, como se viu na secção 2.2.1 no caso das escolas primárias. Ambos estarão correctos na medida em que [305] considera apenas a altura a partir da qual os estudos sobre legibilidade começaram a ser realizados de forma mais sistemática.

Ao longo dos séculos, escreveu-se muito sobre as diferenças entre os estilos do inglês ornamentado e simples [79, p.10]. Em 1880, Lucius Adelno Sherma, professor de Literatura Inglesa da Universidade do Nebraska, começou a ensinar literatura a partir de pontos de vista históricos e estatísticos e no seu livro, em 1893, "Analytics of Literature, A Manual for the Objective Study of English Prose and Poetry" mostrou como o tamanho das frases se tornavam mais pequenas ao longo dos tempos.

No início do século XX, existia uma grande preocupação com a dificuldade de compreensão dos conteúdos de livros de texto e outros materiais escritos [304]. Surge assim a investigação de métodos práticos que associassem os textos às competências dos leitores, estudantes e adultos [79, p.11]. Mais especificamente, a principal finalidade dos estudos sobre legibilidade clássica era desenvolver métodos práticos que pudessem ser usados em materiais de leitura de estudantes e adultos, no sentido de determinarem o seu nível de legibilidade.

Neste período, a investigação da legibilidade decorreu essencialmente em duas vertentes, que correspondiam às preocupações existentes [304]:

- Estudo do controlo do vocabulário;
- Estudo de medição da legibilidade.

Ambas as vertentes (áreas de investigação) tiveram o seu início por volta da mesma altura, embora os estudos de controlo de vocabulário tenham prevalecido durante os primeiros anos.

Através de estudos de controlo do vocabulário pretendia-se avaliar qual o vocabulário mais eficaz para se aprender a ler a partir da leitura de livros de texto. Nesses estudos eram estudadas as novas palavras de cada livro, o número de vezes que apareciam e a sua dificuldade. Estes estudos centraram-se ao nível do ensino primário.

Em 1921, Thorndike publicou "The teacher's Word Book" [275], um livro que fornecia uma forma de medir a dificuldade das palavras. Para isso, tabulou as palavras de acordo com a frequência do seu uso na literatura em geral, assumindo que as palavras que eram encontradas mais frequentemente pelos leitores eram de menor dificuldade de compreensão que as que apareciam mais raramente. O livro de Thorndike foi a primeira lista extensiva de palavras em inglês apresentada pela frequência.

Entretanto apareceram outras listas indicando a dificuldade das várias palavras e houve mesmo diversas lições para ensino da leitura que foram adaptadas tendo em atenção a dificuldade de cada palavra.

A segunda vertente consistiu em estudos de medidas de legibilidade. Nesse sentido concentraram-se esforços de modo a criar fórmulas de legibilidade que os professores e bibliotecários pudessem usar [79, p.11]. Esta área de investigação deu grande importância

aos materiais para graus de ensino mais elevados, incluindo o universitário e o ensino de adultos. Os estudos mais antigos sobre legibilidade eram conduzidos através de questionários a alunos, livreiros, bibliotecários e professores sobre o que achavam que tornava os textos legíveis [305]. Em 1920, a população da escolaridade júnior e sénior foi alterada. Anteriormente a população completava a sua escolaridade formal nos graus elementares. Nesta nova geração, os estudantes já frequentavam o secundário [79, 305].

Em 1921, o psicológo Kitson publicou "The Mind of the Buyer" [153, p.77] onde mostra como e porque é que os leitores das diferentes revistas e jornais são diferentes uns dos outros. Na sua teoria, o tamanho da frase e da palavra, medidas pelas sílabas, são bons indicadores de legibilidade. Ele confirmou as suas teorias através de análises de jornais e revistas. Mais tarde Catalano confirmou esta teoria [47].

Ao longo do século XX foram desenvolvidas várias fórmulas de legibilidade com diferentes propósitos e que englobavam documentos governamentais, artigos de jornal, documentos médicos, livros escolares, entre outros [20]. O desenvolvimento de fórmulas de legibilidade foi feito inicialmente com base na língua inglesa. A segunda língua para a qual foram realizados trabalhos de legibilidade foi o espanhol. Seguiram-se outros trabalhos similares em outras línguas. Em 1980, existiam cerca de 200 fórmulas e mais de mil estudos publicados sobre testes das fórmulas de legibilidade, da sua validade teórica e estatística [79, 22].

Em 1923, Bertha A. Lively e Sidney L. Pressey tiveram dificuldade em escolher livros de texto científicos para os alunos de escolaridade júnior. Os livros tinham muitas palavras técnicas e com isso teriam de gastar muito tempo das aulas a ensinar o vocabulário. No seu artigo é apresentada a primeira fórmula de legibilidade para crianças. O estudo de Lively-Pressey teve grande influência nos estudos de legibilidade que se seguiram.

Entre 1928 e 1945 realizaram-se estudos que mostraram quais as fórmulas de legibilidade mais adequadas para aplicar à literatura infantil com o propósito de determinar a compreensibilidade de material escrito nessa área [130].

A fórmula de "Flesch Reading Ease Readability" publicada em 1948 [120] terá também sido uma das primeiras. Este índice de legibilidade é baseado no número médio de sílabas por palavra e palavras por frase. A taxa de facilidade de leitura de textos da fórmula de legibilidade de Flesch é dada numa escala de 100 pontos, em que quanto maior a

pontuação, mais fácil é entender o texto. Em 1952, Gunning [305, 79, 120] publicou uma fórmula de legibilidade desenvolvida para adultos, o Índice de Fog (*Gunning Fog Index*), a qual se tornou muito popular devido à sua facilidade de utilização. tualiz Com a publicação das fórmulas de Flesch e de Dale-Chall também em 1948 e Gunning em 1952 entre outras, terminou o primeiro período de desenvolvimento dos estudos da legibilidade [305, p.45]. Já [79] considera que este período terminou na década de 40 com a publicação das fórmulas de Flesh e de Dale-Chall. Dada a importância da fórmula de Gunning parece mais aceitável a posição de [305].

Em síntese, por volta de 1920, os educadores descobriram uma forma de usar a dificuldade do vocabulário e o comprimento da frase para prever o nível de dificuldade de um texto. Embeberam este método nas fórmulas de legibilidade, mostrando o seu valor em mais de 80 anos de aplicação [79]. Até aos anos 50 do século XX, a investigação das fórmulas eram uma espécie de segredo e as fórmulas eram usadas tipicamente em jornalismo, investigação, saúde, direito, seguros e indústria [79].

2.2.3 Segundo Período

Mais tarde, mas ainda na década de 50 do séc. XX, surgiu o período de consolidação e estudo aprofundado da legibilidade por parte de um número mais alargado de investigadores. Este número mais alargado permitiu acelerar o estudo da legibilidade. Neste período, houve uma maior preocupação em perceber e melhorar as fórmulas existentes.

Novos desenvolvimentos transformaram o estudo da legibilidade, como as contribuições da psicologia linguística e cognitiva, o que fez com que os investigadores, explorassem também de que modo a motivação, o interesse e o conhecimento prévio afectavam a legibilidade [79]. Os estudos neste período levaram à criação de fórmulas mais fiáveis.

Um dos primeiros trabalhos de investigação dessa época foi conduzido por Fry (1968). Fry criou um dos testes mais populares o qual utiliza um gráfico onde estão representados os diferentes níveis de legibilidade. Posteriormente, em 1969, Harry McLaughlin publicou a fórmula *Simple Measure Of Gobbledygook* (SMOG). Em 1975 foi apresentada outra fórmula bem conhecida na área da legibilidade, a Fórmula Flesch-Kincaid. Esta fórmula é uma recalibração da fórmula de Flesch.

Resumidamente, este período caracterizou-se pelos seguintes aspectos [79, p.25]:

- Existência de uma comunidade de estudiosos que avaliava a evolução das fórmulas desenvolvidas anteriormente;
- O procedimento *Cloze* publicado por Wilson Taylor em 1953 que abriu novos horizontes no estudo da legibilidade ao trazer novas formas mais precisas e detalhadas de medir as propriedades dos textos e dos leitores;
- Inclusão de novas variáveis, como a capacidade de leitura, conhecimento prévio, interesse e motivação, no estudo da legibilidade;
- Estudos sobre a eficiência da leitura que analisavam o efeito da legibilidade na velocidade de leitura e persistência;
- Medição do conteúdo, por influência da psicologia cognitiva e da linguística, através de novos estudos quanto a factores cognitivos e estruturais no texto e sobre como usá-los para estimar a legibilidade;
- Determinação qualitativa e subjectiva do nível do texto (*text leveling*), que com formação e prática poder ser eficaz;
- Utilização das fórmulas na produção e transformação do texto para o colocar num dado nível;
- Novas fórmulas de legibilidade;
- Estudo de discrepância entre fórmulas e como essas diferenças poderiam ser aproveitadas.

Numa fase mais avançada do segundo período, com o aproximar do séc XXI, verificou-se uma utilização mais alargada das fórmulas devido à possibilidade da sua computação. Para além disso, e com a facilidade da computação, as fórmulas mais recentes usam em geral mais parâmetros. Alguns exemplos são a *Framework Lexile* baseada na linguística [12] e a ferramenta informática *Coh-Metrix* [103].

2.3 Fórmulas de Legibilidade

As fórmulas de legibilidade foram criadas inicialmente para prever a dificuldade de leitura dos textos [120, 305, 79]. Estes esforços centraram-se em tornar mais fácil a apli-

cação das fórmulas de legibilidade de modo a que os bibliotecários e professores pudessem usar. O resultado das fórmulas é uma estimativa que representa em geral o número de anos de educação que é necessário as pessoas possuírem para compreender um dado texto [156, 305]. Ou seja, é o nível de escolaridade mínimo, daí que o resultado seja designado por grau de escolaridade, ou simplesmente grau. Noutros casos, o resultado tem de ser ajustado para poder representar um grau de escolaridade.

Ao longo dos anos foram desenvolvidas e utilizadas várias fórmulas com variados propósitos sendo dois exemplos disso as fórmulas Forcast e de Fry [20]. Em 1980 já existiam mais de 200 fórmulas de legibilidade diferentes [79, p.2]. O elevado número de fórmulas e de estudos utilizando-as com sucesso ajudam a credibilizá-las. No entanto, umas são mais conhecidas do que outras [305].

Os estudos iniciais mostraram que as fórmulas classificavam muito bem quando comparadas com uma outra medida psicométrica utilizada, como seja o teste de leitura. As fórmulas de legibilidade também têm sido utilizadas para ajudar os autores de material escrito a tornar o seu material mais legível.

2.3.1 Apresentação Diacrónica das Fórmulas

Antes de aparecer a primeira fórmula de legibilidade, por volta de 1911, Thorndike começou a fazer a contagem da frequência das palavras nos textos de inglês. Em 1921, publicou "The Teacher's Word Book" onde listava 10.000 palavras. Thorndike tinha consciência que a sua lista era muito limitada, pois muitas palavras importantes não constavam na sua lista. Mais tarde, em 1932, actualizou a sua lista para 20.000 palavras, e em 1944, juntamente com Irving Lorge, actualizou a sua lista para 30.000 palavras, conforme descrito mais abaixo [79][13].

A primeira fórmula terá sido criada em 1923 por Bertha A. Lively e Sidney L. Pressey [305, 79]. A fórmula surgiu devido à dificuldade que os seus autores tinham em escolher livros de texto para os alunos de escolaridade mais jovem. Os livros utilizados continham muitas palavras desconhecidas dos alunos e com isso consumia-se muito do tempo das aulas a ensinar esse vocabulário. Lively e Pressey usaram três métricas para avaliar a dificuldade do vocabulário, em que duas delas dependiam da lista de Thorndike:

Na fórmula, para cada 1000 palavras é medido [305, 79, 13]:

1. O número de palavras diferentes;

2. O número de palavras não existentes na lista de Thorndike;

3. A mediana dos índices das palavras encontradas na lista de Thorndike.

Os autores testaram a sua fórmula com 11 amostras, em livros de texto de dificuldades diferentes e num jornal. Testaram para os segundo e quarto graus de leitores [13].

Em 1928, Mabel Vogel e Carleton Washburne realizaram um dos estudos considerados mais importantes sobre legibilidade. Foram os primeiros a estudar as características estruturais do texto, a estrutura da frase e os primeiros a usar a forma de pronúncia das palavras. Estudaram dez factores diferentes, entre eles, os tipos de frases, as frases proposicionais, a dificuldade das palavras e os comprimentos das frases, e para a sua fórmula, designada de *Winnetka*, seleccionaram 4 desses factores. A validação teve por base os resultados de testes de leitura de crianças e verificaram que os resultados obtidos com a fórmula tinham uma correlação elevada com os resultados desses testes de leitura. Esta fórmula foi a primeira a prever a dificuldade por grau de ensino nos Estados Unidos (1 a 12) e tornou-se o protótipo das fórmulas de legibilidade modernas. A fórmula avalia do 1º ao 9º grau [130].

A fórmula é a seguinte:

$$X1 = 0.085X2 + 0.101X3 + 0.604X4 - 0.411X5 + 17.43 \tag{2.1}$$

em que, para uma amostra de 1000 palavras [130, p.42][13]:
$X1$ corresponde ao grau necessário para compreender o texto,
$X2$ é o número de palavras diferentes que ocorrem na amostra,
$X3$ é o número de preposições na amostra incluindo duplicados,
$X4$ é o número de palavras, incluindo duplicados, não existentes na lista de Thorndike,
$X5$ é o número de frases simples numa amostra de 75 frases.

Em 1931, W.W. Patty e W. I. Painter desenvolveram uma fórmula para medir a dificuldade relativa dos livros de texto baseados numa combinação de frequência como determinado pela lista de Thorndike e a diversidade do vocabulário (o número de palavras diferentes num texto) [79, 13].

Apesar de muitos dos estudos se focarem na população jovem, começou a haver um crescente interesse pela população adulta [13].

Também em 1931, Douglas Waples e Ralph W. Tyler publicaram um estudo de dois anos sobre o interesse de leitura do adulto, com o nome de "What People Want to Read About". Descobriram que a capacidade de leitura de muitas pessoas é limitada devido à lacuna de material adequado. Por vezes, os leitores pretendem expandir os seus conhecimentos mas o material de leitura em que eles estão interessados são excessivamente difíceis [79, 13].

Em 1934, Ralph Ojeman criou um método para avaliar a dificuldade dos materiais de adultos. Este método consiste em fazer 16 passagens de cerca de 500 palavras, tiradas de revistas. Atribuiu cada passagem de nível de leitores adultos, àqueles que eram capazes de responder, a pelo menos, metade das questões de escolha múltipla.

Ainda em 1934, Ralph Tyler começou a interessar-se pelos adultos com limitações nas capacidades de leitura [79, 13]. Nesse sentido, em conjunto com Edgar Dale, desenvolveram uma fórmula de legibilidade e apresentaram um estudo de fórmulas de legibilidade para adultos. Na sua fórmula combinaram três factores para prever a proporção de adultos leitores de capacidade de leitura limitada que fossem capazes de entender os materiais.

A fórmula desenvolvida é a seguinte:

$$X1 = -9.4X2 - 0.4X3 + 2.2X4 + 114.4 \pm 9.0 \tag{2.2}$$

em que:
$X1$ representa a percentagem de um grupo de adultos do 3º ao 5º grau,
$X2$ é o número de palavras técnicas diferentes,
$X3$ é o número de palavras não técnicas diferentes,
$X4$ é o número de cláusulas indeterminadas na selecção.

Os trabalhos de Ojeman, Tyler e Dale marcam o início dos estudos com adultos.

Em 1935, William Gray e Bernice Leary [79, 156, 13] publicaram o livro "What Makes a Book Readable" que foi considerado um marco na investigação. Os autores identificaram 288 elementos que afectam a legibilidade e agruparam-nos sob 4 títulos, como se segue:

- Conteúdo: refere-se ao tema do texto (argumentos, estrutura e coerência);

- Estilo da expressão e apresentação: prende-se com os tipos de frases (sintaxe) e palavras usadas no texto (semântica);
- Formato: diz respeito à apresentação dos aspectos visuais tais como a tipografia, o *layout* da página e as ilustrações;
- Características gerais de organização: estas características são os parágrafos, cabeçalhos usados para organizar as ideias no texto e a navegação no texto.

Destes 4 grupos, apontaram o conteúdo do texto como o factor mais importante que afecta a legibilidade, logo seguido do estilo [79, 156]. Baseada nas suas investigações, propuseram a seguinte fórmula [13]:

$$X1 = -0.01029X2 + 0.009012X3 + 0.02094X4 - 0.03313X5 - 0.01485X6 + 3.774 \quad (2.3)$$

em que:
$X1$ refere-se à pontuação média,
$X2$ é o número de palavras difíceis numa passagem,
$X3$ é o número de pronomes pessoais,
$X4$ é o número médio de palavras numa frase do texto,
$X5$ é a percentagem do total de palavras que são únicas,
$X6$ é o número de frases preposicionais no texto.

Em 1938, Irving Lorge, publicou um livro com uma lista relativa à frequência de utilização das palavras na Língua Inglesa com o título "The Semantic Count of the 570 Commonest English Words". Em 1944, juntamente com Thorndike, conforme referido anteriormente, publicaram um novo livro que expandia a lista de palavras para 30.000, intitulado "The Teacher's Word Book of 30.000 Words" [107, p.37][79, 13, p.19].

Em 1943, Rudolph Flesch obteve o seu doutoramento em investigação educacional. Na respectiva tese publicou a sua primeira fórmula de legibilidade para medir a compreensibilidade de textos destinados a adultos e, mais tarde, em 1948, publicou a sua segunda fórmula de legibilidade [305, p.44][79, p.21][120, p.177][71, 264, p.2]. Na sua primeira fórmula usou duas variáveis. Uma relativa a afixos (que são por definição morfemas[2] dependentes), e outra a referências pessoais, tais como pronomes pessoais e nomes.

[2]Na linguística, um morfema corresponde à unidade gramatical mais pequena, com significado.

A primeira fórmula desenvolvida por Flesch foi a seguinte:

$$X1 = 0.1338X2 + 0.0645X3 - 0.0659X4 + 4.2498 \tag{2.4}$$

em que:
$X1$ refere-se à percentagem de respostas de crianças com pelo menos 75% de respostas correctas,
$X2$ é a média do comprimento da frase,
$X3$ é o número de afixos,
$X4$ é o número de referências pessoais.

Quanto à segunda fórmula, o objectivo foi simplificá-la, de modo a ser fácil de aplicar [13]. Flesch dividiu-a em duas partes, cada uma contendo dois factores. Na primeira parte, com o nome *Flesch Reading Ease* deixou de considerar os afixos e passou a usar duas variáveis, o número de sílabas e o número de frases para cada amostra de 100 palavras. Na segunda parte da fórmula faz a contagem do número de palavras pessoais (pronomes e nomes), frases pessoais (citações, exclamações e frases incompletas)[79, p.21]. Esta segunda fórmula (a primeira parte) é considerada uma das fórmulas mais populares e é a fórmula utilizada no Microsoft Word [120, 305].

A fórmula *Flesch Reading Ease* tem a seguinte expressão [305, p.44][79, p.21][120, p.177][103, 13, p.7]:

$$RE = 206.835 - (1.015ASL) - (84.6ASW) \tag{2.5}$$

em que:
RE (*Reading Ease*) é a posição numa escala entre 0 e 100, sendo o intervalo entre 0 e 30 considerado muito difícil e acima de 70 acessível a adultos [79],
ASL é o comprimento médio das frases medido em palavras ($\frac{\text{número de palavras}}{\text{número de frases}}$),
ASW é a média de sílabas por palavra ($\frac{\text{número de sílabas}}{\text{número de palavras}}$).

Em 1944, Lorge publicou uma nova fórmula, conhecida por *Lorge Readability Index*. Este índice é uma estimativa do grau de leitura. Apesar de ter sido criado para medição

da legibilidade em jovens em idade escolar, foi muito usada com adultos. A sua fórmula é a seguinte [79, 13]:

$$Grau = 0.07(w/s) + 13.01(p/w) + 10.73(h/w) + 1.6126 \tag{2.6}$$

em que:
$Grau$ é uma estimativa do grau de leitura,
w é o número total de palavras na amostra,
s é o número de frases,
p é o número total de frases preposicionais,
h é o número de palavras difícieis (não existentes na lista de Dale de 769 palavras[3]).

Em 1948, Edgar Dale juntamente com Jeanne Chall desenvolveram uma fórmula para adultos e crianças acima do 4º nível de escolaridade que pretendia corrigir algumas limitações da fórmula de Flesch. Definiram também uma lista com 3.000 palavras consideradas fáceis, conhecidas por 80% dos leitores do 4º grau [79, 120, 13].

A fórmula é a seguinte:

$$RawScore = 0.1579PDW + 0.496ASL + 3.6365 \tag{2.7}$$

em que:
$RawScore$ corresponde ao nível a que um leitor consegue responder a metade das questões de um teste,
PDW é a percentagem de palavras difíceis, por serem pouco familiares (não existentes na sua lista),
ASL é a média do tamanho das palavras nas frases.

Em 1951, Farr, Jenkins e Paterson modificaram a fórmula de Flesch no sentido de a simplificar [79]. Substituíram o número médio de sílabas por palavra, pelo número de palavras de uma sílaba (por 100 palavras), ficando:

$$Grau = 1.599X1 - 1.015X2 - 31.517 \tag{2.8}$$

[3]também publicada em 1944.

em que:

$X1$ é o número de palavras de uma sílaba por 100 palavras,

$X2$ é o comprimentos médio das frases em palavras.

Em 1952, Gunning publicou a sua primeira fórmula de legibilidade para adultos designada por *Índice de Fog*, a qual se tornou muito popular devido à sua facilidade de utilização [120][79]. Esta fórmula entra em consideração o número de palavras com três ou mais sílabas. Tratou-se da primeira fórmula a utilizar a contagem de polissílabos para medir a dificuldade semântica do texto [184, p.641]. O índice de Fog corresponde ao nível de leitura requerido para compreender o material.

A fórmula é a seguinte:

$$FogIndex = 0.4(ASL + PDW) \tag{2.9}$$

em que:

ASL é a média do comprimento das palavras,

PDW é a percentagem de palavras difíceis (com mais de duas sílabas).

Em 1953, Wilson Taylor publicou uma nova ferramenta para medir a legibilidade designada por *Cloze Procedure* [79, 25]. Propôs a utilização de testes *Cloze* para medir a compreensão individual de um texto. O procedimento *Cloze* consiste em eliminar sistematicamente as palavras numa selecção de parágrafos, avaliando assim o sucesso do leitor ao fornecer com precisão as palavras excluídas. Nestes testes, o leitor tem de preencher os espaços com as palavras omissas. Este tipo de testes continua a ser muito comum nos dias de hoje no ensino da língua inglesa.

No procedimento *Cloze*, Taylor usou um método completamente mecânico na escolha de palavras a serem excluídas e exigindo a substituição exacta da palavra original. Para a construção do teste, elaborou um estudo onde comparou a eliminação aleatória de palavras (ou partes de texto) com a eliminação de cada n-ésima palavra. Concluiu que a eliminação de cada n-ésima palavra era mais fácil de usar. Mais tarde, Taylor sugeriu a possibilidade de escolha de métodos alternativos na marcação do teste *Cloze* [25].

Continuando em 1953, George Spache apresenta uma nova fórmula designada por *Spache* que considera diferir das fórmulas de Lorge, de Flesch e de Dale-Chall [259]. Esta nova fórmula de legibilidade utilizada na avaliação de materiais de leitura destinados aos

primeiros graus. O seu trabalho foi publicado no *The Journal Elementary School*. Spache considerou que a dificuldade de leitura se relaciona com o comprimentos das frases e com a percentagem de palavras difíceis. Nesse sentido, a fórmula calcula o nível de uma amostra de texto tendo por base o comprimento das frases e número de palavras desconhecidas.

A fórmula Spache considera as "palavras desconhecidas" como sendo as palavras em que os individuos do terceiro grau ou abaixo, não reconhecem estas palavras. A fórmula Spache é por isso usada preferencialmente para calcular a dificuldade do texto até ao grau 3 [79].

Apesar da fórmula Spache ser muito similar à fórmula de Dale-Chall, a fórmula Spache foi projectada para a legibilidade dos textos até o final do terceiro nível (grau 3), e no caso da fórmula de Dale-Chall foi projectada para medir a capacidade de leitura de textos mais avançados, a partir do 4º grau.

$$SpacheIndex = (0.141ASL) + (0.086PDW) + 0.839 \tag{2.10}$$

em que:
ASL é a média do comprimento da frase por 100 palavras,
PDW é a percentagem de palavras difíceis.

Em 1958, Powers, Sumner e Kearl produziram um artigo na revista *Journal of Educational Psychology*, da Universidade de Wisconsin, onde apresentaram uma nova fórmula que eles consideraram como uma simplificação das fórmulas de Fog, de Flesch e de Dale-Chall. A nova fórmula designa-se por *Powers-Sumner-Kearl (PSK)* e é uma das melhores fórmulas para calcular o nível de ensino nos EUA, simplesmente a partir de uma amostra de texto com base no comprimento da frase e no número de sílabas. Esta fórmula era adequada para crianças da escola primária (dos 7 aos 10 anos) e, normalmente, não é considerada ideal para crianças com idade acima dos 10 anos.

A fórmula é a seguinte:

$$GL = 0.0778ASL + 0.0455NS - 2.2029 \tag{2.11}$$

em que:
GL é o nível norte-americano,

ASL é a média dos comprimentos das frases,
NS é o número de sílabas.

Em 1965, Edmund Coleman criou quatro fórmulas de legibilidade na sequência de um estudo que realizou. Coleman foi o primeiro a utilizar o procedimento *Cloze* como critério [79]. As suas fórmulas estimam a percentagem de preenchimentos correctos de palavras omissas. As quatro fórmulas são as seguintes:

$$C\% = 1.29w - 38.45 \tag{2.12}$$

$$C\% = 1.16w + 1.48s + 37.95 \tag{2.13}$$

$$C\% = 1.07s + 1.18s + 0.76p - 34.02 \tag{2.14}$$

$$C\% = 1.04w + 1.06s + 0.56p + 0.36prep - 26.01 \tag{2.15}$$

em que:
$C\%$ é a estimativa da percentagem de preenchimentos correctos no *Cloze*,
w é o número de palavras de uma sílaba em 100 palavras,
s é o número de frases em 100 palavras,
p é o número de pronomes em 100 palavras,
$prep$ é o número de preposições em 100 palavras.

Em 1966 ,John Bormuth conduziu vários estudos, e em 1969, realizou um estudo sobre as variáveis da legibilidade e a sua relação com a compreensão. A partir desse estudo produziu uma fórmula com três versões diferentes: a primeira para uma utilização que intitulou de básica, a segunda para uso através de uma máquina e a terceira para utilização manual. A sua fórmula inicialmente designada por *Bormuth Mean Cloze Formula* em que usava o procedimento *Cloze* de Taylor, foi mais tarde, em 1981, adaptada a graus de leitura. A fórmula original é a seguinte:

$$R = R1 - R2 \tag{2.16}$$

$$R1 = 0.886593 - 0.083640(LET/W) + 0.161911(DLL/W)^3 \tag{2.17}$$

$$R2 = 0.021401(W/SEN) + 0.000577(W/SEN)^2 - 0.000005(W/SEN)^3 \tag{2.18}$$

em que:

R é a classificação *Mean Cloze*,

LET é o número de letras numa passagem,

W é o número de palavras numa passagem,

DLL é o número de palavras da lista original de Dale-Chall numa passagem,

SEN é o número de frases numa passagem,

DRP corresponde a graus de leitura numa escala de 0 a 100.

Em 1967, Smith e Senter criaram especificamente para a marinha norte-americana uma nova fórmula designada por *Automated Readability Index (ARI)*, a qual usava uma máquina de escrever eléctrica à qual estavam ligados três comutadores que permitiam fazer a contagem de frases e palavras.

Esta fórmula consiste num teste de legibilidade para avaliar a compreeensão de um texto. Da mesma forma que outras fórmulas de legibilidade, a fórmula ARI gera um valor que corresponde ao grau necessário para compreender um texto [197].

Usa a dificuldade da palavra e dificuldade da instrução/frase. A dificuldade da palavra refere-se ao número de letras por palavra e a dificuldade da frase refere-se ao número de palavras existentes numa frase. O primeiro passo é estabelecer os factores usados e relacioná-los com outros índices. No processo de verificação da relação entre os factores é auto-evidente (*self-evidence*).

Existem dois factores associados com a maioria dos factores de legibilidade. O primeiro factor está relacionado com a estrutura da frase que tem a ver com o número de palavras. O segundo factor está relacionado com a estrutura da palavra que é formada por letras.

A lista de palavras tem vantagens, no entanto, é lenta e relativamente insegura quando aplicada a material de leitura de adultos. O contador de sílabas não é fiável.

A fórmula ARI é a seguinte:

$$Grau = 0.50X1 + 4.71X2 - 21.43 \tag{2.19}$$

em que:

$X1$ é o número de palavras por frase,

$X2$ é o número de caracteres por palavra.

Em 1968, Edward Fry publicou uma fórmula de legibilidade que incorpora um gráfico [79, 120, 20, 184, 13]. No gráfico é registado o tamanho das frases e o tamanho das palavras. O tamanho das frases é obtido pelo número de palavras e o tamanho das palavras é obtido pelo número de sílabas. O gráfico de Fry foi desenvolvido no Uganda para obter uma estimativa do nível de ensino da população. Para isso, foi usada uma representação gráfica do número médio de frases e do número de sílabas por 100 palavras [20, p.199]. O algoritmo para cálculo do nível é o seguinte:

1. Seleccionar uma amostra de 100 palavras de um texto;
2. No eixo dos YY (vertical), marcar o tamanho médio de uma frase na amostra;
3. No eixo dos XX (horizontal) marcar o tamanho médio de uma palavra;
4. A zona do gráfico onde se encontra o ponto corresponde ao grau associado à amostra do texto.

Mais tarde, ainda em 1968, Edward Fry dedicou-se ao estudo de dois factores, o comprimento da frase e a complexidade da palavra. Na sua fórmula, o comprimento da frase foi medido em termos da média do número de frases com passagem em 3 amostras de 100 palavras. A complexidade da palavra foi medida da mesma forma que Flesch, ou seja, através do número médio de sílabas. Os resultados são visualizados através de um grafo com o nome de *Grafo de Legibilidade*. Neste grafo é indicado o nível do texto.

Em 1969, Harry McLaughlin [184] [197] publicou a medida *SMOG Grading*, em que a última letra do nome é um tributo a Robert Gunning criador do *Índice de Fog*. Mclaughlin realizou um estudo em que verificou existir uma correlação negativa quase perfeita entre a quantidade de polissílabos e as suas medições. A medida envolve os quatro passos seguintes para contagem de polissílabos e uma fórmula que permite converter o número de polissílabos obtidos num grau:

1. Contar 10 frases consecutivas no início do texto, 10 no meio e 10 no fim;

2. Em 30 frases seleccionadas contar todas as palavras de três ou mais sílabas;

3. Determinar a raiz quadrada do número de polissílabos;

4. Adicionar 3 ao resultado e obtém-se o grau.

A fórmula SMOG é a seguinte:

$$Grau = \sqrt{X1} + 3 \tag{2.20}$$

em que:
$X1$ é o número de polissílabos.

Foi considerada adequada para avaliação do 4º grau.

Logo no início da década de 70, do séc. XX, Len Mugford [304] desenvolveu um método para prever a legibilidade na forma de um gráfico de legibilidade destinado a crianças entre os 5 anos e meio e os 15 anos. No seu gráfico usou como variáveis o comprimento das palavras medido em sílabas e o comprimento da frase medido em palavras.

Mais tarde, em 1973, um estudo liderado pelas forças armadas norte-americanas (*United States Army*) deu origem à fórmula FORCAST [143, 79, 20]. Esta fórmula foi desenvolvida para avaliação da compreensibilidade dos exames e dos formulários de entrada na armada e baseava-se no cálculo do número de palavras de uma sílaba.

$$Forcast = 20 - (N/10) \tag{2.21}$$

em que:
N é o número de palavras de uma sílaba numa amostra de 150 palavras.

Em 1975, a marinha norte-americana (*United States Navy*) avançou com novo estudo dirigido por John P. Kincaid com o apoio de Fishburne, Rogers, e Chissom [143, 79], tendo como resultado uma nova fórmula de legibilidade. Esta nova fórmula, designada por *Flesch-Kincaid Grade Level Readability Formula* teve por base três fórmulas de legibilidade que foram recalculadas de modo a serem mais adequadas à Marinha. As três fórmulas são a fórmula ARI, o índice de Fog e a fórmula Flesch Reading Ease [143, 79].

A nova fórmula, simplificada, é a seguinte:

$$Grau = (0.4ASL) + (12ASW) - 15 \tag{2.22}$$

em que:
ASL é o valor médio dos comprimentos das frases ($\frac{\text{número de palavras}}{\text{número de frases}}$),
ASW é o valor médio do número de sílabas por palavra ($\frac{\text{número de sílabas}}{\text{número de palavras}}$).

Ainda em 1975, Coleman e Liau desenvolveram uma fórmula de legibilidade designada por Índice Coleman-Liau. Coleman afirmou que criou a fórmula como uma das muitas formas de ajudar o ensino norte-americano a nivelar a legibilidade de todos os livros didácticos para o sistema escolar público. Como outras fórmulas populares de legibilidade, o Índice Coleman-Liau aproxima-se de um nível de grau americano para entender o texto [56].

A fórmula é a seguinte:

$$CLI = 0.0588 * 448L - 0.296S - 15.8 \tag{2.23}$$

em que:
L é o número médio de letras por 100 palavras,
S é o número médio de frases por 100 palavras.

Em 1977, Walter Kintsch propôs uma medida de legibilidade para medir o número de proposições num texto.

Em 1988, devido a controvérsias sobre as fórmulas de legibilidade, os fundadores da MetaMetrix, Inc. publicaram uma nova medida de legibilidade, designada por *Framework Lexile for Reading* [263, 79, 20]. A medida tem por base a teoria linguística e mede a dificuldade dos textos e a capacidade de leitura, permitindo adequar textos aos leitores.

Segundo [263], a comunicação baseia-se na componente semântica e na componente sintáctica. A componente semântica são as palavras e as regras pelas quais as palavras se encontram organizadas é a componente sintáctica. As unidades semânticas variam na familiaridade e as estruturas sintácticas variam na complexidade. Para o autor, a complexidade ou dificuldade de um texto está grandemente relacionada com a familiaridade com as unidades semânticas, e com a complexidade das estruturas sintácticas que são utilizadas nesse texto.

Para os criadores da Lexile, ou se compreende o texto ou não se compreende o texto [79]. Esta medida analisa a compreensão dos textos como um todo - um romance, um artigo de revista, um artigo de jornal - e atribui-lhes uma classificação de leitura que designaram por Lexile (L) [12, p.285]. A escala Lexile varia entre 0L e 1700L, para textos simples (ou leitores iniciados) e textos avançados (leitores avançados), respectivamente.

Em 1995, passados 47 anos, Edgar Dale e Jeanne Chall actualizaram a sua fórmula, onde usaram a fórmula de *Cloze* de Bormuth [13]. É a seguinte:

$$\textit{Dale-Chall Cloze} = 64 - 0.95X1 - 0.69X2 \tag{2.24}$$

em que:
$X1$ é o número de palavras não familiares,
$X2$ é a média do comprimento das frases.

Em 2004, Graesser e os seus colegas criaram a ferramenta Coh-Metrix [103]. É uma ferramenta computacional desenvolvida pela universidade de Memphis que mede a coesão e a dificuldade de textos em vários níveis da linguagem, discurso e análise conceptual. Esta ferramenta foi desenvolvida com o intuito de melhorar a compreensão da leitura na sala de aula fornecendo um meio de melhorar a escrita de livros de texto e tornar os livros de texto mais apropriados aos estudantes [103]. Para a medição da legibilidade considerou importante a coerência e a coesão das relações existentes no texto, o conhecimento prévio do leitor e as características da linguagem e do discurso.

Existem muitas outras fórmulas, no entanto, aqui só se colocaram aquelas que se consideraram mais importantes em virtude de serem as mais referênciadas na literatura consultada.

2.3.2 Resumo das Fórmulas

A tabela seguinte apresenta as fórmulas de legibilidade estudadas.

Tabela 2.3: Fórmulas de Legibilidade.

Ano	**Autores**	**Nome**	**Metodologia**	**Nível**
1923	Bertha A. Lively, Sidney L. Pressey	Primeira fórmula de legibilidade para crianças.	Em cada 1000 palavras é medido o nº palavras diferentes, o nº palavras não existentes na lista de Thorndike e o índice médio do nº palavras encontradas na lista de Thorndike.	Avalia o grau 3 e 4
1928	Mabel Vogel e Carleton Washburne	Fórmula *Winnetka*	Envolvia vários parâmetros: nº palavras diferentes, nº preposições, nº palavras não existentes na lista de Thorndike e o nº frases.	Avalia do grau 1 a 9
1931	W.W. Patty e W. I. Painter	Criaram uma fórmula para medir a dificuldade relativa dos livros de texto baseados na lista de Thorndike.	Determinava o nº palavras diferentes num texto.	Avalia jovens
1934	Ralph Ojeman	Inventou método para avaliar a dificuldade dos materiais.	Consiste em fazer 16 passagens de cerca de 500 palavras tiradas de revistas.	Avalia adultos
1934	Ralph Tyler e Edgar Dale	Criaram fórmula de legibilidade de textos para adultos com dificuldades.	Envolvia os três parâmetros: nº palavras técnicas diferentes; nº palavras não técnicas diferentes e o nº cláusulas indeterminadas.	Avalia adultos (do grau 3 ao 5)
1935	William Gray e Bernice Leary	Livro: "What Makes a Book Readable" e uma fórmula.	Baseado nas palavras difíceis, pronomes, frases preposicionais e no total de palavras do texto.	Avalia adultos
1943	Rudolph Flesch	Publicou a sua primeira fórmula de legibilidade	Utilização duas variáveis: morfemas e referências pessoais (e.g. pronomes e nomes)	Avalia acima grau 7
1944	Irving Lorge	Índice de Lorge	Utiliza o nº palavras e frases.	Avalia jovens e adultos
1948	Rudolph Flesch	Publicou a sua segunda fórmula *Reading Ease Score Formula*	Na primeira parte utiliza duas variáveis: o nº sílabas e o nº frases para cada amostra de 100 palavras. Em seguida, faz a contagem do nº palavras pessoais (pronomes e nomes), frases pessoais (citações, exclamações e frases incompletas).	Avalia todos os graus a partir do 5º
1948	Edgar Dale e Jeanne Chall	Fórmula para adultos e crianças.	Baseia-se no tamanho das palavras, e na percentagem de palavras difíceis.	Avalia do grau 3 ao 5

Continua na página seguinte

Tabela 2.3 – *Continuação da página anterior*

Ano	Autores	Nome	Metodologia	Nível
1952	Gunning	Índice de Fog	Faz a contagem de polissílabos para medir a dificuldade semântica do texto.	Avalia os graus 5 a 8
1953	Taylor	Procedimento *Cloze*	Omissão de partes de texto.	Todos os graus.
1953	George Spache	Fórmula Spache		Avalia até ao grau 3
1958	Powers, Sumner e Kearl	PSK	Utiliza uma amostra de texto com base no comprimento frase e número de sílabas.	Avalia até ao grau 3
1965	Edmund Coleman	Criou 4 fórmulas, usando o procedimento *Cloze* como critério.	Estimam a percentagem de preenchimentos correctos de palavras omissas.	Todos os graus
1969	John Bormuth	*Bormuth Mean Cloze Formula*	Criou uma fórmula com 3 versões diferentes: utilização básica; utilização através de uma máquina; utilização manual.	Todos os graus
1968	Edward Fry	Gráfico de Fry	Criou um teste de legibilidade que utiliza um grafico.	
1969	Harry McLaughlin	SMOG	Consiste na contagem de polissílabos e uma fórmula que permite converter o nº polissílabos obtidos num grau.	Avalia grau 4
1970	Mugford	Gráfico de Mugford		
1973	Estudo comandado pela marinha Norte-Americana (US Navy)	FORCAST	Baseada no cáculo do número de palavras de uma sílaba.	Avalia entre os graus 9 e 10
1975	John Kincaid	Flesch-Kincaid Grade Level	Baseada em três fórmulas para adequar à Marinha.	
1975	Meri Coleman e T.L. Liau	Coleman-Liau Index	Propôs uma medida para calibrar os livros de texto para o sistema de educação público norte-americano.	Aplicado a todos os graus
1988	MetaMetrics, Inc.	Ferramenta Lexile	Mede a dificuldade dos textos e a capacidade de leitura.	Avalia os graus 3 e 4
1995	Dale e Chall	Dale-Chall Cloze	Nova fórmula em que usaram a fórmula *Cloze* de Bormuth.	
2004	Graesser	Coh-Metrix	Mede a coesão e a dificuldade de textos em vários níveis da linguagem, discurso e análise conceptual.	

2.4 Críticas às Fórmulas

As fórmulas de legibilidade foram criadas originalmente para testar o nível de legibilidade dos livros escolares, mas o seu uso estendeu-se à legibilidade dos livros em geral e apesar de muito utilizadas e investigadas, em particular quanto à sua validade, as fórmulas também têm recebido críticas [120, 13]. Uma revisão da literatura em [305] constatou existir um grupo de autores que são a favor das fórmulas de legibilidade e um outro grupo de autores, menos favorável, que considera que as fórmulas de legibilidade têm diversas limitações. Uma parte das críticas refere-se à ausência de análise da compreensão dos textos (ver e.g. [219]). As críticas na sua globalidade não retiram utilidade às fórmulas, que continuam a ser utilizadas, mas permitem reconhecer as suas limitações.

Muitas das fórmulas de legibilidade mais convencionais foram desenvolvidas tendo por base suposições gerais quanto à dificuldade de leitura, nomeadamente, que palavras mais pequenas, frases mais curtas, palavras com poucas sílabas e palavras usadas mais frequentemente serão mais fáceis de ler [20, 120]. As fórmulas também assumem que a legibilidade pode ser avaliada meramente em termos de factores do texto (e.g. [12, 20, 120]) como, por exemplo: o vocabulário, complexidade da frase, percentagem de palavras fáceis, pertencentes a uma lista pré-definida, percentagem de palavras difíceis, que não fazem parte dessa lista, média de palavras por frase, ou a média do número de sílabas por palavra.

Estes atributos tornam um texto, teoricamente, mais ou menos difícil de ler e, de modo a medir a legibilidade de um texto, são usadas fórmulas matemáticas que usam esses factores e assim quantificam a legibilidade num texto [20]. A legibilidade é calculada e o seu resultado é apresentado em termos de uma escala de estimativa. No entanto, existem outros factores a ter em conta na determinação da legibilidade, como por exemplo o interesse do leitor (e.g. [107]), como se procurará mostrar nas secções seguintes.

2.4.1 Validade das Fórmulas

No ensino assume-se que o material do primeiro nível é mais simples que o do segundo nível, que o material do segundo é mais fácil que o do terceiro e assim sucessivamente. Contudo Begeny coloca duas questões [20, p.210]:

- Até que ponto podemos confiar nestas suposições?

- Com que segurança pode ser escolhido o nível apropriado de texto para leitura por um aluno?

Begeny verificou através de um estudo que a resposta era negativa para a maioria das fórmulas que analisou. À partida, o educador poderá escolher o material de ensino de acordo com o resultado obtido pelas fórmulas de legibilidade. No entanto, dado que as fórmulas de legibilidade têm limitações, poderá ser escolhido material menos adequado. Ao mesmo tempo, os professores podem assumir que os livros de texto são adequados a determinados alunos e estes não serem os mais adequados. Apesar de se poder argumentar que componentes de texto usados para avaliar a legibilidade de um texto sejam relevantes, isso não é conclusivo, pois a validade das fórmulas de legibilidade não está comprovada.

Em [20] é apresentado um estudo com o objectivo de identificar, entre um dado conjunto de 9 fórmulas de legibilidade, quais as fórmulas mais válidas na discriminação dos níveis dos textos. Para a análise, os resultados das fórmulas foram comparados com o desempenho da fluência de leitura oral - *Oral Reading Fluency (ORF)*. ORF é considerado um bom indicador da capacidade de leitura, em geral, nos primeiros anos de escolaridade. Cada aluno leu várias passagens de textos e cada leitura produziu um valor. Para cada aluno, um valor mais alto numa passagem corresponderá a maior facilidade na leitura dessa passagem relativamente a outras passagens com valores mais baixos. Para o mesmo aluno, significará que a passagem é mais fácil e, consequentemente, esse valor pode ser relacionado com o nível de legibilidade. Mais concretamente, essas diferenças obtidas nas passagens foram comparadas com os valores obtidos pelas fórmulas.

O estudo também pretendia saber se a validade de uma fórmula depende do nível de capacidade de leitura de um aluno. Como resultado, apenas a fórmula Dale-Chall apresentou valores significativos para as várias comparações em que foi usada, discriminando entre os graus 3 a 5 e, portanto, foi considerada como uma medida válida do nível de dificuldade do texto. As fórmulas Fog, Lexile e Spache mostraram valores significativos mas apenas para uma parte das comparações: a Fog discriminou entre o 5º e o 6º graus; a Lexile entre o 3º e o 4º graus; e a Spache entre os graus 2 e 3.

Como reconhecido em [20], os resultados de aplicação das fórmulas também variam com os estudos, quer devido a diferenças nos métodos de investigação, quer devido a outros aspectos, como por exemplo, o tipo de textos utilizados. Por exemplo, noutro

estudo [261, p.50] as fórmulas de Dale-Chall, Flesch and Lorge foram consideradas medidas válidas para os graus 4, 5 e 6. Nesse estudo, as medidas foram validadas para textos de literatura juvenil. Logo, as fórmulas não se podem considerar universais. De acordo com a investigação realizada por Connatser (e.g.[59]), a dificuldade dos textos é também uma percepção do leitor e por isso não pode ser calculada objectivamente apenas através da contagem de características de texto, como sílabas, tamanho das palavras, ou outras.

Os resultados obtidos com as fórmulas também dependem do contexto em que são usadas. Por exemplo, a marinha norte-americana teve de recalcular 3 fórmulas que pretendia utilizar de modo a serem mais apropriadas para uso militar [143]. As três fórmulas são a fórmula *Automated Readability Index* (ARI), o índice de Fog (*Fog Count*) e a fórmula de Flesch. Como resultado, a fórmula ARI pode ser usada quando o novo material ainda está a ser escrito. A fórmula de Flesch é preferível quando é possível uma contagem automática (*Automated Flesch Count*) e quando o material é classificado segundo graus de dificuldade de leitura. A fórmula de Fog pode ser usada quando não existe equipamento disponível para ajudar a contar.

Portanto, os estudos acerca da validade das fórmulas têm produzido resultados contraditórios e a validade das fórmulas não pode ser considerada conclusiva [20].

2.4.2 Limitações das Fórmulas

A utilização de testes de legibilidade no processo da linguagem é um tópico controverso. Hoje em dia, os resultados são fáceis de obter através da utilização da gramática computadorizada e através de software de verificação.

O aumento do número de fórmulas de legibilidade facilitou de alguma forma a sua utilização para determinar o grau de dificuldade dos textos. Simultaneamente, contribuiu também para que as pessoas as utilizassem como uma forma de modificação de textos com o objectivo de os situar em níveis pré-determinados de legibilidade, uma prática que tem sido criticada por diversos investigadores [20].

Os criadores das fórmulas de legibilidade têm sido acusados de tratar a questão da legibilidade como se se tratasse de um fenómeno relativamente simples e de acordo com o qual o quão fácil um texto é de ler se baseia em geral nos critérios de que é mensurável por uma fórmula estatística e que é redutível a uma pontuação devolvida por essa fórmula (e.g.

[12]). Contudo, como se viu na secção 2.4.1 tem-se verificado que não há uma medida única e simples de legibilidade. O quão fácil é um texto para cada pessoa ler é o resultado da interacção de um número de diferentes factores. É um fenómeno multifacetado, refletindo propriedades de textos, de leitores e da interação entre eles [12].

As fórmulas de legibilidade seguem a ideia, de certo modo intuitiva, de que a dificuldade que um texto apresenta para a sua leitura, parece estar relacionado com o facto de as palavras do texto serem facilmente entendíveis, e de as palavras estarem agrupadas de uma forma que seja fácil de seguir pelo leitor [12].

O grau em que um texto contém, ou não, palavras que não são familiares e/ou que são difíceis de entender refere-se à dificuldade do vocabulário e o grau em que as frases num texto possuem estruturas gramaticais complicadas refere-se à complexidade sintáctica. Tradicionalmente, estes atributos são considerados como tendo um grande impacto na legibilidade [257]. No entanto, trata-se de dois conceitos da linguística relativamente complicados e a sua relação com a leitura não é nada simples, como possa transparecer pelas fórmulas de legibilidade, e que por isso foram analisadas detalhadamente em [12]. Os dois conceitos são explicados de forma mais breve em seguida.

Dificuldade do vocabulário [12, p.286]:

É um critério usado na maioria das fórmulas de legibilidade. É medida pela presença ou ausência de uma palavra numa lista de palavras usadas frequentemente (Thorndike1921, Kucera&Francis1967), e pelo tamanho da palavra, como determinado pela contagem de sílabas (Anderson&Davidson1988).

Segundo [12, 13], as listas de palavras são usadas em várias fórmulas de legibilidade (Dale&Chall1948, Spache1953). A lista de palavras é usada para medir a dificuldade do vocabulário: quanto mais usadas são as palavras, mais familiares são, e consequentemente mais fáceis de entender. Este pressuposto tem dois problemas:

1. a heterogeneidade;

2. a polissemia (certas palavras têm mais do que um significado).

Uma lista de palavras tem de incluir diferentes tipos de palavras, quer palavras familiares, quer palavras difíceis de entender. Para isso é de pressupor que a lista de pala-

vras de uma língua permaneça relativamente estável. No entanto, o vocabulário tende a modificar-se, nomeadamente, nas sociedades em que as mudanças ocorrem rapidamente [12].

Também na sociedade moderna, certas expressões que agora fazem parte do seu quotidiano não existiam anteriormente. Também grupos sócio-culturais diferentes têm vocabulários diferentes. Por último, as listas de palavras não indicam o significado comum, se é familiar, nem o contexto, uma vez que dependendo do contexto a mesma palavra poderá ter significados diferentes [12].

O tamanho da palavra é visto como uma forma de medir a complexidade da palavra. Em muitas fórmulas de legibilidade, quanto maior for a palavra mais difícil será de compreendê-la. Na fórmula de Flesch e na fórmula de Fry, o critério-chave é o número de sílabas por 100 palavras num dado segmento de texto. Para o gráfico de Mugford e para a fórmula de Gunning Fog, o critério-chave é o número de palavras polissilábicas.

A complexidade das palavras evita algumas das limitações encontradas na lista de palavras. Pode-se medir a dificuldade da palavra sem ter de a identificar. No entanto, certas palavras simples, podem levar prefixo e sufixo e continuam a ser palavras simples que se percebem por intuição. Ao mesmo tempo, outras palavras que não sejam compostas, são mais pequenas e contrariamente ao indicado pela fórmula serão mais difíceis de compreender.

Assim, é razoável e concebível que palavras morfologicamente complexas possam realmente ajudar na compreensão fornecendo ao leitor ferramentas que poderão ajudar a desenvolver um ideia plausível quanto ao significado de uma palavra. Isto é possível mesmo não conhecendo o significado da raiz da palavra.

Para além disso, em 1988, Randall realizou experiências sobre a relação entre a compreensão e complexidade morfológica. Testou crianças com idades entre 3 e 7 anos quanto à sua compreensão relativamente a agentes morfologicamente complexos (nomes derivados de verbos). Os resultados sugerem que a compreensão dos agentes complexos não estão directamente relacionados com a extensão da palavra ou a frequência de ocorrência, como seria previsto pelas fórmulas de legibilidade [12].

Em conclusão, as listas de palavras e o tamanho das palavras são critérios questionáveis para a medição da legibilidade.

Complexidade sintáctica:

Grande parte das fórmulas de legibilidade baseiam-se no pressuposto de existir uma correlação entre o comprimento médio das palavras no texto e a dificuldade desse texto, ao considerarem o tamanho da frase na complexidade sintáctica [12].

A forma como determinam a média do comprimento da frase varia, mas como explica Bailin, o maior tamanho de uma frase pode por vezes facilitar a sua compreensão. Por exemplo, através da utilização de conjunções e operadores lógicos que fazem aumentar o tamanho das frases mas podem torná-las mais compreensíveis.

Embora parece intuitivamente plausível que as propriedades sintácticas das frases afectem a compreensão, medir a complexidade sintáctica através do comprimento da frase não é nem um critério útil nem um critério preciso para medir a legibilidade e por isso importa desenvolver outras formas de medir a complexidade sintáctica [12].

Em conclusão, a complexidade sintáctica e a dificuldade do vocabulário são reconhecidos tradicionalmente como tendo um efeito significativo na legibilidade, mas a análise destes dois atributos mostrou que a sua medição não é assim tão simples. Por outro lado, existem outros factores que devem ser considerados e os textos não são apenas fáceis ou difíceis em si, mas dependendo de quem os lê e do contexto dessa mesma leitura [257].

2.4.3 Algumas Vantagens e Desvantagens das Fórmulas

As fórmulas também apresentam vantagens que importa enumerar. Heydari [120] refere um conjunto de vantagens e desvantagens das fórmulas de legibilidade. Como se pode ver em seguida, essas desvantagens têm a ver com as limitações das fórmulas.

Vantagens na utilização das fórmulas de legibilidade, incluem:

1. Por definição, as fórmulas de legibilidade medem o grau de literacia que os leitores devem possuir para ler um dado texto. Os resultados da utilização das fórmulas de legibilidade fornecem ao autor do texto a informação necessária de modo a que o autor consiga captar leitores.

2. As fórmulas de legibilidade não requerem do leitor que leia primeiro o texto para decidir se o texto é muito ou pouco difícil de ler. O objectivo é o leitor saber

antecipadamente se consegue compreender o texto.

3. As fórmulas de legibilidade são fórmulas baseadas em texto, e consideradas geralmente como fáceis de usar.

4. Nos dias de hoje, as fórmulas de legibilidade podem ser executadas por computador, e por isso os processadores de texto podem avaliar o nível de legibilidade do material escrito de forma automática.

5. As fórmulas de legibilidade ajudam os autores a converter material escrito numa linguagem clara.

Desvantagens na utilização das fórmulas de legibilidade:

1. As fórmulas de legibilidade não ajudam muito se se pretende saber em que medida uma audiência irá entender um texto.

2. Uma vez que existem muitas fórmulas de legibilidade, corre-se o risco de se obterem resultados diferentes para o mesmo texto.

3. As fórmulas de legibilidade não podem medir o contexto, o conhecimento prévio, o nível de interesse por parte do leitor, a dificuldade do conceito ou a coerência do texto.

Para concluir, as fórmulas de legibilidade podem ajudar na avaliação da dificuldade de leitura, mas têm sido apontadas diversas limitações e os estudos acerca da validade das fórmulas têm produzido resultados nem sempre coincidentes. Por tudo isto, a validade das fórmulas continua a não ser conclusiva e a sua investigação continua.

Para uma análise mais exaustiva incluindo as limitações e vantagens das fórmulas sugere-se a leitura em [13].

2.5 Leitura e Compreensão

Os educadores e bibliotecários empreendem frequentemente a tarefa de seleccionar o material de leitura para as crianças de uma grande diversidade de aptidões cognitivas e de leitura. É relevante esclarecer como se compreende aquilo que se lê.

A importância da compreensão de material escrito tem sido estabelecida nas áreas da psicologia, da educação e das ciências cognitivas e é considerada essencial para o sucesso das pessoas [257, 136]. Na área da legibilidade de textos, em geral, a compreensão do material lido pode assumir importância crítica. Isso é ilustrado no caso seguinte.

Como referido em [79] dois especialistas em saúde pública notaram que a má compreensão das instruções da instalação de cadeirinhas para bebé nos automóveis levou a que, entre 79 e 94% das cadeirinhas fossem mal colocadas, e que este problema era responsável por grande parte dos acidentes com morte infantil. Também observaram que a utilização correcta das cadeirinhas reduzia os casos de morte em 71% e a hospitalização em 67%. Foram também estudadas as instruções apresentadas e de acordo com os graus de ensino, as instruções revelaram-se difíceis para 80% da população adulta analisada.

2.5.1 Definição de Compreensão

Reduzida à sua forma mais elementar, legibilidade reduz-se a dois termos [24]. A primeira consideração é o leitor: a sua experiência, o seu interesse, os seus sentimentos, a sua motivação, a sua facilidade na utilização da linguagem, as suas necessidades e condições de estudo. Qualquer forma de previsão de legibilidade é válida na medida em que o leitor é tido em conta. A segunda consideração é o nível de interesse, a linguagem, as construções mentais, e as características mecânicas do material de leitura. Todos estes factores aparecem como sendo muito relacionados, no sentido de, intimamente relacionados.

No entanto, a legibilidade deve ter em atenção outros elementos para além do leitor e do texto. Interessa explicitar a actividade de leitura em si e o contexto onde ocorre. Por outro lado, no contexto do presente trabalho, leitura e compreensão são indissociáveis, pelo que importa definir o que se entende por compreensão do material lido, ou simplesmente da leitura.

Uma definição possível pode ser encontrada em [257, p.11]:

Definição 3 (Compreensão da Leitura)**.** *É o processo de, simultaneamente, extracção e construção de significado através da interacção e envolvimento com a linguagem escrita.*

Esta, envolve três elementos, os quais ocorrem num determinado contexto sócio-cultural:

O leitor Inclui todas as suas aptidões, competências, conhecimento e experiência no acto da leitura;

O texto Pode ser em qualquer formato, electrónico ou impresso;

A actividade de leitura Esta actividade inclui o propósito, o processo e as consequências associadas ao acto de leitura.

A figura 2.1 representa o modelo de compreensão segundo a definição anterior. O modelo foi adaptado de [257, p.12] e [202, p.272].

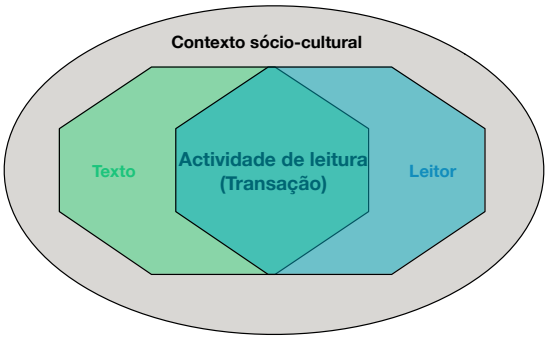

Figura 2.1: Modelo de Compreensão.

Apresenta-se em seguida uma definição diferente de compreensão, elaborada sob uma perspectiva cognitiva [185, p.17]:

Definição 4 (Compreensão)**.** *É a tradução do código escrito em unidades de linguagem com significado que são combinadas para construir uma representação mental coerente do texto. Trata-se de um processo dinâmico que envolve o indivíduo, o texto, a leitura e o contexto sócio-económico.*

Em qualquer das definições, torna-se claro que compreensão, tal como a leitura é um processo e que este processo envolve funções cognitivas do leitor.

2.5.2 Psicologia Cognitiva

A psicologia cognitiva tem sido usada pelo menos desde os anos 70 do séc. XX, no domínio da compreensão do software como modelo explicativo do comportamento dos profissionais de informática (e.g. [248]) e em geral, a sua adequação ao estudo do desempenho individual na manutenção do software (e.g. [140]). No presente caso será do

informático que desenvolve e mantém o software. O modelo cognitivo permite descrever as várias tarefas mentais, como a leitura, ou a escrita de um programa, através de processos cognitivos.

Definição 5 (Psicologia Cognitiva)**.** *É o estudo da forma como o cérebro processa a informação [109, p.3]. Inclui os processos mentais envolvidos na percepção, aprendizagem e armazenamento em memória, pensamento e linguagem.*

Os processos aqui indicados são os envolvidos na utilização e na aquisição do conhecimento [248, p.46] e incluem estruturas representativas de áreas onde se realiza o processamento e o armazenamento da informação. Essas áreas são vistas como áreas de memória.

O estudo da memória por métodos científicos data dos anos 80 do século XIX [226]. O primeiro laboratório de psicologia terá sido montado em 1879, o primeiro trabalho experimental terá sido publicado em 1885 e a teoria com a distinção entre memória de curto prazo e longo prazo foi publicada num livro de 1890 [109]. O desenvolvimento da psicologia cognitiva foi condicionado fortemente pela influência da psicologia comportamental [109, 226]. Foi apenas nos anos 60 do século XX que foram conduzidas experiências que permitiram que a psicologia cognitiva se afirmasse [226]. Essas experiências apontaram para a existência de dois sistemas de armazenamento, consoante se tratava de recordar material memorizado há mais ou menos tempo. No entanto, essa visão tem vindo a aperfeiçoar-se de alguma forma.

Actualmente a psicologia cognitiva conta com avanços tecnológicos, como o da imagiologia médica, para o seu desenvolvimento. Com a ajuda da imagiologia tornou-se claro que certas regiões do cérebro são mais especializadas para umas tarefas do que para outras. A sua utilização já ocorre mesmo na área da informática. Por exemplo, em [253] é descrito um experimento[4] envolvendo 17 participantes em que lhes foram fornecidos pequenos extractos de código e observados através da resssonância magnética enquanto tentavam compreender o código para localização de erros de sintaxe. Como resultado, foi detectada a activação de diferentes regiões do cérebro.

[4]Do latim *experimentum*; Não será usada a palavra experiência em virtude de ter um sentido mais lato. Neste documento será usado o termo estudo experimental.

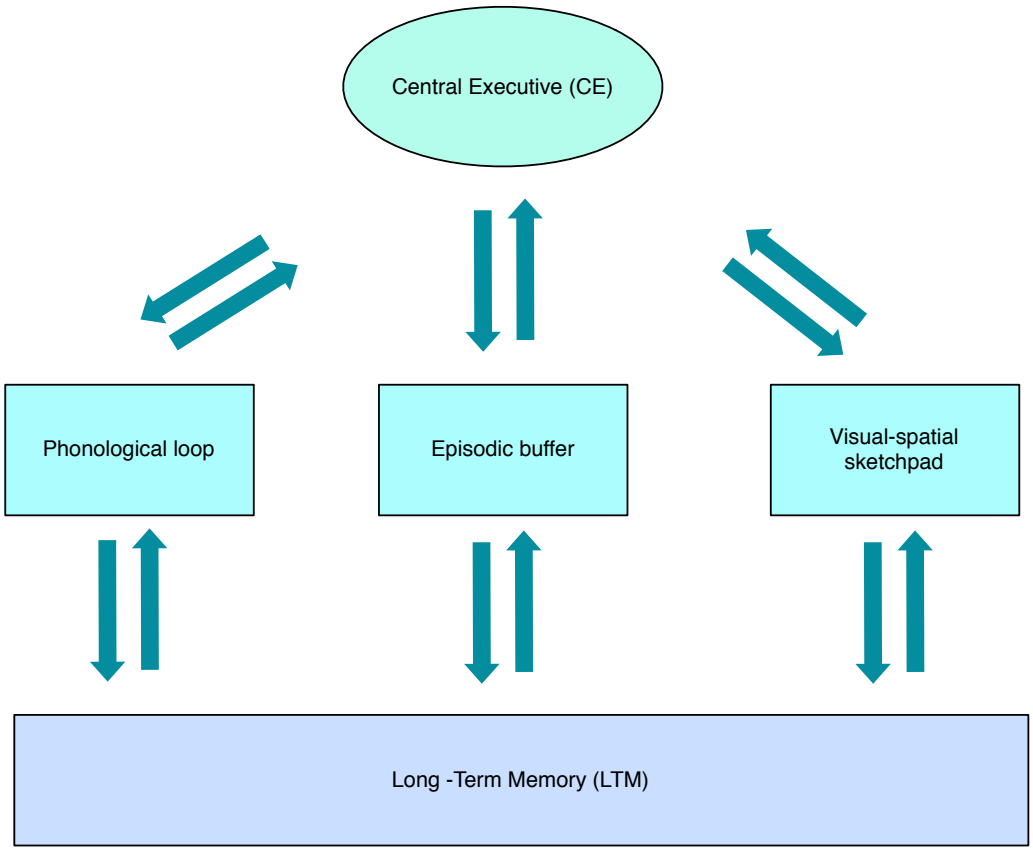

Figura 2.2: Memória de Trabalho (WM). Retirado de [109].

As pessoas apresentam entre si uma grande diversidade de aptidões cognitivas e de leitura [12]. A compreensão da leitura depende da execução e integração de vários processos cognitivos (ver e.g. [136, 282]). O modelo referido actualmente para descrição dos processos cognitivos baseia-se no conceito de memória de trabalho (*Working Memory (WM)*). Este modelo sucede a uma versão anterior com duas estruturas de memória, ou "armazéns", a memória de curto prazo (*short-term memory (STM)*) e a memória de longo prazo (*long-term memory (LTM)*). Esta evolução corresponde a ter-se a STM não como um simples "armazém", mas como uma memória onde é realizado trabalho. A WM é representada na figura 2.2 e inclui duas estruturas de armazenamento, ou "armazéns", um visual e outro auditivo.

Os componentes da WM são definidos em seguida [109][5]:

[5]Para uma apresentação detalhada desta teoria e outras anteriores e ainda de alguns problemas que lhe têm sido levantados ver e.g. [109]. Um dos problemas relaciona-se com a possibilidade de sobreposição das estruturas de armazenamento.

Phonogical loop corresponde a armazenamento breve de sons. Desempenha um papel muito importante no desenvolvimento e uso da linguagem.

Visual-spatial sketchpad corresponde ao armazenamento de imagens. Está envolvido no reconhecimento de padrões, na percepção e controlo do movimento.

Central executive (CE) corresponde à estrutura responsável pelo controlo global do processamento cognitivo. Controla os ciclos fonológico e visual e usa-os para o processamento. É o principal elemento na tomada de consciência. Participa nas capacidades de tomada de decisão, planeamento e resolução de problemas.

Episodic buffer integra informação de diferentes fontes sensoriais e também liga CE com a LTM. Retém pedaços de informação durante um período curto e julga-se que permite ligar entradas e características distintas. Julga-se que este componente também poderá estar envolvido no processo de tomada de consciência (*conscious awarness*), tal como o CE.

A capacidade das áreas de armazenamento fonológico e visual da WM pode ser medida aproximadamente pelo número de dígitos falados, ou objectos apresentados, respectivamente. O desempenho da WM é considerado um bom preditor de outros tipos de desempenho cognitivo, como a compreensão da linguagem ou o desempenho académico. A WM varia com a idade, tendo o seu valor mais alto por volta dos 20 anos, começando a declinar lentamente a partir daí.

A leitura é um processo em grande parte automático (pelo menos para os leitores praticados). Os processos cognitivos tornam-se automáticos, como resultado da prática frequente, como por exemplo, as habilidades envolvidas em palavras de leitura. No entanto, temos a capacidade de substituir essas sequências automáticas quando precisamos [109].

A distinção entre o processamento controlado e processamento automático tem sido útil em várias áreas da psicologia cognitiva. Um exemplo dos diferentes processamentos é quando se vê um rosto familiar. Quando se encontra alguém que se conheceu anteriormente, instantânea e automaticamente reconhece-se o rosto como sendo familiar, mas lembrar onde e quando se encontrou com essa pessoa exige um esforço consciente (Mandler, 1980), como se pode ver em [109].

Assim, a tarefa de leitura de código ficará facilitada quando as instruções já são conhecidas do programador. Pode-se colocar a questão de saber qual a dimensão do código que pode ser reconhecido automaticamente.

Para compreender uma frase, deve-se processar visualmente cada palavra, identificar e aceder às suas representações fonológica, ortográfica e semântica e ligar estas representações de modo a formar uma compreensão do significado da frase [136]. A partir daqui o leitor processa e liga ideias individuais na criação de uma representação mental do texto. A percepção visual de um objecto ou forma pode ser dividida em duas fases, a da sua extracção da cena visual e a do seu reconhecimento [6, p.28].

O processo de leitura pode ser esquematizado como mostrado no diagrama de actividades na figura 2.3. No diagrama pode-se ver a existência de algumas notas explicativas. Entre alguns processos a comunicação é bidireccional. As linhas a traço interrompido representam a comunicação no sentido inverso.

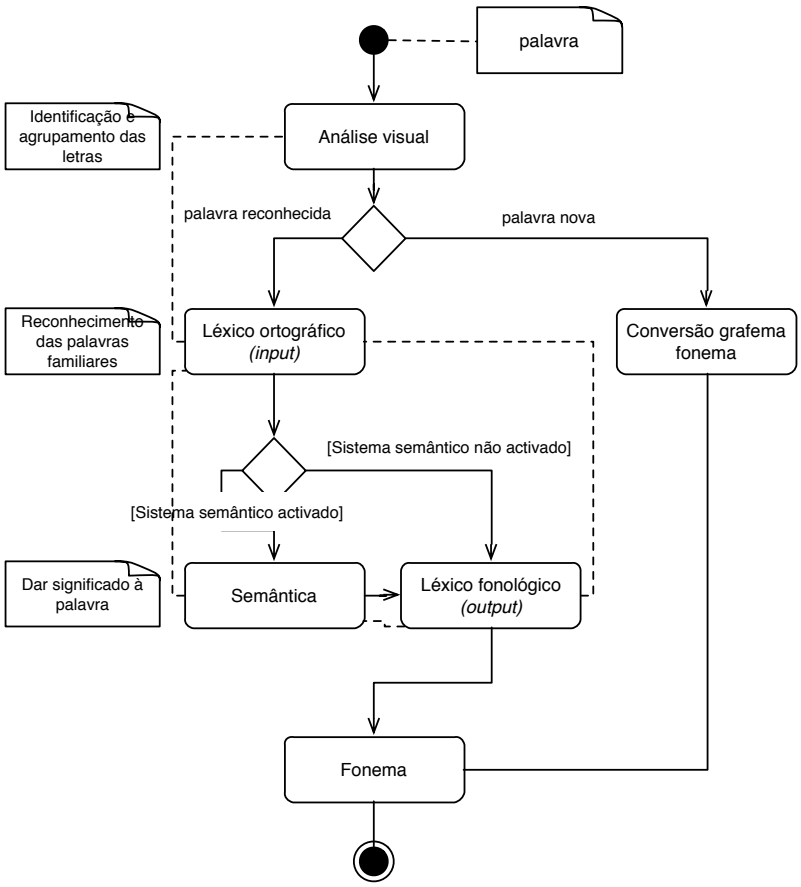

Figura 2.3: Modelo Cognitivo da Leitura. Adaptado de [100].

A leitura e a compreensão podem ser realizadas segundo um processo *top-down*, ou *bottom-up*, ou em ambos os sentidos, isto é, a partir do código, ou partindo-se de uma ideia prévia global do texto (código), ou em ambos os sentidos, o que permitirá que a compreensão vá ocorrendo nas duas direcções. Para uma apresentação de vários modelos de entendimento de programas, tanto o *top-down*, como o *bottom-up*, ver e.g. [284].

Como se compreende, no caso da leitura de código-fonte haverá, em geral, informação inicial acerca da finalidade do código. Nos casos em que o software seja desconhecido para o programador, então uma explicação inicial da sua finalidade global e de cada parte seria útil. Ou seja, o modelo cognitivo justifica a existência de comentários no código. Também em informática e para sistemas orientados a objectos foi mostrado, através de um estudo experimental, ser vantajoso obter primeiro uma ideia do funcionamento do programa antes de se proceder a alterações [134]. No estudo, os participantes não estavam familiarizados com o programa e a familiarização, pela estratégia sistemática, foi feita lendo a documentação, o código e executando a aplicação informática.

Mais, um estudo experimental com 38 alunos dos 3º e 4º anos de informática dos cursos de Ciências da Computação da Universidade de Oslo e da Oslo University College na Noruega, mostrou que os participantes que seguiam uma estratégia sistemática na compreensão do código com que não estavam familiarizados tinham maior probabilidade de produzir alterações correctas [134].

O significado de uma frase é definido em termos de estrutura da frase e, mesmo que os ouvintes tentem extrair tudo o que podem a partir de cada palavra, só serão capazes de colocar as "peças" no lugar quando chegar ao final da frase. Assim, por vezes, as pessoas necessitam de tempo extra após terminar a frase de modo a concluirem este processo. As pessoas têm que manter uma representação da frase na memória porque a sua interpretação pode estar errada, e com isto precisarem de a reinterpretar desde o início. Justa e Carpenter [6] em 1980 realizaram um estudo de tempos de leitura, e descobriram que os participantes tendem a gastar o tempo extra no fim de cada frase, de modo a conseguirem extrair o significado transmitido pela frase.

Com isto introduz-se o princípio da Interpretação Imediata (*Principle of Immediacy of Interpretation*)[6][p.317]:

Definição 6 (Princípio da Interpretação Imediata)**.** *As pessoas tentam extrair o signifi-*

cado de cada palavra quando a recebem e não esperam até ao final da frase.

Alguns autores defendem que a leitura requer processamento letra a letra, enquanto que outros autores defendem que o processamento requer a palavra como um todo. Para a maioria, o consenso recai sobre o processamento da palavra como um todo.

O número de letras duma palavra (comprimento ortográfico) numa palavra escrita afecta o processamento e reconhecimento das palavras, sendo o efeito mais acentuado com "não palavras" do que com palavras. Também as palavras mais irregulares demoram mais tempo a ser soletradas e consequentemente demora-se mais tempo a compreendê-las. A capacidade de ler palavras é afectada fortemente pela apresentação de outros objectos visuais, mas não pela fala [109].

A influência de factores linguísticos nos padrões oculares durante a leitura também tem sido bastante estudada. Sabe-se hoje que factores como o contexto em que a palavra se insere, a frequência dessa palavra, a sua extensão ou a sua estrutura fonológica e morfológica influenciam o local da primeira fixação, a existência ou não de refixações e a duração da fixação, ou fixações, na palavra. Por exemplo, a duração da fixação de determinada palavra pode ser influenciada pelo contexto em que uma palavra se insere. Quando uma palavra é previsível em determinado contexto, o seu tempo de fixação é menor do que num contexto em que é pouco previsível. Por outro lado, uma palavra pouco frequente é quase sempre fixada e normalmente durante mais tempo do que uma palavra frequente, que pode até nem ser fixada.

Nem todas as palavras são fixadas. Geralmente, as palavras não lexicais ou funcionais, sobretudo as mais curtas, não são fixadas, o que não significa que não sejam processadas. Como durante uma fixação extraímos informação da região à volta do ponto de fixação, recolhemos informação para além da palavra fixada e ainda tomamos a decisão de onde fixar a seguir.

Ao mesmo tempo, para compreender um texto como um todo, o leitor necessita de processar as unidades individuais de ideias resultando numa representação mental coerente do texto [136]. Para o sucesso destes processos, contribuem diversos aspectos, tais como, as características do leitor, as propriedades do texto e a exigência da tarefa de leitura em si.

Anderson afirma [6, p.321] que as pessoas combinam informações sintácticas e a semântica das palavras para interpretar uma frase. O peso de cada um destes tipos de informação não é o mesmo para todas as línguas. Ao nível sintáctico, no processo de uma pessoa determinar o sentido de uma frase são consideradas a ordem das palavras e a flexão de cada palavra (i*nflectional structure*).

A relatividade do peso de cada tipo de informação levanta a questão de saber qual a importância de cada uma dessas duas pistas na leitura de uma instrução de código-fonte. No entanto, no caso do software, grande parte dos aspectos sintácticos são resolvidos pelo compilador e possivelmente por actividades de validação e verificação do software. Portanto, na leitura de código-fonte que tenha passado por essas actividades, não deverão existir problemas sintácticos significativos e a semântica dos nomes dos identificadores será o principal factor a condicionar a compreensão da instrução. Seja qual for o caso, pela importância da semântica conclui-se da importância dos nomes.

2.5.3 Processos de Compreensão

Existem diversos modelos que descrevem os processos cognitivos e linguísticos envolvidos na compreensão da leitura [282, 115], cada um realçando diferentes aspectos. Por exemplo, o modelo CC-R (*cognitive-components-resource model*) [115] pretende explicar o desempenho na leitura através das relações entre processos ao nível da palavra, processos de mais alto nível e memória de trabalho[6].

Seja como for, e como referido, os vários modelos têm como noção central a construção de uma representação mental do texto que inclui informação do texto e conhecimentos prévios do leitor interligados através de relações semânticas [136, p.10].

Estas relações resultam de processos de inferência passivos (também chamados automáticos [109]) e estratégicos. Os processos passivos ocorrem automaticamente sem controlo do leitor, enquanto os estratégicos, iniciados pelo leitor, requerem atenção e uso de memória de trabalho [136, p.10]. Ou seja, ocorrem na parte da memória de curto prazo relacionada com processamento perceptivo e linguístico consciente e imediato.

[6]Propõe um conjunto de relações entre (a) os processos de nível inferior que descodificam palavras, (b) os processos de nível superior que extraem informação explícita e implícita de texto e integram informação baseada no texto com conhecimento prévio, e (c) os recursos cognitivos limitados que são partilhados por muitos processos (i.e., a WM).

Definição 7 (Sensação e percepção)**.** *[109, p.36]: Sensação é considerado como a entrada bottom-up no estado natural, "em bruto", a partir dos sentidos e a percepção é considerado como sendo o resultado final do processamento desse material sensorial no sistema visual.*

A sensação e a percepção, encontram-se assim, em extremidades opostas do processo visual.

As inferências permitem ligar ideias do texto, ou ligar as ideias do texto com o conhecimento que a pessoa possui. A representação mental resultante pode ser, por conveniência de análise, separada em dois componentes [144], designados, respectivamente, por texto base (*text base*) e modelo situacional (*situation model*). O primeiro conterá os elementos e relações obtidos apenas com base no texto e que resultará em geral numa compreensão relativamente pobre e incoerente do texto. O segundo envolverá elementos e relações do conhecimento e experiência prévia do leitor que permitirão tornar a estrutura anterior coerente, completá-la, interpretá-la e integrá-la com esse conhecimento anterior do leitor. O modelo situacional persiste mais tempo na memória [6]. Um modelo situacional consiste numa representação global da estrutura da narrativa e contém os seus elementos críticos [6].

No caso do software, o conhecimento do domínio de aplicação deverá favorecer este processo e a compreensão do código-fonte. No caso do código, tal conhecimento estará contido nos nomes e comentários existentes.

As inferências do segundo tipo são apenas necessárias quando a mensagem não pode ser entendida de forma coerente sem conhecimento anterior do leitor. Portanto, a compreensão da mensagem de um texto depende das inferências. As inferências permitem extrair informação de um texto para além do que está expresso de forma explícita nesse texto. Dependendo dos leitores e das suas exigências de compreensão, uns leitores geram mais inferências do que outros [54]. Resumindo, a construção da rede semântica é um processo e envolve diversos (sub)processos. Uma descrição detalhada pode ser encontrada em [282].

A facilidade com que os modelos situacionais são integrados depende, de como os muitos aspectos das dimensões são partilhados entre o conhecimento prévio e as iteracções do modelo situacional que se desenvolve assim que o texto começa a ser lido. A informação congruente entre as dimensões resulta numa actualização mais rápida num modelo

situacional do que a informação incongruente. A informação incongruente ou de conflito retarda ou mesmo interrompe o processo de integração.

Os leitores resolvem as incongruências fazendo um ou mais dos seguintes procedimentos: esclarecer novas informações, revisitando informações antigas, identificando semelhanças, e suprimindo informações desactualizadas. No entanto, os leitores também podem proceder à leitura sem tentar resolver qualquer incongruência, na situação em que a integridade do modelo situacional torna-se mais fraco conforme o processo progride.

As inferências e os modelos fazem parte dos processos de compreensão, isto é, da construção de significado. Tratam-se de processos de alto nível. O processo de identificação das palavras envolve essencialmente 3 subprocessos de mais baixo nível: representação da palavra, ortográfico e fonológico. Estes interagem com os processos de mais alto nível para resolver qualquer ambiguidade.

Os processos cognitivos da compreensão da leitura dividem-se em duas categorias [136, p.11]:

Processos de baixo nível a tradução do "código" escrito em unidades da linguagem com significado. Tem a ver com a codificação, a fluência da leitura e o conhecimento do vocabulário;

Processos de alto nível combina as unidades numa representação mental coerente e com significado. Estes têm a ver com a realização de inferências, processos de função executiva e aptidões de atenção-alocação.

A descodificação (processo de identificação da palavra escrita) afecta a compreensão, porque se ocorre de forma lenta prejudica o automatismo da leitura e consequentemente prejudica a compreensão ao consumir memória de trabalho necessária à compreensão.

Ambos os processos começam a desenvolver-se antes de a criança começar a aprender a ler e são indicadores da aptidão para a compreensão da leitura em idades posteriores. Os processos vão-se automatizando ao longo do tempo sendo a dos de alto nível mais lenta mas com desenvolvimentos importantes até à idade adulta. Portanto, existem diferenças nos processos entre crianças e adultos. O desenvolvimento destes processos varia de pessoa para pessoa o que faz com que estas venham a apresentar capacidades distintas de compreensão dos textos. Perceber as dificuldades de compreensão de leitura de cada

indivíduo é importante, mas sai fora do âmbito deste trabalho que se centra na obtenção de código-fonte legível e não no desenvolvimento da capacidade de compreensão do aluno.

Existem estudos que mostram existir impacto das diferenças cognitivas individuais na compreensão da leitura, sendo maior entre leitores experientes [276].

O processo de ligação entre a informação anterior do leitor e a nova informação pode ser explicada por uma teoria conhecida por teoria *schema* [202, p.273]. A teoria *schema* envolve o armazenamento de vários tipos de informação na memória de longo prazo. Na teoria *schema*, os indivíduos organizam o conhecimento em categorias e sistemas que tornam a pesquisa mais fácil. Ou seja, é como se armazenassem esquemas na sua memória de longo prazo. Quando é encontrada uma palavra-chave ou um conceito, os leitores são capazes de aceder a esse sistema de informação, extraindo as ideias que os irão ajudar a fazer ligações com o texto e criando assim significado [202, p.273]. Novos modelos situacionais irão conduzir à actualização dos esquemas relacionados com esses modelos.

A quantidade e profundidade do conhecimento varia de indivíduo para indivíduo e depende das características individuais de cada um. Cada leitor é único e possui as suas competências, conhecimento, desenvolvimento cognitivo, cultural, e tem um propósito individual ao pretender ler um determinado texto (citando Narvaez2002)[202, p.2].

Em termos de competências podem-se considerar as competências básicas com a sua língua, as competências de descodificação e as competências de pensamento a um nível mais elevado [202]. O conhecimento inclui conhecimento prévio sobre o conteúdo do texto e relaciona-se com o *schema* disponível que o leitor possua para um texto em particular.

Em testes de língua inglesa para estrangeiros, um maior conhecimento do vocabulário, verificou-se ser um factor que ajudava a explicar melhores desempenhos na compreensão da leitura nessa língua [252]. Portanto, será de supôr que o mesmo deverá acontecer com as linguagens de programação.

A compreensão é afectada pela cultura do leitor, na medida em que a deste coincide com a cultura do escritor ou com a cultura exposta no texto. Se o leitor não tiver o mesmo nível cultural do escritor terá dificuldade em compreender o texto. A leitura também depende do propósito pelo qual se lê. Para além disso, também a motivação influencia a compreensão, pois pode influenciar o interesse, o propósito da leitura, a emoção e até mesmo a persistência na leitura do texto [202].

Os leitores mais motivados aplicam mais estratégias e trabalham mais no sentido de conseguir construir significado, enquanto que os leitores menos motivados não trabalham o suficiente e o significado criado não é suficientemente forte de modo a motivá-los.

O processo da compreensão também se altera ao longo do tempo, quer pela maturidade quer pelo desenvolvimento cognitivo do leitor, consoante aumenta a sua experiência e formação [257, 79].

Estudos recentes demonstraram que o papel dos processos *bottom-up* para determinar a competência de compreensão da leitura continua a ser de importância vital para os leitores altamente experientes. A interpretação defendida aqui é que a identificação eficaz da letra contribui para o reconhecimento de palavras de um modo mais eficiente, o que por sua vez permite a compreensão de textos mais eficaz [124]. Assim, os problemas da eficiência da memória de curto prazo pode ter consequências negativas para a aprendizagem a longo prazo.

A perspectiva cognitiva traz consigo outras perspectivas e várias implicações [136, p.13], algumas das quais se apresentam seguidamente. Os processos de alto nível são usados não só na leitura, mas também na audição de um texto, ou na apresentação visual da narrativa do texto. Isto implica que um aluno pode treinar esses processos sem recorrer unicamente à leitura, mas através desses outros meios. A segunda implicação tem a ver com o facto de o conhecimento prévio ser essencial para realizar inferências e que estas sejam correctas. Nesse caso, um exemplo de uma implicação, é a seguinte: se um aluno tem dificuldades de compreensão de leitura, então a introdução de novos conhecimentos que tenham sucesso deverá ser feita de forma gradual e faseada.

2.6 Factores de Legibilidade

2.6.1 O Texto

Os factores associados a cada um dos elementos do modelo na figura 2.4, texto, leitor, contexto e actividade, contribuem para variações na compreensão da leitura. O texto é o elemento mais relevante para este trabalho.

Kintsch e Rawson [145], em 2005, descrevem três níveis de elaboração no texto. O primeiro é um nível linguístico, no qual palavras e frases do texto são identificadas. Esse

nível envolve o reconhecimento de palavras, assim como a análise das palavras e as suas funções nas frases. O segundo nível é o de análise semântica, em que o significado de palavras deve ser combinado em proposições, que são interligadas numa rede chamada de microestrutura.

A microestrutura é constituída por proposições, formadas através das palavras do texto e da sua organização sintáctica, e das relações de coerência entre essas proposições. A microestrutura é também organizada em estruturas de ordem superior, de forma que não apenas proposições, mas também secções maiores de um texto, são ligadas semanticamente. Essas ligações envolvem o reconhecimento de tópicos globais e de suas inter-relações, formando uma estrutura global do texto chamada macroestrutura, que representa o essencial do conteúdo do texto. A microestrutura e a macroestrutura juntas são designadas por texto base.

O texto base permite apenas uma compreensão superficial do texto, enquanto o modelo mental da situação permite a compreensão profunda do texto [145], [200]. A compreensão profunda do texto permite ao leitor a aplicação do que compreendeu, tal não acontece com a superficial.

Em 1935, William S. Gray e Bernice Leary publicaram um trabalho que foi um marco na investigação de leitura, "What makes a Book Readable" com a tentativa de descobrir o que faz um livro ser legível para adultos com capacidades de leitura limitadas. Incluíram no seu estudo 48 selecções de cerca de 100 palavras cada com origem em diferentes fontes, sendo metade delas com origem em ficção, a partir de livros, revistas e jornais. Estabeleceram a dificuldade destas selecções por um teste de leitura-compreensão a 800 adultos.

Os autores identificaram primeiro 228 elementos que afectam a legibilidade e agruparam-nos sob quatro títulos (ver figura 2.4):

- Conteúdo;
- Estilo;
- Formato;
- Características de organização.

Os autores consideraram que o conteúdo, próximo do estilo, era o mais importante. Em terceiro lugar viria o formato, praticamente igual às características da organização que corresponde à aparência do texto. Estas características são capítulos, secções, títulos e os parágrafos que mostram a organização das ideias. Também descobriram que não podiam medir o conteúdo, o formato ou a organização estatisticamente.

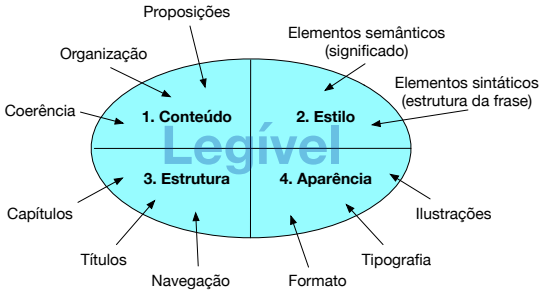

Figura 2.4: Elementos Básicos da Legibilidade.

Podem-se considerar diversos componentes do texto e dimensões em que poderão variar [257, pp.25, 94]:

- Género de discurso (e.g. narrativo, expositivo);
- Estrutura do discurso, que inclui composição retórica e coerência;
- Meio utilizado;
- Dificuldade das frases, que inclui vocabulário, sintaxe e base proposicional do texto;
- Conteúdo (incluindo tipos de modelos mentais, culturas, estrato sócio-económico); faixa etária apropriada para o assunto, e práticas específicas da cultura;
- Textos com vários níveis de envolvimento para determinadas classes de leitores.

Existem diversos factores ao nível do texto que podem afectar a legibilidade e, consequentemente, as fórmulas de legibilidade. Nesta secção listam-se diversos factores retirados de artigos.

Baseados em estudos, especialistas compilaram um conjunto de regras, designadas de ouro, de escrita de documentação que se aplicam independentemente do meio de escrita [79, p.2]:

- Usar palavras familiares, simples e curtas;
- Evitar calão, gíria;
- Usar uma linguagem culturalmente neutra e de género neutra;
- Usar gramática, pontuação e ortografia correctamente;
- Frases simples, voz activa, e no presente do indicativo;
- Começar as instruções no modo imperativo por frases que começam com um verbo de acção;
- Usar elementos gráficos simples, tais como listas com marcadores e passos numerados para tornar a informação visualmente acessível.

Em seguida, apresenta-se uma lista de factores linguísticos apresentados em [12] como factores adicionais a serem tidos em conta na elaboração de medidas de legibilidade.

Gramática: As fórmulas de legibilidade tendem a ver a gramática em termos da questão estilística de complexidade. No entanto, erros ou desvios da gramática padrão podem aumentar a dificuldade de o compreender. Em [12, p.293] são apresentados alguns exemplos.

Estilo: Mesmo que um texto seja gramaticalmente perfeito, certas propriedades estilísticas podem torná-lo mais difícil de compreender para umas pessoas do que para outras. No estilo, incluem-se factores sintácticos, como o número de orações numa frase e o tipo de orações.

O acto da leitura é um processo de compreensão da linguagem escrita, que envolve descodificar símbolos escritos e atribuir-lhes um significado, assim como ouvir a língua falada envolve o descodificar de sinais auditivos e atribuir-lhes significado. Como a leitura é um processo linguístico, a forma como os seres humanos processam a linguagem tem um efeito sobre a compreensão de textos. Por exemplo, [12, p.294] as orações relativas são mais fáceis de processar quando o pronome relativo está claramente presente do que quando não está; frases do tipo *garden-path* que obrigam o leitor a reavaliar a frase; e frases com estruturas de ramificação à esquerda que

se tornam mais difíceis de compreender (em língua Inglesa). As frases *garden-path* são frases em que surge uma ambiguidade num dado ponto da frase o que contraria a expectativa do leitor e, que, por isso, obrigam o leitor a voltar atrás e a reavaliar a frase. Em linguística, ramificação tem a ver com o *parsing* da frase e criação da árvore de *parse*. Se o núcleo da frase estiver no final da frase a ramificação será à esquerda.

A estrutura sintáctica das frases não é a única propriedade estilística de um texto que pode afectar a sua legibilidade. O uso de linguagem figurada é um recurso estilístico não sintáctico que pode representar desafios para os leitores, porque compreender linguagem figurada requer mais do que simplesmente compreender o significado literal do texto. Por exemplo, para compreender a ironia, o leitor deve reconhecer que o escritor está a tentar transmitir o sentido oposto ao (literal) apresentado no texto. A conotação (sentido linguístico) e metáfora fazem muitas vezes referência ao conhecimento que não está contido num texto. Para resolver o problema, ou os leitores possuem o conhecimento que é assumido no texto, mas não está explícito, ou têm de formar hipóteses que lhes permitam interpretar o texto [12, p.295].

Nível de Conhecimento: [12] Existe evidência empírica que mostra que o nível de conhecimento do leitor tem impacto na capacidade que um leitor tem de recordar e realizar inferências. Esse conhecimento auxilia também os leitores na tarefa de leitura e compreensão de textos. Assim sendo, Bailin [12] recomenda que esse impacto deva ser investigado atendendo também às necessidades dos leitores de diferentes ambientes sócio-culturais. Um material que é fácil de ler por um grupo poderá não ser tão fácil para outro.

Coerência textual: Os desenvolvedores de fórmulas de legibilidade limitaram-se a trabalhar por frase, isto é, quando avaliam um texto, ele é avaliado através da combinação das várias frases. No entanto, a leitura de um texto não é redutível a uma simples função, por exemplo, do tamanho médio das suas palavras e do número médio de palavras nas suas frases. A sua legibilidade é antes uma função das ligações e inter-relações entre as frases de um texto, ou seja, através [12, p.297] das suas relações anafóricas; e das relações lógicas e sequenciais entre as frases. Este assunto foi abordado na perspectiva cognitiva na secção 2.5.2.

Correcção: Escrever é um acto de comunicação, e comunicação, seja oral ou escrita, é um acto de negociação entre o orador/escritor e o ouvinte/leitor. É uma negociação em que ambas as partes participam para ter sucesso. Embora existam muitos momentos em que a negociação poderia falhar, há inúmeros exemplos na literatura de formas em que as pessoas tentam reparar os problemas que ocorrem inevitavelmente no acto de comunicação, incluindo na leitura [12].

Também parece óbvio que nem todas as falhas na comunicação serão iguais e, consequentemente, não será possível recuperar com a mesma facilidade de todas elas e, eventualmente, de algumas nem será possível recuperar.

Interacção de factores: Os factores que acabaram de ser apresentados não são necessariamente independentes uns dos outros. Cada factor pode interagir com outros para produzir efeitos que não podem ser produzidos de forma independente. Alguns estudos indicam que o conhecimento linguístico profundo parece afectar a capacidade de entender as estruturas de ramificação à esquerda. Para os leitores jovens, estas estruturas parecem ser mais difíceis de processar do que as estruturas de ramificação à direita. Na verdade, o efeito discutido não deve ser considerado simplesmente uma questão de estilo, mas sim um efeito da interacção de estilo e conhecimento linguístico.

Para concluir, são necessários mais estudos para investigar como vários factores interagem para tornar um texto mais, ou menos, legível [12].

Coesão: É uma característica do texto. É uma propriedade objectiva da linguagem explícita, ou seja, são as características explícitas do texto tais como, palavras, frases, ou as frases que guiam o leitor na interpretação das ideias substantivas (presentes) no texto, na ligação entre as ideias presentes, e na ligação entre as ideias e o nível superior, como por exemplo, o tema. Deste modo, o leitor forma uma representação coerente.

Coerência: As relações de coerência são construídas na mente do leitor e depende das competências e conhecimento que o leitor tem da situação. Se o leitor tem conhecimento adequado sobre a matéria e se são utilizados elementos linguísticos e de discurso adequados então o leitor forma uma representação mental coerente do

texto.

2.6.2 O Leitor

As fórmulas de legibilidade medem apenas as propriedades de textos, não levando em consideração as características dos leitores, mas, no entanto, há evidência empírica que sugere que as características dos leitores têm muito a ver com a sua capacidade de compreender um texto, e que a leitura é uma função da interacção entre as propriedades dos textos e as características dos leitores [12]. Apesar de a preocupação no presente trabalho ser com o texto, não se quis deixar de abordar este elemento pela importância que assume.

A quantidade e profundidade do conhecimento do mundo de um leitor variam de acordo com as características de cada indivíduo [202] e os leitores são diferentes uns dos outros em termos de competências de leitura, conhecimentos que possuem, desenvolvimento cognitivo, cultura e até pela razão que leva cada leitor a ler um texto. As competências de leitura incluem conhecimentos e habilidades linguísticas, capacidade de descodificação e capacidade de pensamento de alto nível. Quanto ao conhecimento, refere-se ao conhecimento sobre o assunto do texto. Quanto maior o conhecimento que o leitor possui relacionado com o texto que está a ler, mais provável será o leitor entender o que está a ler. A compreensão é afectada pela cultura do leitor, na medida em que a sua cultura combina com a cultura exposta no texto pelo autor do texto [202].

Os leitores também leem de um modo particular dependendo do propósito com que estão a ler. A própria motivação de um leitor pode influenciar o interesse, o propósito, a emoção, e até a persistência com que esse leitor se envolve com o texto (e.g. [1]). Os leitores mais motivados aplicam-se mais usando mais estratégias para construir significado. Os leitores menos motivados não se esforçam tanto e em consequência o significado que constroem do texto não será tão correcto. Por sua vez, e de acordo com estudos, a motivação para a leitura é influenciada por factores como a idade, sexo e raça [194] ou mesmo pela personalidade do leitor como mostrado em [186]. Também em [55] foi mostrado que a flexibilidade cognitiva do leitor influencia a compreensão de leitura de passagens de texto e na leitura de palavras isoladas.

Os factores seguintes foram definidos pelos vários investigadores que consideraram importantes na avaliação da dificuldade de leitura de textos [120]:

1. Familiaridade da palavra: Flesch (1948), Dale & Chall (1948 citado em DuBay, 2004), Gunning (1952), Fry (1968), McLaughlin (1969), Bailin & Grafstein (2001)

2. Frequência da palavra: Thorndike (1921), Lively & Pressey (1923), Klare (1968), Chall & Dale (1995), Graesser et al. (2004)

3. Familiaridade com os tópicos e conteúdo: Chang (2006), Pulido (2007), Combs (2008)

4. Comprimento da frase: Kitson (1921), Flesch (1948), Dale & Chall (1948 citado em DuBay, 2004), Gunning(1952), Fry (1968), McLaughlin (1969), Catalano (1990)

5. Densidade do pronome: Graesser et al. (2004)

6. Número de palavras ambíguas: Padak (1993), Graesser et al. (2004)

7. Complexidade Sintáctica: Padak (1993), Bailin & Grafstein (2001), Graesser et al. (2004)

8. Quão concreta ou não abstracta é uma palavra (*Concreteness*): Graesser et al. (2004)

9. Quão fácil é de construir a imagem mental de uma palavra (*Imageability*): Graesser et al. (2004)

10. Clareza do conceito: Bertram & Newman (1981), Padak (1993), Graesser et al. (2004)

11. Tempo: DuBay (2004)

2.7 Conclusão

O capítulo intitula-se teoria da legibilidade, não porque se apresente uma teoria bem definida, mas porque se pretendeu apresentar os principais factores que se considera poderem influenciar a legibilidade de textos e as principais evoluções nesta área. Na realidade, não se pode afirmar que exista uma teoria da legibilidade, mas sim várias, como se pode deduzir não só pelo elevado número de fórmulas distintas que procuram medir a legibilidade de um texto, mas também pelas abordagens diferentes; mais ainda, a forma como

procuram explicar a forma como interpretamos e apreendemos a leitura dos textos. Adicionalmente, a legibilidade recorre a diversas teorias, em particular da psicologia, que ajudam a explicar o processo de leitura e compreensão de textos.

Dois dos factores mais importantes são o texto e o leitor. Também se ficou a saber por esses modelos cognitivos que o conhecimento prévio é fundamental para inferências acerca do texto. As inferências referem-se à extracção de informação de um texto para além do que está escrito no texto para construção do significado do texto. Para o caso do software, pode-se dizer que os modelos cognitivos ajudam a suportar a importância do conhecimento do domínio de aplicação. De qualquer forma, esse conhecimento pode ser fornecido no momento da leitura através dos nomes e comentários existentes no código. Parece lógico que neste caso deverá requerer mais tempo para o programador conseguir compreender o código se já possuísse esse conhecimento previamente.

Como se viu, nas forças armadas norte-americanas o problema da legibilidade foi tratado em duas vertentes: melhoria do nível de literacia dos soldados e a melhoria da legibilidade dos materiais escritos. Para a melhoria da legibilidade foram utilizadas algumas fórmulas de legibilidade devidamente adaptadas que permitiram indicar o nível de legibilidade dos textos. Transpondo para o software, significa que os profissionais e os alunos têm de melhorar as suas capacidades de leitura de código e que é necessário encontrar formas eficazes de conhecer o nível da legibilidade do código que escrevem. A melhoria da capacidade de leitura de código é suportada pela psicologia cognitiva (cf. 2.5.3).

As medidas de legibilidade têm sido amplamente estudadas e utilizadas mas não deixam de apresentar limitações, tal como diversos autores têm mostrado. Neste capítulo foi apresentado um grande número de fórmulas de legibilidade. A investigação da legibilidade no código-fonte tem sido desenvolvida com base nos trabalhos de investigação da legibilidade de textos e em particular adoptando medidas de legibilidade desenvolvidas para textos. O capítulo seguinte ilustra essa tendência.

A psicologia cognitiva e a imagiologia médica contribuem actualmente para o estudo da legibilidade e da compreensão. O estudo da psicologia cognitiva faz-se principalmente segundo quatro vectores de investigação [109, p.4]: a psicologia experimental, a modelação computacional, a neuropsicologia cognitiva e a neurociência cognitiva. É esta última

que envolve a utilização de imagiologia médica para estudo do cérebro, geralmente no diagnóstico.

Capítulo 3

Revisão da Literatura

Neste capítulo é apresentada a revisão do estado de arte sobre a legibilidade no software. Nesta revisão para além de ser apresentada a posição de figuras de relevo na indústria de software, também apresenta artigos de índole científica, que foquem estudos empíricos. Esta última foi realizada em 3 fases. Numa primeira fase, realizou-se um levantamento inicial para identificação de práticas, posteriormente foi realizada uma revisão da literatura seguindo um processo *bola de neve* (*snow-ball*), e por último, foi realizada uma revisão sistemática (*systematic review*), sendo apresentadas no respectivo local. A apresentação é feita por publicação, por ordem cronológica.

3.1 Introdução

A legibilidade, como visto no capítulo 2, prende-se com o julgamento humano sobre a facilidade de compreensão de um texto que é lido [197]. Tanto quanto foi possível apurar, na engenharia de software o termo "readability" tem sido usado a nível do código-fonte e da documentação do software (ver e.g. [39]). Também é verdade que a legibilidade do código está muito relacionada com a manutenção de software. É importante notar que o presente trabalho se centra no código-fonte, não abordando a documentação externa ao código.

Tal como visto anteriormente, para que seja possível realizar a manutenção do software, os programadores têm de entender o código-fonte. A compreensão do código-fonte requer capacidades cognitivas do programador e depende da complexidade do software [50]. Manter o software envolve a leitura do código-fonte e por isso escrever código legível

ou tornar o código-fonte legível poderá promover a facilidade de manutenção do software e desse modo ajudar a assegurar a qualidade deste [197]. Portanto, a legibilidade do software está relacionada com o desenvolvimento e com a qualidade do software. Igualmente, a compreensão do código-fonte afectará também a reutilização do software nas situações em que for necessária a leitura do código-fonte. Mais, é expectável que o tempo requerido na leitura do código-fonte varie em função do seu nível de legibilidade. Para uma organização, mais tempo representa um maior custo e, consequentemente, a legibilidade também poderá ter impacto no custo do software. Para uma análise breve do custo ver e.g. [57].

A manutenção de software corresponde a cerca de 70% do custo de um projecto de software [29][197] o que torna crítica a manutenção do código e da respectiva documentação [3]. Como a legibilidade de um programa está relacionada com a sua manutenção e dada a importância do esforço de manutenção no ciclo de vida do software compreende-se que seja considerada uma característica determinante da qualidade do software. É hoje consensual que a legibilidade do código é uma característica essencial e determinante para a qualidade do código (e.g. [38, 21, 197]).

Estima-se que mais de 50% do tempo gasto pelos programadores profissionais seja com a manutenção, que envolve modificações e actualizações a programas escritos previamente [120].

Uma vez que esses programas são muitas vezes escritos por outros programadores, a compreensão tem um papel fundamental neste esforço [207].

Como se viu, quanto maior o conhecimento que o leitor possui relacionado com o texto que está a ler, mais provável será o leitor entender o que está a ler. Transpondo para o software significa que o conhecimento que o engenheiro informático, ou o aluno, possuem acerca da linguagem e domínio de aplicação do software, mais provável será cada um compreender o que está a ler. Portanto, não se pode esperar que um aluno que desconhece a linguagem usada possa compreender o código-fonte.

O aspecto global do código-fonte pode afectar a legibilidade [39] ou seja, as propriedades visuais também podem afectar a legibilidade na medida em que favorecem ou limitam a facilidade de leitura. Nesse sentido, surgiram diversos trabalhos e ferramentas, designadas por *pretty-printing*, que permitem formatar (e visualizar) o aspecto do código-fonte

de acordo com diversos estilos (ver e.g. [128]).

Também o termo "legibility" tem sido usado na engenharia de software com o sentido definido no capítulo 2, ou seja, relacionado com propriedades visuais associadas às tipográficas. Na interacção pessoa-computador, sendo os computadores concebidos para apresentar a informação visualmente é muito importante ter em consideração requisitos de conforto visual. Uma condição para conseguir um conforto visual é a legibilidade da informação visual [306]. Para Zuffi, a legibilidade eficiente (*readability*), requer uma boa legibilidade (*legibility*) do texto apresentado no ecrã. Esta última, conforme já dito anteriormente, refere-se às propriedades visuais de um carácter ou símbolo e apesar de ser muito importante, não é o tema deste trabalho. A legibilidade pode ser medida de diferentes formas. Legge (citado por Zuffi) considerou a velocidade de leitura dos textos como uma medida de legibilidade [306].

Para Dijkstra, a facilidade de compreensão de um programa depende da simplicidade das suas sequências de controlo e propôs uma disciplina de sequenciação restrita para manter os programas que se tornariam mais complexos, legíveis [273].

Já Kerninghan e Plauger apresentam regras de estilo de programação como linhas de orientação para produção de código legível [273].

Aggarwal [3] propõe um modelo de medição da qualidade de software e considera que a capacidade de manutenção de software é baseada em três aspectos importantes do software: legibilidade do código-fonte (LCF), qualidade da documentação (QDD), e compreensão do software (CDS).

Em [65], De Gyurky afirma que algumas linguagens de programação se lhe afiguravam mais difíceis do que outras. A razão que encontrou para algumas linguagens serem mais fáceis para esse autor do que outras relacionava-se com a sua sintaxe e estrutura, sendo mais fáceis de entender e usar. Chegou à conclusão de que, os indivíduos que inventaram as linguagens de que gostava deviam ter um processo mental semelhante ao seu, ao passo que as que não gostava, não. O processo cognitivo tem dois elementos: o reconhecimento subjectivo e o reconhecimento objectivo. Isto também é verdadeiro na literatura[1]. A ideia de cognição, e a compreensão de como nós, seres humanos sentimos, percebemos, ouvimos, observamos, compreendemos, comunicamos, decidimos e agimos, tornou-se de

[1]Tal como visto no capítulo 2, a legibilidade é também função do leitor.

grande interesse para o autor porque considera estarem ligadas à forma como realiza o seu trabalho, individualmente, ou com outros [65].

A revisão do estado da arte apresentada aqui engloba não só artigos de índole mais científica, como também a apresentação de posições assumidas por algumas figuras de relevo na indústria do software e veiculadas essencialmente em livros. Estes últimos, baseiam-se em geral na experiência dos seus autores. Dada a importância determinante que têm assumido no desenvolvimento de software não se poderia deixar de incluir as suas posições.

A apresentação do estado da arte é feita por publicação e seguindo uma ordem cronológica. Optou-se por apresentar as publicações desta forma e não por tema para que ficasse mais claro o que cada artigo ou livro apresenta e a sua contribuição para o tema. De qualquer modo, foi criada uma tabela que agrupa os estudos por tópico. Nos casos em que o primeiro autor surge em mais do que uma publicação, então essas publicações surgem todas na posição correspondente ao ano em que foi publicado com o respectivo nome do autor. Importa referir que certos artigos, dada a sua antiguidade, não se referem a linguagens OO, e por vezes as suas conclusões são postas em questão por publicações posteriores e em vários casos não foi possível aceder ao documento original.

No final de cada uma das revisões, na indústria e dos estudos, as práticas encontradas foram compiladas numa tabela.

3.2 Legibilidade na Indústria

3.2.1 Introdução

Nesta secção deverá ficar clara a importância que a indústria dá à legibilidade do código-fonte. Essa importância é de tal ordem que alguns autores consideram que o código deve ser escrito a pensar na sua leitura posterior. Os nomes pela sua prevalência serão um dos factores mais importantes, se não o mais importante, para a legibilidade do software. Por exemplo, em [178] é referido que representam 90% do que torna o software legível.

3.2.2 Revisão

Como se verá nesta secção, os diversos autores aqui referidos são de opinião que para produtos textuais a legibilidade é um aspecto-chave na facilidade de manutenção do software. Como afirma Robert Martin, [178] "Se pretende que o código seja fácil de escrever então faça-o fácil de ler."

As opiniões são apresentadas por ordem cronológica. Mesmo quando o mesmo autor apresenta diferentes livros, como é o caso de Bertrand Meyer, o primeiro de 1997 e o segundo de 2009.

Shyam Chidamber e Chris Kemerer, 1991 [51] Existem duas categorias de criticismos que podem ser aplicados às correntes métricas de software.

A primeira categoria tem uma posição relativa com as métricas de software convencional sendo aplicadas ao desenvolvimento de software não orientado a objectos.

A segunda categoria é mais específica com desenvolvimento e concepção orientada a objectos. Suportam conceitos como por exemplo, herança, encapsulamento e polimorfismo. Neste sentido é importante falar na Lei de Demeter, que representa uma tentativa formal de definição de regras de estilo de programação orientada a objectos correctas, baseada nos conceitos de acoplamento e coesão.

Bertrand Meyer, 1997 [189] É um livro orientado para a linguagem Eiffel. Na selecção dos aspectos associados com a legibilidade procurou-se seleccionar apenas aqueles que pareceram ser transversais às várias linguagens orientadas a objectos.

No seu livro, o autor diferencia entre factores de qualidade internos e externos. Os factores de qualidade externos são aqueles em que a sua presença ou ausência são detectadas pelo utilizador final do software. Para os factores de qualidade externos temos por exemplo, a facilidade de utilização e o desempenho. Os factores internos são todos os aplicáveis ao produto de software, e só perceptíveis pelos profissionais que têm acesso ao código, como por exemplo, ser modular e ser legível. Apesar de, em última instância, só as qualidades externas interessarem, a chave para as conseguir alcançar está na obtenção das qualidades internas.

Meyer [189] considera que a sobrecarga de funções tem impacto negativo na legibilidade do código. Relativamente à notação, para Meyer existe um conjunto de regras que ele considera essenciais para a legibilidade do código-fonte:

- Os nomes das classes e dos parâmetros genéricos formais devem estar todos em maiúsculas;
- Entidades pré-definidas e expressões e os atributos constantes devem começar com uma letra maiúscula e as restantes letras serem minúsculas;
- Todos os outros identificadores devem estar em minúsculas. Fazem parte deste grupo, os atributos não constantes, os argumentos formais das funções e entidades locais.

Na perspectiva do engenheiro de software as declarações explícitas são uma ajuda, não uma falha. Os tipos deverão ser claros não só para o compilador mas também para o leitor. *Static typing* é essencial para a fiabilidade, legibilidade e eficiência.

Definição 8 (*Static Typing*)**.** *Uma linguagem é estaticamente tipada se o tipo de uma variável é conhecido em tempo de compilação.*

Uma regra do estilo de software é o *Princípio da Constante Simbólica*, que estabelece que quando é necessário usar um valor constante, nunca se deve usá-lo directamente. Deve-se associar um nome a esse valor e usar o nome em vez do valor. Com a utilização deste princípio aumenta-se a legibilidade e a extensibilidade do código.

Martin Fowler, 2000 [97] Martin Fowler refere que as técnicas usadas no *refactoring* do código também podem contribuir para melhorar a legibilidade. Por vezes a aplicação das técnicas de *refactoring* fazem com que o código funcione apesar de não estar idealmente estruturado. Se for gasto algum tempo a aplicar devidamente essas técnicas é possível fazer código que melhor comunique o seu propósito.

Uma forma de melhorar a legibilidade consiste em criar código sem encadeamento de métodos e que obedeça à "Lei de Demeter".

Quando se tem um literal numérico, este deverá ser substituído por uma constante simbólica usando um nome com significado. A utilização de constantes não acarreta qualquer custo em termos de desempenho e melhora a legibilidade.

Quando se passam diversos valores de um objecto para uma função então será preferível passar o próprio objecto. Fowler justifica esta sugestão com a facilidade em proceder a alterações futuras ao método no caso em que seja necessário passar novos dados do objecto. A justificação faz todo o sentido, mas é questionável quanto à legibilidade. A redução de parâmetros facilita a leitura, mas simultaneamente o cabeçalho da função, por não especificar os dados, pode reduzir a identificação do propósito da função.

Também existem situações em que o *refactoring* tem dificuldade em melhorar a legibilidade. Por exemplo, um método com muitos parâmetros e variáveis temporárias pode ser difícil de ser lido e a utilização de *refactoring* conduzirá à passagem de tantos argumentos e variáveis temporárias como parâmetros que o resultado será muito improvavelmente mais legível que o original.

Kim Bruce, 2002 [37] Este autor também aborda a legibilidade. No contexto das expressões lambda, considera que a sua utilização torna o código mais legível.

Bruce Wampler, 2002 [285] Para Wampler a legibilidade também é importante. Realça a importância de uma boa escolha dos nomes para os identificadores quer a nível das classes, métodos e variáveis mas chama a atenção em particular para as classes. O nome deve ter significado de modo a evitar o uso de comentários desnecessários. Também considera muito importante o *refactoring* para redução dos tamanhos dos métodos de modo a ajudar a melhorar a legibilidade.

Para Wampler é importante maximizar o encapsulamento. Quanto mais independente for uma classe, melhor. Ao mesmo tempo, é importante minimizar o acoplamento. Isto é, as classes devem ser o mais independente possível umas das outras.

Finalmente considera que muitos dos problemas do software *open source* tem a ver com a falta de legibilidade e de possibilidade de manutenção.

Robert Lafore, 2002 [157] O livro de Lafore apresenta os conceitos da programação orientada a objectos em C++. Para além de explicar a linguagem tem a preocupação de

referir certos aspectos de modo a tornar o código dos programas mais legíveis, nomeadamente:

- Declaração das variáveis. O programador deve preocupar-se com o local onde declara a variável. Idealmente é junto ao local onde é usada;
- Instrução *for*. Lafore realça a utilização da instrução *for* para o facto de poder ser utilizada de diferentes formas. Chama a atenção para, no sentido de não diminuir a legibilidade, dever ser usada de uma forma simples;
- Inclusão de chavetas nos blocos de instruções. Por vezes, dentro de uma instrução de repetição existe uma instrução condicional *if*. Nesta situação não é necessário a colocação de chavetas uma vez que a instrução *if* e as instruções do seu corpo são consideradas uma só instrução. Para aumentar a legibilidade deve-se colocar o bloco da instrução *if* dentro de chavetas;
- Tipo retornado nas funções. Deve-se explicitar sempre o tipo retornado da função mesmo que seja do tipo *int*, não sendo permitido a sua omissão;
- Sobrecarga de operadores. É uma boa prática a implementação de operadores nas classes em vez de métodos que realizem a mesma funcionalidade. No entanto, certas linguagens, tal como o caso do Java não permite esta funcionalidade.

John Mitchell, 2003 [192] Reforça a opinião de John Backus que em 1977 afirmou que a exactidão, a legibilidade e a fiabilidade são os factores mais importantes de um programa. Para Mitchell, outro aspecto importante que permite aumentar a legibilidade do código é a utilização de variáveis do tipo booleano no contexto de retorno de funções. Como se compreende, tal utilização implicará o retorno através dos parâmetros dos métodos e, por vezes, aumentar em 1 o seu número.

Diomidis Spinellis, 2003 [260] Spinellis defende que a legibilidade do código e a eficiência não são incompatíveis. Para o autor, não há necessidade de sacrificar a legibilidade do código a favor da eficiência. No entanto, acrescenta que muitas vezes os algoritmos eficientes tornam o código mais complicado. Mesmo se existirem desempenhos diferentes,

a economia que está por trás dos custos de manutenção de software, os salários dos programadores e o desempenho da máquina são a maioria do tempo a favor da legibilidade sobre a eficiência. Este autor também refere a importância da boa escolha dos nomes das variáveis e a indentação de código, de modo a aumentar a legibilidade do código. Padrões de codificação fornecem orientação estilística adicional com o objectivo de aumentar a robustez do código, legibilidade e facilidade de manutenção. A organização criativa do código pode ser usada para melhorar a legibilidade do código.

O autor refere a seguinte lista de elementos básicos da programação para melhorar a legibilidade:

- Reorganizar o código para torná-lo mais legível; a organização criativa do código pode ser usada para melhorar a legibilidade do código;
- Aumentar a legibilidade de expressões usando espaços brancos, variáveis temporárias e parêntesis;
- É possível melhorar a legibilidade de código mal escrito melhorando a indentação e aplicando nomes apropriados às variáveis.

Steve McConnell, 2004 [182] É possível adicionar pelo menos dois níveis de flexibilidade e legibilidade na utilização de variáveis de estado [182, p.292]. Não usar variáveis booleanas como variáveis de estado. Em vez disso usar tipos enumerados. É comum adicionar um novo estado às variáveis de estado (*status*) e com a adição de um novo tipo a um tipo enumerado requer unicamente uma nova compilação em lugar de se reverem todas as linhas do código que contêm a variável.

Dar preferência a rotinas de acesso à variável em vez de verificar directamente a variável. Desta forma possibilita-se uma detecção de estado mais sofisticada. Por exemplo, para verificar combinações de uma variável de estado de erro com uma variável de estado da função corrente, deverá ser mais fácil de o fazer se o teste está numa rotina (invisível do exterior) e mais difícil de fazer se o teste estiver codificado ao longo do programa.

Criar funções curtas. Funções pequenas têm várias vantagens. Uma delas é que aumenta a legibilidade.

Os programas que usam excepções como parte do seu processamento normal têm problemas de legibilidade e manutenção, problemas estes característicos do código dito "spaghetti". As excepções têm uma semelhança com a herança: usadas com muito critério e bom senso, podem reduzir a complexidade, mas usadas de forma indiscriminada, podem tornar o código praticamente impossível de ser seguido.

Deve usar-se sempre chavetas para delimitar as instruções dos ciclos. Não tem custos de velocidade nem de espaço em tempo de execução. No entanto, ajudam na legibilidade, e ajudam a prevenir erros quando o código é modificado. É considerada uma boa prática de programação defensiva.

Barry Boehm, 2005 [29] Para Boehm, a legibilidade é uma componente central de cada um dos aspectos da qualidade do software, tais como a sua manutenibilidade, a sua facilidade de modificação e de reutilização resultando numa redução correspondente de custos associados para com os utilizadores.

Ken Pugh, 2005 [215] Este autor considera que se a legibilidade fosse levada ao extremo, o código poderia ser lido pelo cliente. Apresenta várias linhas de orientação para a legibilidade, mas faz notar que algumas poderão ser extremas e por isso cada um deverá encontrar um equilíbrio na sua utilização tendo em atenção a situação concreta de utilização:

- Criar um nome adequado a cada conceito num sistema;
- O desenho de uma interface pode ser mais poderoso do que uma descrição;
- O código deve comunicar o seu propósito;
- Ser explícito reduz erros de interpretação;
- Programação declarativa pode fornecer flexibilidade sem alterações no código;
- Usar modelos ou *scripts* para classes e métodos de modo a criar lógica consistente;
- Nunca se deve escrever funcionalidades que já existam;

- Usar a linguagem do cliente no código para tornar mais fácil comparar a lógica do código com a lógica do cliente.

Para além dessas orientações inclui outras ao longo do texto, essencialmente acerca da concepção do software, como, por exemplo, cada classe representar uma abstração. Também chama a atenção para a necessidade de se inserir um comentário de uma linha em cada classe sobre o seu propósito. Se o comentário exceder uma linha, considera que a classe representará provavelmente mais do que uma abstracção e terá de ser revista. Também considera que usar um nome simbólico para todos os valores tem impacto positivo na legibilidade.

Ian Sommerville, 2006 [258] Sommerville refere que melhorar a legibilidade do programa foi uma razão chave para a introdução de programação estruturada por Dijkstra e para a proposta deste da eliminação do desvio incondicional (*Goto*) nas linguagens de programação de alto nível.

Robert Martin, 2006 [177] Martin neste seu livro "Agile principles, patterns, and practices in C#" afirma que todos os módulos de software têm três funções:

1. Executar a tarefa para a qual foi criada durante a sua execução;

2. Estar apta a modificações;

3. Comunicar com o seu leitor de modo a que este consiga entender o seu código sem grande esforço.

Franklyn Turbak e David Gifford, 2008 [280] Os autores apresentam várias características que consideram importantes para melhorar a legibilidade do código. Relativamente às características que se aplicam ao paradigma orientado a objectos, algumas já são bem conhecidas, de que se destacam a importância de utilização de espaços para separação dos *tokens* nas expressões e a atribuição criteriosa de nomes aos identificadores.

Bertrand Meyer, 2009 [190] Bertrand Meyer no seu livro "Touch of Class" também refere muitos factores importantes para a legibilidade do código. Em primeiro lugar, sugere que, desde o engenheiro de software ao programar deve ter consciência que não está a "falar" só para o computador mas também para as pessoas. São as pessoas quem vão ler os seus programas, posteriormente. Este aspecto humano é um aspecto central para a engenharia do software [190].

Refere um conjunto de práticas que todo o programador deve seguir. Para o autor este conjunto de práticas é muito importante para criar bons programas, permitindo assim que as pessoas consigam lê-los. Para os programas serem legíveis, devem seguir as seguintes regras:

- Instruções e expressões: uma instrução por linha;
- Escolha dos identificadores: escolher sempre os nomes dos identificadores que identificam claramente o seu papel;
- *Break* e indentação: usar cuidadosamente para melhorar o *layout* e consequentemente ajudar na legibilidade;
- Eliminação do *Goto* e programação estruturada: as construções de alto nível, condicionais e de ciclo, são mais claras. Limitar as imbricações por forma a manter os programas legíveis;
- *Princípio da constante simbólica*: usar constantes simbólicas em lugar de usar directamente o valor. Existem dois argumentos que suportam esta regra: a legibilidade e a facilidade de evolução;
- *Princípio da utilização de nomes padrão*: sempre que aplicável, a utilização de uma terminologia consistente, ajudará a que os mais novos aprendam a usá-la e dessa forma, as pessoas que vão ler os programas poderão mais facilmente identificar as características-chave e os seus propósitos.

Robert Martin, 2009 [178] No seu livro "Clean Code" [178], Robert Martin apresenta a opinião de diferentes autores/profissionais bastante conhecidos acerca do que entendem

por *clean code*[2]. Por aqui se verá também a importância da legibilidade.

- Bjarne Stroustrup

  O código deve ser elegante e eficiente. As dependências devem ser mínimas para facilitar a manutenção, o tratamento de erros completo de acordo com uma estratégia articulada, e o desempenho próximo do ideal de modo a não induzir as pessoas a tornar o código confuso devido a optimizações sem um fio condutor. Finalmente, o código para ser limpo deve fazer uma coisa bem, o que significa que um módulo deve ter uma funcionalidade/responsabilidade e não várias.

  A partir do termo elegante pode concluir-se que o código deve ser agradável de ser lido, enquanto eficiente poderá estar associado a simples e directo, ou seja, focalizado numa certa finalidade e sem pormenores que distraiam dessa finalidade.

- Grady Booch

  Clean code é código simples e directo que se lê como prosa bem escrita. Esse código nunca obscurece a intenção do projectista, mas sim está cheio de abstrações claras (sem pormenores desnecessários) e linhas directas de controlo. Grady toca em alguns dos pontos tocados por Bjarne, mas tem uma perspectiva de legibilidade.

- Dave Thomas

  Clean code é código que pode ser lido, e melhorado por um desenvolvedor que não o seu autor original. É código que tem testes unitários e de aceitação. Tem nomes significativos. Uma única responsabilidade. *Clean code* tem uma quantidade muito reduzida de dependências, que são explicitamente definidas, e para além disso, proporciona uma API clara. O código deve ser *literate* porque não é possível expressar todas as informações necessárias de forma clara, unicamente com o código.

  De acordo com [178, p.9], código *literate*[3], é uma referência a Donald Knuth e por isso terá como consequência que o código deverá ser escrito de forma a ser legível. Devem existir testes como forma de tornar o código limpo (menos erros) e,

[2]Este termo não será traduzido.

[3]Código cujo objectivo principal deve ser criado em primeiro lugar para ser lido e entendido pelas pessoas e em segundo lugar para ser executado pelo computador.

dessa forma, talvez ajudar a melhorar a sua legibilidade porque será mais fácil ler e compreender código sem erros e pormenores que distraem da sua finalidade.

- Michael Feathers

  Destaca a seguinte qualidade entre todas as possíveis. *Clean code* tem sempre o aspecto de ter sido escrito por alguém que deu importância à qualidade da sua escrita. Como tal, não há nada de óbvio que se possa fazer para torná-lo melhor.

  Se o desenvolvedor se preocupou com o código, então este será simples e organizado. A dificuldade em encontrar formas de melhorar o código é também uma forma de perceber se o código é limpo.

- Ron Jeffries

  Este autor já escreveu código em quase todas as linguagens e para quase todos os sistemas [178, p.10]. Considera importantes as quatro regras de Kent Beck, pela seguinte ordem de prioridade:

  1. Executar todos os testes;
  2. Não possuir código duplicado;
  3. Expressar todas as ideias da concepção do sistema;
  4. Minimizar o número de entidades, tais como classes, métodos, funções ou outras.

  Relativamente à expressividade refere ainda o seguinte: identificadores com significado; cada objecto e cada método deve realizar apenas uma tarefa. Caso um objecto não o faça então deverá ser dividido; e no caso de um método deve aplicar-se "*refactoring*". Também aconselha a que se usem abstrações para tarefas (implementações) que se programam frequentemente, pois permite alterar a implementação de acordo com as necessidades e saber qual a tarefa sem se perder nos detalhes da implementação. Portanto, estas abstrações favorecem a legibilidade.

- Ward Cunningham

  Para este autor sabemos que estamos a trabalhar com *clean code* quando uma rotina que se lê é exactamente como se esperava que fosse. E pode-se designar por *Beautiful Code* o código que faz parecer que a linguagem foi feita para o problema.

> Para o código ser como se esperava, implica que se lê com facilidade, ou pelo menos com a facilidade expectável para o problema em causa. Para isso, haverá que manter o código simples, ou seja, a linguagem que o código "usa" deverá ser simples. Como consequência, o código parecerá bonito nos termos desse autor. No extremo pode-se dizer que com *clean code* se percebe qual o problema a partir da solução.

Martin [178] também apresenta muitas recomendações que, de acordo com a sua experiência, poderão permitir melhorar a qualidade do código, especialmente em termos de manutenção. Dessas podem-se encontrar muitas recomendações acerca da escrita do código-fonte com possível impacto na legibilidade, as quais se apresentam em seguida.

Pode-se dizer que os nomes são omnipresentes no código-fonte. Os nomes estão em todo o lado. Nomes das variáveis, das funções, dos argumentos e dos parâmetros, dos ficheiros fonte e das pastas que os contêm. Desse modo, o nome de uma variável, função ou classe deverá responder à seguinte questão [178, p.18]: *porque* existe, *o que* faz e *como* é usado.

Para isso, Martin apresenta algumas orientações que podem ajudar a responder a estas questões:

- Um nome deve revelar o propósito da sua utilização. Se um nome requer um comentário então esse nome não revela o seu intuito;
- Nomes devem ser informativos e distinguíveis. Um exemplo marcante de desinformação consiste na utilização de maiúsculas e minúsculas como nomes, especialmente o uso da sua combinação; outro exemplo de nomes não informativos, e não distinguíveis quanto ao propósito, são as séries de números, a1, a2, a3, ... Estes nomes não dão qualquer informação ao leitor acerca do seu propósito; nomes redundantes também caem nesta categoria e não devem ser usados. São as chamadas *noise words*, como por exemplo, o nome "variável". Com os ambientes de desenvolvimento modernos usar o tipo de dado no nome também será em geral redundante. Indicar o contexto no nome apenas se necessário;
- Dar nomes a literais. Isso torna-se ainda mais importante quando uma constante aparece mais do que uma vez no corpo do código;

- Usar uma e só uma palavra para um conceito e mantê-la ao longo de todo o programa;
- Código relacionado com o domínio do problema deve usar termos desse domínio.

Martin apresenta ainda outras orientações já usadas mais comummente, como: usar substantivos para os nomes das classes; iniciar os nomes das classes com maiúscula; incluir verbos nos nomes dos métodos/funções; tal como quaisquer outros nomes, os nomes das funções também devem ser descritivos. Acerca de funções (métodos) estes devem ser de pequena dimensão. Para Martin, pequeno significa 3 ou 4 linhas.

Existem estratégias que podem ajudar a conseguir manter uma função curta:

1. Escrever cada bloco de um *if* numa linha, por exemplo, através da invocação de uma função;
2. Garantir que cada função realiza apenas uma tarefa. E uma e só uma, se se pretende evitar duplicação de código. Pode ser testado, verificando se as instruções se encontram todas no mesmo nível de abstracção. O tratamento de excepções, a interrogação e a alteração (*set*) também contarão cada uma como uma tarefa;
3. Reduzir a utilização do $switch/if$ a favor do polimorfismo.

Na medida do possível, as funções devem encontrar-se pela ordem em que são invocadas e num nível decrescente de abstração. O número de argumentos de uma função deve tender para zero e 3 já serão de evitar. Quando o número é de 3 ou superior, será preferível, se possível, passar um objecto. Também devem ser evitadas as sentinelas como parâmetros.

Se uma função chama outra, devem ser verticalmente fechadas, e a função chamadora deverá estar acima da função chamada, sempre que possível (na medida do possível). Isto dá ao programa um fluxo natural. Ao usar uma determinada convenção deverá ser consistente, isso é muito importante para quem lê o código.

A duplicação pode estar na raiz de muitos dos problemas no software. Muitos princípios e práticas foram criados com a finalidade de controlar ou eliminar a duplicação. A programação orientada a objectos serve para concentrar código em classes de base que seriam redundantes. A Programação Estruturada, a Programação Orientada a Aspectos,

a Programação Orientada a Componentes, são, em parte, as estratégias para eliminar a duplicação.

Martin considera os comentários um mal necessário. Por isso, o seu uso deve ser muitíssimo criterioso e adopta a sugestão de Kernighan e Plaugher de que não se deve comentar mau código mas sim reescrevê-lo. Quando tem de se escrever um comentário este deverá ser escrito da melhor forma possível. Também sugere que devem ser informativos e/ou justificar a decisão da implementação. Não devem existir instruções comentadas. As instruções que não são usadas devem ser removidas. Em síntese, os comentários devem existir apenas para ajudar a perceber, apesar do código dever ser autoexplicativo.

Martin considera fundamental que o desenvolvedor garanta a legibilidade do código. Para isso, a formatação do código desempenha um papel muito importante.

Apresenta algumas orientações nesse sentido. Como se verá, algumas dessas orientações estão relacionadas com a estruturação do código-fonte tendo como critério a sua semântica e não unicamente a sua formatação:

- Cada ficheiro não deve ser demasiado grande;
- Cada ideia no código deve ser separada por uma linha em branco;
- Evitar separar as instruções que se relacionam entre si;
- Ideias relacionadas entre si devem estar verticalmente próximas;
- O mesmo se passa com as funções que são invocadas por outras: a função que chama deve surgir antes da que é chamada;
- As variáveis devem ser declaradas o mais próximo possível do local da sua utilização;
- As linhas devem ser curtas. Sugere 120 caracteres como o máximo;
- Separar os operadores com um espaço. Mas no caso de expressões com vários operadores, não aplicar o espaço nos operadores de maior precedência;
- No caso de expressões maiores quebrá-las e usar expressões intermédias;
- Usar indentação. Sem indentação, os programas poderão ser virtualmente ilegíveis. Nesta situação não sugere valores.

Quanto aos objectos, Martin sugere que se evite usar insersores (*set*) e extractores (*get*) para todos os atributos, pois isso fá-los parecer automaticamente como públicos, quando antes foram definidos como privados. A solução é usar esses métodos apenas de acordo com as necessidades.

Martin, discute ainda outros factores que segundo ele, influenciam o *clean code*, nomeadamente, a lei de Demeter e objectos criados unicamente para transferência de dados a partir de uma base de dados, mas que são tratados como objectos de negócio.

Clean code além de ter de ser legível também deverá ser robusto. Estes dois objectivos não são incompatíveis entre si. É portanto possível escrever *clean code* robusto. Para isso, haverá que seguir a orientação anterior de considerar o tratamento de excepções como uma tarefa em si e iniciando a respectiva função com a palavra reservada *try*. Apesar da importância do tratamento de erros, Martin considera que é errado que o tratamento de erros obscureça a lógica do código.

Produzir código legível é tão importante quanto torná-lo executável. No entanto, não é boa ideia tentar escrever *clean code* se não se sabe o que isso significa.

Escrever *clean code* requer o uso disciplinado de um grande número de pequenas técnicas aplicadas cuidadosamente de modo a conseguir alcançar o código de melhor qualidade (e.g. menos erros, mais coeso). Esta sensibilidade para criar assim o código é a chave. Alguns de nós nascemos com ela, outros têm de lutar muito para a adquirir. Não só nos permite ver se o código é bom ou mau, mas também nos mostra a estratégia a seguir na aplicação da nossa disciplina para transformar código de má qualidade em *clean code*.

Clean code significa, para além de legível, código sem erros (na medida do possível), o que conduz à existência de testes. Portanto, quando o código-fonte vem acompanhado de testes, estes têm obrigatoriamente de ser legíveis. Para Martin, nos testes unitários, a legibilidade é talvez ainda mais importante que na produção de código. O que faz com que os testes sejam legíveis são as 3 características seguintes: clareza, simplicidade e densidade das expressões. Acrescenta que num teste quer-se dizer muito com o menor número de expressões. No entanto, não especifica em concreto como conseguir atingir cada uma destas 3 características. Apesar disso, fornece diversas orientações acerca de como conseguir *clean test*.

Recentemente, foi apresentado trabalho de carácter mais científico acerca da legibili-

dade de testes [62]. Trata-se de um modelo de legibilidade de testes unitários para código orientado a objectos, ou seja, um modelo que permite predizer a legibilidade dos testes.

Martin Fowler, 2010 [96] Martin Fowler neste seu livro, "Domain Specific Languages", afirma que a chave para uma linguagem específica de domínio[4] é a legibilidade, apesar da existência de uma série de ferramentas para ajuda na escrita do código. O que realmente conta é a leitura do código [96].

No código, a legibilidade também é afectada através do encadeamento de métodos[5] pois obriga à modificação dos nomes dos métodos, não seguindo as convenções dos nomes de métodos. Deste modo, torna o código mais confuso, reduzindo assim a legibilidade.

O encadeamento de métodos para além de afectar a legibilidade, normalmente quebra o *Princípio Command-Query Separation (CQS)*. Este princípio é extremamente importante e muito comum na programação orientada a objectos e foi proposto por Bertrand Meyer [96]. Segundo este princípio, um método pode ser de dois tipos:

1. *Command* Mudando o estado do objecto que o define, mas não retornando valor algum (métodos *void*);

2. *Query* Retornando o resultado de algum processamento, mas não alterando o estado do objecto que o define (*no side-effects*).

Kevlin Henney, 2010 [119] Neste livro são apresentados vários testemunhos de profissionais. Aqui serão apresentadas apenas as opiniões relacionadas com a legibilidade do código.

- Jorn Ølmheim

  Afirma que existem quatro aspectos que devem estar presentes no desenvolvimento de código:

  - Legibilidade;

[4]Uma linguagem específica de domínio (*domain-specific language (DSL)*) é uma linguagem especializada para um domínio em particular.

[5]O encadeamento de métodos consiste numa sequência de chamadas de métodos onde cada chamada actua no resultado da chamada anterior. Os métodos são separados por um ponto ".".

  - Facilidade de manutenção;
  - Velocidade de desenvolvimento;
  - Qualidade indescritível de beleza.

  Para o autor, bom código é aquele que os objectos são simples com uma só responsabilidade contendo similaridade simples; em que os métodos têm nomes descritivos e o seu tamanho varia entre 5 e 10 linhas.

- Yechiel Kimchi

  Para Kimchi o software deve ser livre de erros. Nesse sentido deve::

  - Evitar a utilização de instruções *goto*;
  - Evitar variáveis globais que possam ser modificáveis;
  - Cada variável deverá ter um âmbito reduzido;
  - Tornar o código legível usando espaçamento vertical e horizontal;
  - As funções/métodos devem ser pequenas e só com uma tarefa com um limite de 24 linhas;
  - As funções/métodos devem ter poucos parâmetros. Sugere um limite de quatro;
  - Evitar os métodos de acesso *get* que retornam o estado interno do objecto. O encapsulamento é tudo;
  - A utilização dos métodos de acesso *set* deve ser desencorajado por forma a evitar a quebra do estado de um objecto.

  Desta forma mais fácil será de entender o código.

- Kirk Pepperdine

  O polimorfismo é uma das grandes ideias que é fundamental na programação orientada a objectos. O polimorfismo usado cuidadosamente permite criar pequenos contextos de execução sem necessidade de recorrer aos blocos *if-then-else*. O uso cuidadoso das implementações alternadas, é possível capturar contexto que pode ajudar a produzir menos código e mais legível.

- Karianne Berg

  A autora diz que apesar de escrever código ser uma tarefa divertida, a leitura do código é difícil, por vezes impossível. Então ler código de outras pessoas ainda é mais difícil. A autora considera algumas situações em que o código é de difícil leitura:

  - Formatação pobre;
  - Nomes inconsistentes e pobres;
  - Mistura de assuntos/tarefas no mesmo bloco de código.

- Kevlin Henney

  Para este autor os testes são muito importantes. Considera que um bom teste deverá ser legível. Um bom teste deverá ser suficientemente compreensível e suficientemente simples que se consegue ver de uma forma legível se está ou não correcto.

John Hunt, 2012 [127] John Hunt diz o seguinte no seu livro:

"Independentemente da linguagem, um bom estilo de programação ajuda a promover a legibilidade, a compreensão e a clareza do código."

Com esta frase pode-se ver a importância que a legibilidade do código-fonte tem no desenvolvimento do software.

3.2.3 Lista de Práticas de Legibilidade na Indústria

Os vários artigos e livros analisados na secção 3.2.2 são listados na tabela 3.1 apresentada em seguida. Para cada prática é indicado o(s) autor(es) que a referem.

Tabela 3.1: Lista de Práticas de Legibilidade na Indústria.

Descrição	**Fonte**
Os nomes das classes e dos parâmetros genéricos formais devem estar todos em maiúsculas.	Meyer[189]
Entidades pré-definidas e expressões e os atributos constantes devem começar com uma letra maiúscula e as restantes letras serem minúsculas.	Meyer[189]
Todos os outros identificadores devem estar em minúsculas. Fazem parte deste grupo, os atributos não constantes, os argumentos formais das funções e entidades locais.	Meyer[189]

Descrição	**Fonte**
Utilização de constantes simbólicas em substituição de constantes numéricas (Princípio da Constante Simbólica)	Meyer[189], Fowler[97], Martin[178],
Instruções e expressões: uma instrução por linha.	Meyer [189]
Escolha adequada dos nomes dos identificadores. Nomes descritivos e distinguíveis.	Raymond[218], Meyer[189], Wampler[285], Spinellis[260], Turbak[280], Gifford[280], Thomas[178], Jeffries[178], Martin[178], Pepperdine[119], Ølmheim[119]
Evitar variáveis globais que possam ser modificáveis.	Kimchi[119]
As linguagens devem ser estaticamente tipadas.	Meyer[189]
Break: usar com critério.	Meyer[189]
Usar Indentação.	Raymond[218], Meyer[189], Spinellis[260], Martin[178]
Eliminação do desvio incondicional *Goto* e preferência pela programação estruturada: as construções de alto nível, condicionais e de ciclo, são mais claras.	Meyer[189], Sommerville[258], Kimchi[119]
Evitar imbricações profundas.	Meyer[189]
Princípio da utilização de nomes padrão: sempre que aplicável, a utilização de uma terminologia consistente.	Meyer[189]
O código deve obedecer à "Lei de Demeter".	Chidamber [51], Fowler [97], Martin[178]
Evitar o uso de encadeamento de métodos	Fowler [96]
Redução do número de parâmetros dos métodos/funções. Martin sugere que o número de argumentos de uma função deve tender para zero, sendo já de evitar 3. Kimchi sugere um máximo de 4.	Fowler[97], Martin[178], Kimchi[119]
Redução do número de variáveis temporárias.	Fowler[97]
Promover utilização de expressões lambda.	Fowler[97]
Redução dos tamanhos dos métodos (métodos extensos).	Wampler[285], Ølmheim[119], Fowler [97]
Maximizar o encapsulamento.	Wampler[285]
Minimizar o acoplamento.	Wampler[285]
Código idêntico ou semelhante existente em mais de um local.	Fowler[97]
Não usar nomes de tipos de dados em nomes de métodos.	Fowler[97]
Evitar nomes de métodos que não descrevem o que faz.	Fowler[97]
Evitar grandes blocos lógicos condicionais.	Fowler[97]
Local de declaração das variáveis sem critério.	Fowler[97]
Declarações das variáveis junto ao local onde são usadas (âmbito reduzido de uma variáve).	Lafore[157], Martin[178], Kimchi[119]
Evitar local de declaração das variáveis sem critério.	Fowler[97]
Linhas de código relacionadas devem aparecer juntas.	Martin[178]
A instrução *for* deve ser usada da forma mais simples. Evitar usar sem expressões.	Lafore[157]
Inclusão de chavetas nos blocos de instruções.	Lafore[157], McConnell[182]
Deve-se explicitar sempre o tipo retornado nas funções.	Lafore[157]

Descrição	**Fonte**
Sobrecarga de operadores: a implementação de operadores nas classes em vez de métodos que realizem a mesma funcionalidade nas linguagens que o permitam.	Lafore[157]
Organização criativa do código.	Spinellis[260]
Nas expressões uso de espaços brancos, variáveis temporárias e parêntesis.	Spinellis[260]
Espaço branco entre os argumentos/parâmetros nas funções	Martin[178]
Espaço branco para acentuar as precedências dos operadores numa expressão	Martin[178]
Utilização de espaços para separação dos tokens nas expressões.	Turbak, Gifford[280]
No operador atribuição, incluir espaço branco antes e depois do operador	Martin[178]
Utilização de enumerações em lugar de variáveis de estado.	McConnell[182]
Rotinas de acesso à variável em vez de verificar directamente a variável.	McConnell[182]
Criar funções curtas. Ølmheim sugere um tamanho máximo de 5 a 10 linhas. Kimchi sugere um máximo de 24 linhas.	McConnell[182], Martin[178], Ølmheim[119], Kimchi[119]
Usar as excepções com muito critério e bom senso.	McConnell[182]
Espaçamento vertical e horizontal.	Kimchi[119]
A lógica deve ser simples de modo a tornar difícil a ocultação dos erros.	Stroustrup[178]
Dependências mínimas.	Stroustrup[178], Thomas[178]
Cada função / método com uma só tarefa / responsabilidade.	Stroustrup[178], Ron Jeffries[178], Martin[178], Ølmheim[119], Pepperdine[119], Kimchi[119]
Testes unitários e de aceitação.	Thomas[178], Jeffries[178], Martin[178]
Código literado.	Thomas[178]
Não possuir código duplicado. Controlar e diminuir.	Jeffries[178], Martin[178]
Expressar todas as ideias da conceção do sistema.	Jeffries[178]
Minimizar o número de identidades.	Jeffries[178]
Usar a regra: *porque* existe, *o que* faz e *como* é usado.	Martin[178]
Usar uma e só uma palavra para um conceito e mantê-la ao longo de todo o programa.	Martin[178]
Código relacionado com o domínio do problema deve usar termos desse domínio.	Martin[178]
Usar substantivos para os nomes das classes e iniciar os nomes destas com maiúscula; incluir verbos nos nomes dos métodos/funções.	Martin[178]
Cada bloco de um *if* deve estar numa linha.	Martin[178]
Reduzir *if/switch* a favor do polimorfismo.	Martin[178], Pepperdine[178]
As funções devem encontrar-se pela ordem em que são invocadas, a função que chama deve estar antes da função que é chamada.	Martin[178]
Evitar as sentinelas como parâmetros.	Martin[178]
Utilização criteriosa de comentários.	Martin[178]
Não comentar *bad code*, reescrevê-lo.	Martin[178], Kerninghan&Plaugher[178]
Cada ficheiro não deve ser demasiado grande.	Martin[178]
Cada ideia no código deve ser separada por uma linha em branco.	Martin[178], Kerninghan&Plaugher[178]

Descrição	Fonte
Separar os operadores com um espaço branco.	Martin[178]
Ideias relacionadas entre si devem estar verticalmente próximas.	Martin[178]
As linhas devem ser curtas. Sugere 120 carateres como máximo.	Martin[178]
Quebrar expressões maiores, usando intermédias.	Martin[178]
Evitar a utilização de *set* e *get* para todos os atributos.	Martin[178]

3.3 Estudos

A revisão da literatura que agora se apresenta pretende descrever o estado da arte na área da legibilidade do software no que se refere às práticas na codificação que possam influenciar a legibilidade do código-fonte.

3.3.1 Introdução

A revisão da literatura foi realizada com base em bibliotecas digitais de artigos científicos publicados em revistas e conferências científicas. A revisão foi realizada em 3 fases. Foi feito um levantamento inicial que permitiu identificar um conjunto de práticas de legibilidade. Em seguida foi realizada uma segunda revisão da literatura. Esta revisão seguiu essencialmente um processo designado por *bola de neve* (*snow-ball*) usando as listas de referências dos artigos que iam sendo consultados bem como as respectivas referências que citavam esses artigos. Este processo encontra-se facilitado pela existência de bases de dados digitais com opção de consulta quer das referências quer das citações, como, por exemplo, a da ACM, a da IEEE, a da Springer e do motor de pesquisa especializado para fins académicos, *scholar*, da Google. Por último, em virtude do objectivo do trabalho foi também realizada uma revisão sistemática para identificação de estudos empíricos envolvendo práticas de legibilidade. Esta revisão sistemática encontra-se descrita na secção seguinte. O processo *bola de neve* e a revisão sistemática podem ser usadas conjuntamente, com o *bola de neve* como primeira estratégia de pesquisa, está de acordo com as recomendações de [296].

Para o caso mais específico do termo "legibility" foi feita uma pesquisa através dos termos "software legibility" e "source code legibility" nas bases de dados digitais da ACM e IEEE. A pesquisa permitiu verificar que o termo não é actualmente usado no sentido de "readability" (e.g. [33]) e, como tal, o resultado dessa pesquisa prospectiva não consta da

presente revisão. Uma excepção ocorre no modelo de qualidade do software apresentado por Boehm et al [30, p.595] o qual inclui um conjunto de características de qualidade. Note-se que se trata de um artigo de 1976. Entre essas características, encontra-se a "maintainability" que se divide em "testability", "understandability" e "modifiability". Por sua vez, a "understandability" depende de 4 outras características, sendo uma delas a "legibility". "Legibility" é aí definida como a medida em que a função do código é facilmente discernida pela leitura do código [30, p.604].

A presente revisão procurou focar-se em estudos empíricos, portanto essencialmente em artigos científicos. No entanto, também se incluíram estudos descritos em livros e, por outro lado, um ou outro artigo que ainda que não descrevesse um estudo mas se foi considerado relevante para o actual estado da arte da legibilidade.

A apresentação de cada artigo procura realçar as práticas de legibilidade aí constantes. Por ter sido essa a principal preocupação e não o artigo em si, as apresentações variam entre si. Adicionalmente, nem todos os artigos apresentam o mesmo nível de complexidade, o que se reflecte frequentemente numa apresentação mais extensa para os artigos mais complexos, com o intuito de facilitar ao leitor a eventual leitura do artigo em causa, mas que também revela ter existido uma maior dificuldade na sua compreensão.

3.3.2 Descrição da Revisão Sistemática

Esta secção descreve a última parte da revisão da literatura, a qual foi realizada como uma *mapping study*. Uma *systematic mapping study* ou simplesmente *mapping study (MS)* define-se como [147, p.vii]: *"broad review of primary studies in a specific topic area that aims to identify what evidence is available on the topic."*

Aqui serão pesquisados estudos empíricos sobre legibilidade do software. O processo seguido baseado em [147] e [230] é apresentado em seguida.

Segunda revisão da literatura (*Systematic Mapping Study (SMS)*):

Objectivo:

- Quais as práticas de programação que afectam a legibilidade do código;
- Procurar estudos empíricos sobre práticas de legibilidade.

Critérios de inclusão:

- Estudos que analisem as práticas na legibilidade do código-fonte;
- Ser um estudo empírico.

Critérios de exclusão:

- Não envolve práticas;
- Não é um estudo;
- Não analisa impacto de qualquer prática sobre a legibilidade do código.

Após tentativa de várias *queries*, optou-se por uma *query* que pesquisasse a nível do corpo do texto e nas palavras-chave. A *query* escolhida foi a 12ª e é a seguinte:

Query 12: content.ftsec:(ethnography “case study” experiment empirical survey “research method” “research approach” “post-mortem analysis” “historical study” “longitudinal study” “systematic literature review” “mapping review” “mapping study” “systematic review” “quasi-experiment” “observational study” simulation “trend study” “participatory research” “pilot study” “cross-sectional” “controlled study” “structured interview” “semi structured interview” “expert opinion” “action research” “focus group” “naturalistic enquiry” ethnography “ex post facto research”) AND

content.ftsec:(“software readability”“code readability” “readable program” “readable software” “readable code” “readability of program” “readability of software” “readability of code” “readability of the program” “readability of the software” “readability of the code” “readability practice” “program readability” “program clarity” “code clarity” understandability “programmming style”) AND

keywords.author.keyword:(“software readability” “code readability” “readable program” “readable software” “readable code” “readability of program” “readability of software” “readability of code” “readability of the program” “readability of the software” “readability of the code” “readability practice” “program readability” “program clarity” “code clarity” “quality code” “programmming style” “code comprehension” “software comprehension” “program comprehension” “code understandability” “software understandability” “program understandability” “code understanding” “software understanding” “program understanding”)

Resultado da *query* 12: 313 entradas.

3.3.3 Revisão

Como Dijkstra fez questão de realçar ainda em 1965 [76, **?**] a programação é em primeiro lugar uma actividade de pessoas. Aliás, Dijkstra fala inclusivamente em código elegante inspirado pela ideia de beleza do método do matemático George Boole e em contraponto com a rejeição de elegância dos métodos por parte do físico Ludwig Boltzmann. É muito possível que tenha sido a primeira vez que o termo foi utilizado. Na pesquisa efectuada, os primeiros estudos empíricos acerca do desempenho de programadores terão sido publicados em 1966 [86][6], em 1967 [236] e depois em 1968 [227]. Neste último, ao compararem o desempenho na depuração com acesso *online* versus *offline* os resultados mostraram a superioridade do trabalho *online*, mas também revelaram a importância das diferenças individuais no desempenho. No entanto, terá sido só após a publicação por Gerald Weinberg do seu livro "The Psychology of Computer Programming" [289] que a ideia de a programação ser uma actividade humana foi reconhecida em toda a sua amplitude de forma generalizada [247].

Melhorar a legibilidade é apontada como uma das razões para o desenvolvimento de linguagens de programação de mais alto nível. Essa preocupação foi referida logo na primeira conferência de engenharia de software (ver [198]). Acerca da concepção de linguagens de programação, desde há muito que vários autores têm definido a legibilidade como um dos critérios a ter em consideração. Por exemplo, Hoare[122, pp.11-2]:

"*Since in principle programs should be read by others, or reread by their authors, before being submitted to the computer, it would be wise for the programming language designer to concentrate on the easier task of designing a readable language to begin with.*"

Houve outros autores que apontaram esse critério, e.g. [288].

O primeiro artigo relativo ao estudo da legibilidade do código que foi possível encontrar data de 1967 [223]. Também em 1967, no fórum de PL/1 da ACM foi apontado [225, p.22]: que a razão para linguagens de alto nível existirem é a facilidade de alteração e manutenção, e não a de produzir código mais eficiente; que os programas têm de ser

[6]Não foi possível obter este relatório.

legíveis; e que a facilidade de leitura devia ser um dos critérios a considerar em qualquer inovação à linguagem PL/1.

A importância de se obter código bem legível é clara em vários artigos clássicos compilados em [302] do início da década de 70 do séc. XX, (e.g. a substituição do *goto* por *while* melhoraria a legibilidade, num artigo de 1971, p.71, ou a definição de programação estruturada como regras que melhoravam a legibilidade e a manutenibilidade, em artigo de 1972, p.65). Também orientações para a codificação começavam a incluir referências à legibilidade. Por exemplo, as orientações originais de Kernighan e Plauger de 1974 [137] ainda não afirmavam explicitamente a preocupação com a legibilidade, uma situação que se alterou com a segunda edição de 1978 [138]. Já Yourdon revelava essa preocupação em 1975 [302].

Mais a título de curiosidade, pode ser encontrada em [92] uma revisão de medidas da complexidade do software, com enfoque na "ciência do software" de Halstead [113], como forma de medição da compreensão do código, e dos estudos realizados durante a década de 70 do séc. XX.

Ainda na década de 70 do séc. XX foram realizados diversos estudos empíricos que não são aqui apresentados (e.g. [106]) por se centrarem na concepção de linguagens, mais concretamente, por dizerem respeito ao estudo da compreensão do código utilizando determinadas estruturas das linguagens. Por exemplo, $if-goto$ versus a estrutura imbricada $if-then-else$; ou linguagens tipadas e não tipadas. Relativamente ao primeiro exemplo, foi mostrada por Green, a superioridade do $if-then-else$ sobre o *goto* tanto com alunos (num estudo anterior), como com programadores com experiência. Ainda nesse artigo, de destacar ter-se verificado ser a utilização de $if-then-else$ com redundância (repetição da condição no ramo), a versão com melhores resultados de entre as variantes experimentadas. Uma revisão detalhada e crítica desses estudos pode ser encontrada em [243]. Pela revisão, as estruturas $if-then-else$ permitiram resultados superiores ao *goto* quanto à compreensão do código.

Weissman, 1974 [290] O autor conduziu diversos estudos experimentais, alguns publicados originalmente em 1973, para determinar factores que influenciassem o entendimento e manutenção dos programas. Testou a interacção entre a indentação e os comentários,

usando PL/I e a indentação de 2 espaços [191, p.861]. A preocupação do autor surge devido ao facto de muitas das ideias acerca do assunto não serem suportadas por evidência científica, i.e, não decorrerem de estudos científicos, mas sim de opiniões, mais ou menos bem fundamentadas. Como à época ainda eram muito raros os estudos experimentais no desenvolvimento de programas, o trabalho do autor, pela sua quantidade (10 estudos experimentais) e sistematização, ganhou ainda maior relevância.

O artigo lista vários factores que o autor considera afectarem a complexidade dos programas. Dividiu estes factores em 4 grupos. Os factores que considerou como podendo influenciar a (p.26) "legibilidade e aparência" de um programa e agrupou-os sob o título "formato do programa". Os factores são os seguintes:

1 - Comentários com significado;

2 - Localização das declarações;

3 - Colocação de parágrafos nas listagens de código dos programas;

4 - Escolha dos nomes das variáveis;

5 - Redeclaração de um nome de variável num âmbito interior;

6 - Uso de constantes simbólicas.

Dos estudos experimentais realizados, os primeiros dois envolveram os comentários, a colocação dos parágrafos e as mnemónicas nos nomes das variáveis. Foram considerados apenas 2 níveis para cada factor: presente ou ausente. Usaram 2 programas. A indentação foi de 2 espaços. No primeiro estudo participaram alunos do 1º ano. O investigador concluiu que tais alunos não eram adequados devido aos seus poucos conhecimentos das construções da linguagem.

Tal problema levanta uma questão que o autor não discute. O conhecimento necessário ao programador não é apenas o do domínio do problema, mas também o do domínio da programação. Pensando-se num texto literário, a situação tem algum paralelo com a do escritor que tem de saber escrever e conhecer a história. Tal necessidade parece evidente e existe evidência que aponta para a necessidade de contemplar o conhecimento de ambos domínios para a compreensão de programas (e.g. [241]).

O segundo estudo foi idêntico ao primeiro mas com 48 alunos do 2º ano do curso. Os programas foram atribuídos aleatoriamente aos participantes. Foi pedido a cada aluno para ler o programa e fazerem a sua traçagem em 30 minutos, o máximo que fossem capazes. No fim, foi-lhes pedido para avaliarem entre 0 e 9 quanto ao entendimento do programa. No caso de programas com comentários, a auto-avaliação registou resultados superiores relativamente aos programas sem indentação, mas as tarefas de modificação, simulação e questionário não registaram melhorias.

Num dos estudos experimentais, Weissman (citado por [92, p.14]) testou a compreensão por parte de alunos de programas com e sem funções. A partir de 2 programas criou para cada um 3 versões com funções e outras 3 sem funções, portanto num total de 6 versões. Foram dados 5 minutos para cada aluno poder estudar o programa que lhe fora atribuído e auto-avaliar a sua compreensão do programa. Em seguida, foram dados mais 15 minutos de estudo, seguidos de um teste e de nova auto-avaliação. Finalmente, mais 20 minutos e novo teste. Foram então produzidos 4 resultados. Em [92] foram calculadas as correlações entre a medida de esforço de Halstead e os resultados obtidos por Weissman. A correlação foi significativa para a situação em que o estudante teve mais tempo para analisar o programa. Tratou-se de uma correlação negativa, o que se compreende porque mais esforço implica supostamente maior dificuldade na compreensão.

Weissman descobriu que o principal efeito da indentação não era significativo em nenhuma da suas medidas, mas que já era significativo em interacção com os comentários. Quando os comentários estavam ausentes nos programas, a indentação ajudava apenas ligeiramente. Quando os comentários estavam presentes, a indentação influenciava de uma forma drástica. Weissman ficou surpreso com esses resultados negativos e tentou explicá-los pelo facto de que os programas tinham que ser divididos dentro dos limites da página, o que não tinha acontecido. Além disso, os programas continham instruções *Goto* que em nada ajudavam na indentação. Enquanto Weissman pensava que isso poderia explicar parcialmente os efeitos negativos da indentação, não sentia o mesmo relativamente à interacção com os comentários. Noutra experiência, testou a interacção entre a indentação e o controlo de fluxo. Esta experiência suporta a hipótese de que a indentação auxilia na compreensão do programa e na satisfação do utilizador, uma vez que na maioria dos casos melhorou o desempenho (às vezes significativamente) e não prejudicou significativamente

o desempenho [191, p.861].

Como factos também já conhecidos, o autor afirma que é importante que os programas estejam bem documentados e bem estruturados.

Ledgard, 1976 [166] O artigo tem como tema o que os autores designam por uma das fases do ciclo de desenvolvimento do software, a escrita de programas.

Trata-se de um dos primeiros trabalhos que surgiram motivados pela legibilidade. Na altura já existiam várias propostas de orientações de codificação (e.g. [137]). Nas palavras dos autores (p. 601), a "legibilidade de um programa é muito mais importante que a capacidade de escrita". Também referem que a manutenção dos programas tem um custo tão elevado, que é necessário um esforço significativo em reduzi-lo. Os autores propõem diversas regras para a escrita de programas em COBOL (norma de 1974) divididas por tipos. Destacam-se: a definição de nomes padrão típicos em programas COBOL a serem usados no projecto e que devia ser concretizada antes da codificação para, posteriormente, não se perder tempo na sua escolha; a definição dos espaçamentos; a utilização de hífen na separação das partes lógicas de nomes de variáveis, específicas do utilizador/cliente; e a definição de uma uniformização na identificação de partes do programa. Existem outras relacionadas com a legibilidade, mas muito específicas do COBOL.

Essencialmente tratou-se de um esforço de normalização da utilização do COBOL com preocupação com a legibilidade.

Clifton, 1978 [53] Para Clifton, um dos atributos mais importantes dos programas é a legibilidade. Segundo o autor, um programa fácil de ler e de entender será mais fácil de testar, manter e modificar. Também afirma que apesar de a legibilidade dever ser um dos benefícios da programação estruturada, a utilização de estruturas de controlo com imbricação elevada, faz com que os programas se tornem difíceis de ler e consequentemente de entender.

Por isso, no artigo, é proposta uma técnica para tornar os programas estruturados mais fáceis de ler. Essa técnica designa-se por conector de linha e consiste em criar uma linha de ligação entre o início da estrutura e o seu fim. Por exemplo, para o caso do *do-while*, seria criada uma linha de ligação entre o *do* e o *while*. Estes conectores poderiam ser

gerados automaticamente nas listagens de código e permitiriam ajudar o leitor a visualizar a estrutura lógica dos programas. O autor afirma que devem ser usados em combinação com a indentação. O conector de linhas seria uma opção disponível no ambiente de desenvolvimento. Hoje em dia muitos ambientes de desenvolvimento já possuem esta característica.

Finalmente é criticada a possibilidade de se restringir o grau de indentação e o número de linhas de uma função por poder contribuir para soluções pouco naturais e menos legíveis ao fazer aumentar o número de procedimentos. Essa crítica é interessante, na medida em que, actualmente, como se viu anteriormente para a revisão da prática actual (e.g. [298, 178]), se defende que os métodos sejam curtos e satisfaçam uma única finalidade. Mas pode acontecer que em algumas situações possam ocorrer soluções de leitura mais difícil devido à proliferação de funções.

Curtis, 1979 [60, 131] Foram realizados dois estudos experimentais. No primeiro estudo desenvolveram uma fórmula que avaliava seis características de estilo de programação para analisar a legibilidade dos programas. As características foram as seguintes:

- comentários;
- indentação;
- linhas em branco;
- tamanho dos nomes das variáveis;
- operadores aritméticos;
- instruções *goto*.

No segundo estudo foram avaliados aspectos metodológicos. Neste estudo experimental, os autores mediram a complexidade psicológica das tarefas de manutenção do software com as métricas de análise da complexidade dos programas Halstead e McCabe. Os valores das métricas foram comparados com o desempenho do programador para duas tarefas de manutenção. Os programadores estudaram os programas com vários níveis de comentários e nomes de variáveis com mnemónicas e estruturas de controlo. Como conclusão, não foram encontradas diferenças nos desempenhos nos diferentes níveis dos estilos de programação referidos.

Sheppard, 1979 [244] Existe um consenso geral que o código bem estruturado é mais fácil de entender e de modificar. Tem, no entanto, havido desacordo relativamente às estruturas de controlo de fluxo. Muitos programadores não estão de acordo com as três estruturas de Dijkstra: a sequência, decisão e repetição. Relativamente às mnemónicas como nomes das variáveis também tem havido alguma controvérsia. Com os comentários também não existe consenso.

Os autores realizaram três estudos. No primeiro, que deram o nome de compreensão, examinaram o efeito da codificação estruturada e dos nomes das variáveis com mnemónicas. Este primeiro estudo experimental envolveu 36 participantes. Como resultado deste estudo verificaram que o controlo de fluxo afecta o desempenho. O código menos estruturado foi o mais difícil de compreender mas não tiveram efeito estatisticamente significativo. As mnemónicas nos nomes das variáveis também não tiveram efeito significativo.

No segundo estudo, ao qual deram o nome de modificação, examinaram a influência da codificação estruturada e os estilos de comentários nas tarefas de modificação. Neste estudo participaram 36 programadores. Aqui os participantes teriam de modificar os programas. Como resultados deste estudo, também não houve diferença estatisticamente significativa em termos de controlo de fluxo estruturado.

No terceiro e último estudo, chamado de depuração, examinaram o efeito do comprimento dos programas no desempenho da depuração. Neste último estudo participaram 54 programadores. Aqui, os participantes tinham de localizar e corrigir erros num programa em Fortran.

As conclusões encontradas foram as seguintes: O principal benefício do código estruturado no desempenho é a sua visibilidade. Os comentários e as mnemónicas dão ajuda limitada, pelo menos em programas curtos. Parece que os resultados não mostram vantagem de mnemónicas.

Jorgensen, 1980 [131] No artigo é referido que a legibilidade e a modificabilidade (facilidade de modificação) são dois tópicos muito importantes na manutenção do software e que os custos da manutenção são cerca do dobro dos custos de desenvolvimento de software. É portanto importante conhecer quais os factores que dificultam a legibilidade e a modificabilidade dos programas.

O objectivo desta investigação foi determinar de que modo será possível expressar a legibilidade e a modificabilidade dos programas de computador em termos de características do estilo de programação. Visto de outra forma, pretende saber quais destas características poderão influenciar a legibilidade e modificabilidade dos programas. Para isso, é apresentada uma metodologia que permite detectar características de legibilidade e de modificabilidade. Neste contexto interessa apenas a legibilidade. Como se verá, estas características correspondem ao que neste texto se designa por práticas de legibilidade. Neste artigo é apresentada a definição do conceito de legibilidade em linguagem natural dada pelo Bjornsson [131]: "A legibilidade de um texto é a soma dos factores estilísticos que fazem o texto mais ou menos entendível para o leitor."

A metodologia foi testada através de 2 estudos experimentais. Foram considerados dois tipos de estilos de programação. O primeiro tipo pode ser avaliado por um programa. A estas características chamaram características **C**.

O segundo tipo de características requer avaliação humana, como por exemplo, perceber o propósito de um nome. A essas características chamaram características **H**. Foram consideradas as características definidas por Weissman [290] (vistos acima) que são as seguintes:

H1 - Escolha do nome da variável

H2 - Estrutura do programa

H3 - Uso de comentários

H4 - Escolha das estruturas de controlo

H5 - Inserir parágrafos nas listagens

No primeiro estudo foram analisados, quanto à legibilidade, 48 programas distintos por 10 peritos informáticos e usando uma escala de 1 a 7, em que 1 era excelente e 7 muito pobre. A média destes valores para cada programa resulta num índice de legibilidade R. Em seguida, definiram 57 características C. Em terceiro lugar, mediram automaticamente as 57 características nos 48 programas.

Finalmente, após análise estatística, o resultado foi uma fórmula de aproximação de legibilidade (equação de regressão linear), R', contendo seis características C relacionadas

com o estilo de programação. A selecção da aproximação foi um compromisso por parte dos autores que consideraram que a variância residual do desvio $R - R'$ é muito pequena. As seis características são as seguintes:

C1 - Extensão dos comentários (+)

C2 - Extensão de brancos na margem esquerda (+)

C3 - Extensão das linhas em branco (+)

C4 - Comprimento médio do nome das variáveis (+)

C5 - Número médio de operadores aritméticos por 100 linhas (-)

C6 - Número médio de instruções *goto* (-)

A fórmula resultante é a seguinte:

$$R' = 5.7 - 0.061 * C1 - 0.047 * C2 - 0.023 * C3 - 0.15 * C4 + 0.0065 * C5 + 0.21 * C6 \quad (3.1)$$

Os valores de R' variam entre 1 e 7 e quanto mais próximos forem de R melhor porque explicam melhor R. Nos resultados acima, o sinal '+' significa que quando esta característica aumenta, a legibilidade aumenta. O sinal '-' significa o contrário, quando a característica aumenta, a legibilidade diminui. Como resultados do primeiro estudo experimental referem que os comentários em geral ajudam na compreensão dos programas e que C5 deverá ter significado apenas em programas com grande número de operadores aritméticos.

O segundo estudo experimental teve em atenção os aspectos metodológicos, por exemplo, variações causadas pela natureza da tarefa. Assim, foram seleccionados 10 programas que realizam a mesma tarefa e analisados por 100 estudantes de informática. Cada programa foi analisado separadamente por 10 estudantes. As 6 características encontradas no estudo 1 foram determinadas também no estudo 2. Também foram classificadas pelos estudantes as 5 características H. Foram também colocadas 9 questões de avaliação da compreensão acerca dos programas. Usaram novamente técnicas de regressão linear para análise dos resultados.

Concluíram que, apesar dos cuidados que os autores dizem ter tomado na realização deste estudo experimental, este não forneceu resultados similares com a fórmula do estudo 1. Consequentemente, a fórmula 3.1 necessitaria ser mais bem estudada. O segundo estudo confirmou apenas os comentários como sendo relevantes para a legibilidade segundo os critérios definidos no artigo, como seja a utilização de uma fórmula de regressão linear. O estudo 2 forneceu também informação sobre os pares C e H que envolviam o mesmo aspecto estilístico. Apenas C1 e H3 obtiveram uma correlação alta, de 0.84, interpretado como tendo mais significado os comentários mais extensos do que esparsos. Por último, verificou-se existir uma correlação forte, 0.85, entre o julgamento da legibilidade pelo aluno através de uma escala de 7 pontos e a avaliação da compreensão dos programas através de 9 questões acerca dos programas.

As 6 características C reflectem apenas quantidades. Por exemplo, C é o número total de caracteres em comentários dividido pelo número total de caracteres, incluindo espaços, e depois multiplicado por 100 (o seu valor médio para os 48 programas foi 7.2).

Em conclusão, este artigo permite mostrar a importância de algumas características para a legibilidade, como é o caso dos comentários, mas não permite saber a forma como essas características devem ser usadas.

Chaudhary, 1980 [48] O autor considera que as medidas de complexidade do código que já haviam sido propostas não correspondiam à globalidade da complexidade psicológica de um programa. Nesse sentido, identificaram cinco propriedades de um programa que afectam o seu entendimento e corresponderiam a essa complexidade: tamanho, estrutura de controlo, estruturas de dados, estrutura de execução e significância (*meaningfulness*).

Estas propriedades permitiriam relacionar o programa com o domínio do problema e assim reduzir o esforço de reconhecimento da estrutura do programa. Ainda, segundo os autores, essas propriedades correspondem a várias características dos programas, como, por exemplo, comentários, nomes, dados e organização do controlo, indentação e formatação, a que conjuntamente chamaram significância.

No artigo, é descrito um estudo experimental e referido um outro estudo experimental anterior realizado pelo primeiro autor. De acordo com o estudo anterior, as quatro pro-

priedades estudadas, estrutura de controlo, tamanho, estruturas de dados e estrutura de execução, contribuem para a complexidade do programa, por essa ordem, decrescente de influência.

O estudo descrito visou estudar o efeito da significância, em termos de nomes das variáveis. O estudo compreendeu 105 estudantes atribuídos aleatoriamente aos 5 programas escolhidos. O teste envolveu os participantes a escreverem uma descrição acerca do que o programa fazia e como isso era feito, e posteriormente foi-lhes pedido para reconstruírem a funcionalidade do programa e ainda detectar os defeitos nos programas.

Os resultados revelaram a influência dos nomes, mais sugestivos e compridos, com até 6 caracteres, relativamente aos de 2 caracteres que eram neutros, mas o programa que serviria de controlo foi o segundo em que os alunos obtiveram melhores resultados o que não era esperado. Também se verificou que a utilização de vectores não prejudicava a compreensão. O resultado inesperado foi justificado pelo maior esforço que os alunos tiveram de realizar na tentativa de obterem um bom resultado, visto o exercício ter peso na classificação final.

No entanto fica uma dúvida quanto ao artigo, pois os autores referem nas conclusões que o resultado para o tamanho não foi significativo.

Shneiderman, 1980 [248] Shneiderman descreve diversos estudos que realizou, alguns já de 1974, sendo vários relacionados com práticas de legibilidade, mas tendo como enfoque a análise da compreensão.

Relativamente à importância dos comentários, Shneiderman refere 4 estudos anteriores com resultados contraditórios. Por não ter sido possível localizar os documentos originais dos estudos, estes são aqui abordados brevemente.

O primeiro, não publicado, revelou que os comentários não melhoravam a compreensão do código por parte de estudantes. O segundo é um estudo-piloto de 1970 que mostrou que comentários incorrectos dificultam a compreensão do código relativamente aos correctos, mas que a compreensão era ainda mais rápida quando não existiam comentários. Um terceiro, são na realidade os vários estudos experimentais de Weissman, descritos acima, com resultados favoráveis ao uso de comentários significativos e bem colocados relativamente a comentários muito breves. O quarto estudo revelou que os participan-

tes obtinham resultados significativamente melhores em questões acerca da compreensão de programas quando estes possuíam bastantes comentários do que em programas com apenas um comentário simples no início.

Após a revisão desses quatro estudos, Shneiderman apresenta um estudo experimental que realizou posteriormente. Nesse estudo com 62 alunos foram usados 2 programas com tipos de comentários distintos. Num programa existia um comentário inicial (HI) de mais alto nível descrevendo o programa e a sua finalidade. No segundo existiam 19 comentários curtos (LO), em diferentes linhas do programa. Foram solicitadas 3 alterações a cada 1 dos programas e posteriormente que se recordassem das linhas de código. Verificou-se que o grupo de alunos com os comentários HI obtiveram melhores resultados (estatisticamente significativos) em ambas as tarefas.

O autor concluiu que os comentários HI ao descreverem a finalidade facilitam a passagem da informação da memória de longo prazo para a memória de trabalho e confirmam que os comentários devem descrever o significado e a funcionalidade do código e não ser uma mera repetição do código. Também refere que os comentários não são armazenados na memória semântica de longo prazo. Sugere ainda que o efeito negativo dos comentários terá a ver com o facto de interromperem a leitura do código e aumentarem a dimensão do código obrigando à "mudança" de página.

Em conclusão, os comentários devem referir-se ao domínio do problema. Aliás, também em [48] foi mostrado a importância da referência ao domínio do problema tanto para os comentários como para os nomes dos identificadores.

A maior parte dos programadores continua a preferir a utilização de alguns comentários. Por esta razão, é inportante criar a ligação entre um comentário e a parte de código ao qual pertence. Esta ligação irá dar uma opção de remoção dos comentários do código, ou então sobrepô-los no código, para ambos tipos de visualização.

Shneiderman também analisou a importância dos nomes para a compreensão. Primeiramente, refere um estudo em que os resultados de compreensão, usando várias questões, foram superiores para nomes sem usar mnemónicas (sem significado) do que usando mnemónicas (com significado). O resultado pode ser explicado pela existência de um parágrafo que descrevia as variáveis. Em segundo lugar, descreve estudos experimentais anteriores realizados por Weissman.

Num dos seus estudos, foram analisados programas em FORTRAN que usavam como nomes de variáveis mnemónicas, por exemplo, ISUM, e não-mnemónicas, como I1, e verificou-se que os grupos de estudantes inexperientes obtinham melhores resultados na compreensão no caso dos programas com mnemónicas. Um segundo estudo com alunos de anos intermédios consistia em detectar e corrigir um defeito em versões de programas em PASCAL com e sem mnemónicas, respectivamente. Os resultados com não mnemónicas foram inferiores, mas não significativos. Em termos cognitivos, Shneiderman considerou que esses nomes representam uma sobrecarga devido à necessidade que o programador tem de encontrar o seu significado, na memória semântica.

A utilização de mnemónicas para os nomes das variáveis dá a impressão de ajudar na compreensão dos programas. No entanto, as mnemónicas têm que ser de tal modo que consigam adicionar informação semântica relevante ao código. É muito provável que mnemônicas diferentes tenham significados diferentes para programadores diferentes.

A substituição sistemática dos nomes das variáveis de acordo com as especificações do programador permitirá adaptar o código à sua preferência pessoal. Tendo mnemônicas significativas aliviaria a memória de "curto prazo" do programador, tornando a sua tarefa mais fácil.

Por último, Shneiderman analisou a indentação do código. Começou por rever 2 estudos que mostravam que a indentação não melhorava a compreensão de programas, respectivamente, em PL1 e FORTRAN. Assim, realizou um estudo experimental idêntico ao segundo para os nomes, mas desta vez de programas, também em PASCAL, com indentação e sem indentação. Não foram encontradas diferenças significativas. O autor concluiu que o benefício da indentação não é claro e que programas com indentação profunda poderão implicar dividir a instrução. Sugere a utilização de linhas em branco para demarcar unidades funcionais.

Shneiderman não o refere, mas para além da eventual necessidade de dividir a instrução por mais do que uma linha, o estender o código em largura também poderá dificultar a sua leitura ao obrigar os olhos a percorrerem uma distância superior. Mais, actualmente as linhas podem "esconder-se" para lá do limite físico do ecrã, isto caso não se opte pela sua quebra, o que também pode dificultar. Isto tudo faz supor que devam existir limites para a indentação.

Sheil [243] questiona as conclusões de Schneiderman acerca da indentação. Refere que a indentação não é um atributo inerente de um programa. Uma boa indentação para uma pessoa pode ser fraca para outra. Não se pode concluir que a indentação seja uma técnica ineficaz enquanto não for demonstrado.

Também [131] refere que alguns estudos de programas curtos mostram que os comentários no código interferem no processo de compreensão e requerem uma maior filtragem na leitura do código. Se este não está actualizado, pode dar uma ideia errada do indicado e consequentemente causar erros na representação semântica do código. Os estudos experimentais com programas mais longos e afinal os mais realistas, não são reportados por Shneiderman. Pode bem ser o caso de que a importância dos comentários aumenta com o tamanho do programa.

Tanto nos estudos de Schneiderman como nos de Sheppard, a dimensão dos programas é reduzida. Por isso, e relativamente aos nomes, é razoável pensar que os nomes com mnemónicas serão de utilidade reduzida em programas curtos, nos quais haverá menos possibilidade de confusão entre nomes, ou na sua memorização. O contrário ocorrerá em programas maiores ou naqueles em que haja menor familiaridade [243].

Por último, o autor refere que segundo Hansen, nomes mais longos são mais apropriados para as variáveis que são utilizadas raramente ou variáveis globais e que os nomes mais curtos são mais apropriados para as variáveis locais ou variáveis de controlo de ciclo.

Woodfield, 1981 [298] Este autor realizou um estudo experimental para investigar como os diferentes tipos de modularização e os comentários se relacionam com a capacidade dos programadores compreenderem os programas. Foram dadas 8 versões diferentes de um programa em FORTRAN a 48 programadores com experiência. O estudo foi realizado de forma independente por cada programador e teve uma duração de 75 minutos. Os participantes foram divididos aleatoriamente em 8 grupos, e foi dado, a cada um, uma listagem de programa. As oito versões tinham quatro tipos de modularização:

- Monolítico - programa com apenas um módulo lógico e físico;
- Funcional - programa onde cada módulo lógico se encontra num módulo físico separado;

- Super - programa separado em vários módulos físicos pequenos, cada um com cerca de 5 a 10 linhas;
- Tipo de dado abstrato - programa com modularização funcional em que um módulo físico é composto por vários módulos lógicos.

Para cada uma destas quatro versões, existia uma versão com comentários por módulo físico e outra versão sem comentários. Às variáveis foram dados nomes sem significado e retirada toda a indentação. Os participantes tinham posteriormente de responder a 20 questões relativas ao programa num período máximo de 60 minutos.

Os resultados dos grupos com programas comentados foram significativamente superiores aos grupos com programas sem comentários. Quanto à modularização, as que permitiram melhores resultados foram, a modularização com o tipo de dado abstracto e a modularização funcional. Concluíram que a modularização e os comentários influenciam a compreensão dos programas. No entanto, questionam se o valor dos comentários se manteria caso a indentação não tivesse sido retirada.

O artigo lista um programa modularizado funcionalmente. Os comentários do exemplo variam entre, aproximadamente, as 2,5 e as 4 linhas. Uma linha de comentário possui cerca de 60 caracteres.

DeYoung, 1982 [73] Um medida computacional da legibilidade poderá ser útil aos desenvolvedores de programas durante a codificação e para aqueles que assumem a responsabilidade pela manutenção do software desenvolvido por outros.

Inicialmente os autores seleccionaram um conjunto alargado de características de código para avaliação automática. Após uma pré-avaliação usando o analisador automático concluíram que algumas não tinham valor preditivo da legibilidade e por isso foram descartadas. As características restantes a serem usadas no estudo e a medir pelo analisador automático são apresentadas em seguida.

1. Variáveis:
    - $LENGHT$ - Média dos comprimentos dos nomes das variáveis;
    - $WIDTH$ - Número médio de linhas entre a primeira utilização e a última utilização de uma variável;

- VAR = média de $LENGHT/WIDTH$ - Comprimento médio normalizado das variáveis. Os autores assumem como positivo o comprimento do nome de uma variável variar de acordo com a dimensão do seu âmbito. No entanto, não é fácil medir o grau em que as mnemónicas são utilizadas.

2. Comentários e Complexidade estrutural:

   - NC - Volume dos comentários em caracteres;
   - $CYCLO$ - Soma das ramificações dos programas mais 1. Corresponde ao número de estruturas de controlo (IF, WHILE, e restantes), ao número de operadores lógicos usados nas condições e ao número de alternativas $CASE$ usadas na estrutura de controlo condicional.

3. Medidas de Halstead (as 4 primeiras) e derivadas:

   - $N1$ - Número total de operadores num programa;
   - $N2$ - Número total de operandos num programa;
   - $NU1$ - Número de operadores únicos;
   - $NU2$ - Número de operandos únicos;
   - $N = N1 + N2$ - Comprimento medido do programa;
   - $NP = NU1 * \log 2NU1 + NU2 * \log 2NU2$ - Comprimento estimado;
   - $L = 2 * NU2/NU1 * N2$ - Nível de implementação estimado;
   - $V = N * \log 2(NU1 + NU2)$ - Volume do programa;
   - $ERR = |N - NP|/NP$ - Erro absoluto do comprimento estimado;
   - $LEVEL = L * *2 * V$ - Nível da linguagem utilizada;
   - $EFFORT = V/L$ - Esforço mental requerido para compreender o programa.

4. Comprimento do programa:

   - NSL - Número de linhas com instruções.

Em seguida, após reverem a literatura consideraram os parâmetros seguintes como passíveis de avaliação através do julgamento humano.

- *IDNTF* - Adequação dos identificadores;
- *INDNT* - Uso adequado da indentação;
- *CMNTS* - Conteúdo dos comentários adequado;
- *PRCDR* - Utilização adequada de procedimentos;
- *STRCT* - Utilização adequada das estruturas de dados;
- *CNTRL* - Controlo de fluxo bem estruturado;
- *STYLE* - Utilização de características estilísticas úteis ao programa (pessoal).

Foram examinados vários programas usando um analisador automático. Os seus resultados foram depois comparados com os obtidos por julgamento humano, neste caso de 21 alunos mais licenciados. Cada um analisou 8 programas de um total de 30 programas. Todos os programas tinham a mesma função. Utilizou regressão múltipla para encontrar os factores que melhor predizem a legibilidade humana. A regressão foi repetida várias vezes até ser encontrada a melhor combinação de parâmetros preditores. Estes preditores foram depois usados num conjunto de programas não usados na análise de regressão.

As duas características que os autores consideraram excelentes preditores da legibilidade foram o número de linhas com instruções (*NSL*) e o comprimento médio normalizado das variáveis (*VAR*), com pesos de -0,499 e 0,295, respectivamente. E das características de julgamento humano, a que permitiu melhores resultados significativos quando usada conjuntamente com as duas características anteriores foi a característica controlo de fluxo (*CNTRL*).

Através da leitura dos resultados parece difícil extrair informação quantitativa mais útil que pudesse servir como parâmetro para o programador. Por exemplo, quanto ao comprimento dos nomes, seria muito útil a indicação de pelo menos um intervalo.

Elshoff, 1982 [84, 85] Elshoff estudou a legibilidade de 120 programas comerciais escritos em PL/I. Declarou que "Basicamente os programas são ilegíveis. Um programador com experiência irá ter extrema dificuldade em lê-los." Concluiu que a legibilidade dos programas não pode ser medida automaticamente. No entanto, admitiu que alguns dos atributos que afectam a legibilidade podem ser identificados. Em [85] é listado um conjunto de práticas para melhorar a legibilidade que os seus autores aplicaram a um sistema

tendo obtido resultados positivos, como por exemplo, uma melhoria de 40% na complexidade do código, medida pela complexidade ciclomática de McCabe. Trata-se de um caso de estudo.

Para os autores, o ciclo de modificação dos programas envolve os seguintes passos:

1. Pedido do utilizador para que o programa seja alterado.
2. As especificações para as modificações são redigidas e é feita a estimativa dos custos.
3. É decidido quais as alterações que valem a pena ser realizadas.
4. O programa é alterado de modo a contemplar as novas especificações.

A modificabilidade dos programas põe em risco a legibilidade desses programas. É necessário inverter essa situação. A abordagem que sugerem para melhorar a legibilidade consiste em: ler o código; aplicar as transformações; adicionar comentários; e reajustar a indentação. As transformações devem ser seguidas do teste do programa. O processo é iterativo. As transformações usadas foram garantir blocos de uma entrada e uma saída, usar mnemónicas como nomes de variáveis, modularização para eliminação de blocos repetidos, adição do ramo senão (mesmo vazio), substituir *goto* por ciclos, usar variáveis de estado para controlar a execução ao longo de blocos para tornar explícito que se verifica o pressuposto para execução do próximo bloco (ou seja, que a variável não foi alterada entretanto); usar *switch* para reduzir ramificações; e reduzir a distância entre a definição e a utilização de variáveis.

Miara, 1983 [191] No artigo é considerado que a indentação de programas está relacionada com o aumento da compreensão do programa. Por isso, estudaram o efeito da indentação na compreensão dos programas.

É apresentado um estudo relativo à indentação do código dos programas. Trata-se de um estudo experimental com o objectivo de avaliar o efeito da indentação na compreensão de programas e satisfação do utilizador. O objectivo deste estudo era ter evidência experimental de modo a suportar a noção de indentação intermédia, ou seja, 2 ou 4 espaços.

Fizeram um estudo experimental envolvendo programadores com e sem experiência. Os programadores sem experiência tinham menos de dois anos de experiência profissional

e/ou menos de três anos de programação na universidade. Os programadores com experiência tinham três ou mais anos de programação na universidade e/ou dois ou mais de experiência profissional.

Foram testados dois estilos de blocos, bloco com início e fim explícitos, e sem bloco, e com quatro níveis possíveis de indentação: 0, 2, 4 e 6 espaços.

Foi usado um programa em Pascal retirado do livro de Grogono, *Programming in Pascal*. programa foi alterado de modo a obterem-se sete versões diferentes. Cada versão do programa tinha 102 instruções e não tinham nem comentários nem linhas em branco. Cada uma das versões ocupava duas páginas impressas em papel, e a quebra de página era exactamente no mesmo local. As diferenças entre as sete versões residiu nos graus da indentação. Para cada nível de indentação usaram os dois estilos de bloco descritos antes. Os programadores tinham de responder a um questionário sobre o código e classificar a dificuldade em entender o programa.

Concluíram que o nível de indentação que produz resultados de compreensão óptima está entre 2 e 4 espaços e que são significativos tanto para programadores inexperientes como para os experientes. Para os programadores com experiência, quando aumenta o número de espaços, aumenta o nível de compreensão. No entanto, aumentando ainda mais a indentação, então o nível de compreensão diminui, o que, segundo os autores se poderá dever ao facto de dificultar a visão devido a maior distância a percorrer.

Os programas com indentação apresentavam um resultado de 20 a 30% superior aos programas que não usavam indentação em testes de compreensão. Para o caso dos blocos, os resultados não foram significativos. Os programadores experientes obtiveram em geral resultados superiores aos dos programadores sem experiência.

Dunsmore, 1985 [80] Dunsmore realizou vários estudos com objectivos distintos. Pretendia estudar a eficácia dos comentários e analisar a importância da utilização das mnemónicas (por exemplo, *HOME-ADDRESS*). Os resultados permitiram concluir que os comentários são uma ajuda eficaz na compreensão e na manutenção de programas e que é mais fácil ao programador recordar-se de nomes criados com mnemónicas com 5 a 8 caracteres, do que em nomes sem mnemónicas ou nomes mais extensos. Todos os resultados foram estatisticamente significativos.

Os estudos foram do tipo estudo experimental e com alunos com experiência em programação. Os alunos foram atribuídos aleatoriamente aos vários grupos que correspondiam às várias situações a estudar, como por exemplo, no caso do factor comentários, as situações seriam duas: programa com comentários e programa sem comentários, e os alunos teriam de alterar os respectivos programas.

No caso dos nomes foram realizados vários estudos, mas a finalidade foi apenas a de recordar os nomes, e não teve directamente a ver com código-fonte. Os resultados indicam que mnemónias entre 1 a 5 caracteres são recordadas mais rapidamente e com menos erros do que nomes sem mnemónicas, ou com mnemónicas maiores, e.g. 5 a 9. As mnemónicas de 1 a 4 caracteres suscitam menos erros.

Estes resultados, tal como os estudos descritos acima de Sheneiderman mostram a importância das mnemónicas que, naturalmente, será tanto maior quanto maior forem os programas [256]. Como refere Dunsmore, nomes muito curtos podem não ser suficientes para ajudar a recordar o seu significado, enquanto nomes muito compridos tornam-se difíceis de recordar e obrigam a um esforço adicional de processamento interno para o fazer. Então a escolha das mnemónicas deve ser feita com cuidado de forma a conter apenas informação útil. Escolher nomes adequados requer tempo [256].

Deimel, 1985 [69] No artigo é realçada a importância da leitura do código para o programador e nesse sentido é afirmado que as competências de leitura deveriam ser desenvolvidas nos cursos de programação. Quando o programador aprende uma nova linguagem, ler programas escritos por outros programadores é uma forma eficaz de aprender, não só a sintaxe mas também as subtilezas associadas à linguagem.

Existem muitas circunstâncias em que o programador deverá ler os programas. Para os autores existem três componentes necessários a ensinar na leitura de programas. Primeiramente é importante explicar a sua importância. Em segundo lugar, os alunos devem ser encorajados a ver os programas em diferentes níveis de abstracção e em diferentes perspectivas. Por último, devem ser fornecidas linhas de orientação aos alunos para escreverem programas mais legíveis.

Em [68] pode ser encontrado um relatório sobre a implementação da leitura de código em cursos de programação com orientações e exercícios.

Pennington, 1987 [207] O trabalho descrito no artigo tinha como finalidade estudar a função do conhecimento de programação na compreensão de programas e a natureza das representações mentais dos programas, isto é, se assumiam maior importância as relações de natureza procedimental, ou as de natureza funcional. A natureza procedimental refere-se ao controlo de fluxo e a natureza funcional à hierarquia de objectivos.

O conhecimento de programação é analisado sob dois componentes, e que são, o conhecimento sobre programação e o conhecimento sobre planos de programação. O segundo, refere-se ao conhecimento sobre os grupos de instruções que são habitualmente utilizadas em conjunto para a realização de certas funções, ou seja, corresponde a conceitos de nível intermédio, como, por exemplo, pesquisa, ou somatório.

A autora refere que a programação estruturada promove a organização dos programas de acordo com um número de construções de controlo de fluxo que são mais fáceis de entender e de modificar por corresponderem à organização mental do programador. Essas construções são a sequência e a repetição. Reflectem portanto a natureza procedimental. Considerando o código como texto, essas construções corresponderão a palavras-chave de frases que serão unidades de texto. Os planos correspondem à natureza funcional.

Foram realizados 2 estudos. Os programas a serem avaliados estavam escritos em COBOL. O primeiro estudo contou com 80 programadores profissionais e o segundo com 40. No primeiro, os participantes foram testados quanto à compreensão e reconhecimento de código-fonte de pequenos programas. No estudo, os autores tratam o programa como um texto por considerarem que a teoria e métodos no estudo da compreensão de textos estavam bastante desenvolvidas e poder ser um ponto de partida para a compreensão de programas. No segundo estudo, os participantes analisaram e alteraram o código de programas de tamanho médio.

Em ambos os estudos, os resultados mostraram que a base da representação mental é formada pelas relações procedimentais dos programadores profissionais, ou seja, os programas são compreendidos, em primeiro lugar, quanto aos seus "episódios" procedimentais. A finalidade das tarefas pode influenciar as relações que dominam as representações mentais posteriores na compreensão. Estas requerem a integração de unidades procedimentais.

Como se pode ler, a autora comparou o código-fonte com texto e recorreu à psicologia cognitiva para ajudar a explicar o processo de compreensão do software. Este artigo situa-

se unicamente no contexto da compreensão, mas esta revisão não ficaria completa sem abordar artigos desse âmbito e a importância que a psicologia cognitiva tem tido como suporte teórico.

Baecker, 1988 [10] Neste artigo é defendido que por forma a tornar os programas mais legíveis a apresentação do código-fonte necessita ser melhorada. O objectivo deste trabalho é melhorar o código-fonte do programa, e desse modo torná-lo:

- mais legível;
- mais compreensível;
- mais vivo (quanto à aparência);
- mais atractivo;
- mais memorizável;
- mais útil;
- e de mais fácil manutenção.

Os autores apresentaram sete princípios que consideram poder melhorar a aparência de um programa em C. Esses princípios são os seguintes:

1. Características a nível do papel e da página.
2. Composição da página e do *layout*.
3. Vocabulário tipográfico.
4. Parâmetros de definição do tipo.
5. Símbolos e elementos diagramáticos.
6. Cor.
7. Meta-texto: comentários.

As listagens dos programas são demasiado longas podendo ser reduzidas através da diminuição do tamanho da fonte da letra, mas esta medida provocaria uma diminuição da legibilidade do código. No artigo, defende-se que os ambientes de desenvolvimento devem ser mais parametrizados, de modo a obter-se uma aparência óptima do código. Como os

ambientes de desenvolvimento eram muito dispendiosos na época, os autores receavam que essas melhorias dos elementos visuais aumentassem ainda mais o seu preço. Como se sabe, os ambientes actuais já implementam de alguma forma esses princípios. Esses princípios estão relacionados essencialmente com aspectos tipográficos.

Tenny, 1988 [273] A codificação é apenas uma parte da engenharia de software mas muito importante no que diz respeito à manutenção. Quanto mais legível é o código mais rápida e seguramente conseguirá o programador obter informação crítica do programa através da leitura do código do programa. Para o autor, um programa é legível se a informação necessária à sua manutenção é fácil de encontrar lendo o código.

O objectivo do estudo era analisar a legibilidade segundo a modularidade e os comentários. Assumiu existir uma diferença qualitativa entre a leitura de um programa com procedimentos internos, a leitura de um programa com procedimentos externos, e a leitura de um programa com todo o código num bloco único e, portanto, sem funções. Para o estudo foi utilizado um programa que permitia guardar os registos de atletas. O programa foi codificado em PL/I pelo facto de permitir funções internas e externas. Para cada um dos 3 tipos de programas referidos foram criadas duas versões, uma com comentários e outra sem. O resultado foram seis versões do mesmo programa:

- sem procedimentos e sem comentários;
- sem procedimentos e com comentários;
- procedimentos internos, sem comentários;
- procedimentos internos, com comentários;
- procedimentos externos, sem comentários;
- procedimentos externos, com comentários.

O estudo envolveu 157 estudantes de engenharia de software, sendo a maioria deles de anos mais avançados (seniores) e alguns (não é referido quantos) com experiência profissional de programação. O autor assume que quanto mais legível for um programa, mais rapidamente e mais exacta será a informação obtida pela sua leitura. Por isso, mede a legibilidade pelo número de respostas correctas dos participantes a questões acerca do programa, num dado tempo. Apenas 148 apresentaram resultados.

Como resultados, os programas sem procedimentos (o código para as sub-tarefas está todo inserido no módulo físico do programa principal, ou seja, é um programa monolítico) e com comentários mostraram ser os mais legíveis, enquanto que os programas sem procedimentos e sem comentários foram os menos legíveis. Este resultado sugere uma interacção entre a modularização e os comentários.

Concluíram que os procedimentos têm pouco efeito sobre a legibilidade. Também referem que os comentários melhoram a legibilidade dos programas, mas o efeito dos comentários é significativo apenas na ausência de funções.

Uma possível explicação para o facto de o resultado da modularização não ser significativo será por esta fazer aumentar a dimensão do programa dificultando a tarefa de o recordar. Ao mesmo tempo, havia um tempo máximo para responder, igual para todas as versões. Os resultados transpostos para a programação OO fazem supor que a divisão dos programas num grande número de métodos prejudica a legibilidade. Ou seja, seriam preferíveis métodos de alguma dimensão comentados. Isto é discutível, em primeiro lugar devido à dimensão actual dos programas e à organização em classes.

Benander, 1990 [21] Foi realizado um estudo empírico sobre a utilização da instrução *goto*. Neste artigo são apresentados os resultados. Apesar de autores consagrados serem da opinião da não utilização da instrução *goto*, existem outros autores que são favoráveis à sua utilização. Afirmam até que a sua utilização é essencial. Argumentam que com a sua utilização é possível produzir soluções mais eficientes para os problemas de programação comuns.

O objectivo deste estudo deveu-se a três factores e para isso foram analisados 300 programas em COBOL. Em primeiro lugar, pretendeu-se investigar qual a relação entre a instrução *goto* e os programas sem erros. Em segundo lugar, pretendeu-se investigar qual a relação entre a instrução *goto* e o tempo necessário na depuração dos erros. Por último, pretendeu-se determinar se houve diferença significativa na medição das estruturas e estilo em programas que usavam *goto* relativamente a programas que não usavam.

Gellerich [102] realizou um estudo e os resultados apontaram que em grandes projectos de software usando linguagens com baixo índice de instruções *goto*, são mais legíveis.

Oman, 1990 [201] No artigo considera-se que é muito importante para os programadores que estão a iniciar-se lhes seja ensinado os princípios gerais de estilo de programação.

Para o autor, um bom estilo de programação é escolher a expressão e arranjo que melhor transmite a intenção do programador e estrutura subjacente(s). Por isso, um bom estilo é a criação da expressão mais compreensível do algoritmo a ser codificada [201].

O estilo da programação com a indentação é ensinado nos cursos de programação e é discutido em muitos livros de programação. Nas aulas é incentivado um bom estilo de programação, mas os autores referem que o ensino de um bom estilo de programação é apenas uma pequena parte do curso. O mesmo acontece em muitos livros de programação com uma secção onde se apresenta o estilo que consiste, normalmente, de regras simples ilustradas com pequenos fragmentos de código de programas.

Os autores consideram que ensinar o estilo de programação desta forma cria dois problemas: Em primeiro lugar, não realça a importância da legibilidade. Isto poderá ser ultrapassado atribuindo tarefas de manutenção de programas considerados pobres, isto é, pouco legíveis. Adicionalmente, isto deverá convencer os alunos que um mau estilo pode comprometer a manutenção do programa [201].

O segundo problema é que os programadores ficam com a impressão que o estilo da programação é meramente uma colecção de regras específicas da linguagem que devem seguir, mas que normalmente não existe o melhor estilo de programação para todas as aplicações. Em vez disso, sugerem o seu ensino partindo-se, por exemplo, da taxonomia que desenvolveram.

Neste artigo, o autor apresenta uma taxonomia de estilos da programação. Para isso elaborou uma análise sistemática do estilo da programação em livros e através de analisadores do código. Começaram por compilar uma lista de todos os estilos presentes nos exemplos, melhorados, pertencentes ao livro "The Elements of Programming Style" do Kernighan e Plauger. Basearam-se neste livro por ser o livro mais citado e utilizado. Também reescreveram os exemplos e criaram regras gerais para situações específicas. Nas situações que foram reescritas explicitaram as modificações.

A taxonomia foi dividida em três categorias principais. As três categorias principais, definidas pelo autor, são as seguintes:

1. Estilo tipográfico.

2. Estilo das estruturas de controlo.

3. Estilo das estruturas de informação.

Consideram que a categoria de boas práticas de programação em geral deve ser tratada à parte (grupo 0). É essa categoria que se apresenta em seguida:

A. Definição: Regras e linhas de orientação aplicáveis ao processo de programação que afecta directamente o estilo do produto.

B. Critérios de classificação: Restrições na metodologia de programação, normalmente de natureza temporal individual.

C. Exemplo de subclasses:

(a) Técnicas de concepção;

(b) Selecção da linguagem e restrições;

(c) Testes e práticas de depuração;

(d) Aspectos relacionados com a manutenção.

D. Guia de aplicação: Não pode ser especificado porque as práticas são dependentes dos métodos locais e restrições.

E. Exemplo de regras:

(a) Definir o problema completamente, resistindo à vontade de começar logo a codificar;

(b) Usar unicamente características da norma ANSI;

(c) Implementar técnicas de depuração;

(d) Não comentar mau código, mas sim reescrevê-lo.

Como se constata, várias das regras de programação são referidas por outros autores.

Uma vez que os princípios e as regras correspondentes e as directrizes estejam desenvolvidas, muitas ferramentas de estilo podem ser criadas e ajudar nas tarefas de codificação e manutenção. Sugerem a análise taxonómica de programação que acabará por levar à criação de padrões de programação e ferramentas de estilo associadas.

Raymond, 1991 [218] Este autor afirma que a leitura do código-fonte é uma actividade fundamental na manutenção de software porque, para ser possível fazer a manutenção do código, é preciso primeiro entendê-lo. Para os autores, a leitura é uma actividade física e mental.

O esforço físico pode ser aliviado através da escolha de uma fonte e de espaçamento que menos provoque um esforço. Um segundo aspecto que provoca esforço físico é quando o programa, mais especificamente um módulo do programa, ocupa mais do que uma página, e é necessário aceder às várias páginas, umas a seguir às outras e por vezes, várias vezes. Um terceiro aspecto que provoca esforço físico é o tempo gasto para a leitura do programa. Este último tem efeitos na actividade mental. A velocidade de leitura irá ter efeitos psicológicos.

Já o esforço mental da leitura seria caracterizado pela dificuldade dos conceitos envolvidos no texto, e em particular o código confuso, tipo *spaguetti*.

Os autores estudaram o problema do código-fonte no contexto da manutenção de software, mas não chegaram a nenhuma conclusão. Apesar de não terem chegado a nenhuma conclusão, considerou-se importante incluir aqui o artigo pois considerou-se que o tema é importante.

Laitinen, 1996 [158] Laitinen refere que os nomes em linguagem natural entendem-se melhor do que as suas abreviaturas. Também diz que a complexidade é inversamente proporcional ao entendimento de um programa. Ou seja, se é possível aumentar o entendimento de um programa então a sua complexidade diminui.

Para o autor é importante estimar a eficácia das práticas de desenvolvimento de software. Nesse sentido, quando se considera o desenvolvimento de software como um processo de documentação, deve-se ter meios para estimativa do entendimento dos documentos de software de modo a comparar diferentes práticas de documentação. Deverá então existir um método que seja capaz de comparar o entendimento do software.

Os nomes para as variáveis, as constantes, as funções são símbolos importantes num programa. Conforme referido anteriormente, já foi comprovado que os nomes afectam significativamente a aparência visual de um programa e consequentemente a sua legibilidade.

Apesar de alguns testes realizados por outros autores sugerirem que a utilização de mnemónicas nos nomes melhora significativamente o entendimento do código dos programas, os autores consideram que os resultados estatísticos não têm sido convincentes.

Os autores usaram linguagem natural nos seus casos de estudo. Para os autores, os programas cujos identificadores estão em linguagem natural são mais entendíveis que os programas cujos identificadores usam abreviaturas.

Haneef, 1998 [114] Neste artigo considera-se que a legibilidade é semelhante em alguns aspectos à usabilidade, com a diferença de a usabilidade se aplica ao utilizador final enquanto que a legibilidade se aplica à manutenção. A usabilidade tem atraído a atenção porque a sua necessidade torna-se aparente no curto prazo. Os autores afirmam que a legibilidade ainda é negligenciada porque o software é utilizado na produção, e não um bem de consumo para venda. O maior impacto de leitura difícil estaria na produtividade que é um problema a longo prazo, e mais fácil de ignorar [114].

Segundo os autores existe a premissa que a produtividade e a qualidade do desenvolvimento e manutenção do software, em particular, para grandes projectos, está relacionado com a legibilidade. Neste artigo é proposta a criação de grupos de documentação/legibilidade nas empresas de software. Estes grupos deveriam ser de programadores e deveriam produzir a documentação e assegurar a legibilidade dessa documentação. Declaram que não conseguiram provar que esta assumpção funcionava.

Wiedenbeck, 1999 [291] Foram realizados dois estudos experimentais para comparação das representações mentais relativamente à compreensão dos programas por parte de programadores sem experiência nos paradigmas procedimental e OO.

Neste artigo são apresentados os resultados dos estudos. O primeiro estudo comparou as representações mentais e a compreensão de pequenos programas nos dois paradigmas. O segundo estudo estendeu o estudo a programas maiores e com características mais avançadas.

As linguagens utilizadas para os estilos procedimental e OO foram o Pascal e C++, respectivamente. Os intervenientes nos estudos eram estudantes do 2º ano de um curso de programação.

Como conclusões, os autores referem o seguinte: Para os programas curtos não houve diferença significativa nas respostas às questões sobre os dois paradigmas. No entanto, na indicação de qual a funcionalidade dos programas, o paradigma OO teve mais respostas acertadas.

Os autores consideram que isto acontecia por a informação da função estar mais disponível nas suas representações mentais dos programas e apoiam o argumento nos destaques da notação OO ao nível da classe individual.

Para os programas longos a compreensão do paradigma OO por parte dos intervenientes foi superior em qualquer tipo de pergunta. Os autores referem ainda que, as dificuldades sentidas pelos inquiridos a responder a questões sobre um grande programa do paradigma OO são resultantes de eles enfrentaram problemas em mobilizar informações e fazer inferências a partir dele. E que o resultado pode estar relacionado com uma curva de aprendizagem mais longa para os mais inexperientes do paradigma OO, bem como às características e à notação utilizadas.

Mengel, 1999 [187] Entender a qualidade dos programas por parte dos estudantes, é uma tarefa difícil mas que deverá ser auxiliada pelos seus professores. A análise estática dos programas de alunos pode ajudar a melhorar o processo de classificação realizado pelos professores.

A análise estática dos programas pode ser realizada pela inspecção visual do código de um programa. Uma vantagem da utilização da análise estática é que o programa não necessita ser executado. Aqui, o objectivo é avaliar a qualidade do código e compreendê-lo. Foi realizado um estudo que através de um conjunto de métricas permitiu estimar a qualidade dos programas. Os autores consideraram várias métricas de diferentes autores, David Jackson e De Sheung-Lun Hung [187].

As métricas consideradas pelo primeiro autor, David Jackson, são as seguintes [187]:

- Correcção;
- Estilo - tamanho do módulo, tamanho do identificador, linhas de comentários e a indentação;
- Eficiência;

- Complexidade.

Os autores De Sheung-Lun Hung et al., consideram as seguintes métricas [187]:

- Competências de programação;
- Complexidade;
- Estilo de programação;
- Eficiência de programação.

Também utilizaram a métrica de Halstead sugerida pelo autor Ronald Leach [187]. Foram analisados 90 programas escritos em C++.

Como resultado, o estudo revela que não é eliminada a necessidade de rever os programas manualmente, mas reduz a necessidade de inspeccioná-los tediosamente. As métricas podem ajudar os professores a escolher as tarefas de programação mais adequadas e ajudá-los a entender por que é que os alunos têm desempenho mais fraco numa dada tarefa. Como conclusão, podem servir para motivar o aluno no sentido de melhorar a sua qualidade na programação.

Shima, 2002 [246] No artigo são apontadas diversas razões que justificam a importância da compreensão do código. Muitas vezes é necessário alterar o software para o reutilizar, para o melhorar, corrigir erros ou adaptar a novas circunstâncias. A compreensão do software é uma das características mais importantes da qualidade do software porque pode influenciar o custo ou a fiabilidade na evolução, na reutilização, ou na manutenção do software.

Os desenvolvedores tentam reutilizar um sistema de software desenvolvido por outros e a dificuldade de compreensão limita a sua reutilização. Mesmo que os desenvolvedores do sistema original façam parte da organização inicial, eles poderão ser transferidos, ou mudar de emprego ou até mesmo reformarem-se.

As alterações aos sistemas de software são chamadas de evolução de software na área de investigação da manutenção de software. As alterações a sistemas reutilizados podem ser consideradas como uma evolução de sistemas reutilizados. A compreensão pode ser colocada como um factor de evolução na reutilização ou manutenção.

Num estudo experimental que realizaram de inspecção de código, verificou-se que 60% dos tópicos que os indivíduos reportavam estavam relacionados com a compreensão. Esta verificação reforça a importância da compreensão. Não é fácil medir a compreensão do software porque a compreensão é um processo interno das pessoas.

Para o autor, a compreensão de um sistema de software pode ser considerada como "*...reading necessary information via the software system to evolve it...*" [246].

Stamelos, 2002 [262] Os autores apresentam o resultado de um caso de estudo-piloto para compreensão das implicações da qualidade estrutural.

O núcleo das actividades *open source* acontecem a nível do código. O propósito deste artigo é reportar e discutir os resultados de um caso de estudo-piloto que examina a qualidade do código-fonte distribuído pelo desenvolvimento *open source*. O estudo tem por finalidade perceber os benefícios da análise da qualidade estrutural do código *open source*.

Também tem como objectivo identificar as métricas estruturais que ajudam a distinguir mais do que uma versão candidata de um componente de software quando a determinação do conteúdo de uma versão está relacionada com o tamanho. Para isso, mediram as características de qualidade de 100 aplicações.

Os autores referem que o estilo da programação já foi reconhecido como estando relacionado directamente com algumas das características de qualidade de um programa, como é o caso da clareza do código e o seu entendimento e que a imbricação excessiva reduz a legibilidade e a capacidade de teste de um componente. Também afirmam que a satisfação do utilizador é provavelmente a forma mais directa de medir a qualidade externa de um programa.

Conforme referido pelos autores, limitaram a análise ao nível do componente. Num programa em linguagem C, um componente corresponde a uma função. Os autores seleccionaram um conjunto de métricas que consideraram relevantes para medir a qualidade dos componentes, sendo aqui só apresentadas aquelas que os autores consideraram que estão relacionadas com a legibilidade:

1. Complexidade ciclomática: é um indicador do esforço necessário para compreender e testar os componentes.

2. Tamanho do programa: mede o tamanho do programa através da soma do número de ocorrências dos operandos e operadores.

3. Número máximo de níveis de indentação: um excessivo número de indentação reduz a legibilidade e a capacidade de teste de um componente.

4. Tamanho médio: mede o tamanho médio das instruções de um componente.

Mais uma vez, o elevado grau de indentação é colocado em causa.

Como conclusões, os autores apresentam um conjunto de práticas, que eles consideram práticas-chave a seguir, tendo em consideração a qualidade do código estrutural, quando se analisa o código *open source* já criado:

1. Solicitar aos programadores para terem em conta a qualidade do código estrutural quando intervêm no código.

2. O coordenador do projecto poderá avaliar a qualidade do código resultante da intervenção por parte dos programadores, de acordo com um padrão pré-definido. Isto implica que certos componentes não estejam em conformidade com esse padrão, sejam rejeitados ainda que estejam correctos.

3. O coordenador do projecto tomará decisões quanto à reengenharia do código sempre que o projecto pareça ter problemas.

Chhabra, 2003 [50] Neste artigo são apresentadas duas medidas de complexidade espacial, baseadas no código e nos dados. A complexidade espacial do código mede o esforço requerido para compreender os módulos e a sua utilização. A complexidade espacial dos dados medirá o esforço necessário para perceber o propósito e a utilização das variáveis globais e constantes.

O estudo foi realizado em quinze projectos, tendo sido aplicadas as duas medidas de complexidade espacial. Como resultado, os autores referem que o uso adequado de dados globais e práticas de engenharia de software ajudam a reduzir os valores da complexidade espacial e com isso compreender melhor o software. O estudo permitiu concluir que o software deverá ter um máximo de 3700 linhas de código.

Dagpinar, 2003 [61] Neste artigo é descrito um estudo realizado com o objectivo de determinar quais as métricas que podem ser usadas como estimativa da facilidade de manutenção do software orientado a objectos. Os autores usam a classificação de [89] que divide as métricas em três categorias: métricas de processo, métricas de produto e métricas de recursos.

Por sua vez, as diferentes métricas de produto foram divididas em:

- métricas de medida;
- métricas de herança;
- métricas de coesão;
- métricas de acoplamento.

As métricas de medida referidas são as mais frequentes e são as seguintes:

- *LOC* - linhas de código;
- *NIM* - número de instâncias dos métodos;
- *TNOS* - número total de instruções.

Este tipo de métricas depende muito do estilo de codificação por parte dos programadores. As outras métricas são também importantes, mas por estarem mais associadas à concepção não serão aqui descritas.

Através do estudo realizado, usando análise de regressão multivariada, chegaram aos seguintes resultados. As métricas *TNOS* e *NIM* mostram poder ser usadas com eficácia para estimar a facilidade de manutenção dos sistemas de software.

As métricas de acoplamento dividem-se em:

- herança vs não herança;
- importação vs exportação;
- directo vs indirecto.

O estudo também mostra que as métricas de medida e as métricas de acoplamento directo de importação são estimadores/preditores muito significativos, enquanto que as métricas de acoplamento de herança, de coesão e de acoplamento indirecto de exportação não o são.

Dolado, 2003 [77] O artigo apresenta um estudo sobre o impacto dos efeitos colaterais (*side-effect*) na compreensão dos programas. Como resultado do estudo verificou-se que os efeitos colaterais têm impacto negativo, inibindo a legibilidade dos programas e consequentemente prejudicando a manutenção e evolução do software.

Alguns exemplos de expressões com efeito colateral:

$x++$

$--v$

$(x+1, y = x+3)$

$x == 2 \;\&\&\; y = 2 * x$

Os autores como conclusões afirmaram que não tiveram por objectivo adicionar *tabu* à comunidade de engenharia de software, mas acrescentar conclusões relevantes baseadas em prática empírica de modo a reduzir a complexidade do software. É importante ter presente que um pequeno fragmento de programa pode ter efeitos muito fortes na compreensão dos programas.

Os resultados mostram que existe uma degradação clara e significativa do desempenho para estas formas limitadas (dentro dos procedimentos) de efeito colateral. As formas mais complexas dos efeitos colaterais poderão produzir degradação similar ou mesmo pior desempenho.

Mantyla, 2003 [173] É apresentada uma taxonomia que categoriza os *bad smell*. A finalidade da taxonomia era melhorar a compreensão sobre os *bad smells* e as suas inter-relações.

Apesar dos resultados serem considerados muito preliminares, os *bad smells* foram categorizados da seguinte forma:

- *Bloaters* - código que tem crescido muito e por isso não pode ser manipulado eficazmente: métodos longos, classes com mais de uma responsabilidade e extensas, longas listas de parâmetros, *primitive obssession*[7], *data clumps*[8];

[7]Utilização de tipos de dados primitivos para representar ideias de domínio.

[8]Grupos de itens de dados que estão relacionados e são sempre usados ou passados juntos.

- Abusos da programação orientada aos objectos: *switch*, *field*, *refused bequest*[9], classes alternativas com *interfaces* alternativas, hierarquias de herança paralela;
- Prevenção de alteração: *divergent change*, *shotgun surgery*[10];
- Dispensáveis: *lazy class* e *data class*, código duplicado e *speculative generality*[11];
- Encapsulamento: *message chain* e *middle man.*[12] Nestes *smells* muita vezes a solução é reestruturar a hierarquia das classes movendo os métodos e adicionando subclasses;
- Acoplamentos: *Feature Envy*[13], *inappropriate intimacy.* Estes representam um elevado acoplamento o que vai contra os princípios de concepção OO;
- Outras situações: bibliotecas de classes incompletas, comentários.

.

Também é apresentado um estudo empírico da utilização dos *smells* para avaliação da qualidade de código numa pequena organização de software. Este estudo pretendeu determinar a correlação entre os *bad smell* de diferentes grupos. Para o estudo empírico foi realizado um teste dirigido aos programadores.

Concluíram que existe uma maior correlação entre os *bad smells* dentro dos grupos do que entre os *bad smells* dos diferentes grupos.

Mantyla, 2004 [175, 174] Neste artigo é apresentado um estudo empírico de avaliação subjectiva dos *bad smells* a nível do código.

Em primeiro lugar, foi estudado o efeito avaliador na avaliação subjectiva dos *bad smells* a nível do código. Em segundo lugar, foram aplicadas métricas para identificação de três *bad smells* (classes muito extensas, listas de parâmetros longas, código duplicado) e depois foram comparados os resultados deste estudo com os da avaliação subjectiva.

[9]Herança entre classes para reutilizar o código de uma superclasse. Mas a superclasse e subclasse são completamente diferentes.

[10]Alteração numa classe é feita em várias classes simultaneamente.

[11]Código não usado e redundante.

[12]Classe que delega a maior parte de seu trabalho noutras classes.

[13]Métodos que fazem uso extensivo de outra classe.

Na investigação para o estudo dos *bad smells* foi usado um questionário no qual se pediu aos programadores para avaliarem em que medida é que cada *bad smell* estava presente num dado módulo.

Questões de investigação apresentadas:

1. De que modo a experiência do desenvolvedor afecta a avaliação dos *bad smells*?
2. Os desenvolvedores têm uma opinião uniforme acerca do nível de *bad smells* no código-fonte?
3. Existe correlação entre a avaliação dos desenvolvedores e as métricas existentes?

O objectivo da investigação foi aumentar a compreensão sobre os indicadores de qualidade de código. Os indicadores de qualidade propostos foram os seguintes:

- Reutilização;
- Flexibilidade;
- Compreensão (*understandability*);
- Extensibilidade;
- Eficácia;
- Funcionalidade.

A escolha dos *bad smell* teve a ver com o facto de considerarem mais simples a operacionalização desses *bad smells* e a existência de ferramentas mais apropriadas para medi-los.

Para os autores, a avaliação subjectiva dos *bad smells* e o efeito demográfico (conhecimento, experiência dos programadores e o seu papel) parecem explicar algumas das avaliações nas avaliações. Verificaram que as métricas e as avaliações subjectivas não se correlacionam. Este estudo torna questionável a utilização dessas medidas.

Deissenboeck e Pizka, 2005 [67] Deissenboeck e Pizka concluiram que a escolha dos nomes dos identificadores é crucial para a compreensão de um programa. Implementaram uma ferramenta *IDD (Identifier Dictionary)* como *Plug-In* para a plataforma de desenvolvimento Eclipse em Java. Este *Plug-In* faz uma análise de cada ficheiro em código-fonte para determinar automaticamente os tipos associados com os identificadores encontrados.

Todos os identificadores são listados com o seu nome, tipo, descrição e o número de declarações dos identificadores com o mesmo nome. Durante este processo são indicados potenciais problemas de consistência. Para além disso, o *Plug-In* tem outro recurso chamado de *global rename* que propõe a mudança de nome de todos os identificadores que têm um determinado nome. Este recurso apresenta uma visualização de todas as mudanças propostas e faz uma verificação automática da sua validade.

No estudo que envolveu vários programas verificaram que cerca de 70% do código consiste de identificadores.

No artigo é sugerida a existência de três importantes razões para nomes inapropriados dos identificadores:

1. Os identificadores podem ser escolhidos arbitrariamente pelos desenvolvedores e iludir a análise automática.
2. Os desenvolvedores têm conhecimento limitado sobre os nomes já usados no sistema.
3. Os identificadores estão sujeitos a deixarem de ser válidos durante a evolução do sistema.

Convenções de nomenclatura fazem parte de inúmeros estilos de codificação. Essas convenções geralmente centram-se em aspectos sintácticos, como é o caso de, em Java, *packages* em minúsculas e classes em *CamelCase* (definição em Sharafi, 2012, pag. 157).

Segundo os autores, quando se trata dos aspectos realmente importantes da nomenclatura, como seja a semântica dos nomes, geralmente há pouca orientação e afirmam que os nomes devem ter significado no domínio da aplicação e não no da implementação.

Segundo o artigo, um modelo formal baseado em mapeamento bijectivo entre conceitos e nomes, fornecerá uma fundação sólida para a definição de regras precisas para nomes.

Desenvolveram uma definição formal de designação concisa e consistente para a definição de regras para a criação dos nomes.

Regra 1: Consistência. Quanto à consistência são indicadas duas formas diferentes de inconsistência: homónimos e sinónimos. Os homónimos são palavras que se escrevem e pronunciam da mesma maneira e têm significado diferente. Os sinónimos são palavras diferentes com o mesmo significado. Os sinónimos têm um impacto muito negativo

aumentando significativamente o esforço de compreensão. Isto porque para cada identificador, o programador tem de considerar todos os conceitos associados. A mistura de sinónimos e homónimos, que é comum no código-fonte, maximiza a confusão a agrava o esforço da compreensão.

Regra 2: Concisão: Os autores definem concisão em dois passos. Primeiro definem exactidão. É muito comum um identificador possuir o nome correcto e não ser suficientemente conciso. Deste modo, para evitar práticas de identificação fracas adiciona-se a propriedade concisão. Esta definição requer um identificador que tenha exactamente o mesmo nome do conceito que vai representar.

Os autores também introduzem os conceitos de permutação e transformação e consideram que uma permutação é um subconjunto de transformação e que a transformação é uma generalização de permutação. O reforço destas regras é suportado com a ferramenta IDD. Para o autor, esta ferramenta não só reduz o esforço necessário para aplicar as regras dos nomes como também fornece suporte para tarefas simples tais como determinação de nomes para os identificadores nas declarações.

Raskin, 2005 [217] Neste artigo, o autor chama a atenção para o código auto-documentado e para programas que fazem documentação automática. O autor afirma que esses métodos não podem fornecer a documentação necessária para um código confiável e sustentável. São apenas uma ajuda.

O código auto-documentado e os geradores de documentação automática não podem mostrar a razão pela qual o programa está a ser escrito, nem as razões pelas quais foi escolhida uma determinada metodologia. Não conseguem argumentar sobre certas abordagens que foram tomadas perante várias alternativas.

Raskin alerta para a importância de não se ser doutrinário sobre este assunto. Um programador competente que aprendeu o estilo de documentação às vezes pensa numa solução em que primeiro faz o código, e depois cria a documentação. Uma alternativa será aplicar uma estratégia mista - especialmente quando não há um projecto de algoritmo complicado envolvido. Esta abordagem não deve ser desencorajada, desde que documentação que o programador defenda (e sinceramente apoie) seja a abordagem da documentação.

Como conclusão, uma boa documentação inclui informações de base e de decisão que não podem ser derivadas do código. A documentação prévia, clara e extensa é um elemento-chave na criação de software que pode sobreviver e adaptar-se. A documentação de acordo com padrões elevados irá diminuir o tempo de desenvolvimento, resultando num trabalho melhor.

Relf, 2005 [220] Neste artigo é apresentado um estudo empírico para verificar se a legibilidade do código-fonte melhora caso exista no editor uma ferramenta que identifique os nomes usados no código-fonte. Foi realizado um estudo experimental que envolveu estudantes e engenheiros informáticos. O estudo consistiu de três exercícios de programação:

1. Exercício de teste: No primeiro exercício foi solicitado aos participantes para substituirem num programa os nomes dos identificadores que eles considerassem pobres por nomes de identificadores mais significativos.
2. Exercício de manutenção: No segundo exercício foi solicitado para fazer a modificação de uma unidade software existente que foi propositadamente escrita com um mau nome de identificador.
3. Exercício de produção: No último exercício é pedido aos participantes para desenvolver um método.

Os dois últimos exercícios, o de manutenção e o de produção, foram escolhidos por serem representativos das duas principais fases do ciclo de vida do software.

As linhas de orientação utilizadas para dar nomes aos identificadores foram encontradas ou na literatura científica como conducente à melhoria da legibilidade do código-fonte ou então por existir evidência empírica que estas linhas de orientação têm efeito na melhoria da legibilidade do código-fonte. Estas dezanove linhas de orientação, umas a seguir, outras a evitar, são as seguintes:

- Constante sem nome;
- *Underscore* múltiplo;
- *Underscore* externo;

- Dígito(s) numérico;
- Nome curto;
- Nome comprido;
- Contador de palavras;
- Codificação do identificador;
- Qualificação da classe/tipo;
- Qualificação da constante/variável;
- Palavras abstractas;
- Palavras em inglês;
- Nome numérico;
- Palavra em plural;
- Convenção de nomenclatura;
- Nomes duplicados;
- Nomes semelhantes;
- Identificador não utilizado;
- Mesma palavra em identificadores distintos.

Os participantes demonstraram considerável imaginação na criação dos nomes dos identificadores, sendo 60% dos nomes dos identificadores únicos. Tiveram a capacidade de criar pelo menos sete nomes de identificadores durante o exercício de produção e pelo menos três identificadores durante o exercício de manutenção. A identificação de um nome de identificador pobre pode influenciar individualmente a capacidade de modificar correctamente o código-fonte.

Como conclusão, o autor considera a magnitude do resultado notável, uma vez que, nalguns casos houve um número considerável de restrições que deveriam limitar a escolha dos nomes dos identificadores. A legibilidade do código-fonte beneficiaria se se utilizasse um conjunto limitado de palavras em inglês na construção dos nomes dos identificadores e um programador beneficiaria de uma maior assistência, possivelmente fornecida por um editor de código-fonte, na construção dos nomes dos identificadores.

Collar, 2006 [57] Os autores, afirmam então que a reutilização é afectada pela legibilidade, a qual por sua vez influencia a compreensão do código. Pretendem saber o que influencia a legibilidade.

Nesse sentido, os autores referem no artigo um estudo, da autoria do primeiro autor, sobre o impacto da legibilidade nas várias fases do desenvolvimento do software. O estudo recorre ao modelo COCOMO, que contempla custos com reutilização, e na teoria cognitiva da legibilidade, considerando o código como um conjunto ordenado de proposições. De acordo com os resultados, a legibilidade do software tem um efeito a nível global (em todas as fases) nos custos de desenvolvimento e é independente da dimensão do software.

Ou seja, quando a legibilidade aumenta, é gasto menos tempo na leitura do código o que reduz os custos desta fase do ciclo, o que na reutilização, ou manutenção é uma actividade constante do programador. Também de acordo com o estudo, o número de argumentos repetidos (AR) e o número de novos argumentos (NA) e a densidade proposicional (DP) aumentam significativamente a legibilidade do software e que a sua contabilização é um preditor da legibilidade.

Conclusões:

DP Uma grande DP requer um maior esforço de processamento por parte do leitor; Este esforço torna o código mais difícil de ler.

NA Um grande número de argumentos requer que o leitor utilize mais conceitos na memória; Este esforço torna o código mais difícil de ler.

AR Um grande número de argumentos repetidos dentro e entre linhas de código lógicas torna o programa mais coerente e consequentemente mais fácil de ler.

Interligação entre partes de código Provoca um esforço cognitivo tornando o código mais difícil de ler.

Lin, 2006 [171] Neste artigo, o autor propõe um modelo chamado de compreensibilidade integral que permite medir a compreensão do software e que se explica em seguida, após uma breve contextualização.

Quando os programadores tentam reutilizar um sistema de software desenvolvido por outros programadores, muitas vezes a dificuldade de compreensão do software dificulta a

sua reutilização. Não é fácil medir a compreensão do software uma vez que é um processo interno do ser humano.

Boehm definiu compreensão do software como uma característica da qualidade que significa a facilidade de compreensão dos sistemas de software.

Se os programas são difíceis de compreender então as alterações podem causar problemas graves e desse modo as alterações podem custar mais tempo que refazer o software. Como forma de ajudar a resolver o problema os autores propõem um modelo de compreensão do software composto por diversos factores.

Factores que influenciam a compreensão do software:

1. Compreensibilidade da documentação: o software é mantido através do uso integrado do código-fonte com a sua documentação. A legibilidade do código-fonte e a qualidade da documentação deve ser tida em consideração quando se mede a manutenção do software.
2. Compreensibilidade da estrutura: o software ser entendível não depende só da compreensão do sistema de software mas também do facto de ser entendível por parte de quem o está analisar. O número médio de tentativas necessárias para reconstrução correcta que vários programadores corrigiram um sistema de software, significa compreensão do sistema de software.
3. Compreensibilidade dos componentes: usam o modelo de tamanho funcional cognitivo (CFS - *cognitive functional size*) para medir a compreensibilidade dos componentes do software.
4. Compreensibilidade do código-fonte: usam a métrica de complexidade de Halstead para avaliar a facilidade de compreensão do código do software.
5. Compreensibilidade dos dados: utilizam análise de complexidade espacial dos dados na avaliação.

Lawrie, 2006 [164] Quando as funções não são comentadas, o que é muito comum, a compreensão é quase exclusivamente dependente dos nomes de identificadores.

Partindo do princípio de que quem faz os programas pretende criar identificadores de qualidade (por exemplo, identificadores que incluem o conhecimento de domínio rele-

vante), deve-se perguntar, por exemplo: As iniciais de um nome de um conceito fornecem informações suficientes para representar o conceito? E se um identificador já não é necessário? Qual é o efeito de identificadores mais longos na limitada capacidade de memória de curto prazo (*working memory*)?

Os autores deste artigo fizeram um estudo que envolveu 100 programadores onde lhes foi pedido para descreverem 12 funções diferentes e darem um novo nome aos identificadores existentes em cada uma dessas funções. As funções usaram três níveis de identificadores: uma letra, abreviatura e nome completo. As funções utilizadas para o estudo foram retiradas de livros de texto de área de engenharia informática.

Como resultados do estudo, os identificadores com o nome completo revelaram-se os de mais fácil compreensão. Em alguns casos não havia diferença entre o nome completo e a abreviatura. Mais, em certas situações, uma abreviatura bem escolhida era preferível uma vez que se memorizava mais facilmente.

Kondoh, 2006 [155] No seu artigo, Kondoh refere que pelo que viu em artigos de estudos sobre a instrução *goto*, a maior parte das opiniões e trabalhos realizados apareceram durante a controvérsia do *goto*, não tendo sido estudado através da medida facilidade de correcção (*easiness of correctness proof*), a qual era a intenção original de Dijkstra. Para além disso, refere que Dijkstra apontou que a eliminação do *goto* que estava na base do trabalho de Bohm e Jacopini era inútil para a melhoria do entendimento dos programas. Também Knuth propôs a programação estruturada usando a instrução *goto*.

Lawrie, 2007 [165] Este artigo descreve um estudo realizado com o objectivo de melhor entender o efeito da criação do nome do identificador na capacidade do programador de manipular o código, como por exemplo, a capacidade de compreender o código ou a capacidade de se lembrar de um identificador em particular. Participaram no estudo 128 pessoas.

O estudo foi realizado em duas partes. Na primeira parte é estudado o impacto de três níveis da qualidade do identificador: nome completo, abreviatura e uma letra. A abreviatura é formada com base na variante de nome completo, sendo formada pelas duas primeiras letras de cada uma das palavras. Na segunda parte é estudado a compreensão

das diferentes variantes do identificador, a memória do identificador, o impacto da experiência e escolaridade na compreensão do identificador, o papel do género (se homem ou mulher) na capacidade e confiança em descrever o código e por último se o tamanho do identificador tem impacto na memória.

Como resultados do estudo tem-se o seguinte: Na compreensão das diferentes variantes dos nomes dos identificadores existe mais evidência para o nome completo e em muitos casos, a abreviatura é de melhor compreensão na situação de descrição do código pelos dois géneros, homem e mulher. Em todos as situações os resultados apontam para melhor compreensão do nome completo e abreviatura, em vez de letra simples.

Relativamente à memória do identificador, não é claro qual a melhor variante.

Quanto à experiência e escolaridade, a confiança na compreensão do código têm menos impacto negativo quando a experiência e o nível de escolaridade são elevados. Deste modo, a confiança na compreensão aumenta com o aumento da experiência e escolaridade. Nesta situação, as mulheres têm menos confiança que os homens.

Por último, relativamente à memória do identificador, os nomes completos são mais difíceis de memorizar.

Lawrie, 2007 [162, 163] Os identificadores são claramente importantes para a compreensão dos conceitos num programa.

Nestes artigos, é caracterizada a utilização dos identificadores ao longo de três décadas, quatro linguagens de programação e 186 programas. Estes últimos incluem 48 milhões de linhas.

De modo a fazer a manutenção do código de forma eficaz é importante que os identificadores e os comentários mostrem claramente os conceitos que representam. São apresentados os resultados de um estudo para verificação de quando os identificadores estão bem formados sem qualquer informação adicional.

Os nomes dos identificadores devem ser compostos por várias palavras, sequências de caracteres com algum significado associado. São considerados dois tipos de palavras: palavras *hard* e palavras *soft*.

As palavras *hard* são separadas por marcadores de palavras como é o caso de *Camel-Case* e do *underscore*. Este último também denominado por *snake case*. Por exemplo,

nomeVariavel ou *nome_variavel* em que nome e variável são palavras *hard*.

Sempre que as palavras *hard* são palavras do dicionário ou abreviaturas é suficiente a divisão com palavras *hard*. Quando não caem numa dessas categorias então o identificador pode conter palavras *hard* que podem ser divididas em mais do que uma palavra *soft*. Um exemplo é a palavra *hashtable_entry* em que *hashtable* pode ter várias utilizações. Para verificação se os identificadores estavam ou não bem formados usaram a sintaxe baseada em concisão e consistência de Deissenbock e Pizka. Se o nome de um identificador está contido dentro de outro, existe então uma violação do requisito da consistência de sinónimos ou do requisito da concisão sintáctica ou ambos.

Definiram duas regras de violação aos identificadores:

Definição 1 (violação do Tipo I): Seja o identificador $id1$ a sequência de palavras *soft* $sw1sw2...swn1$. Os identificadores $id1$ e $id2$ falham no requisito consistência de sinónimo sintáctico ou o requisito de concisão sintáctica (violação do Tipo I) se $id2$ inclui a sequência de palavras *soft* $w1w2...sw1sw2...swn1...wn2$ ($id2 = w1w2...id1...wn2$).

Definição 2 (violação do Tipo II): Seja o identificador $id1$ a sequência de palavras *soft* $sw1sw2...swn1$. Os identificadores $id1$, $id2$ e $id3$ falham o requisito de concisão sintáctica e podem falhar o requisito de consistência de sinónimos (violação do Tipo II) se $id2$ inclui a sequência de palavras *soft* $w1w2...sw1sw2...wn2$ e $id3$ inclui a sequência a sequência de palavras $soft\ u1u2...sw1sw2...swn1...u3$. Esta definição para estar completa requer que $id2$ não contenha $id3$ e vice-versa.

Os resultados dos estudos mostram estatisticamente que, para as violações dos tipos I e II, há um aumento significativo em violações quando são incorporados sinónimos de linguagem natural. Uma vez que o aumento para as violações do tipo I e do tipo II é muito pequeno para ambos, então suportam a observação que os programadores usam um vocabulário limitado, não usam um número significativo de sinónimos em linguagem natural.

Por último, para as violações do tipo I, existe evidência que o código *open source* inclui mais violações. Para as violações do tipo II, os modelos de regressão indicam que a percentagem de violações diminuiu com o amadurecimento da área. Em sistemas de código aberto, esta percentagem aumenta.

Borstler, 2007 [32] No artigo é proposta uma medida de legibilidade, baseada na fórmula de Flesch, designada por *Software Readability Ease Score (SRES)*. Tendo em consideração que a fórmula de Flesch usa as sílabas, as palavras e as frases, a fórmula SRES é definida da seguinte forma com indicação dos parâmetros correspondentes à fórmula de Flesch):

1. Lexemas[14] são as sílabas.
2. Instruções são as palavras.
3. Unidades de abstracção são as frases.

$$SRES = ASL - 0.1 * AWL \tag{3.2}$$

em que:

ASL é o número médio de palavras por instrução ou bloco e AWL é o tamanho médio dos lexemas (identificadores, palavras reservadas e os símbolos).

Realizaram um estudo e nos resultados observaram uma correlação da fórmula com a qualidade dos exemplos de livros de texto de programação e que poderá ter interesse investigar a relação entre a SRES e a qualidade dos programas escritos pelos estudantes.

Sendo a legibilidade do software um factor tão importante na manutenção do software em geral, também poderá ser interessante investigar a utilidade da fórmula na previsão de vários aspectos na manutenção do software.

Segundo os autores, apesar de esta versão ainda não estar "refinada", comporta-se tão bem ou melhor que outras medidas de legibilidade, tais como as de Buse e de Posnet. É menos sensível em certos aspectos, como é o caso dos comentários e dos espaços brancos.

Fluri, 2007 [94, 95] A tarefa de comentar o código-fonte é por vezes negligenciada. A leitura do código-fonte é uma tarefa fundamental na engenharia de software e ler o código é mais frequente que escrevê-lo. Bons comentários permitem entender mais rapidamente o código e com mais detalhe e aumentam a sua legibilidade, sendo aspectos fundamentais na manutenção e na reengenharia de software.

[14] Lexema é uma instância de um *token*

É apresentado um estudo que consistiu em averiguar se o código e os comentários eram alterados ao mesmo tempo. Como resultado dos estudos concluíram que sempre que o código e o respectivo comentário são alterados em simultâneo, então são modificados 97% dos comentários. Se as alterações não são em simultâneo é muito raro haver alteração aos comentários.

Tan, 2007 [270] Neste artigo é abordado o problema de os programas serem mal comentados. Indicam dois tipos de inconsistências que designaram por *bugs* e "maus" comentários:

- *bugs*: o código não segue as assunções e requisitos especificados pelos comentários correctos do programa;
- "maus" comentários: comentários inconsistentes com código correcto, o qual pode confundir e "enganar" programadores de forma a introduzirem erros nas versões seguintes.

O objectivo dos autores era analisar automaticamente comentários escritos em linguagem natural por forma a extrair regras de programar implícitas e usar essas regras para detectar automaticamente inconsistências entre comentários e código-fonte, indicando quer os defeitos quer os "maus" comentários. Para isso, desenvolveram a ferramenta *iComment*.

A ferramenta foi usada nas versões mais recentes do Linux, Mozilla, Wine e Apache. A ferramenta inferiu 1832 regras muito precisas e detectou 60 inconsistências de comentários no código (33 *bugs* e 27 "maus" comentários), em que 19 deles foram confirmados pelos desenvolvedores.

Segundo os autores, os resultados da aplicação da ferramenta *iComment* demonstraram a sua eficácia, sendo um primeiro passo com resultados promissores para inspirar e motivar mais trabalho de investigação nesta direcção.

Tonella, 2008 [278] Neste artigo, o autor divide a qualidade do código em qualidade interna e externa. Faz a apresentação dos dois tipos, considerando as suas diferenças, e para cada um dos tipos é apresentado o contexto. É feita uma revisão às ferramentas e

técnicas disponíveis para a verificação e melhoria da qualidade de código interno, tendo em atenção a perspectiva do programador.

Concluem existir aspectos da qualidade (interna) do código, que são em grande parte negligenciados, mas que afectam profundamente a capacidade do programador para realizar modificação de código e as tarefas de depuração e correcção dos erros. Tratam-se de aspectos sobre a forma como a linguagem natural é incorporada no código como uma forma de modelação do domínio.

Na sequência dessas constatações, realizaram um estudo empírico ao código da plataforma ALICE quanto ao léxico utilizado por programadores (da plataforma ALICE). Foram estudados três aspectos:

- Estabilidade do léxico ao longo da evolução do sistema;
- Relação entre os grupos léxicos nos diferentes elementos no código-fonte;
- Conceitos representados por termos frequentes.

Com os resultados concluíram que o léxico tende a manter-se estável ao longo do tempo. No entanto, não ficaram a saber se isso é devido a uma lacuna de ferramentas de suporte ou se é uma estabilidade intrínseca. A classe léxica e os termos usados com mais frequência representam conceitos fundamentais do domínio de aplicação e pode ser visto como um bom ponto de partida para a extracção de conhecimento semântico a partir do código.

Butler, 2009 [42] Os nomes usados nos identificadores são componentes cruciais do código-fonte, com impacto na compreensão de programas por parte dos seus leitores. Foi realizado um estudo empírico que consistiu em avaliar a qualidade dos nomes no código-fonte do software, isto é, foi avaliada a correcta atribuição de nomes aos identificadores. Relf [220] desenvolveu 21 linhas de orientação para os nomes das variáveis quer para Java quer para Ada. Estas linhas de orientação focam-se na tipografia e no comprimento do nome do identificador. O autor usou a lista de orientações devidamente actualizada de Relf para o seu estudo e que se apresentam em seguida.

- Anomalia na capitalização: os identificadores devem ser capitalizados adequadamente;

- *Underscores* consecutivos: o uso de *underscores* consecutivos não é recomendado;
- *Underscores* externos: os identificadores não devem conter *underscores* no início e no fim;
- Palavras do dicionário: os nomes dos identificadores devem ser compostos por palavras que existam no dicionário e no caso de abreviaturas e acrónimos usar quando é mais comum a forma abreviada em vez da forma não abreviada;
- Palavras excessivas: os nomes dos identificadores não devem conter mais do que quatro palavras ou abreviaturas;
- Ordem da declaração dos identificadores nos tipos enumerados: as constantes das enumerações deve respeitar uma ordem, alfabética, ou de acordo com o que representam;
- Codificação do identificador: a informação do tipo do identificador não deve estar presente no nome do identificador;
- Nome do identificador demasiado comprido: deve ser evitada a utilização de nomes demasiado compridos para os identificadores;
- Anomalia da convenção de nomes: os nomes não devem ter mistura sem critério de maiúsculas e minúsculas;
- Nome do identificador demasiado curto: os nomes dos identificadores não devem ter menos de oito carateres. A excepção é para os seguintes nomes: *c, d, e, g, i, in, in.out, j, k, l, m, n, o, out, t, x, y, z.*

A qualidade do nome depende de diversos factores. O uso de tipografia como definido nas convenções de programação dá ao leitor pistas para o papel de cada identificador. No entanto, o uso da tipografia isoladamente é insuficiente; um bom nome para identificador deverá mostrar logo qual o conceito que representa e a sua função através do uso de linguagem natural.

Os autores também desenvolveram uma ferramenta para automatizar a extracção e análise de identificadores de código-fonte em Java. As linhas de orientação de *underscores* consecutivos e a ordem das enumerações foram excluídos da análise estatística.

Como conclusão, os autores adoptaram um conjunto de 11 linhas de orientação para os nomes em linguagem natural para Java. Foram escolhidas estas linhas de orientação por terem sido avaliadas empiricamente e serem mais detalhadas que outras propostas na literatura. Geralmente, os programadores podem cometer erros mais significativos, independentemente da qualidade dos identificadores. No entanto, devido a possivelmente às metodologias de desenvolvimento ou convenções de nomenclatura utilizadas, determinados tipos de defeitos dos identificadores podem indicar a presença de defeitos mais graves no software.

Binkley, 2009 [27] As convenções de nomenclatura são normalmente adoptadas no sentido de melhorar a compreensão de nomes. De acordo com o artigo, apesar da tendência recente da utilização do estilo *CamelCase* em vez do *underscore*, os investigadores na área da psicologia argumentam que é uma má escolha. Em discussões informais, argumentam que esta prática deverá aumentar a dificuldade de leitura.

Foi realizado um estudo empírico com o objectivo de estudar o impacto do estilo do identificador na legibilidade do código. Para avaliar o impacto do estilo, o estudo avalia a velocidade e precisão com que os indivíduos encontram um dado identificador. O estudo descrito no artigo também mostra que os nomes usados no código existente são demasiado longos para a memorizar na memória de curto prazo de um grupo de programadores. Isto fornece evidência que os engenheiros de software necessitam considerar os nomes curtos e mais concisos. No entanto, uma vez que o estudo considera os nomes individuais extraídos de código, há uma tendência de não serem valorizados, uma vez que não necessitam de se lembrar do contexto do nome.

O estudo envolveu 135 programadores e não programadores, dos participantes programadores 32% estavam inscritos em cursos de ciências da computação, 16% tinham menos de um ano de formação, 7% tinham um a dois anos de formação e 8% com mais de dois anos de formação.

No final do estudo foi perguntado aos participantes se tinham preferência por um estilo ou por outro. Dos que não tinham formação 46% preferiram os *underscores* e 45% não tinham preferência. Conclui-se que os restantes não responderam. Dos que tinham formação, independentemente do número de anos, 38% manifestaram preferência pelos

underscores. Dos restantes, deram preferência ao *CamelCase*, 32% dos que tinham menos de um ano de formação e 55% com mais de dois anos de formação. Mais uma vez, uma vez que o autor não referiu, presume-se que houve participantes que não responderam.

Como resultados, os indivíduos sem treino têm mais dificuldade a reconhecer identificadores que usam *CamelCase*. Os indivíduos com formação demoram menos tempo a encontrar os identificadores com *CamelCase* do que os identificadores com *underscores*.

Uma vez que o tipo de estilo a utilizar varia com as pessoas, não se deve definir um estilo do nome para *CamelCase* ou *underscore* como uma prática.

Apesar de não haver uma separação clara entre estilos de programação e práticas de programação, os estilos podem ser considerados com aspectos de gosto pessoal e/ou tipográfico, enquanto que as práticas, tal como usadas neste texto, são em geral, ou em grande medida, independentes do gosto pessoal do programador, ainda que possam variar, como variam as pessoas.

Como conclusões, o número de sílabas num identificador influencia fortemente o tempo requerido para o programador compreender o código-fonte. Os resultados indicam que o comprimento excessivo tem impacto negativo no tempo e pode levar a correcções. Também é necessária precaução para garantir a escolha de vocabulário consistente. Os princípios de bons nomes para além de limitar o tamanho dos nomes também reduz a necessidade de vocabulário especializado. Para os autores, a ideia nuclear é que num programa um conceito deverá ser sempre referenciado pelo mesmo nome. Os nomes deverão conter informação suficiente para se perceber o conceito que representam e ao mesmo tempo não serem difíceis de memorizar na memória de curto-prazo.

Binkley, 2009 [28] Neste artigo são apresentados os resultados de um estudo sobre a interacção entre as limitações da memória de curto prazo do programador e os identificadores num programa. É também alvo de estudo o impacto das limitações da memória humana tendo em consideração que as actividades de compreensão por parte do programador se tornam cada vez mais difíceis à medida que a memória de curto prazo de um programador se torna sobrelotada.

Foram estudadas cinco hipóteses:

1. O comprimento aumenta o tempo de estudo? - Esta hipótese estabelece que os identificadores mais longos requerem mais tempo de entendimento.
2. O comprimento reduz a exactidão? - Esta hipótese refere que os identificadores longos reduzem a capacidade das pessoas se recordarem das diferentes partes do nome.
3. Partes de código que o programador está familiarizado melhoram a exactidão? - Esta terceira hipótese considera que as partes de código que já existam na memória persistente ajudam a melhorar a compreensão com exactidão.
4. A experiência do programador melhora a exactidão? - Esta hipótese reforça a anterior, na medida em que se o programador é experiente significa que partes do código já existem na memória persistente.
5. O impacto da exactidão pela memória persistente, pelo comprimento dos nomes e pela experiência do programador são independentes?

Os resultados dos estudos mostram que o número de sílabas num identificador influência bastante o tempo requerido para o programador compreender o código-fonte. Também os identificadores com os quais os programadores se encontram familiarizados (que já se encontram na memória persistente), melhoram a exactidão e quando o comprimento do identificador aumenta, a experiência do programador influencia o desempenho.

Os resultados também reforçam as propostas anteriores de outros autores para o uso de vocabulário limitado, consistente e sistemático em nomes de identificadores. É importante escolher um vocabulário consistente. Também o comprimento excessivo afecta negativamente o tempo e a exactidão. Bons princípios de atribuição de nomes aos identificadores limitam o comprimento e reduzem a necessidade de vocabulário especializado.

Buse, 2010 [38, 39] Neste artigo é explorado o conceito de legibilidade do código e é estudado a sua relação com a qualidade de software. Os autores realizaram um estudo que se pode considerar dividido em duas partes. A primeira pretendia extrair um grande número de julgamentos/avaliações de legibilidade de pequenas amostras de código, designadas por *snippets*, cada um com 7 linhas. Para o estudo era importante que estes fossem relativamente pequenos para permitir uma avaliação mais correcta e logicamente coerentes.

Na segunda fase desta primeira parte, desenvolveu um modelo utilizando métodos de aprendizagem máquina. O modelo utilizava valores de 25 características obtidas para cada um dos *snippets*. Os resultados obtidos na primeira parte foram usados para avaliar a métrica que desenvolveu. Aí, realizou uma análise em componentes principais em que 95% do total da variância acumulada pode ser explicada por apenas 8 componentes. Ou seja, deverá existir uma sobreposição significativa entre as características que utilizou.

As 8 características são as seguintes:

- Número médio dos identificadores por linha;
- Comprimentos médio da linha;
- Número médio de parêntesis e chavetas;
- Comprimentos máximo da linha;
- Número médio do ponto ".";
- Número máximo de identificadores;
- Tamanho médio da indentação.

Num artigo anterior o mesmo autor refere 98% da variância explicada por 6 componentes [38]. Aparentemente, refere-se ao mesmo estudo o que é no mínimo estranho, mas não foi possível encontrar qualquer explicação para essa alteração.

Outras conclusões:

- Factores como o comprimento médio da linha e número médio de identificadores por linha são importantes para a legibilidade;
- Por outro lado, tamanho médio do identificador não constitui um factor preditivo da legibilidade, acontecendo o mesmo com a estrutura de decisão if, ciclos e operadores de comparação.

Na segunda parte, foi estudada a relação da métrica com medidas de qualidade, como: defeitos encontrados, alterações ao código entre versões e ainda a evolução da legibilidade ao longo do tempo dos projectos. Tratou-se sempre de projectos de código aberto. Como principais conclusões, a métrica correlacionava-se com essas noções de qualidade e no caso da tendência nem todos os projectos evoluíram da mesma forma.

O autor afirmou não ter encontrado outros trabalhos similares na área da legibilidade de código-fonte. É importante referir que foram usadas as referências de dois artigos do autor e que existem algumas discrepâncias relevantes entre os dois artigos como a que foi apontada. Para além de não apresentar de forma clara quais os objectivos da sua investigação, o que dificulta a leitura. Foram identificados vários factores que influenciam a legibilidade, mas o artigo não identifica valores para esses factores, e.g., qual o número médio de identificadores por linha.

Mais importante, o modelo de Buse foi apenas usado com pequenos fragmentos de código. Na presença de fragmentos maiores produz sempre o resultado 0 [135].

Haiduc, 2010 [112] Para além da grande quantidade de código e da sua complexidade, os nomes, os comentários e a aparência, também irão afectar a legibilidade do código-fonte.

Considerando que a legibilidade do código envolve a aparência sintáctica do código, havendo uma fraca legibilidade isso é entendido como uma barreira para compreensão de programas, tendo repercussão na semântica.

No artigo, é proposta uma técnica para resumir automaticamente o código-fonte, aproveitando a informação léxica e estrutural, com base na qual foi desenvolvida uma ferramenta. Propõem que a ferramenta seja integrada nos IDE's. Portanto, este artigo não se insere no âmbito do presente trabalho.

Schulte, 2010 [238] Existe um grande debate no ensino/aprendizagem da programação acerca de como os alunos podem progredir, em termos de tópicos da sequência de aprendizagem, tarefas de aprendizagem, métodos de ensino, entre outros.

Os resultados obtidos mostram que os seguintes aspectos são críticos para o processo ensino/aprendizagem:

- Domínio de conhecimento para compreender os programas;
- Tarefas de leitura e compreensão dos programas;
- Representação mental flexível.

Green, 2011 [105] O autor afirma que, tal como a arte, o código é simultaneamente subjectivo e não subjectivo. Os aspectos não subjectivos da codificação incluem ideias que devem ser seguidas para criar código de qualidade, tais como, padrões de concepção e a utilização de bibliotecas. No entanto, os conceitos referidos são o alicerce para o desenvolvimento de software de qualidade e de fácil manutenção. E são as nuances das técnicas do programador que realmente fazem a grande diferença para se conseguir produzir código claro, de fácil manutenção e compreensível e ao mesmo tempo com a capacidade de comunicar claramente a sua funcionalidade e uso.

Também para o autor isto acontece porque as pessoas têm afinidade com o seu estilo de programação pessoal, da mesma forma que a criança quando cresce cria o seu próprio estilo de escrita. Quando isto acontece, a pessoa cria os seus hábitos () bons e maus) e as suas preferências. Isto é importante na medida em que cada pessoa pode dar um significado único ao código.

O autor considera as boas práticas:

- Considerar um programa como uma tabela;
- Deixar que o inglês simples seja o guia;
- Confiar no contexto de modo a simplificar o código;
- Usar espaços em branco para melhor visualizar a estrutura;
- Deixar as estruturas falarem por elas próprias;
- Enfoque no código e não nos comentários.

Estas linhas de orientação são usadas na educação tendo também mérito no ambiente industrial. Os alunos que aprenderam estas orientações, a maioria deles gosta de as usar, ou com algumas variantes.

Podugu, 2011 [211] A legibilidade do código-fonte está relacionada com a facilidade de manutenção. Neste artigo, é apresentada a relação entre métricas de software e a facilidade de manutenção. Consideram muito importante este tópico, uma vez que, 50 a 60% do ciclo de vida de desenvolvimento de software é gasto na manutenção (como se referiu antes, existem autores que referem valores superiores). Os autores definem

legibilidade como sendo a facilidade de leitura e compreensão do código. A legibilidade de um programa, será portanto um julgamento sobre a compreensão do programa.

Num ambiente académico onde os estudantes usam programas como ferramentas de aprendizagem, então a legibilidade desses programas deve ser boa e que ajude nesses mesmos programas. Estudos sobre a legibilidade de textos mostram que se um texto é de legibilidade fraca então é difícil de entender. Aplicando o mesmo princípio aos programas de computador, então um programa que é difícil de ler também será difícil de compreender.

Os autores apresentam um modelo descritivo de legibilidade do software baseado em características simples que podem ser extraídas automaticamente dos programas. Este modelo de legibilidade no software relaciona-se fortemente com o modelo cognitivo das pessoas e também com noções externas da qualidade do software, tais como detectores de defeitos e alterações do software. Utilizou a métrica definida por Borstler, SRES para gerar o modelo de legibilidade de software.

Para efeitos do estudo considerou que a legibilidade seria uma medida estática baseada nos elementos individuais independentes tais como, os identificadores.

Posnett, 2011 [213] No artigo é referido que a legibilidade do código é um aspecto central para os engenheiros informáticos. Código legível é frequentemente considerado de mais fácil manutenção e o código que hoje é mais legível presume-se que permanecerá mais fácil de ler, compreender e manter.

É assumido que a relação entre legibilidade e compreensão é análoga à relação entre as análises sintáctica e semântica, respectivamente. A legibilidade é uma propriedade subjectiva do código e não é passível de medição automática objectiva.

O artigo tem por base o trabalho de Buse, o qual apresentou um modelo de medição composto por 8 características. Isto sugeria que algumas características contribuíam para a legibilidade mas também que um modelo mais simples poderia contribuir para uma teoria mais geral de legibilidade. Relativamente ao modelo de Buse verificaram que não era generalisável a *snippets* com dimensão bastante superior às 7 linhas dos *snippets* de código usados por Buse.

Nesse sentido, é apresentada uma teoria de legibilidade (intuitiva segundo os autores), baseada no tamanho total do programa e entropia do código. Para um dado nível de

entropia, um aumento no tamanho contribui positivamente para a legibilidade.

Posnett identificou 3 variáveis que considera que desta forma o seu modelo supera o de Buse. Usando a seguinte função lógica:

$$1/(1 + e - z) \tag{3.3}$$

Tem-se um modelo mais simples usando a seguinte expressão de z:

$$z = 8.87 - 0.033V + 0.40Linhas - 1.5Entropia \tag{3.4}$$

Em que:

V é o volume do programa das Métricas de Halstead.

Posnett e os seus colegas argumentam o facto que esta abordagem fornece uma teoria bem fundamentada para medição da legibilidade. Também referem que a medida apresentou melhores resultados que a medida de Buse, pelo facto de ter sido treinada com um conjunto de dados bastante limitado.

Wang, 2011 [286, 287] No artigo é referido que as boas práticas de programação e linhas de orientação apontam para a utilização de espaçamentos horizontal e vertical para delimitar de forma clara os vários segmentos do código que representam diferentes passos dos algoritmos ou acções de alto nível. Os autores consideram que, infelizmente, os programadores nem sempre seguem essas mesmas linhas de orientação.

Com o objectivo de aumentar a legibilidade do programa para o tornar mais fácil de entender, as linhas de orientação para o código incluem habitualmente normas de formatação como, por exemplo, indentação e inserção de chavetas nas instruções condicionais e de repetição. Ao mesmo tempo, as boas práticas de programação sugerem o uso de linhas em branco. Neste artigo, os autores apresentam uma solução heurística para o problema da inserção automática de linhas em branco num programa.

Semelhante à indentação, o espaçamento vertical (isto é, a inserção de linhas em branco) permite melhorar a legibilidade por separar visualmente segmentos de código, em que cada segmento de código possui instruções relacionadas logicamente entre si. O autor afirma que estudos recentes de avaliação da legibilidade do software referem que as linhas em branco simples são mais importantes do que comentários.

Na prática, o espaçamento vertical não é usado como deveria. Num estudo de 16,236 métodos que têm 16-40 linhas de código, 35% dos métodos contêm apenas 0-1 linha em branco.

A inspecção manual mostrou que muitos métodos que são maiores têm poucas linhas em branco, resultando em grandes blocos de código, sem segmentação vertical para facilitar a leitura. Os autores também verificaram que o espaçamento vertical é utilizado de forma inconsistente.

No artigo, é apresentado o primeiro sistema automático para inserção de linhas em branco no código-fonte para melhorar a legibilidade do código. O resultado da avaliação por parte de programadores que avaliaram as linhas em branco geradas pela ferramenta, considera que separou adequadamente os diferentes blocos que estão relacionados logicamente entre si. O primeiro estudo de avaliação mostra que 88,7% das vezes, as linhas em branco geradas pelo sistema são tão bem geradas quanto aquelas criadas por desenvolvedores originais, podendo até em certas situações ser melhor. O segundo estudo de avaliação mostra que as linhas em branco geradas pelo sistema automático está de acordo com a maioria das opiniões dos programadores sobre o local onde o espaçamento vertical deve ser colocado.

Dhillon, 2012 [74] Este artigo apresenta o resultado de um estudo empírico para investigar a relação entre os *bad smell* e a probabilidade de erro nas classes. No estudo foram considerados três níveis de severidade do erro. A investigação foi conduzida no contexto de sistema já disponibilizados ao cliente (*pos-release*) em processo de evolução.

Como resultado do estudo é mostrado que alguns *bad smell* estão associados positivamente com a probabilidade de erro na classe nos três níveis. Neste contexto, o resultado suporta a utilização dos *bad smell* como método sistemático na identificação e *refactoring* de classes problemáticas. Os *bad smell* usados no estudo são:

- GC - *god class*
- GM - *god method*
- RB - *refused bequest*
- SS - *shotgun surgery*

- FE - *feature envy*
- LMC - *long message chain*
- Violação do ISP - *Interface Segregation Principle*

Como conclusões, a utilização de modelos de identificação podem ajudar a prever os erros e que o *refactoring* de uma classe para além de melhorar a qualidade da arquitetura reduz a probabilidadede a classe ter erros no futuro.

Dorn, 2012 [78] O autor propôs uma nova medida/modelo de legibilidade por considerar que as anteriores não mediam as várias dimensões da legibilidade.

Aplicou o modelo a vários *snippets* de código de 3 tamanhos distintos, aproximadamente 10, 30 e 50 linhas de código. O modelo proposto usa aprendizagem automática para determinar os seus parâmetros pelo que os valores encontrados são válidos para as (360) amostras de treino, mas a possibilidade de generalização é considerada questionável pelos autores.

Posteriormente, comparou os resultados com os resultados de uma avaliação humana. A participação das pessoas foi feita através de um inquérito via Web e 5468 pessoas que avaliaram pelo menos uma das 20 amostras de código (*snippets*) disponibilizadas a cada pessoa (do total de 360 amostras existentes), mas por diversas razões, algumas dessas contribuições, foram eliminadas pelo autor do artigo.

Após a fase de avaliação humana, o autor determinou quais as características que melhor explicavam esta avaliação, tendo resultado nas características seguintes (para todas as linguagens):

- Comprimento da linha;
- Linhas compridas;
- Área do operador;
- Transformada Discreta Fourier (TDF) da sintaxe;
- Transformada Discreta Fourier (TDF) dos comentários;
- Comprimento mínimo do alinhamento;

- Área de string para área das palavras reservadas.

Também foi verificado que características visuais e espaciais podem melhorar a exactidão das métricas de legibilidade e que as características que influenciam a legibilidade variam com as linguagens utilizadas, conforme indicado no estudo descrito.

Nem os resultados, nem o artigo contêm informação adicional que permita perceber o significado concreto de cada característica. Por exemplo, o que se entende por uma linha comprida? O modelo envolve vários métodos estatísticos/matemáticos (e.g. transformadas de Fourier). Adicionalmente, os valores das transformadas não permitem inferir acerca dos valores originais com base nos quais foi calculada. Portanto, este trabalho acaba por não fornecer resultados utilizáveis. Pode-se dizer que a sofisticação estatística/matemática não é suficiente para se obterem resultados com valor para a investigação.

Também mostrou que a medida de Buse pode produzir valores inválidos, possivelmente por não captar aspectos visuais e linguísticos, ou quando o *snippet* está bastante acima das 7 linhas.

Namani, 2012 [197] Para os autores, a legibilidade de software está relacionada com a qualidade do software. No artigo é definida uma nova métrica para medir a legibilidade do código-fonte.

A nova métrica de legibilidade é representada por:

$$CR = LOC + LL + NOCL + NOBL + NLAS + BSAD + NOM \tag{3.5}$$

sendo,

LOC - as linhas de código,
LL - o comprimento da linha,
NOCL - presença de linhas de comentário no programa,
NOBL - o número de linhas em branco,
NLAS - a quebra de linha a seguir ao ponto e vírgula,
BSAD - o espaço branco a seguir à instrução,
NOM - o número de métodos.

Os autores referem que o resultado é apresentado em percentagem de legibilidade, no entanto, não indicam como obtêm esta percentagem. Quanto maior é esta percentagem mais legível é o código-fonte.

Também aplicaram algumas métricas já existentes, nomeadamente, a fórmula ARI a fórmula FOG e a fórmula SMOG. Os resultados foram comparados entre si. Para isso, foi desenvolvido um protótipo que recebe como entrada o código-fonte e aplica-lhe as métricas. As métricas geram estatísticas de legibilidade. Ao mesmo tempo também é disponibilizado esse código de entrada a 50 engenheiros de software e pedido que indiquem um valor (em %) da legibilidade do código.

Os resultados das métricas e dos engenheiros foram então comparados, verificando-se serem idênticos.

Sharafi, 2012 [242] Neste artigo são apresentados os resultados de um estudo experimental realizado com quinze homens e nove mulheres para estudar o impacto da utilização do estilo *CamelCase* versus *underscore* na construção dos nomes dos identificadores.

Os autores apresentam as duas definições destas convenções.

Definição 9 (*CamelCase*)**.** *A convenção CamelCase para nome de identificador é a prática de criação de nomes pela concatenação de termos com a primeira letra em maiúsculas. O nome está associado ao efeito das boças do camelo.*

Definição 10 (*Underscore*)**.** *A convenção underscore para nome de identificador é a prática de criação de nomes pela inserção de um underscore na concatenação de termos do nome.*

Apesar de ser a convenção usada na linguagem Java, tal não acontece com as outras linguagens. Segundo os autores, muitas das outras linguagens usam a convenção *underscore* para formar a nomencãtura.

Como conclusão, os autores verificaram que, no estudo, as mulheres esforçam-se mais na leitura e examinam com detalhe. Pode-se dizer que é uma observação de baixo nível. Os homens fazem uma análise mais rápida correndo mais o risco de escolha de opções erradas. No entanto, não existe diferença significativa entre os homens e mulheres no tempo gasto

nas tarefas de compreensão. Também concluíram que o estilo não tem impacto no esforço de compreensão dos programas.

Sivaprakasam, 2012 [255] No artigo é referido que, pela análise de diferentes actividades de desenvolvimento, é mostrado que legibilidade do software tem um efeito global nos custos de desenvolvimento de software e é independente do tamanho do software.

Criaram uma medida de legibilidade automática que, segundo os autores, pode ser 80% eficaz, e melhor que o o ser humano na previsão de legibilidade. A medida criada baseia-se na fórmula de *Flesch Reading Ease* mas para textos com mais de 100 palavras.

Quanto aos *snippets* consideram que é uma pequena parte do código. Um *snippet* não inclui linhas precedentes nem pode estar entre linhas que não são declarações simples, tais como comentários, cabeçalhos de função, as linhas em branco, ou cabeçalhos de instruções compostas como *if-else*, *try-catch*, *while*, *switch*, e *for*. Os *snippets* devem ser pequenos. No entanto, se demasiado pequenos, podem obscurecer considerações importantes de legibilidade. Em segundo lugar, os *snippets* devem ser logicamente coerentes para permitir aos anotadores identificar o contexto e analisar a sua legibilidade. Os *snippets* são dados aos anotadores para os avaliarem. Os anotadores são as pessoas que escrevem qual a funcionalidade do código.

Os autores consideram as seguintes como propriedades do código: Comprimento da linha em carateres, identificadores, comprimento do identificador, indentação (precedida de espaços), palavras reservadas, parêntesis, números, comentários, ponto (“.”), ramificações e ciclos.

Binkley, 2013 [26] No artigo é afirmado que os identificadores são nucleares na compreensão de programas.

São investigados dois estilos de atribuição de nomes aos identificadores: *CamelCase* e *underscore*. Para isso foram realizados cinco estudos. O primeiro estudo serviu para analisar como as pessoas lêm os dois estilos. Os restantes quatro estudos serviram para analisarem as implicações do estilo. O estudo envolveu 150 participantes de duas universidades formando três grupos.

O primeiro grupo era de 135 estudantes e incluia programadores e não programadores.

Os programadores deste grupo estavam mais habituados ao estilo *CamelCase* e as idades estavam compreendidas entre os 17 e os 22. 54% dos participantes era do sexo masculino, sendo dos programadores 67% do sexo masculino. Dos participantes que não eram programadores, 46% preferiram o estilo *underscore* e 45% não tinham preferência. Já entre os participantes que tinham conhecimentos de programação, 38% preferiram o estilo *underscore*. O segundo grupo é um subconjunto de programadores do primeiro grupo de 19 anos, frequentando maioritariamente o segundo ano do curso. Portanto, 47% dos elementos do grupo tinham 1 a 2 anos de treino de programação. Os restantes elementos tinham 4 anos de treino. Neste grupo, 79% dos participantes eram do sexo masculino. O terceiro e último grupo incluiu 15 programadores de uma universidade diferente dos dois grupos anteriores, sendo apenas 2 do sexo feminino. Estes programadores tinham experiência dos dois estilos de nomes. Deste grupo, faziam parte 7 alunos sem o ensino secundário, 6 alunos com o ensino secundário, e 2 alunos da faculdade. 47% dos elementos deste grupo preferiram o estilo *CamelCase* e 47% preferiram *underscore*. Os restantes não tinham preferência.

No primeiro estudo foi pedido aos participantes para identificarem nomes num conjunto de *snippets* de código contendo elementos distractores. No segundo estudo foi pedido aos participantes para procurar num fragmento de código todas as ocorrências de um dado identificador. No terceiro estudo, o contexto foi um parágrafo em linguagem natural. O parágrafo foi modificado de modo a incorporar um dos dois estilos. Neste estudo o objectivo foi examinar as diferenças entre a compreensão da leitura em geral e da leitura de código-fonte específico. No quarto estudo, foi pedido aos participantes para lerem e tentarem compreender *snippets* isolados de código que incluíam ambos os estilos. Foi também pedido para identificarem os identificadores encontrados. No último estudo foi pedido ao participantes para estudar uma função e verbalmente dizer o que faz o código. O objectivo deste estudo foi determinar o efeito do estilo do identificador na compreensão de um dado pedaço de código.

Foram usadas três técnicas de modelação estatística, uma para variáveis de resposta numérica e as outras duas para variáveis de resposta por categoria.

Como conclusões, os resultados dos estudos referem que, de um modo geral, a escolha do estilo de criação de nomes dos identificadores recai sobre *CamelCase* sobretudo nos

programadores iniciados.

Hansen, 2013 [116] Que factores têm impacto na compreensibilidade do código é a questão colocada pelos autores. Para isso, os autores realizaram um estudo experimental em que os participantes com experiência em programação prevêm qual o *output* exacto de dez pequenos programas escritos em Phyton. A maioria dos programas tem menos de 20 linhas de código. Para cada um dos dez programas, existem duas e três versões, sendo as diferenças entre eles muito subtis. O objectivo é analisar o impacto ao nível da correcção e do tempo de resposta para a correcção. Juntaram 1602 testes de 162 participantes durante cerca de 3 meses (de 20 de Novembro 2012 a 19 de Janeiro de 2013). A análise dos dados foi executada em R, e todas as regressões foram feitas com as funções *built-in* "lm" e "glm".

A notação dos aspectos físicos é frequentemente considerada superficial e pode ter um impacto profundo no desempenho. No estudo, muitas vezes os programadores agruparam o código em blocos, cujas instruções estavam relacionadas entre si. Para isso usaram o espaçamento vertical.

Os resultados mostram que na maior parte dos casos o desempenho aumenta com a experiência dos programadores. No entanto, os resultados mostram que este espaçamento superficial pode, mesmo a programadores com experiência, esconder um programa incorrecto. A notação pode também fazer com que um programa simples se torne mais dificil de ler.

O aspecto físico da notação é muitas vezes considerado superficial. Uma conclusão a que chegaram os autores é que o espaço vertical é mais importante que a indentação quando os programadores avaliam as instruções que pertencem ou não ao mesmo corpo de uma repetição.

Os resultados também mostram que o espaçamento horizontal pode realçar a estrutura entre os cálculos relacionados.

Sasaki, 2013 [234] Neste artigo é proposta uma técnica para reorganização das instruções em blocos de código de modo a melhorar a legibilidade do código. Para isso, foi conduzido um estudo experimental com 44 participantes em que são reordenadas instru-

ções em cada um de 215 métodos. Os resultados experimentais revelaram que na maior parte dos casos os métodos reordenados têm melhor legibilidade que os originais.

A forma de reordenação foi a seguinte:

1. Mover as instruções para os blocos mais interiores de modo a reduzir o âmbito das variáveis.
2. Juntar as instruções que estão relacionadas entre si, que pertencem ao mesmo bloco, de modo a diminuir a distância entre a definição de uma variável e a sua referência.

Propõem para o futuro determinar mais factores para a ordenação de instruções, melhorar a sua técnica e desenvolver um ambiente de reordenação interactivo.

Scanniello, 2013 [235] Neste artigo é apresentado o resultado de um estudo experimental controlado para investigar quando é que a presença de identificadores abreviados existentes no código-fonte afecta a capacidade de programadores inexperientes identificarem erros no código-fonte. Foram usadas 4 aplicações implementadas em linguagem C. Após se terem omitido os comentários no código, criaram-se 4 tipos de mutações ao código:

- LCO - operador de mudança da literal (*literal change operator*);
- LOR - substituição do operador de linguagem (*language operator replacement*);
- CFD - operador de interrupção do controlo de fluxo (*control flow disrupt operator*);
- VRO - operador de substituição da variável (*variable replacement operator*).

O estudo foi conduzido em condições controladas com alunos cuja média das suas idades era de 21 anos. Os alunos participaram no regime de voluntariado e foram informados antes do início do estudo. Após a sessão de esclarecimento, os participantes foram divididos em 4 grupos: A, B, C e D. Os grupos A e B tinham ambos 13 estudantes, o grupo C tinha 12 e o grupo D tinha 11. Os alunos foram distribuídos pelos diferentes grupos de acordo com as notas. Os que tinham melhores notas estavam no grupo A, enquanto que os com piores notas estavam no grupo D.

Tiveram de realizar as seguintes tarefas:

1. Identificar os erros.
2. Corrigir os erros.
3. Preencher um questionário após o estudo.
4. Expandir as abreviações.

Os resultados indicam que a diferença da utilização de nomes abreviados e de nomes completos não têm significado estatístico relativamente ao tempo necessário para completar tarefas e relativamente ao número de erros detectados e corrigidos pelos participantes.

Tashtoush, 2013 [271] No artigo é descrito um inquérito a programadores experientes de modo a avaliar 25 características[15] de legibilidade de código-fonte considerando se tinham efeito positivo ou negativo.

Do estudo concluíram a importância de várias características e ausência de impacto de outras tal como ilustrado pela tabela 4.1. A direcção da relação com a legibilidade é indicada pela segunda coluna. O sinal '+' indica que a característica tem efeito positivo. O sinal '-' indica que tem efeito negativo. A ausência de relação é indicada com o valor 0.

Do inquérito concluíram que o código legível é menos erróneo, mais reutilizável, mais fácil de manter, rápido de modificar e mais consistente.

Tabela 3.2: Características de Legibilidade.

	Característica	**Direcção**
1	Média das variáveis com nomes com significado	+
2	Média das funções com nomes com significado	+
3	Linhas em branco por linha de código	0
4	Indentação	0
5	Comentários	+
6	Âmbito	0
7	Comprimento das linhas	0
8	Fórmulas aritméticas	-
9	Média dos *if-else*	0
10	*if* imbricados	-
11	Média dos ciclos *for*	0
12	Ciclos imbricados	-
13	Funções recursivas	-

[15] ao longo do texto os autores referem apenas 22

14	Arrays	0
15	Distribuição das classes	0
16	Herança	0
17	Redefinição (*overriding*)	0
18	Consistência	+
19	Comprimento do nome do identificador	0
20	Frequência do identificador	0
21	Espaçamento	0
22	Instrução *switch*	0

Lee, 2013 [167] Para os autores, a legibilidade do código é extremamente importante pois afecta o entendimento do contexto do código, facilitando desta forma a comunicação e a colaboração entre os membros da equipa de desenvolvimento de software. Os autores apresentam duas definições de legibilidade, a legibilidade de textos (de acordo com a definição de Buse) e a legibilidade na engenharia de software. Esta última é a seguinte:

Definição 11 (Legibilidade na Engenharia de Software)**.** *Na engenharia de software a legibilidade é definida como um julgamento do desenvolvedor de quão fácil o código-fonte é de entender, trabalhar em equipa e fazer a sua manutenção.*

Os autores referem que num questionário realizado por LaToza [160], 66% dos desenvolvedores concordam que o entendimento do código é um dos problemas mais sérios que afecta o desenvolvimento de software. Existem várias linhas de orientação e convenções com o objectivo de produzir código melhor e com mais qualidade. Tendo em consideração que a manutenção representa 70% dos custos (de todos os estágios do desenvolvimento de software), é uma das razões pela qual é tão importante não esquecer a legibilidade do código. Mais ainda, o software não é mantido sempre pela mesma pessoa durante todo o seu tempo de vida.

Neste artigo, os autores apresentam os resultados de um estudo para verificar se o código ilegível tem mais regras de violação que o código legível. Para o seu estudo foram analisados quatro projectos de código aberto *open-source* escritos em Java. Para medir a legibilidade do código usaram o modelo de Buse [39]. Os projectos tinham níveis diferentes de maturidade. Determinaram um conjunto com um total de 126 regras de violação de acordo com as directrizes e convenções de codificação em 1.870 ficheiros.

Como resultados, este estudo identificou várias violações no código que tinham influência na sua legibilidade. Os resultados do estudo mostraram que as regras de violação que tornam o código menos legível são:

- Código com modificadores que não são usados;
- Código com modificador redundante;
- Verificação do nome de constante, devido à não utilização da convenção de nomes.

As regras de violação que tornavam o código mais legível são as seguintes:

- Verificação da chaveta esquerda;
- Verificação da necessidade de utilização de chavetas na instrução condicional *if-else*;
- Verificação de espaço branco a seguir à vírgula. Por exemplo, numa lista de elementos;
- Verificação da correcta utilização de tabulação.

Para os autores, os estilos de codificação podem diferenciar entre os programadores, do domínio dos projectos e da maturidade do projecto.

Aman, 2014 [4] De um modo geral, os comentários melhoram a legibilidade dos programas, sendo portanto "inofensivos" para a qualidade do software. Mais, por vezes os comentários podem ser adicionados para compensar a falta de legibilidade em certos programas.

Este artigo apresenta um estudo empírico sobre a relação entre comentários e defeitos de programas escritos em Java. Usaram para o efeito quatro aplicações de software aberto escritas em Java. No estudo foram analisados dois tipos de comentários, os comentários dentro do corpo do método designados por comentários internos, e os comentários antes da declaração do método designados por comentários de documentação. Cada um dos tipos de comentários tem um propósito específico:

- Comentários internos - são utilizados como ajuda no código;
- Comentários de documentação - são utilizados como manual do programador.

Para o estudo foram formuladas duas questões de investigação:

- A propensão para defeitos dos métodos difere significativamente com o tipo de comentário? Para os autores, se houver uma diferença significativa na propensão para defeitos, a categorização dos comentários e a respectiva análise são relevantes.
- A quantidade de comentários tem alguma relação com a propensão a falhas? Se existir relacionamento notável com a propensão para defeitos é importante dar ênfase à quantidade de comentários durante o desenvolvimento ou revisão de código.

Os resultados empíricos revelaram que os comentários internos têm impacto significativo na propensão para defeitos dos métodos.

Hill, 2014 [121] Para a técnica de *CamelCase* dos nomes dos identificadores não existe nenhuma convenção para a inclusão de acrónimos nos identificadores com *CamelCase*. O acrónimo pode ter todas as letras em maiúsculas ou apenas a primeira letra. A decisão de adopção quer de um quer de outro depende exclusivamente da legibilidade. Este artigo descreve um estudo empírico do estado de arte das técnicas de quebra dos nomes dos identificadores e de algoritmos automáticos de quebra de construção dos nomes dos identificadores.

O objectivo dos algoritmos de divisão de identificadores é tomar o identificador como *input*, gerando uma lista das palavras existentes no nome produzindo assim um dicionário de palavras. Este estudo apesar de referir a importância dos nomes dos identificadores na legibilidade, não estabelece uma relação entre os resultados e a legibilidade.

Para o *CamelCase*, não existe nenhuma convenção da inclusão de acrónimos no nome dos identificadores quando todas as abreviaturas são em maiúsculas ou quando a primeira letra é em maiúscula. Quando envolve preposições e conjunções, as regras de *CamelCase* são alteradas para melhorar a legibilidade. Os autores dão como exemplo, DIAdoMES.

Siegmund, 2014 [253] Entender a compreensão de programas não está limitada à teoria e pode ter efeitos devastadores na melhoria da educação, treino e na avaliação e concepção de ferramentas e linguagens para programadores. Para os autores, a compreensão de programas, a qual descreve o processo de como os desenvolvedores compreendem

o código-fonte, é um factor humano importante na engenharia de software. Considerando que os custos de manutenção são o factor custo principal para o desenvolvimento de um sistema de software, se se puder melhorar a compreensibilidade do código-fonte é possível reduzir o tempo e os custos de todo o ciclo de vida do software.

Foi então conduzida uma revisão da literatura a artigos de 13 revistas e conferências ao longo dos últimos 10 anos (de 2001 a 2010):

- Empirical Software Engineering (ESE);
- Journal of Software: Evolution and Process (JSEP);
- Transactions on Software Engineering and Methodology (TOSEM);
- Transactions on Software Engineering (TSE);
- International Conference on Program Comprehension (ICPC);
- International Conference on Software Engineering (ICSE);
- International Conference on Software Maintenance (ICSM);
- International Symposium on Empirical Software Engineering and Measurement (ESEM);
- Symposium on the Foundations of Software Engineering (FSE);
- Symposium on Visual Languages and Human-Centric Computing (VLHCC);
- Conference on Human Factors in Computing Systems (CHI);
- Cooperative and Human Aspects of Software Engineering (CHASE);
- Working Conference on Reverse Engineering (WCRE).

Escolheram estas publicações por serem as principais plataformas para publicar resultados relativos (empíricos) de engenharia de software e compreensão de programas. Incluiram 872 (de 4935) artigos na selecção inicial e como resultado identificaram 39 parâmetros que geram confusão.

Como resultado, os autores criaram um catálogo de parâmetros que geram confusão, incluindo uma visão geral das medidas e das técnicas de controlo. Dada a extensão das

tabelas, não serão aqui reproduzidas. no entanto, o leitor interessado poderá consultá-las no respectivo artigo.

Com este catálogo, os investigadores ficaram com uma ferramenta que lhes permitirá obter resultados válidos e fiáveis.

Salviulo, 2014 [228] Os autores realizaram um estudo com o objectivo de compreender como os identificadores e os comentários no código orientado a objectos são utilizados na compreensão e modificabilidade do ponto de vista de quem faz a manutenção. Para isso, escolheram código-fonte desconhecido dos participantes. O objectivo consistiu também em avaliar essa utilização tanto por estudantes de informática, como de profissionais recém-formados.

Como conclusões, os profissionais não deram importância aos comentários. Só se preocuparam com o código-fonte. Os alunos consideraram importantes ambos, os nomes e os comentários. Quer os alunos quer os profissionais concluíram que as técnicas de convenção de nomes para os identificadores são essenciais para compreender e fazer manutenção ao código. Também ambos preferem os identificadores com nomes curtos, com significado e mnemónicas.

Como trabalho futuro consideram que poderá ser significativo investigar quantitativamente se e como a presença de comentários no código-fonte melhora ou não a compatibilidade e manutenção do código-fonte.

Suzuki, 2014 [268] Os nomes dos métodos são importantes no processo de desenvolvimento de software. Os estudos mostram que a qualidade dos nomes dos métodos afecta a compreensão do software. O modelo *n-gram* é um tipo de modelo de linguagem que assume o modelo de Markov na ocorrência de palavras, a ocorrência de cada palavra no nome de um método depende unicamente das *n-1* palavras que a precedem. Só um conjunto de n palavras consecutivas domina os nomes dos métodos. Estas n palavras consecutivas chamam-se *n-grams*.

Realizaram um estudo para avaliação da compreensibilidade dos nomes dos métodos usando esta metodologia. Como conclusão, consideram que esta abordagem permite avaliar 75% dos nomes dos métodos e permite sugerir 92% de novas palavras para o nome do

método. No entanto, consideram que a metodologia pode ser melhorada considerando o contexto do método. Realço o facto que perante a leitura do artigo o valor 75% não me pareceu correcto.

Avidan, 2015 [8] Tendo em consideração que por vezes o software é muito complicado e difícil de entender, o autor refere que Frederick Brooks[132] dividiu a complexidade do software em duas partes:

- Complexidade essencial - está relacionada com os requisitos do sistema e reflete a dificuldade do problema e por isso não pode ser evitada ou ignorada;
- Complexidade acidental - mostra como um problema é conduzido na sua resolução. Esta complexidade pode e deve ser reduzida pela utilização de padrões de concepção, *refactoring*, e por alterações ao código de modo a melhorar a legibilidade, melhorando assim a sua apresentação e consequentemente a sua legibilidade, tornando o código mais compreensível e de mais fácil manutenção.

Os autores fizeram um estudo usando técnicas de *obfuscation*. A *obfuscation* é a degradação intencional do código, a fim de evitar a engenharia reversa, e, deste modo, proteger, em primeiro lugar, o esforço intelectual investido na criação do programa. Ao longo dos anos um grande corpo de conhecimento sobre *obfuscation* de código tem sido acumulada, incluindo o desenvolvimento de novas técnicas de *obfuscation*, a sua classificação, descrição, e trabalho empírico sobre a sua eficácia.

Os autores definem legibilidade como uma propriedade local do código, em que código legível tem linhas razoavelmente curtas, indentação consistente, e nomes dos identificadores e das funções com significado.

Os autores concluem que as técnicas desenvolvidas para *obfuscation* do código podem ajudar a identificar impedimentos na compreensão do código e uma vez identificados podem ser usados para derivar novas métricas para o código.

Batool, 2015 [19] O objectivo principal deste artigo é encontrar características de programação e o seu impacto na legibilidade do código-fonte. Os autores consideram que a sua contribuição é importante uma vez que vai ajudar os desenvolvedores a escolher uma das duas linguagens, C# ou Java, de acordo com a sua legibilidade.

Neste artigo, são avaliadas algumas das características de programação que tornam o código legível, em Java e C#. Para isso, foram considerados *snippets* em ambas as linguagens. Usaram métricas para calcular a legibilidade do código: o índice ARI, a fórmula SMOG e o índice de Fox Gunning. No final fizeram uma comparação entre as linguagens com base nos resultados.

Pela leitura do artigo, os autores usaram, para além das métricas acima mencionadas, o índice de legibilidade Flesch-Kincaid e o índice Coleman-Liau.

De acordo com os resultados, as métricas e os inquéritos mostraram que a linguagem Java é uma linguagem de programação mais legível do que a linguagem C#. Existem muitas outras linguagens de programação que estão a tornar-se populares e, por isso, é possível fazer a comparação da legibilidade das diferentes linguagens para uma boa decisão dos programadores de modo a poderem escolher a melhor linguagem para desenvolvimento do seu software.

Borstler, 2015 [33] No ensino, os exemplos a ser utilizados são uma ferramenta importante. Os autores referem que a escolha dos exemplos é importante no sentido de ajudar os alunos a desenvolver generalizações das estruturas em lugar das características superficiais. Deste modo, apresentando-se aos alunos exemplos "bem feitos", exemplares, reforçando as propriedades desejáveis, os alunos irão eventualmente reconhecer padrões de uma "boa" concepção e ganhar experiência no reconhecimento de propriedades desejáveis e indesejáveis.

Com a utilização de exemplos desenvolvidos cuidadosamente, os professores podem minimizar o risco de más interpretações e conclusões erradas.

Para os autores, as propriedades de bom exemplos são as seguintes:

1. Compreensível pelo computador: para ser usado por este, senão não é um verdadeiro programa de computador.
2. Compreensível pelos alunos: para os alunos conseguirem criar um modelo mental do programa.
3. Comunicar eficazmente os conceitos a serem ensinados.

Um pré-requisito básico para a inteligibilidade é a legibilidade. Os elementos sintácticos básicos devem ser fáceis de detectar e fáceis de reconhecer. Só assim, é possível estabelecer relações entre os elementos. E só quando as relações com significado terem sido estabelecidas, é que o texto fará sentido.

Os autores propõem uma medida de legibilidade baseada na fórmula de Flesh e num estudo comparam-na com outras fórmulas de legibilidade. De notar que a fórmula aqui apresentada é a mesma do artigo de Borstler [32]:

$$SRES = ASL - 0.1 * AWL \tag{3.6}$$

em que:

ASL é o número médio de palavras por instrução ou bloco de instruções (delimitado por chavetas e ponto e vírgula);

e AWL é o comprimento médio dos lexemas (identificadores, palavras reservadas e os símbolos) em número de caracteres.

Os autores afirmam que o parâmetro ASL é mais importante que o parâmetro AWL porque é mais difícil ler instruções de um programa que um simples lexema. Os autores consideram que quanto menores forem os valores de SRES, maior é a legibilidade.

Também fazem distinção entre *readability* e *legibility* considerando que o objectivo do estudo é capturar a legibilidade independentemente de aspectos ou características do código. O tamanho do tipo de letra, o estilo do tipo de letra têm efeito visual. A estrutura e comprimento da frase, o vocabulário e a organização afectam a mente.

A principal diferença entre a fórmula SRES e as teorias de Buse e de Posnett é que os dois últimos são baseados em dados empíricos de a legibilidade percebida de pequenos *snippets* de código que não continham declarações. Uma parte significativa dos programas orientados a objectos contêm declarações. Os exemplos usados no estudo são classes completas contendo portanto declarações.

De acordo com os resultados do estudo dos autores consideram que a medida poderá ajudar os professores na selecção e desenvolvimento de programas adequados para exemplos. Para além disso sugerem como trabalho futuro, o estudo da relação entre a métrica e a qualidade dos programas escritos pelos alunos podendo ser integrada no conjunto de

ferramentas a dar aos alunos, e desse modo estes poderiam ter logo "feedback" sobre certas qualidades dos seus programas.

Busjahn, 2015 [**41**] A leitura de código é uma competência muito importante na programação. A leitura é baseada na linearidade que as pessoas exibem enquanto leem textos de linguagem natural. A linearidade consiste na leitura da esquerda para a direita e de cima para baixo. Ao contrário dos textos em linguagem natural, o código-fonte é executável e requer uma leitura específica. Os autores criaram medidas de visão local e global que caracterizam a linearidade na leitura do código-fonte.

Para validar as métricas compararam os movimentos dos olhos de jovens programadores e de programadores com experiência (peritos). A ambos foi pedido para lerem texto em linguagem natural e programas em Java. Os resultados mostram que os mais novos leem o código-fonte com menos linearidade que os textos em linguagem natural. E que os peritos em programação leem o código-fonte ainda com menos linearidade que os jovens programadores. De acordo com estes resultados, existem diferenças específicas na leitura de textos e na leitura de código-fonte. Os autores sugerem que as competências de leitura não linear aumenta com a experiência.

Deste modo a utilização de medidas de legibilidade para leitura de textos é diferente da legibilidade na leitura de código-fonte. Os programas em código-fonte diferem dos textos de linguagem natural de várias formas. Em primeiro lugar os programas são lexicalmente e sintacticamente diferentes: por exemplo, no vocabulário utilizado. Em termos léxicos, por exemplo, os nomes para os identificadores devem ser escolhidos criteriosamente de modo a terem significado. Em termos sintácticos, a organização dos programas é diferente dos textos. Existem estruturas pré-definidas tais como a indentação.

Em segundo lugar, os programas são diferentes em termos semânticos: O texto natural é tipicamente compreendido em duas fases concorrentes: o texto propriamente dito (o que está escrito) e o domínio (o que significa). No caso do código-fonte, tem uma terceira dimensão de compreensão: a execução do programa .

Normalmente não se ensina a compreensão de algoritmos da mesma forma que se ensina compreensão da leitura. Os autores defendem que a leitura do código deveria fazer parte dos curricula dos cursos de informática.

Daka, 2015 [62] Neste artigo é proposto um modelo de legibilidade para testes unitários baseados no julgamento humano. e a utilização do modelo para melhorar a geração automática de testes unitários.

Em termos de características concretas que podem afectar a compreensão, o primeiro teste é maior, utiliza mais classes diferentes, define mais variáveis, tem mais parêntesis e linhas mais compridas. Ou seja, em termos de legibilidade, o seu código é considerado não legível, uma vez que intuitivamente é mais difícil de realizar uma vez que as tarefas requerem maior esforço no entendimento do código.

De acordo com o estudo realizado, os autores consideram que a legibilidade dos testes unitários depende de vários factores, tendo-os agrupado por categorias:

- Estrutural;
- Complexidade lógica;
- Densidade do código.

Para medir a legibilidade, basearam-se nas métricas de Buse e de Posnet, e acrescentaram as seguintes:

- *Assertions*: esta característica conta o número de asserções *JUnit* (estrutura simples para escrever testes repetíveis). Incluiram uma característica binária que toma valor 1 se tem asserções, e valor 0 se não tem asserções;
- Excepções: esta característica permite medir as excepções, considerando que o comportamento da excepção é diferente de comportamento regular;
- Identificadores não utilizados: os testes unitários são tipicamente curtos e contêm poucas variáveis;
- Comentários: esta característica permite contar o número de linhas com comentários de linha ("//"), e com comentários de multi-linha ("/* ... */");
- Características dos *tokens*: identificaram várias características adicionais não capturadas por outros modelos de legibilidade. Observaram que as ferramentas de geração de testes unitários por vezes tendem a incluir *casts* defensivos;

- Características dos tipos de dados: os diferentes tipos de dados primitivos têm impacto diferente na legibilidade. Propõem características baseadas na ocorrência do valor nulo, de valores booleanos, de acessos a arrays, constantes, números de vírgula flutuante, dígitos, strings, caracteres e o comprimento das strings;
- Características das instruções: propõem características para contar diferentes tipos de instruções, tais como, chamada dos construtores, acesso aos atributos e invocação dos métodos;
- Diversidade da classe e método.

O estudo confirma que os nomes dos identificadores no código-fonte são muito importantes. De acordo com o estudo sobre os efeitos da legibilidade na compreensão dos testes, que nem todos os testes que têm boa aparência são também de fácil compreensão.

Liu, 2015 [172] Neste artigo, os autores consideram que os comentários para descrever o código têm um papel muito importante na medição da legibilidade dos programas. Para avaliarem se os programas estão devidamente comentados recorreram à ferramenta WordNet (base de dados lexical). Através desta ferramenta, compaRAram os nomes dos métodos extraindo os nomes e verbos dos nomes dos métodos e dos valores de retorno, respectivamente.

Para avaliação seleccionaram um pacote do projecto JEdit (software de edição de texto), composto por 19 classes, 203 métodos incluindo 175 métodos e comentários contractualmente ligados. Com o WordNet comparam as palavras extraídas com os comentários para verificar se são ou não sinónimos.

Definiram uma escala com três valores possíveis:

- comentário válido;
- comentário não recomendado;
- comentário inválido.

Com base nos resultados obtidos calculam a relação dos três tipos de resultados nos *packages* e nas classes de um programa de modo a medir a sua legibilidade.

Os resultados mostram que 55.5% dos comentários são válidos, 1.1% dos comentários são inválidos e 43.4% dos comentários são não recomendados. De referir que, os resultados empíricos do estudo elaborado são muito semelhantes à análise humana que efectuaram sobre as mesmas classes e é de 54.2%, 2.9% e 42.9%, respectivamente.

Ribeiro, 2015 [222] Os atributos qualitativos estão relacionados com os aspectos do código-fonte que podem ter uma avaliação subjectiva. Um exemplo é a qualidade dos identificadores, comentários e organização do código-fonte. Consequentemente, uma revisão da literatura baseada nos procedimentos da revisão da literatura sistemática foi realizada com o objectivo de analisar os atributos do código-fonte com o propósito de caracterização relativamente ao seu impacto na legibilidade e compreensão do código-fonte para o desenvolvedor de software no contexto da indústria de desenvolvimento de software.

Através do estudo de 236 artigos, seleccionaram 18 artigos. Desta selecção de artigos identificaram 59 atributos do código-fonte, sendo 23 qualitativos e 36 quantitativos, tendo sido identificados com evidência explícita considerando o seu impacto na legibilidade e compreensibilidade.

Lista de práticas a seguir no código-fonte apresentadas no artigo:

- Ficheiros e pastas:

    1. Organizar coesivamente as pastas e os ficheiros dos projectos.
    2. Usar extensões do nome do ficheiro de acordo com o tipo de conteúdo do ficheiro.
    3. Cada módulo deve ser composto pelo menos por um ficheiro de definição e outro de implementação.
    4. Usar a ordem a seguir para organizar os ficheiros de definição.
    5. Usar a ordem a seguir para organizar os ficheiros de implementação.

- Apresentação (*Layout*):

    6. A indentação deve ser consistente.
    7. Usar indentação nas expressões lógicas com dois ou mais operadores não iguais.

8. O uso de chavetas para identificação dos blocos deve ser consistente (2 tabulações).
9. Ter em atenção para a utilização de espaçamento consistente (1 espaço) diferente nas estruturas sintácticas e elementos do programa.
10. Usar parêntesis para operandos nas expressões.
11. Usar linhas em branco para separar instruções extensas de outras instruções a para separar blocos de instruções com propósitos distintos.
12. Cada linha deverá ter só uma declaração.
13. Cada linha deverá ter só uma instrução simples.
14. O tamanho da linha deverá ter no máximo 80 caracteres.
15. As variáveis deverão ser declaradas junto e no início do seu âmbito válido.
16. As constantes devem ser declaradas junto e no início do seu âmbito válido.

- Comentários:

17. Seleccionar uma única lingua (português ou inglês) para escrever os comentários.
18. Escrever um comentário de identificação para cada ficheiro *header*.
19. Escrever um comentário para explicar o propósito de cada função.
20. As linhas de comentário devem ser usadas só em casos específicos.

- Nomes de identificadores:

21. Seleccionar uma única lingua (português ou inglês) para os nomes dos identificadores.
22. Os identificadores devem ser de fácil memorização.
23. Utilização correcta de prefixos e sufixos nos elementos do programa.
24. Os elementos identificadores de um programa devem ser consistentes no seu estilo.
25. As constantes devem ser simbólicas.

- Paradigma de programação:

26. Evitar a utilização de *goto*.

27. Evitar o uso do operador ternário (*? :*) quando o tamanho da instrução excede o tamanho estabelecido para a linha.
28. Evitar a presença de código comentado que já não se vai usar.
29. Tamanho da solução em termos de código, incluindo blocos, funções e ficheiros. Isto poderá ser um indicador que ou o problema não está bem estruturado ou então a complexidade estrutural da solução é elevada.

Como se pode constatar uma parte destes atributos não dizem respeito ao código-fonte.

Borstler, 2016 [31] Neste artigo, os autores apresentam um estudo realizado sobre o papel do encadeamento de métodos e dos comentários na legibilidade do software e na compreensão. Para o encadeamento usaram dois níveis, existência e ausência de encadeamento. Para os comentários utilizaram três níveis:

1. "bons" comentários
2. "maus" comentários
3. sem comentários

Para o estudo, os autores definiram duas questões de investigação:

1. De que modo a quantidade e a qualidade de comentários afectam a legibilidade do software e a compreensão?
2. De que modo o encadeamento de métodos afecta a legibilidade do software e a compreensão?

Foi realizado um estudo experimental. Os participantes envolvidos eram alunos do primeiro e do segundo ano de um curso de ciências da Computação. Destes alunos, 42.3% declararam ter elevada ou muito elevada experiência de outras linguagens de programação para além do Java e do C++. Para além disso, 19.2% destes alunos declararam ter experiência média a muito elevada como programadores profissionais. Para o estudo, os participantes levaram exercícios para trabalho de casa. Este trabalho foi realizado no computador.

Como conclusões os autores referem o seguinte: A ausência e presença do encadeamento de métodos não é estatisticamente significativo relativamente à legibilidade e à compreensão. Quanto aos comentários o estudo mostrou que o código comentado afecta a legibilidade mas não afecta a compreensão. Também referem que, contrariamente ao esperado, o código mal comentado é mais legível.

3.3.4 Lista de Práticas de Legibilidade nos Estudos

A tabela 3.3 apresentada uma compilação das práticas encontradas nos estudos analisados na secção anterior (secção 3.3.3). Para cada prática é indicado o(s) autor(es) que a referem.

Tabela 3.3: Lista de Práticas de Legibilidade nos Estudos.

Descrição	**Fonte**
Conetor de linha gerado pelos compiladores e combinado com a indentação.	Clifton[53]
Usar indentação. Miara sugere 2 a 4 espaços.	Jorgensen[131], Hansen[116],Buse[38, 39], Miara[191], Ribeiro [222]
Indentar as linhas na proporção da sua posição na hierarquia.	[38], Miara[191], Tashtoush[271]
Evitar muitos níveis de indentação.	Stamelos[262]
Usar Indentação em expressões lógicas com dois ou mais operadores diferentes.	Ribeiro [222]
Tamanho adequado do nome do identificador.	Jorgensen[131], DeYoung[73], Tashtoush[271], Buse [38], Shneiderman[248], Scanniello[235], Butler[42], Borstler[33]
Escolha criteriosa dos nomes dos identificadores tendo em atenção a visão global.	Weissman[290], Jorgensen[131], Wang[287], Tashtoush[271], Buse[38], Caprille[44], Tonella[278]
Usar nomes compridos para variáveis pouco usadas.	Shneiderman[248]
Usar nomes curtos para variáveis locais ou de ciclo.	Shneiderman[248]
Usar mnemónicas para nomes dos variáveis.	Elshoff [85]
Nomes dos identificadores concisos, consistentes e com significado.	Deissenboeck[67], Haiduc[112], Tashtoush[271], Kim[142]
Evitar nomes de identificadores só com uma letra.	Lawrie[164, 165]
Usar abreviatura ou nome completo para nome do identificador.	Lawrie[164, 165]
Capitalizar os nomes dos identificadores apropriadamente.	Butler[42]
Evitar *Underscores* consecutivos no nome do identificador.	Butler[42]
Evitar *Underscores* externos início e fim.	Butler[42]
Os nomes dos identificadores deverão ser compostos por palavras do dicionário, abreviaturas, e acrónimos que são comummente usados na forma não abreviada.	Butler[42]
Evitar o uso de palavras excessivas no nome do identificador: (mais de quatro palavras ou abreviaturas).	Butler[42]
Não usar identificador numérico.	Butler[42]

Descrição	**Fonte**
Não usar identificador demasiado curto.	Butler[42]
Consistência da variável de acordo com todas as suas utilizações.	[271]
Usar nomes para os identificadores de fácil memorização e numa única língua.	Ribeiro[222]
Evitar a presença de identificadores abreviados no código-fonte.	Scanniello[235]
Utilização de comentários: com critério e consistentes.	Weissman[290], Jorgensen[131], Sheppard[244], Woodfield[298], Baecker[10], Haiduc[112], Tashtoush[271], Wang [287], Elshoff [85], Fluri [94], Tan[270], Ribeiro[222], Tenny[273], Namani [197], Borstler[31]
Usar comentário para explicar propósito de cada função.	Ribeiro[222]
Linhas de comentário só usadas em casos específicos.	Ribeiro[222], Namani [197]
Utilização consistente em termos de estilo dos comentários.	Ribeiro[222]
Evitar comentar código.	Ribeiro[222]
Comentários: só numa lingua, em cada ficheiro *header* um comentário de identificação.	Ribeiro[222]
Evitar *GOTO*.	Jorgensen[131], Ribeiro[222]
Substituir *GOTO* por ciclos.	Elshoff [85]
Controlo na passagem de argumentos/parâmetros das funções.	Weissman[290]
Utilização de parêntesis nas expressões.	Buse[38]
Evitar o encadeamento dos métodos.	Borstler[31]
Uniformidade na notação,terminologia e simbologia utilizada.	Shneiderman[248]
Usar variáveis de estado para controlar a execução ao longo de blocos.	Elshoff [85]
Controlar tamanho da solução em termos de código.	Ribeiro[222]
Escolha adequada das estruturas de controlo de fluxo.	Jorgensen[131], Chaudhary[48], DeYoung[73]
Inserção parágrafos nas listagens.	Jorgensen[131]
Usar modularizações de tipo de dado abstrato.	Woodfield[298]
Usar modularização funcional.	Woodfield[298]
Controlo do número de linhas com instruções.	Weissman[290], DeYoung[73], Stamelos[262], Namani[197], Posnett[213]
Controlo do tamanho do programa.	Chaudhary[48], DeYoung[73], Stamelos[262], Haiduc[112], Posnett[213]
Optar pela programação estruturada.	Pennington[207]
Melhorar a apresentação dos programas.	Baecker[10]
Evitar complexidade ciclomática.	Stamelos[262]
Cuidado em manter a qualidade do código estrutural quando se intervem no código.	Stamelos[262]
Software deverá ter um máximo de 3700 linhas de código.	Chhabra[50]
Usar *switch* para reduzir ramificações.	Elshoff [85]
Incorrecta definição/utilização da herança.	[271]

Descrição	**Fonte**
Evitar expressões com *side-effect*.	Dolado[77]
Evitar argumentos repetidos.	Collar[57]
Evitar densidade proposicional.	Collar[57]
Ordem nos tipos enumerados: alfabética ou de acordo com o que representam.	Butler[42]
Limitar o comprimento da instrução (linha).	[248], Benander[21], Tashtoush[271], Buse[38], Namani [197]
Linhas em branco a isolar blocos de instruções.	Jorgensen[131], Ribeiro[222], Wang[286, 287], Namani[197], Tashtoush[271], Buse[38]
Quebra de linha a seguir ao ponto e vírgula.	Namani[197]
Usar linhas em branco a seguir à instrução.	Namani [197]
Optar por linhas curtas.	Buse [38, 39]
Comprimento linha máximo 80 carateres.	Ribeiro [222]
Uma declaração por linha.	Ribeiro[222]
Uma única instrução por linha.	Ribeiro[222]
Usar espaçamento horizontal e vertical de forma a delimitar de forma clara segmentos de código.	Wang[286, 287], Hansen[116]
Controlo número de métodos.	Weissman[290], Namani [197]
Utilização correcta das funções recursivas.	Buse[38, 39], Tashtoush[271]
Optar por blocos de uma entrada e uma saída.	Elshoff[85]
Adição do ramo *Senão* (mesmo vazio).	Elshoff[85]
Evitar o excesso de ramificações das decisões.	Weissman[290], Tashtoush[271], Buse[38]
Reorganização das instruções dentro de um módulo de modo a reduzir o âmbito da(s) variável(eis).	Weissman[290], Elshoff [85], Sasaki[234], Tashtoush[271], Ribeiro[222]
Declarações variáveis e constantes junto ao local onde são usadas.	Elshoff[85], Tashtoush[271], Ribeiro[222]
Mover as instruções para os blocos mais interiores para reduzir âmbito.	Sasaki[234], Tashtoush[271]
Juntar instruções relacionadas entre si reduzir distância entre definição e referência.	Sasaki[234]
Evitar aritmética de apontadores dentro de expressões.	Weissman[290]
Modularização para eliminação de blocos repetidos.	Elshoff[85]
Incorrecta definição/utilização de herança.	Tashtoush[271]
Utilização correcta da redefinição de funções (“overriding”).	Buse[38, 39], Tashtoush[271]
Evitar a utilização de vectores, substituir por contentores.	Tashtoush[271]
Uso consistente de chavetas para identificação blocos.	Ribeiro[222]
Espaçamento consistente.	Ribeiro[222]
Usar parêntesis nos operandos.	Ribeiro[222]
Utilização correcta de prefixos e sufixos.	Ribeiro[222]
Utilização de constantes simbólicas.	Ribeiro[222]
Evitar o operador ternário quando tamanho excede limite estabelecido.	Ribeiro [222]

3.4 Conclusão

Na revisão da literatura apresentada neste capítulo, assim como em todo o trabalho, procurou-se centrar a investigação na legibilidade do software. No entanto, a legibilidade está por definição associada à compreensão. Para se determinar se um pedaço de código é legível é necessário determinar se foi compreendido. Uma forma simples de o fazer é procurando saber a opinião do programador acerca da legibilidade do código. Esta associação fez com que o trabalho incluísse com frequência estudos de compreensão que, no entanto, deveriam referir o impacto na legibilidade das práticas aí estudadas. Uma das conclusões deste trabalho de revisão é a constatação da dificuldade, se não mesmo impossibilidade, em investigar a legibilidade ignorando-se completamente a compreensão.

Neste capítulo deverá ter ficado clara a importância que a indústria dá à legibilidade do código-fonte. Essa importância é de tal ordem que alguns autores consideram que o código deve ser escrito a pensar na sua leitura posterior. Os programadores em geral e os alunos em particular, nem sempre escrevem programas legíveis. Importa, portanto, conseguir transmitir-lhes a importância de escreverem programas legíveis. Isto pode ser ultrapassado colocando os programadores/alunos a executar uma tarefa de manutenção num programa de pouca legibilidade. Isso deve convencer os alunos que a escrita de programas pouco legíveis dificulta seriamente a manutenção dos programas [201]. Esta afirmação pode ser suportada pela importância que a motivação do leitor tem na legibilidade, como visto na secção 2.6.2.

Vários dos estudos analisam ou propõem medidas de legibilidade. Através delas verifica-se a importância das características de legibilidade em causa mas fica-se sem saber como se deve usar essa característica para melhorar a legibilidade. Aliás, muitos desses estudos não apresentam sequer valores absolutos concretos para as características de legibilidade que referem. A medida de complexidade ciclomática de McCabe também foi proposta para a medição da legibilidade. No entanto, um estudo, que compara as medidas com a opinião de programadores acerca da complexidade do código, mostrou existirem medidas superiores [135].

São muitas as práticas de legibilidade. Referem-se agora duas, comentários e nomes, a título de exemplo. Apesar de serem consideradas importantes, a sua investigação ainda não se pode considerar completa, pois existem opiniões divergentes quanto à sua

utilização. Existe algum consenso quanto à importância dos comentários desde que usados criteriosamente, mas alguns autores rejeitam-nos. Os nomes pela sua prevalência no código-fonte serão um dos factores mais importantes para a legibilidade do software [272]. Por exemplo, em [178] é referido que os nomes representam 90% do que torna o software legível. Também se sabe que cerca de 70% do código-fonte consiste em identificadores [70]. A qualidade dos identificadores é relevante para a documentação do código, a tal ponto que para alguns autores deveria permitir evitar os comentários. Tem sido uma prática muito estudada, mas também aqui não existe unanimidade quanto à forma e dimensão dos identificadores.

O estudo do comportamento na programação, ou psicologia da programação, dependendo dos autores, é dividido por Détienne [72] em duas fases. A dos anos 70 do séc. XX e um segundo período, posterior ao primeiro, mas sem explicitar uma data de início, pelo que se poderá supor que inicia por volta de 1980. Nestes períodos encontram-se muitas referências à legibilidade pelo que parece razoável enquadrar também a investigação da legibilidade nesses períodos, quanto mais não seja, pela ligação que tem com a compreensão do software.

O primeiro período pretendia avaliar o efeito de diversos factores da programação no desempenho, recorrendo a estudos experimentais. Este período caracterizou-se pela ausência de um enquadramento teórico e por uma experimentação deficiente. O segundo caracteriza-se pelo desenvolvimento de modelos cognitivos de programação e uma melhoria da qualidade da experimentação. No entanto, não é completamente correcta a ausência de enquadramento teórico (aliás como reconhecido de forma limitada por Détienne), pois em alguns trabalhos ele já existia, baseado na psicologia (e.g. [106, 248]). Por exemplo, Schneiderman [248] apresenta um modelo cognitivo. Já a componente experimental foi criticada por outros (e.g. [243]). De qualquer forma, a divisão por Détienne em 2 grandes períodos não parece desajustada.

O segundo período referido por Détienne irá pelo menos até à data de publicação do seu livro em 2002. Pela revisão aqui apresentada, pode-se considerar um terceiro período de inve stigação, mais direccionado para a leitura do código em si. Os trabalhos de Lawrie já em 2006 [164] com preocupações claras com a leitura e de Borstler em 2007 [32] dedicado explicitamente à legibilidade poderão marcar o início desse período. Mas

talvez seja mesmo este último que marca este terceiro período que se caracteriza por vários trabalhos com o intuito de medir a legibilidade. Assim sendo, pode-se considerar que o segundo período terminaria pouco antes.

Numa fase mais recente, para além da experimentação mais tradicional, também se constata a utilização de outros métodos na investigação da legibilidade, como sejam a imagiologia e a observação dos movimentos dos olhos, ou oculares, durante a leitura do código. A automatização é outro caminho que tem sido seguido. Por exemplo, a produção automática de resumos do código-fonte através da identificação de propriedades do código é uma linha de investigação em curso (e.g. [300]) que pretende servir na elaboração de documentação do código. Esta automatização poderá ter um interesse especial no caso de código produzido por outros e/ou que ainda não esteja devidamente (docu)comentado.

Capítulo 4

Estudo Inicial

É apresentada uma lista preliminar de 33 práticas para a legibilidade do código que poderão ser ensinadas nos programas de programação orientada a objectos dos cursos de informática. Neste conjunto, estão incluídas práticas com impacto positivo e negativo. Foi realizado um inquérito com professores de programação orientada a objectos de modo a avaliar a importância de ensinar um conjunto de boas práticas. Os resultados são apresentados, bem como a respectiva análise e interpretação. Alguns resultados adicionais são apresentados.

4.1 Introdução

A legibilidade do software é tida como uma característica importante da qualidade de software devido ao potencial impacto na sua manutenção e evolução (e.g. [29]). Além disso, é de facto comum o software ser desenvolvido por equipas cujos membros precisam de ler o código uns dos outros, nomeadamente nas revisões de código. Ler o código de outros é uma necessidade permanente em projectos de código aberto, mas não é menos verdade que muitos destes projectos de código aberto prestam atenção à legibilidade do código.

Especificamente, no caso do software de código aberto, o código é frequentemente mantido por um grande número de pessoas com a agravante de se encontrarem geograficamente dispersas o que torna a legibilidade ainda mais importante. É importante notar que os engenheiros de software passam cerca de 75% do seu tempo a ler código-fonte [208].

A escrita de código legível deve ser iniciada e reforçada durante os cursos de programação informática do ensino superior, de modo a que os futuros profissionais sejam capazes de o fazer desde o início das suas carreiras. Por outras palavras, pode dizer-se que a responsabilidade de ensinar e ajudar os estudantes a desenvolver a supracitada competência pertence aos cursos de ensino superior. Essa competência é também relevante nos trabalhos de estudantes, nomeadamente quando estes participam em projectos de grupo e quando os professores precisam de analisar o trabalho dos estudantes.

É importante reunir não só as boas práticas, como também aquelas que poderão ter um impacto negativo e devem ser evitadas. Especificamente, também seleccionámos uma lista de *bad smells* cujas descrições parecem deixar claro que têm impacto sobre a legibilidade. Este subconjunto é maioritariamente composto por *bad smells*. As práticas compiladas e respectivas fontes estão listadas na Tabela 4.1. Essas práticas cobrem diferentes aspectos do software, tais como estrutura (e.g. linhas em branco), organização (e.g. âmbito dos identificadores), lógica e a complexidade do código (e.g. excesso de ramificações de decisão). Na tabela, os números das práticas "negativas" são sufixados com um hífen. Esta é a lista final usada no inquérito.

Tabela 4.1: Lista preliminar de práticas compilada a partir da revisão da literatura.

N.	Descrição	Fonte
1	Limitar o comprimento da instrução (linha)	[248, 21, 271, 38]
2	Indentar as linhas na proporção da sua posição na hierarquia	[38, 191, 271]
3	Utilização de parêntesis nas expressões	[38]
4	Uniformidade na notação,terminologia e simbologia utilizada	[248]
5	Quebra de linha a seguir ao ponto e vírgula	[197]
6	Linhas em branco para isolar blocos de instruções	[271, 38, 197]
7	Linhas de código relacionadas devem aparecer juntas	[178]
8	Declarações das variáveis junto ao local onde são usadas	[178]
9	A função que chama deve estar acima da função que é chamada	[178]
10	No operador atribuição, incluir espaço branco antes e depois	[178]
11	Espaço branco entre os argumentos/parâmetros nas funções	[178]
12	Espaço branco para acentuar as precedências dos operadores numa expressão	[178]
13	Utilização de constantes simbólicas em substituição de constantes numéricas	[178]
14	Utilização de enumerações em lugar de variáveis de estado	[182]
15	Utilização de chavetas no bloco de ciclo	[182]

16	Escolha adequada dos nomes para os identificadores tendo em atenção a visão global	[271, 38, 44, 278]
17	Tamanho adequado do nome do identificador. Idealmente entre 9 a 16 caracteres. As variáveis muito utilizadas com menor número de caracteres	[271, 38, 182, 248, 235]
18	Controlo do uso dos comentários: consistentes com o código e controlados	[271, 38, 270]
19	Evitar o excesso de ramificações de decisões	[271, 38]
20	Reorganização das instruções dentro de um módulo de modo a reduzir o âmbito da(s) variável(eis)	[234]
21	Utilização correcta das funções recursivas	[271, 38]
22	Utilização correcta da redefinição de funções ("overrriding")	[271, 38]
23	Evitar a utilização de vectores, substituir por contentores	[271]
24	Âmbito pequeno das variáveis	[271, 234]
25	Consistência da variável de acordo com todas as suas utilizações	[271]
1-	Métodos extensos	[97]
2-	Métodos com listas de parâmetros extensas	[97]
3-	Código idêntico ou semelhante existente em mais de um local	[97]
4-	Nomes de tipos em nomes de métodos	[97]
5-	Nomes de métodos que não descrevem o que faz	[97]
6-	Incorrecta definição/utilização da herança	[271]
7-	Grandes blocos lógicos condicionais	[97]
8-	Local de declaração das variáveis sem critério	[97]

Como se pode notar na tabela, os pontos de vista nos dois grupos de práticas não são disjuntos. Foi feito um esforço por reduzir a redundância na elaboração da lista de práticas, no entanto, com algumas práticas tal não foi possível. As práticas 8, 20 e 24, que podem ser vistas como diferentes formas de alcançar um âmbito limitado, são caso disso, pelo que foi decidido manter as três. O mesmo acontece com as práticas 16 e 5-, especificamente acerca dos nomes dos identificadores.

4.2 Inquérito

O estudo empírico foi conduzido através de uma sondagem. O inquérito é considerado um método de investigação adequado quando os investigadores pretendem reunir dados de um grande número de respondentes [209]. No nosso caso, a intenção era disponibilizá-la para o maior número possível de respondentes num período de tempo relativamente reduzido. Por estas razões, decidiu-se usar, também, um inquérito *online*.

4.2.1 Objectivo e Questões de Investigação

O principal objectivo desta investigação foi apurar a importância do ensino de um conjunto definido de "boas práticas" de legibilidade do código-fonte, pela comunidade académica de professores de programação orientada a objectos (POO). O foco está na POO, mas várias práticas são transversais a linguagens de outros paradigmas de programação. Sendo um estudo inicial, decidiu-se limitá-lo ao universo de professores portugueses de programação orientada a objectos.

A partir dos objectivos da investigação, foram definidas várias questões de investigação:

QI1 Que nível de importância é que os professores de POO dão ao ensino de boas práticas para a legibilidade do software?

QI2 Os professores levam em conta a legibilidade quando avaliam os trabalhos dos estudantes?

QI3 Os estudantes já produzem código de grande legibilidade? Por outras palavras, qual é o nível médio de legibilidade do código produzido pelos alunos?

QI4 A experiência dos professores afecta a importância que dão à legibilidade?

QI1 é a principal questão de investigação. Embora não seja possível representar todas as variáveis que podem intervir na sondagem, a experiência de ensino pode ter alguma importância. Portanto, foi adicionada uma questão, QI4, de modo a ajudar a representar essa variável. Quanto a QI3, se os estudantes produzem código de elevada legibilidade, este projecto de investigação deve ser repensado. O objectivo de QI2 é saber o estado actual da avaliação da legibilidade.

4.2.2 Concepção do Inquérito

A concepção do inquérito pode ser classificado como maioritariamente descritivo, no sentido de que o principal objectivo é obter informação acerca da importância de um conjunto de práticas. Contudo, é também exploratório, já que quase nada era já conhecido sobre o julgamento dos professores quanto a essas práticas. Ao perguntar aos participantes sobre a importância que dão às práticas, de acordo com a sua experiência anterior, a concepção do estudo, seguindo [152], é um estudo transversal (no momento em que se realiza o estudo), e controlo de casos (relativo à experiência).

4.2.3 O Instrumento

Para o inquérito foi desenvolvido um questionário como instrumento de recolha dos dados necessários. O questionário foi desenvolvido de acordo com o objectivo do estudo, baseado nas práticas e orientado para professores de programação orientada a objectos (POO). O questionário foi construído após uma revisão da literatura sobre legibilidade do software. A sua construção seguiu as principais recomendações encontradas da literatura sobre inquéritos (ex. [209, 9]).

O instrumento consiste de um total de 40 questões, agrupadas de acordo com os respectivos objectivos, como explicado adiante, e está escrito em português. O grupo principal é composto por 33 questões não-opcionais, sobre potenciais factores de legibilidade de software adquiridos da literatura, como explicado na secção 4.1. Cada questão corresponde a uma prática da Tabela 4.1, de modo a que estas possam ser avaliadas. A tabela foi, de facto, o resultado da avaliação do instrumento abaixo descrita. Seguindo as práticas, as questões foram também divididas em dois conjuntos de factores; os factores com impacto positivo na legibilidade e os factores com impacto negativo, com 25 e 8 questões respectivamente.

Todas as 33 questões começavam com "nível de importância que dá ao seu ensino?", seguido da respectiva prática. Esta questão foi formulada para reflectir, tanto quanto possível, o principal objectivo do questionário, conforme expresso por QI1. Para cada questão sobre legibilidade, o questionário usa uma escala ordinal de cinco pontos. Foi usada uma escala de cinco pontos, por se considerar que aumenta a fiabilidade das estimativas [133]. A escala é do tipo *Likert*, com valores que variam do 1 (valor mais baixo) ao 5 (valor mais alto). O significado dos valores é o seguinte:

1 – Muito Baixo

2 – Baixo

3 – Médio

4 – Alto

5 – Muito Alto

Foi intencional evitar o ponto "não sabe" na escala de classificação, uma vez que os participantes-alvo, sendo professores de POO, devem ter uma opinião formada sobre cada prática.

Há uma questão adicional, que interroga os respondentes sobre outra prática que usem. Após isso, o respondente pode encontrar uma caixa de texto para comentários e sugestões. Estes dois itens eram opcionais. De modo a possibilitar a resposta das restantes questões de investigação (QI2, QI3 e QI4), foram adicionadas as seguintes questões obrigatórias no início do questionário:

1. Número de edições em que lecciona(ou) POO?
2. Na avaliação dos trabalhos dos alunos tem em conta a legibilidade do código-fonte?
3. Como é que avalia em média a legibilidade do código-fonte produzido pelos alunos?

Número de edições de POO que ensina (não necessariamente o mesmo curso). A primeira questão é numérica e aceita valores de 1 a 10 ou mais. A segunda questão usa uma escala booleana de verdadeiro/falso, enquanto a terceira usa a habitual escala de 5 pontos. Espera-se que a aprendizagem de POO em cursos de informática ocorra durante a licenciatura. Para verificar isso, uma questão sobre o nível do curso (licenciatura ou mestrado) foi adicionada ao questionário.

Neste caso, havia uma só questão demográfica. Era uma questão opcional, que pedia aos respondentes para fornecer os seus endereços de e-mail caso desejassem receber os resultados da sondagem. De modo a assegurar o anonimato dos respondentes, o questionário não requeria quaisquer dados de identificação, excepto o campo adicional para os seus endereços de e-mail. Foi inserida, no início do questionário, uma breve introdução do mesmo, incluindo o seu objectivo. O tempo para o seu preenchimento foi de 7 minutos.

O instrumento foi desenvolvido e inicialmente revisto, para o formato das questões, inteligibilidade, número de questões e tempo requerido para o completar. A seguir, o questionário foi enviado para dois professores de informática, para ser revisto. Com a informação dos dois revisores, o instrumento foi aprovado, principalmente quanto ao fraseamento e redundância. Consequentemente, algumas questões foram removidas e outras foram reescritas. O questionário pode ser consultado no apêndice B.

Como último passo de avaliação, um professor completou o questionário revisto. Como resultado, mais uma questão foi removida, outras foram melhoradas e o tempo de questionário foi analisado. A conclusão foi que os 7 minutos deveriam ser suficientes.

4.2.4 Participantes

Conforme o objectivo do estudo, a população-alvo é composta por todos os professores de programação orientada a objectos (POO) em cursos de informática de ensino superior português. Contudo, tanto quanto sabemos, tal lista não existe. Ainda assim, o ministério da educação possui, no seu site, uma lista de todos os cursos. Portanto, decidiu-se seleccionar e aceder directamente a todos cursos de informática e pedir aos responsáveis de cada um que tornassem o questionário acessível aos seus professores de POO. O resultado consistiu numa lista de 44 cursos de informática.

4.2.5 Hipóteses

Algumas hipóteses foram desenvolvidas, a partir das questões de investigação, para a análise estatística.

QI1: O nível de importância que os professores dão a um conjunto de boas práticas para a legibilidade do código-fonte é o constructo. Este constructo é operacionalizado através de um conjunto de questões acerca das práticas. De modo a respeitar o tipo de escala, a análise dos resultados pode ser feita em termos de medianas e percentagens e testes estatísticos baseados em categorias [150, 245], isto é, testes não-paramétricos.

Para QI1, o objectivo é, em última instância, saber se os professores de POO consideram que ensinar este conjunto de práticas é importante. Na escala de importância de 5 pontos, o ponto 3 é um valor médio. Assim, para considerar o conjunto de práticas importante, a mediana das respostas deveria ser acima de 3. As respectivas hipóteses são:

$H1_0$: O conjunto de boas práticas não é importante para os professores de POO. Isto é, a mediana é 3 ou menos, $\theta <= 3$

$H1_a$: O conjunto de boas práticas é considerado importante para os professores de POO. Isto é, a mediana é superior a 3, $\theta > 3$

Por haver dois subconjuntos, hipóteses idênticas aplicam-se a cada subconjunto. Para os testes, a mediana será usada com o teste de Wilcoxon, para uma única amostra, com

um nível de significância de 5% para unilateral. O resultado será comparado com o valor crítico[1].

QI2 e QI3: Estas duas questões foram consideradas como maioritariamente descritivas, pelo que não foram formuladas hipóteses para o seu teste.

QI4: Para QI4, a experiência dos professores no ensino de POO pode ser medida pelo número de edições que um professor ensinou POO. As hipóteses para QI4 são:

$H4_0$: O número de edições é independente do nível de importância atribuído ao ensino de um conjunto de práticas. Isto é, as variáveis são mutuamente independentes, $\rho = 0$.

$H4_a$: Existe uma correlação entre o número de edições e o nível de importância atribuído ao ensino de um conjunto de práticas. Isto é, $\rho <> 0$.

O coeficiente de correlação não-paramétrico de *Spearman* vai ser usado com um valor-p[2] com um *alfa* (nível de significância) de 5% para bi-lateral.

4.2.6 Recolha de Dados

Após a avaliação, o instrumento foi publicado por via do mesmo sítio web usado para o seu desenvolvimento. Foi preparado um curto texto sobre o inquérito. O apontador e o texto foram enviados, por e-mail, aos directores de curso e directores de departamento, pedindo que disponibilizassem a ligação aos seus professores de POO. O questionário esteve disponível durante aproximadamente um mês. Não houve seguimento dos respondentes para obtenção de informação adicional, o que está de acordo com o anonimato.

4.2.7 Resultados

Trinta e três (33) professores responderam ao questionário e onze (11) deles forneceram o endereço de e-mail. Todos os 33 questionários preenchidos eram sobre o ensino da programação orientada a objectos ao nível da licenciatura.

QI1: A Tabela 4.2 apresenta a mediana, o intervalo de variação e a amplitude interquartil (IqR) para cada questão. Na tabela, as práticas a evitar estão ainda numeradas com um hífen no fim. A mediana global para todas as classificações foi de 4.

[1]Os testes estatísticos foram realizados no Excel usando RealStats.xaml de www.real-statistics.com e verificados com funções de R.

[2]Valor de prova (*p-value*).

A Tabela 4.3 contém a mediana para cada professor. O teste de Wilcoxon forneceu um resultado $T = 24.5$ para uma $mediana = 3$ tendo em consideração que 7 professores apresentaram mediana igual a 3. Dado que o valor crítico de T para $n = 26$ e que o coeficiente unilateral é 110, a hipótese nula foi rejeitada.

Os dois subconjuntos de medianas dos professores para as práticas positivas e negativas, respectivamente, estão listados na Tabela 4.4. Foi aplicado o mesmo teste estatístico nas mesmas condições. Para o primeiro conjunto de medianas (positivas), o resultado do teste foi $T = 21.5$, para um $T_c = 75$, e para o segundo conjunto foi $T = 16.5$, para um $T_c = 151$. O resultado foi estatisticamente significativo para ambos.

QI2 e QI3: Trinta e dois (32) professores levam em conta a legibilidade quando avaliam trabalhos dos estudantes e um (1) não leva. A mediana para a questão sobre o nível de legibilidade do código dos estudantes foi de 3.

QI4: Os totais para a experiência de ensino da programação orientada a objectos, contados em números de edições, são mostrados na Tabela 4.5. A entrada 10+ significa 10 ou mais edições.

Para maior rigor, foram calculadas não só a correlação entre número de edições ensinadas e as medianas totais dos professores, como a correlação entre o número de edições com as medianas dos professores para factores positivos e correlação entre o número de edições com as medianas dos professores para factores negativos usando os valores da Tabela 4.4.

Todas as correlações foram calculadas usando o coeficiente de correlação de *Spearman*. Foi também determinado o valor-p para um *alfa* de 5% bi-lateral. A Tabela 4.6 mostra os resultados para as 3 correlações. A partir dos valores, a hipótese nula não pôde ser rejeitada para qualquer um deles.

4.2.8 Discussão

Conforme foi dito, os questionários não foram enviados directamente para os potenciais respondentes e, consequentemente, não há forma de saber quantos professores receberam o questionário, nem o tamanho da população. Desse modo, não há valor concreto para a taxa de resposta.

Tabela 4.2: Estatística descritiva.

Q#	Mediana	Intervalo	IqR
1	3	4	1
2	5	3	1
3	3	4	1
4	5	2	1
5	4	3	1.5
6	4	3	1
7	4	4	0.5
8	3	4	2
9	3	4	1.5
10	3	4	2
11	2	4	2
12	2	3	2
13	4	4	1.5
14	3	4	1
15	4	4	1
16	5	4	1
17	4	4	1
18	4	4	2
19	4	4	0
20	4	3	1
21	4	4	1
22	4	4	2
23	3	4	1
24	4	4	1.5
25	4	2	1

Q#	Mediana	Intervalo	IqR
1-	4	3	2
2-	4	4	1
3-	4	2	1
4-	4	4	2
5-	5	2	1
6-	5	3	1
7-	4	4	1
8-	4	4	2

Tabela 4.3: Mediana dos respondentes.

Professor	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Mediana	5	3	3	4	4	4	4	4	4	3	4	4	4	4	4	5	
Professor	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
Mediana	4	4	3	4	4	4	3	4	3	1	4	4	4	5	4	5	3

Tabela 4.4: Medianas dos professores (33), para as práticas positiva e negativa.

Mediana positiva	5	4	3	3	4	4	4	3	4	4	4	3	4	4	4	4	
cont.	4	4	3	4	4	3	3	4	4	4	3	4	3	1	4	3	3
Mediana negativa	4	5	3	3	5	4.5	4.5	4	5	5	4	4	4	3.5	4	5	
cont.	4	5	4.5	4	5	4	4	5	5	5	3	5	3.5	1.5	3.5	5	4

Tabela 4.5: Número de edições.

N. de edições	1	2	3	4	5	6	7	8	9	10+
Respostas	5	5	2	2	4	0	3	1	0	11
Percentagem(%)	15	15	6	6	12	0	9	3	0	33

Começando com a questão de investigação **QI1**, houve um valor de 4 para a mediana total de respostas às questões sobre práticas de legibilidade. As medianas para as classificações atribuídas por cada professor para todas as questões (Tabela 4.3) foram altas. Para as três hipóteses nulas, os valores obtidos de T para *alfa* 5% e unilateral permitem rejeitar a hipótese nula nos 3 casos. Além disso, para os 3 testes, $\Sigma(categorias\_positivas) > \Sigma(categorias\_negativas)$. Concluindo, os professores consideram importantes o conjunto de práticas global e cada um dos subconjuntos.

Só duas práticas (11 e 12) sobre a separação com espaços brancos tiveram o valor 2 de mediana para uma moda 2 e 3, respectivamente. Isto significa que os professores em geral atribuem baixa importância ao seu ensino. Outras 7 práticas tiveram pontuação média. Uma delas, a 11, era também sobre espaços brancos. De acordo com o intervalo na Tabela I4.2, existe, para muitas questões, uma grande variabilidade entre as respostas dos professores. Por outras palavras, embora os professores concordem que as práticas

Tabela 4.6: Correlações entre a experiência e a importância das medianas.

	medianas globais	medianas pos.	medianas neg.
ρ	-0,085817811	0,026635753	-0,064897985
valor-p	0,634890513	0,883024042	0,719734081

são importantes, a importância que dão a cada prática varia.

Nenhuma prática a evitar (Tabela 4.2, à direita) obteve uma mediana abaixo de 4, mas para as positivas, observaram-se as medianas 2 e 3. Porém, o mesmo teste estatístico nas mesmas condições foi aplicado aos dois subconjuntos de medianas das respostas para cada professor e as respostas foram estatisticamente significativas. Isto mostra que, apesar de algumas diferenças, cada tipo de práticas, individualmente, é considerada importante.

Quanto a **QI2**, trinta e dois (32) de trinta e três (33) professores responderam afirmativamente. Isto significa que os professores levam em conta a legibilidade quando avaliam os trabalhos dos estudantes, e certamente dão importância à legibilidade, caso contrário, não a avaliariam. Isto é importante, pois embora os resultados para **QI1** se refiram a um conjunto específico de práticas em estudo, a importância da legibilidade torna-se mais generalizável com este resultado.

Quanto a **QI3**, a mediana foi 3. Este valor significa que, no geral, a legibilidade do código dos estudantes é considerada razoável, nem boa, nem má. Nesta questão, houve apenas um professor que classificou com 5, mas 3 classificaram com 1. É razoável dizer que há espaço para melhorar a legibilidade do código dos estudantes.

Para **QI4**, as 3 correlações entre experiência, medidas pelo número de edições ensinadas, e as medianas globais dos professores, as medianas parciais dos professores para factores positivos e as medianas parciais dos professores para factores negativos foram todas muito baixas e sem importância estatística (os 3 valor-p eram acima do $alfa = 0.05$). Portanto, não se conseguiu rejeitar a hipótese nula e pode ser concluído que a experiência dos professores parece não ter qualquer efeito sobre a importância que dão ao ensino de boas práticas de legibilidade e, mais ainda, independentemente de serem boas práticas com impacto positivo ou negativo.

Apesar de as correlações para responder a **QI4** para os 2 subconjuntos de práticas com positivo e negativo sobre legibilidade sejam aproximadamente 0, uma foi positiva e a outra foi negativa. Além disso, olhando para Tabela 4.2 para as medianas das questões dos subconjuntos, pode ser visto que nenhuma mediana é menor que 4.

Mais ainda, a partir de **QI1**, há uma diferença aparente entre a importância que os professores dão aos dois subconjuntos. Desse modo, importa saber se estes subconjuntos – que se revelaram importantes quando considerando cada subconjunto individualmente

e em conjunto – são vistos como igualmente importante pelos professores ou não. Por outras palavras, é necessário comparar as medianas das classificações dos professores para cada grupo de questões.

Para esta situação, os mesmos professores estão a responder a dois grupos de questões, o que significa que a análise deve ser para uma amostra emparelhada. Usando Wilcoxon para amostras emparelhadas dá-nos um resultado significativo de $T = 22$, para bi-lateral e $alfa = 5\%$, o que é menos que o valor crítico $T_c = 89$. A hipótese nula pode ser rejeitada e a importância de práticas positivas e negativas não pode ser considerada idêntica (proveniente da mesma distribuição) pelos respondentes. Por outras palavras, as práticas negativas são consideradas mais importantes.

Esta diferença pode ser explicada por várias razões. Em primeiro lugar, os *bad smells* são bem conhecidos e bastante estudados, e a comunidade de software, em geral, e os professores, em particular, reconhecem a sua importância. Em segundo lugar, este conjunto de *bad smells* resultou de uma selecção feita durante a revisão, que permitiu obter um subconjunto de *bad smells* mais "fiável" relativamente à legibilidade. Além disso, também se pode especular que as práticas de *bad smells* estarão mais directamente relacionadas com erros no código que as práticas positivas, que se relacionam indirectamente com os erros no código e com a manutenção através da legibilidade. Possivelmente, este foi também o caso, anteriormente, com as práticas relacionadas com o uso de espaços brancos. Para finalizar, pode ser considerado que é necessário um maior esforço para tornar os factores de legibilidade positivos mais conhecidos.

Para concluir, os respondentes consideraram, no geral, a legibilidade e o ensino de boas práticas para a legibilidade do software importantes. Acreditamos que é necessária mais investigação e que o desenvolvimento de um conjunto de boas práticas é de interesse. É verdade que, apenas um subconjunto dos professores de programação orientada a objectos foi contactada e que respondeu ao inquérito, mas de acordo com a experiência da autora como professora, pode ser esperada, de uma maior população de professores, a obtenção de resultados semelhantes.

4.2.9 Limitações

A parcialidade das respostas significa que as respostas ao questionário não representam amostras de respostas, ou a população de respostas, e é frequentemente associada a taxas baixas de resposta. Este problema pode limitar a validade externa [161].

É sabido que, os inquéritos têm frequentemente uma taxa de resposta reduzida [209]. Ainda assim, como explicado anteriormente, o número de professores não é conhecido. Mesmo que não se tivesse obtido uma taxa de respostas elevada, se tivermos a certeza de que os respondentes são representativos de uma população mais alargada, a análise pode ser feita [151]. Neste caso, não há qualquer razão para suspeitar que não-respondentes responderiam de outra forma.

A validade, no geral, e a validade do conteúdo, em particular, foram, de certa forma, fortalecidas pela avaliação feita antes do inquérito. A validade do constructo não pode ser assegurada, mas todas as práticas compiladas da literatura dá algum grau de validade às questões. Adicionalmente, a questão formulada para todas as práticas foi a mesma, excepto uma nota menor acerca da direcção do impacto. Contudo, este inquérito é, também, exploratório. Também é verdade que, para uma escala do tipo Likert, a tendência é para o valor central. Esta tendência não pode ser medida, mas a mediana para a maioria das questões foi acima do ponto central, não demonstrando problemas significativos com a tendência natural de escolha do centro da escala (uma posição neutra).

Resumindo, as grandes limitações para a validade/fiabilidade do questionário foi a estratégia de triagem, que não foi probabilística mas, ainda assim, tentou cobrir tantos professores quanto possível, indirectamente, através dos cursos. Sendo assim, houve aleatoriedade na triagem dos respondentes. Além disso, o questionário só consultou professores de programação orientada a objectos (POO) de um país. Finalmente, as variações sobre a posição dos cursos de POO no currículum de cada curso e a existência de disciplinas de programação prévias não foram apuradas.

4.3 Conclusão

Neste capítulo foi apresentado um estudo acerca da importância de um conjunto de práticas obtidas da literatura. Dos resultados, pode concluir-se que existe entre os docen-

tes nacionais de programação orientada a objectos um interesse significativo pela legibilidade.

As principais contribuições deste estudo inicial são um conjunto de práticas para a legibilidade do código e um inquérito para apurar a importância do seu ensino em disciplinas de POO em cursos superiores de informática. Foi feita uma revisão da literatura inicial sobre práticas de legibilidade, resultando num conjunto inicial de 33 práticas. De seguida, foi realizado um inquérito através de um questionário com o conjunto das práticas, para ser preenchido pelos professores de POO.

Para a questão de investigação principal, sobre a importância de ensinar as práticas, a resposta foi positiva, com uma mediana global de 4, numa escala de 1 a 5, e estatisticamente significativo. Pode concluir-se que, os professores reconhecem a importância do ensino de práticas de legibilidade e, em particular, indirectamente, se os professores as consideram importantes, isso dá um valor mais geral às práticas.

Porém, há 2 práticas com medianas negativas e o intervalo de resultados para cada prática é algo grande. Este problema leva a acreditar que o conjunto pode ser melhorado. O segundo subconjunto, maioritariamente composto por *bad smells*, sugere que pode ser mais fácil para os professores reconhecer a importância das práticas mais directamente ligadas a erros no código, bem como pelo facto de serem mais conhecidas.

Os resultados são um encorajamento para a investigação aprofundada de boas práticas para a legibilidade do software, com o objectivo de produzir um conjunto de práticas a ser ensinadas em disciplinas de POO. Portanto, após a realização do estudo e confirmado o interesse dos docentes, foi dada continuidade à revisão da literatura sobre práticas de legibilidade, aprofundando-a e expandido-a. O capítulo 3 é o resultado dessa revisão.

Como nota final, na sequência da continuidade da revisão foram encontrados diversos trabalhos, dos quais se deve destacar a taxonomia de factores de legibilidade apresentada em [201]. Esse trabalho, apesar de ser relativamente antigo, poderia, com os devidos ajustes, ter servido de base para a elaboração do questionário usado no estudo.

Capítulo 5

Estudo Experimental

Neste capítulo é apresentado um estudo experimental para validação de duas das práticas de legibilidade apresentadas anteriormente. Para o referido estudo são apresentados os resultados e realizada a sua análise e interpretação. Por último, são também apresentadas as ameaças à validade do estudo. O estudo é antecedido por uma introdução que apresenta a motivação e a estratégia seguida, uma descrição resumida de estudos experimentais e de estudos de replicação.

5.1 Introdução

No capítulo 3 de revisão da literatura foram tabeladas duas listas de práticas de legibilidade. Uma tabela (secção **??**) contém práticas analisadas através de estudos empíricos descritos em publicações de carácter científico enquanto a outra (secção **??**) contém práticas referidas noutro tipo de publicações que dão de certo modo uma visão mais profissional, ou seja da indústria. Existe uma grande quantidade de literatura sobre codificação e quais as práticas para melhorar a legibilidade do código e a sua compreensão, mas os resultados não têm sido sempre no mesmo sentido, e nem sempre conclusivos, isto é, estatisticamente significativos. Consequentemente, é importante continuar a realizar estudos empíricos sobre a matéria.

Numa fase inicial, foi ponderada a possibilidade de se validar, através de um estudo experimental, a lista das boas práticas que resultariam da revisão da literatura científica. Contudo, uma vez que essa lista cresceu muito, a sua validação afigurou-se muito difícil como se explica em seguida. A primeira dificuldade diz respeito à necessidade de

os alunos experienciarem as práticas da lista e a segunda em conseguir-se extrair conclusões objectivas e válidas não só do conjunto, mas também de cada prática, a partir dos resultados obtidos. Mais concretamente, o estudo requeria a criação de extractos de código a apresentar aos alunos que obedecessem a todas as práticas e de outros extractos de código que não obedecessem a qualquer prática. Essa dificuldade também ocorreria se, em alternativa, fosse solicitado aos alunos que desenvolvessem código que obedecesse, e outro que não obedecesse, a essa lista extensa de práticas, e que depois poderia ser lido por outros alunos.

Quanto ao segundo problema, sabe-se que os efeitos de umas práticas podem afectar os efeitos de outras, o que dificulta a extracção de conclusões como pretendido. Esta interferência designa-se por "interferência de múltiplos tratamentos" [221, p.52]. Também é sabido que, a dificuldade de interpretação dos resultados aumenta com o número de tratamentos [129, p.25]. Uma alternativa consistiria na divisão dessas práticas por diversos extratos de código a testar separadamente. Quantos mais extractos de código, mais grupos seria necessário formar e, consequentemente, maior seria o número total de participantes, necessário para se manter uma certa potência para o teste estatístico a realizar [277]. Sem se conhecer previamente o número de participantes e dado o número elevado de práticas esta estratégia não se mostrou viável.

Definição 12 (Potência do Teste)**.** *A* ***potência*** *estatística de um teste [83, p.52] "... descreve a probabilidade de um teste identificar correctamente um efeito genuíno."*

A potência de um teste depende de vários factores, incluindo o número de participantes. Na determinação da dimensão N de uma amostra, é calculado para cada condição um n correspondente ao número de participantes dessa condição [277] e, portanto, para a mesma potência do teste, mais condições implicam mais participantes.

Dadas as dificuldades apresentadas e limitações das alternativas apresentadas, optou-se por estudar apenas um número muito reduzido de práticas, mais concretamente duas. Esta estratégia reduz em particular o problema potencial de um número insuficiente de alunos e do código a produzir. A principal limitação é o número de práticas a estudar. A opção por apenas duas práticas possibilita mais participantes por situação a estudar, ou seja, um maior número de dados por situação, e consequentemente aumenta a potência dos testes estatísticos.

As práticas seleccionadas foram o encadeamento de métodos e a utilização de comentários no corpo de métodos. Não se pode dizer que a escolha tenha sido completamente objectiva e inquestionável. O encadeamento suscitou interesse por ser uma prática estudada apenas muito recentemente quanto ao seu impacto na legibilidade. Apesar de a adesão à Lei de Demeter ser tão importante para a engenharia de software, ainda não existiam evidências empíricas dos seus benefícios na legibilidade [111]. Por outro lado, o impacto dos comentários tem sido analisado desde os primeiros estudos sobre legibilidade do software, mas ainda continua a ser um ponto de alguma discórdia, como se conclui da revisão da literatura apresentada.

O encadeamento de métodos refere-se a métodos que retornam um objecto que pode ser usado como fonte para a chamada de outro método. Por exemplo:

$$objecto.metodo1(...).metodo2(...).metodo3();$$

O encadeamento de métodos é usado com muita frequência na programação OO e é também aconselhado como um bom estilo de programação [96, 139]. O encadeamento dos métodos poderá ser intuitivo se as chamadas aos métodos forem chamadas a métodos idênticos ou então se retornarem o mesmo tipo. Se os métodos estão encadeados de um modo *ad-hoc*, o encadeamento de métodos pode tornar o código pouco intuitivo, afectando a legibilidade, e poderá ainda violar a Lei de Demeter (LoD) [96, 31].

Esta lei, que não é realmente uma lei, requer que um objecto cliente só envie mensagens aos objectos que estão imediatamente no seu âmbito, e dessa forma ajuda a assegurar o encapsulamento da informação e a tornar o acoplamento mais explícito. Guo et al. mostraram que as violações à lei de Demeter conduzem a código com mais defeitos, tendo impacto negativo na qualidade do software [111]. Também em [214], o autor refere que os sistemas que não obedecem à Lei de Demeter são mais difíceis de implementar, de testar e de fazer a sua manutenção ao longo do tempo. No entanto, não foram encontrados estudos que relacionem a Lei de Demeter com a legibilidade [33].

Relativamente ao impacto do uso de encadeamento de métodos na legibilidade, [33] refere que foi encontrado apenas um estudo empírico muito recente, em [176]. De acordo com [33], nesse artigo, os autores Marinescu e Marinescu, verificaram que as classes que apresentam encadeamento de métodos são mais propensas a falhas e tornam o código

mais difícil de entender. No entanto, tendo lido o artigo original, essa informação não é explícita. No respectivo artigo, os autores afirmam, com base no estudo realizado, que as classes que utilizam classes com defeitos são mais propensas a falhas. Independentemente disso, o estudo não trata directamente da legibilidade.

A segunda prática escolhida consiste na utilização de comentários no interior de métodos. A utilização de comentários tem sido estudada desde há muito, mas com resultados contraditórios como indicado na revisão da literatura. Dos vários estudos sobre comentários analizados na revisão da literatura, a localização dos comentários varia. Em [33] também é estudada a utilização de comentários nos métodos. Sendo um estudo muito recente, mas não orientado para alunos de programação orientada a objectos, considerou-se relevante repeti-lo no contexto real de sala de aula de ensino de programação orientada a objectos.

Uma questão que se coloca com a utilização de comentários no código é o da sua qualidade, o que foi feito em [31]. É expectável que comentários com qualidade e sem qualidade contribuam de forma diferente para a legibilidade do código onde se inserem. Nesse sentido, para além da existência de comentários decidiu-se verificar a importância da qualidade do comentário, ou seja, a importância da existência de comentários com qualidade e de comentários sem qualidade. De uma forma simples, entende-se por comentários sem qualidade os comentários genéricos que não ajudam a entender o bloco de código.

No artigo acima referido foi realizado um estudo que envolveu os comentários e o encadeamento dos métodos [31]. O presente estudo irá repetir essa análise. A repetição de um estudo designa-se por replicação. Na replicação, os investigadores estudam a mesma questão de investigação estudada num estudo anterior e tentam que seja o mais idêntico quanto possível do estudo a ser replicado [45].

Apesar de tanto o estudo a realizar como o referido antes [31], apresentado na secção 3.3.3, possuirem como objectivo o estudo das mesmas práticas de legibilidades, o estudo a realizar difere em certa medida desse estudo. A principal divergência tem a ver com o contexto. Isto é, o estudo original não tinha como preocupação os alunos de programação orientada a objectos, nem mesmo que todos os participantes fossem alunos, contrariamente ao estudo a realizar em que todos os participantes terão de ser alunos de programação

orientada a objectos de um curso de informática. No entanto, tentou-se utilizar os mesmos materiais, mas apenas na medida do possível atendendo à concepção adoptada para o presente estudo e que será explicada durante a apresentação do estudo. Outra diferença está no desenho do experimestudo experimental. A terceira diferença tem a ver com a análise estatística.

Foi usado um estudo para validar as práticas seleccionadas por se saber que este tipo de estudo é superior aos restantes tipos em termos de validade [90, 297]. O estudo terá de ocorrer na sala de aula e serão usados pedaços de código (*snippets*) seleccionados previamente. É importante notar que com este estudo não se pretende desenvolver um modelo de legibilidade conforme fizeram outros autores, tais como, Buse [39] e Posnett[213]. Pretende-se antes estudar a importância das práticas na legibilidade.

5.2 Estudos experimentais

Existem diferentes estratégias de investigação cuja escolha depende do propósito e das condições da investigação. As estratégias encontradas habitualmente na literatura de estudos empíricos em engenharia de software são o inquérito, o caso de estudo, o experimento (estudo experimental) e o quasi-experimento [297, 90]. De todos, o estudo experimental é o que permite resultados mais válidos e generalizáveis.

- Inquérito: Um inquérito é usado em geral como um sistema para recolha de informação de forma mais alargada. Por vezes é uma investigação executada em retrospectiva. Os meios primários de recolha de dados qualitativos e quantitativos são as entrevistas e os questionários. Se possível, o inquérito é realizado numa amostra representativa da população a ser estudada. Posteriormente os resultados são analisados e obtidas as respectivas conclusões que podem ser descritivas ou explanatórias. No final, os resultados são generalizados à população de onde foi retirada a amostra, caso esta seja aleatória [297, 91];

- Caso de estudo: Consiste numa investigação empírica que se baseia em múltiplas fontes de evidência para estudar uma instância (ou um pequeno número de casos) de um fenómeno de engenharia de software no seu contexto real, especialmente quando a fronteira entre o fenómeno e o contexto não pode ser especificada claramente [297];

- Estudo experimental: Um estudo experimental (ou experimento) é um estudo empírico que manipula factores ou variáveis do ambiente estudado. Quando se conduz um estudo experimental formal pretende-se avaliar a consequência da variação das variáveis de entrada no processo em estudo. Mantendo-se as outras variáveis constantes, medem-se os efeitos resultantes. Têm-se então relações de causa-efeito. O objectivo de um estudo experimental é fundamentalmente avaliar uma hipótese [297]. Este tipo de estudo também se designa por estudo experimental formal, real, ou verdadeiro, ou ainda ensaio controlado. Aqui usar-se-á o termo estudo experimental.

- Quasi-experimento: É similar ao experimento controlado mas em que se relaxam algumas das suas condições. Em geral, a aplicação dos tratamentos não é feita de forma aleatória.

Os estudos experimentais são conduzidos quando se pretende controlar e manipular directa, sistematicamente e com precisão o comportamento do fenómeno e envolvem mais do que um tratamento de forma a permitir comparar os resultados observados entre eles [297]. No entanto, pode ser possível controlar certos factores e outros não. Os estudos experimentais são em geral limitados ao estudo de um número reduzido de variáveis. Pelo nível de controlo que requerem, os estudos experimentais são o tipo de estudo mais difícil de realizar. Apesar de um estudo experimental não ser simples, tem vantagens. Através da sua aplicação é possível a obtenção de conclusões mais válidas e generalizáveis [90]. As hipóteses formuladas são testadas através de análise estatística [297].

Num estudos experimentais verdadeiro está garantida a validade interna. A validade interna refere-se à relação causa-efeito entre as variáveis independentes e dependentes. Um estudo tem validade interna se o tratamento efectuado provocou efectivamente o efeito observado nas variáveis dependentes [90, 143-4]. A validade interna é conseguida através da atribuição aleatória das condições do tratamento aos participantes [277, 2]. A isto, chama-se randomização e o respectivo estudo, randomizado, ou aleatorizado. Com isso, características específicas (sistemáticas) dos participantes e não controladas no estudos experimentais serão distribuídas aleatoriamente pelas condições. Dessa forma, não haverá condições que apresentem participantes com uma característica específica [277, 14]. Contudo, todos os estudos empíricos têm limitações que ameaçam a sua validade.

Os estudos experimentais podem ser orientados às pessoas ou orientados à tecnologia, sendo os primeiros de controlo mais difícil, uma vez que as pessoas se comportam de modos diferentes em ocasiões/situações diferentes [297].

Num estudos experimentais consideram-se dois tipos de variáveis, as variáveis independentes e as variáveis dependentes [90, 297]. As variáveis independentes participam no processo e são manipuladas ou controladas pelo investigador. As variáveis independentes são também chamadas de factores. As variáveis dependentes são o resultado da manipulação das variáveis independentes, por isso, também designadas de variáveis de resposta.

As variáveis independentes podem ser, por exemplo, o método de desenvolvimento, a experiência profissional, as ferramentas de suporte e o ambiente. Um tratamento corresponde a um valor particular de um factor. Habitualmente, existem tantos grupos quantos os tratamentos, ou então mais um que funciona como grupo de controlo. Existem, no entanto, outras variáveis que podem afectar a variável dependente de forma indesejável, daí a randomização.

Os tratamentos são aplicados pela combinação de objectos e participantes. Um objecto pode ser um documento que deverá ser revisto com diferentes técnicas de inspecção. As pessoas que recebem o tratamento são chamadas de participantes ou sujeitos. As características de ambos os objectos e sujeitos podem ser variáveis independentes no estudo experimental.

Os estudos experimentais necessitam passar por diferentes fases bem definidas para assegurar que todos os aspectos importantes são tidos em conta e os resultados úteis [297, 90]. Na tabela 5.1 são apresentadas as fases do processo de experimentação como indicadas em duas obras da área da engenharia de software. Mas fora da engenharia de software também existem muitas obras sobre a concepção de estudos experimentais (e.g. [232]).

Apesar de idênticos, como seria de esperar, não existe uma correspondência exacta entre os 2 processos. Fenton [90] sugere que um verdadeiro estudo experimental deve envolver um processo de seis fases, enquanto Wohlin [297] indica 5 fases. Para o presente estudo decidiu-se não seguir de forma única qualquer um dos processos indicados por estes dois autores. No entanto, considerou-se importante utilizar ambos na definição de

Tabela 5.1: Fases do processo de experimentação indicadas em duas obras.

Fenton and Bieman [90]	Wohlin et al [297]
• Concepção • Desenho • Preparação • Execução • Análise • Disseminação e tomada de decisão	• Âmbito do estudos experimentais • Planeamento • Realização/Operação • Análise e Interpretação dos resultados • Apresentação dos resultados

um processo a ser seguido de forma a assegurar que nenhum passo ficava esquecido e obter-se assim um processo tão rigoroso quanto possível.

5.3 Replicação

A replicação de estudos tem sido apontada por diversos autores (e.g. [18][249]) como muito importante para o desenvolvimento da engenharia de software. Num estudo de replicação, os investigadores estudam a mesma questão de investigação estudada num estudo anterior utilizando material e participantes tão idênticos quanto possível do estudo a ser replicado [45].

Dada a dificuldade em obter amostras aleatórias, os estudos de replicação acabam por ser a forma de conseguir generalizar os resultados [232]. As replicações podem ser realizadas por várias razões. Uma razão óbvia para a replicação consiste em simplesmente verificar o trabalho de outros no sentido de evitar a existência de erros não intencionais [36]. Em [36] é apresentada a citação seguinte de Martin Goldstein e Inge Goldstein (p.3):

> "We now take for granted that any observation any determination of a fact even if made by a reputable and competent scientist might be doubted. It may be necessary to repeat an observation to confirm or reject it. Science is thus limited to what we might call public facts. Anybody must be able to check them experimental observations must be repeatable."

Um estudo de replicação pode ser definido como se segue [46, p.267]:

Definição 13 (Replicação)**.** *Repetição deliberada do mesmo estudo empírico com o propósito de determinar se os resultados do primeiro estudo podem ser reproduzidos.*

Carver [46, p.269-70] elaborou um conjunto de linhas de orientação para estudos de replicação. Essas linhas descrevem quatro tipos de informação que devem estar presentes num estudo de replicação e são apresentadas em seguida. No caso presente, parte da informação já foi dada a conhecer anteriormente na secção 3.3.3 e a restante será descrita durante a descrição do estudo.

A informação disponibilizada sobre o estudo original deve ser suficiente para permitir entender o estudo de replicação, e deve incluir:

- Questões de investigação;
- Participantes;
- Plano do estudo experimental;
- As variáveis de contexto;
- Resumo dos resultados.

A informação fornecida sobre a replicação deve permitir ao leitor entender detalhes específicos da replicação e deve incluir:

- Motivação para a replicação;
- Nível de interacção com o estudo experimental(ais) original;
- Modificações ao estudo experimental original;
- Comparação dos resultados da replicação com os resultados do estudo original, incluindo resultados consistentes e não consistentes.

As conclusões que comparam os estudos são importantes pois fornecem informações que podem ser extraídas apenas de uma série de estudos sobre a mesma questão mas que não podem ser obtidas a partir de um único estudo. Um estudo de replicação é tão importante, quer produza resultados idênticos aos do estudo original, quer produza resultados distintos [249, p.212].

Uma replicação pode ser designada como replicação exacta, ou replicação conceptual. A replicação exacta procura seguir os procedimentos do estudo original da forma mais fiel possível. A replicação conceptual usa a mesma questão de investigação, mas segue um procedimento experimental distinto [250, p.211]. Dadas as diferenças de procedimento entre este estudo e [31], o estudo actual será apenas um estudo de replicação conceptual.

O estudo original possuía as seguintes questões de investigação [31, p.888]:

RQ1: De que modo a quantidade e a qualidade dos comentários do código-fonte afectam a legibilidade e compreensão do software?

RQ2: De que modo o encadeamento de métodos afecta a legibilidade e compreensão do software?

No estudo actual não se explicita a compreensão nas questões de investigação porque esta está subjacente à legibilidade. O estudo original consistiu num estudo experimental único com um plano 3x2 com blocagem. Mais informação sobre o estudo original pode ser encontrado no capítulo 3 de revisão da literatura. Esse estudo será referido neste capítulo sempre que necessário.

5.4 Processo Seguido

O processo adoptado para o estudo tem por base os processos sugeridos por [297, 90] como referido no final da secção 5.2. Com base nesses processos, o processo adoptado para o estudo dividir-se-á então nas seguintes fases, com cada uma a corresponder a uma secção deste capítulo:

- Objectivo e Questões de Investigação;
- Planeamento do Estudo Experimental;
- Preparação;
- Execução;
- Análise e Interpretação dos Resultados;
- Limitações e Ameaças à Validade.

5.5 Objectivo e Questões de Investigação

De uma forma genérica, o principal objectivo deste estudo consiste na validação das práticas escolhidas quanto à sua legibilidade. Sendo um estudo experimental uma tarefa trabalhosa e complexa não se deve avançar sem uma definição mais clara dos seus objectivos. Existe um conjunto importante de aspectos que contribuem para uma melhor identificação dos objectivos da investigação que devem ser definidos antes do planeamento e execução do estudo experimental [297].

Wohlin [297] sugere um modelo (*template*) de modo a assegurar que tais aspectos não são esquecidos. Os itens seguintes correspondem a esse modelo (*template*) de definição do objectivo. Os itens são na realidade questões que devem ser respondidas pelo investigador.

- <*objecto de estudo* > - o que vai ser estudado?
- <*propósito*> - qual é a intenção?
- <*enfoque*> - qual o efeito do que vai ser estudado?
- <*perspectiva*> - sob que ponto de vista?
- <*contexto*> - em que situação é que o estudo vai ser conduzido?

Seguindo o *template*, tem-se então para o primeiro item que no estudo serão estudadas duas práticas, a de encadeamento de métodos e a utilização de comentários no interior dos métodos.

Estas práticas serão estudadas quanto à sua influência na facilidade de leitura do código produzido pelos alunos e é, portanto, este o objecto que irá ser "observado".

Um aspecto considerado importante na realização de qualquer estudo empírico é a definição da unidade experimental do estudo [152]. A unidade experimental é definida da seguinte forma em [146, p.1]:

Definição 14 (Unidade Experimental)**.** *Uma unidade experimental é a entidade que é atribuída a uma condição experimental independentemente de outras entidades.*

De uma forma mais simples, a unidade experimental é a unidade a analisar [11], ou seja, a unidade que irá receber o tratamento [82]. A entidade que irá receber os tratamentos (e que é independente das outras entidades) é o aluno, e, por isso, é ele a unidade

experimental. Neste texto, usar-se-ão indistintamente os termos, aluno, participante e unidade experimental.

Quanto ao segundo item, este estudo tem como propósito verificar o impacto que a aplicação destas práticas poderá ter na legibilidade do código-fonte. A legibilidade é portanto o efeito a observar, ou seja, o terceiro item. Relativamente ao item seguinte, o quarto, a perspectiva do estudo é a da investigadora e dos docentes de programação orientada a objectos e os principais destinatários dos resultados do estudos são os alunos. Por último, o contexto, envolve os alunos de programação orientada a objectos dos cursos de informática do ensino superior.

Daqui foram produzidas as questões de investigação seguintes, uma para cada prática a estudar e para o contexto indicado.

QI1: Em que medida é que o código-fonte que *não contém encadeamento de métodos* (NMC) é mais legível que o código que *contém encadeamento de métodos* (MC)?

QI2: Em que medida é que o código com *comentários* (GC) é mais legível que o código *sem comentários* (NC) ou com *"maus" comentários* (BC)?

Cada uma das práticas será avaliada através de um estudo experimental específico para o efeito. O principal motivo para esta decisão foi tentar simplificar a análise estatística. Nesta altura já estava a ser feito um levantamento de possibilidades para a análise estatística, a qual cresceria em complexidade com o aumento do número de variáveis, como referido na introdução do capítulo. Ter-se-ão assim dois estudos experimentais, um para cada prática. Os dois estudos experimentais são descritos em paralelo nas secções seguintes.

5.6 Planeamento do Estudo Experimental

Os estudos experimentais, e no caso do software não é excepção, requerem planeamento de modo a fornecerem resultados com significado e úteis [90]. O planeamento do estudo experimental também é designado por concepção do estudo, dependendo dos autores. No planeamento de um estudo experimental em engenharia de software vários autores (e.g. [297, 90]) sugerem o seguimento de várias fases. Por exemplo, Wohlin sugere as seguintes fases para o planeamento [297]:

1. Âmbito
2. Formulação das hipóteses
3. Selecção das variáveis
4. Selecção dos participantes
5. Escolha do tipo de desenho experimental
6. Instrumentação
7. Avaliação da validade

Relativamente a desenho experimental (do Inglês "*experimental design*") será apenas uma das fases. Em Português, tanto é possível encontrar o termo "desenho experimental" (e.g. [221]) como o termo "plano experimental" (e.g. [129]).

5.6.1 Âmbito

O contexto do estudo pode ser caracterizado de acordo com 4 dimensões:[297]

- *off-line* vs *on-line*;
- estudante vs. profissional;
- problema ficticio vs. problema real;
- específico vs. genérico.

Num estudo, para se conseguir uma maior generalização dos resultados é importante realizar o respectivo estudo em projectos reais [297]. Nesse sentido, foi definido que os estudos experimentais seriam realizados no contexto de sala de aula de uma disciplina de introdução à programação orientada a objectos de um curso de informática.

Pretende-se que os resultados do estudo não sejam específicos deste estudo, mas sejam o mais generalizáveis possível. No entanto, existem sempre aspectos que limitam uma generalização, como se verá para este estudo na secção 5.10 sobre ameaças à validade. Os estudos experimentais realizados com alunos, como o actual, são considerados habitualmente estudos experimentais relativamente fáceis de controlar dado o ambiente contido em que se realizam [297]. Para o estudo serão usados pequenos pedaços de código (*snippets*)[1], de código aberto tal como no estudo original.

[1]Estes dois termos serão usados indistintamente.

5.6.2 Hipóteses

Nesta fase de planeamento há que traduzir os objectivos num conjunto de hipóteses concretas. As hipóteses incluem: as hipóteses nulas e as hipótese alternativas para cada hipótese nula.

Como foi referido, o único estudo sobre o impacto do encadeamento na legibilidade, mostra existir alguma superioridade na legibilidade de código sem encadeamento. A utilização de encadeamento tenderá a tornar as respectivas instruções mais extensas e, eventualmente, mais complexas. Como visto na revisão da literatura a existência de linhas de código compridas poderia explicar uma menor legibilidade. Tendo em linha de conta este resultados, a hipótese alternativa terá de mostrar a tendência para um efeito positivo na legibilidade da não utilização de encadeamento. A hipótese alternativa tem por base a questão de investigação.

$H1_0$ Hipótese Nula: Não existe diferença significativa na legibilidade do código-fonte que não possua encadeamento de métodos ou que possua encadeamento de métodos.

$H1_a$ Hipótese Alternativa: O código-fonte que não possui encadeamento de métodos será significativamente mais legível do que o mesmo código quando possui encadeamento de métodos.

A hipótese alternativa traduz então o efeito positivo esperado da prática de não utilização de encadeamento.

No estudo original os comentários "bons" tiveram um efeito positivo significativo na legibilidade. O mesmo não aconteceu com os comentários "maus" e com a ausência de comentários. No entanto, código com "maus" comentários obteve um resultado superior ao código sem comentários. Contudo, de acordo com a revisão da literatura não existe consenso acerca da utilização de comentários. Pareceu então mais razoável hipotetizar pela superioridade do efeito de comentários "bons", sobre os restantes níveis.

$H2_0$ Hipótese Nula: Não existe diferença significativa na legibilidade do código-fonte com ou sem comentários nos métodos.

$H2_a$ Hipótese Alternativa: A legibilidade do código-fonte varia significativamente consoante se usam comentários "bons" (GC), comentários "maus" (BC), ou não se usam comentários (NC). Isto é, $GC > BC \approx NC$.

5.6.3 Variáveis

Para o presente estudo consideram-se como variáveis independentes as práticas em estudo. Note-se que as variáveis independentes também poderão ser úteis na definição do âmbito do estudo. Para o estudo, a legibilidade do código-fonte é o conceito em análise e que constituirá a variável dependente, cuja resposta se pretende avaliar. A legibilidade é um conceito abstracto.

A utilização de encadeamento pode variar em duas vertentes. Primeiro, um programa pode possuir encadeamento de muitos níveis, *2* a *n* métodos encadeados. Em segundo lugar, a percentagem de utilização de encadeamento no código pode variar desde utilizar o encadeamento apenas numa instrução até o utilizar em todas as instruções em que tal seja possível. Para além disso, a dimensão dos programas é variável. Estes factores criam um gama infinita de valores possíveis para o encadeamento. Seria interessante estudar todas essas possibilidades, mas isso não é viável. Tal como em [31] também aqui se optou por considerar dois valores para a variável. A variável independente relativa ao encadeamento possui então 2 níveis: existência de encadeamento de métodos (MC); e inexistência de encadeamento de métodos (NMC).

Os comentários podem variar muito em detalhe e na forma como são escritos. Mas em geral, pode-se dizer que há comentários melhores e outros piores. A qualidade de um comentário depende da informação que pode transmitir. Quando essa informação está associada à semântica do código, ou seja, à sua relação com o problema associado ao código, o comentário contribui para se compreender a razão de ser do código e considera--se que o comentário tem qualidade, ou seja, é um "bom" comentário. Caso o comentário sirva apenas para descrever o código então o comentário pouco ou nada contribui para se justificar o código lido e é considerado de má qualidade, ou seja, é um "mau" comentário. Por outro lado, há quem defenda a inexistência de comentários. Existirão muitos níveis intermédios, mas é necessário limitar a gama de valores para tornar o estudo exequível. A variável relativa aos comentários possuirá então 3 níveis: comentário "bom" (GC); comentário "mau" (BC); e inexistência de comentário (NC).

5.6.4 Medidas

A legibilidade é um conceito abstracto, algo não mensurável directamente. A legibilidade é o conceito objecto de estudo. Este tipo de conceito abstracto designa-se por constructo[2]. Será portanto necessário proceder à sua operacionalização [179]. Aliás, o software é pródigo em conceitos desse tipo, sendo a complexidade do software outro exemplo.

Operacionalizar o conceito (a variável) consiste em tornar o conceito "palpável", ou seja, passível de medição. Para isso, é preciso encontrar outra variável, dita operacional, que represente o melhor possível o conceito e que possa ser medida.

Uma das formas de medir a legibilidade do código-fonte será através do julgamento humano, isto é, perguntando aos alunos qual o grau de legibilidade desse código. Esta prática tem sido usada desde há muito em estudos de legibilidade no software e foi usada no estudo original [31]. Segundo um estudo empírico [131, p.402] (cf. secção 3.3.3) verificou-se existir uma correlação considerada forte, de 0.85, entre o julgamento de legibilidade do código pelo aluno e a avaliação da compreensão dos programas através de questões sobre o programa.

Como referido no capítulo 4, uma escala do tipo Likert de cinco pontos tem uma fiabilidade elevada [133]. Também em [31] foi usada uma escala de cinco pontos. Por essas razões, optou-se por usar no presente estudo uma escala do tipo Likert de cinco pontos como medida de legibilidade. É, portanto, uma escala ordinal[3]. O aluno terá de classificar a legibilidade do código segundo essa escala. Esta medida será designada simplesmente por *Escala*. A escala a usar será de 5 níveis de classificação, que variam de "Muito difícil" a "Muito fácil".

1. Muito difícil
2. Difícil
3. Neutro
4. Fácil
5. Muito fácil

[2]Do Inglês *construct* e como usado por outros autores de língua Portuguesa e.g. [188].
[3]Em bom rigor é apenas um item Likert [267]

Muitas das medidas de legibilidade estudadas no capítulo 2 são de aplicação ao nível do próprio texto. Uma das excepções é o chamado "Teste *Cloze*" que é usado ao nível do leitor e, como tal, esta medida pode ser usada no presente contexto. Ao não se aplicar directamente ao texto, não parece existir motivo para que não se possa aplicá-lo à leitura de código-fonte. Este teste foi também o escolhido em [31] como medida da compreensão. A medida para o teste Cloze será uma percentagem obtida pela razão entre o números de casos preenchidos correctamente e o total de casos a preencher. Uma percentagem é considerada uma medida que pode ser usada como sendo do tipo rácio [307]. Esta medida será designada apenas por *Cloze*.

A legibilidade e a compreensibilidade estão relacionadas mas são conceptualmente diferentes [33]. A legibilidade é requerida para a compreensibilidade mas a legibilidade não implica necessariamente compreensibilidade. Isto torna dificil medir objectiva e independentemente legibilidade da compreensibilidade.

Existem vários procedimentos de elaboração do teste Cloze, dependendo do tamanho do texto a analisar. Em situações de textos de pequena dimensão é recomendado eliminar as palavras do texto de modo aleatório. Não havendo nada contra, neste estudo replicou-se a implementação do teste Cloze usada no estudo [31].

O teste Cloze também poderá ajudar a avaliar a fiabilidade das respostas dos alunos à Escala, ou seja, se existe uma relação entre o valor da medida Escala indicado pelo aluno e o resultado da aplicação da medida Cloze. No entanto, essa utilização dependerá de se saber se existe uma correlação entre ambas. Contudo, tanto quanto foi possível investigar, não se encontraram estudos que estabelecessem essa correlação, o que fez com que se prescindisse dessa avaliação da fiabilidade.

É expectável que um texto seja lido tanto mais rapidamente quanto mais legível ele for [13]. As partes de um texto que não são compreendidas numa primeira leitura requerem novas leituras até à sua compreensão, ou seja, requererão mais tempo de leitura. O mesmo é de esperar no caso de código-fonte. Tendo essa expectativa em consideração, decidiu-se medir o tempo de leitura. Esta última medida será designada por *Tempo de Leitura*. A medida para o Tempo de Leitura será obtida pela diferença do tempo de início de leitura e o tempo de término da leitura do código-fonte antes de responder a qualquer questão sobre o mesmo. Esta medida poderá ter relação com a medida Cloze. Durante o estudo,

o aluno terá de anotar o tempo necessário para cada tarefa.

5.6.5 Plano Experimental

De uma forma genérica, o plano experimental tem a ver com a forma como se organizará a aplicação dos tratamentos, o que depois condicionará a análise estatística. Num estudo experimental cada um dos tratamentos pode ser aplicado a um grupo distinto de participantes, ou em alternativa, cada participante pode receber os vários tratamentos:

- plano *intra-sujeitos* ou *relacionado*[4] (*whithin subject design* ou *repeated measures*);
- plano *inter-sujeitos* ou *não-relacionado* (*between-subjects design*);
- plano *emparelhado* (*paired*).

Um plano *intra-sujeitos* é aquele em que são aplicados os diferentes tratamentos, ou níveis do tratamento, ao mesmo participante [104]. Num plano *inter-sujeitos* cada tratamento, ou nível de tratamento, é aplicado apenas uma única vez a cada participante [104]. Também se pode ter um plano *emparelhado* em que se procura que grupos distintos possuam participantes com características idênticas. Como os grupos se tornam idênticos este plano é idêntico ao relacionado [104]. Na concepção de um plano experimental existem alguns princípios gerais que podem ser usados de forma combinada. Esses princípios são:

- Randomização;
- Blocagem;
- Balanceamento.

No caso presente, apresentar o mesmo código, respectivamente, com e sem encadeamento, a cada participante (plano intra-sujeitos) seria problemático porque o participante ficaria a conhecer o código após o primeiro tratamento o que influenciaria o resultado na segunda apresentação. Este tipo de efeito designa-se por *carry-over*. Uma possibilidade para resolver o problema seria fornecer um *snippet* de código-fonte diferente ao mesmo aluno. Contudo, esse código teria de ser muito semelhante em termos de legibilidade, o

[4]Adopta-se aqui a terminologia usada em [104].

que se afigurou demasiado arriscado se não mesmo inviável. Também seria possível proceder ao contra-balanceamento, isto é, aplicar os 2 níveis por diferente ordem a diferentes participantes[5]. Contudo isso implica duplicar o tempo necessário para cada participante. Segundo [180] este tipo de plano requer cuidados especiais na definição da unidade experimental.

O plano inter-sujeitos de apresentação do código-fonte uma única vez a cada participante elimina o problema de *carry-over* e a dificuldade em encontrar pedaços de código idênticos em termos de legibilidade. Esta concepção implica ter tantos grupos de participantes quantas as condições de tratamento. O principal inconveniente é poder acontecer existirem condições de tratamento que "recebam" apenas um certo tipo de participantes. No entanto, este problema é resolvido, ou minimizado, com a randomização. Quanto ao plano emparelhado não parece ser apropriado para o estudo por não se perceber que característica distintória poderia servir para o emparelhamento.

Aparentemente a solução inter-sujeitos será a mais viável. O que se considerou determinante para a escolha foi o facto de o plano intra-sujeitos obrigar o aluno a ler mais pedaços de código que no plano inter-sujeitos. O plano intra-sujeitos tem a vantagem de requerer menos participantes. No entanto, a leitura de vários *snippets* requereria mais tempo, para além do esforço por parte dos alunos, aumentando o risco de o estudo ultrapassar a duração de uma aula. De notar, que o estudo deverá realizar-se em sala de aula. A aula do tipo teórico é aquela que permite reunir o maior número de alunos de uma só vez. Este tipo de aulas tem habitualmente a duração de 50 minutos. Em [31] foram analisados 5 *snippets* por cada aluno. Mas na situação do presente trabalho os alunos ainda estão a aprender POO por isso seria muito arriscado tentar usar essa quantidade de *snippets*. Neste estudo adoptou-se então a solução inter-sujeitos.

Randomização A randomização é considerada uma actividade muito importante para a validade interna do estudo pois permite resolver o problema de muitas variáveis *nuissance*. Existem características individuais dos participantes que podem influenciar os resultados de forma sistemática. Algo que interfere certamente é a inteligência dos participantes. A prática de randomização, ao distribuir aleatoriamente as condições pelos participantes,

[5]Para 2 níveis, A e B, requeriria AB e BA.

faz com que os resultados sejam independentes das características de cada participante [110]. Por outras palavras, a probabilidade de haver pessoas com capacidades distintas é igual para todas as condições experimentais [104].

Dessa forma, "anula-se" o efeito de factores não controlados e consegue-se assim um maior isolamento das variáveis independentes relativamente a outros factores. Consequentemente, há mais segurança que o resultado observado na variável dependente se deva à manipulação das variáveis independentes e com isso aumenta a validade interna do estudo. A randomização aplica-se independentemente do delineamento escolhido, isto é, inter-sujeitos ou intra-sujeitos.

Existem vários métodos/instrumentos para se conseguir a randomização, uns recorrem a tabelas pré-definidas de números aleatórios que depois são escolhidos segundo algum critério [11], outros utilizam um algoritmo para combinar aleatoriamente as condições com os participantes. É possível encontrar programas informáticos que implementam os segundos, pelo que se tornam mais apelativos.

É expectável que os alunos de uma disciplina se encontrem distribuídos por mais do que uma turma. No entanto, não parece que essa divisão implique diferenças entre os alunos e, logo, tenha impacto na atribuição das condições aos participantes. Por isso, o agrupamento turma será ignorado e neste estudo as situações experimentais serão atribuídas aos alunos de modo aleatório e independente da turma. Trata-se de uma atribuição aleatória simples.

Para o estudo foi decidido que seria preferível usar um método automático para assim facilitar o processo. A sua escolha será feita durante a preparação.

Blocagem A técnica de blocagem (*blocking*) surgiu na área agrícola, onde se definiam blocos (áreas) de terra para os estudos experimentais aos quais eram depois atribuídos os tratamentos de forma aleatória [224]. O plano experimental com blocagem e randomização deve-se ao estatístico Sir Ronald A. Ficher (1890-1962) [224]. A técnica serve para eliminar fontes conhecidas, que não as variáveis independentes, de efeitos sistemáticos nos resultados, ou mais concretamente para blocar esses efeitos [52]. Habitualmente, trata-se de factores relacionados com os participantes. Esses factores designam-se por variáveis

nuissance[6] [146]. A técnica consiste em agrupar os participantes segundo esses factores. Esta técnica contribui assim para aumentar a precisão do estudo [52].

A escolha da blocagem apresenta vantagens e desvantagens [52]. A vantagem da blocagem depende de a variação entre tratamentos dentro de cada bloco ser inferior à variação entre blocos. Das vantagens destaca-se propiciar em geral a redução do erro experimental e o aumento da precisão na comparação dos tratamentos. Por outro lado, se houver uniformização dos tratamentos para todos os blocos, o que quer dizer que não há variação entre os blocos, então a blocagem torna-se uma desvantagem relativamente a um plano de randomização simples. Finalmente, a existência de dados em falta também pode afectar a análise estatística com blocagem.

Neste estudo foi aplicada esta técnica sobre os *snippets* com o objectivo de se conseguir maior generalização dos resultados, como se explica em seguida. Existe um número infindável de programas escritos em linguagens orientadas a objectos. Esses programas diferem entre si de muitas formas. Quando se pretende estudar a legibilidade, impõe-se questionar se a legibilidade não variará entre programas distintos, ainda que apliquem as mesmas boas práticas de legibilidade. Para tornar os resultados mais generalizáveis torna-se necessário aplicar as práticas em estudo a mais do que um pedaço de código.

Os pedaços de código, *snippets*, são então a variável, ou factor, de blocagem e cada pedaço de código diferente será um nível desse factor. Este factor irá ser referido por *snippet*. A diferença entre os *snippets* servirá para indicar que o efeito observado não depende de um pedaço específico de código. A opção entre usar um factor como uma variável independente ou de blocagem não é sempre clara e fica muita vezes ao critério do investigador [90]. Neste caso, o factor de blocagem *snippet* não será considerado uma variável independente porque não é o objecto de estudo e não se procura que os *snippets* possuam níveis definidos de alguma característica própria distintória com eventuais efeitos conhecidos na legibilidade. As variáveis externas, para além das independentes, que podem interferir no efeito e não tem a ver com a randomização designam-se por variáveis de confusão (*confounding*) [82].

Para os estudos foram escolhidos 3 *snippets* de entre os 5 utilizados em [31] para um estudo experimental e os 2 restantes para o outro estudo. Relembre-se que uma das

[6]Este termo não será traduzido de modo a reduzir o risco de perda de informação.

preocupações do presente estudo é conseguir obter um número elevado de observações por situação experimental. Portanto, para um número fixo de participantes, quantos mais *snippets* houver menor será o número de observações por situação. Foi com isso em mente, que se decidiu, usar 3 dos *snippets* para o estudo relativo ao encadeamento de métodos e reservar os 2 restantes para o estudo relativo aos comentários.

A utilização de 3 níveis para o factor de bloco, triplicará o número de células do plano experimental. O resultado são 6 células, como mostrado na tabela 5.2. Este plano experimental designa-se por *Randomized Block Design*. No caso dos comentários, o número de blocos é 2, mas os níveis são 3 de que resultam 6 situações (tabela 5.3).

Tabela 5.2: Plano experimental para encadeamento.

Blocos	Com Encadeamento	Sem encadeamento
B1 (S1)	$Situação_{11}$	$Situação_{12}$
B2 (S4)	$Situação_{21}$	$Situação_{22}$
B3 (S5)	$Situação_{31}$	$Situação_{32}$

Tabela 5.3: Plano experimental para comentários.

Blocos	Comentários "Bons"	Comentários "Maus"	Sem Comentários
B1 (S2)	$Situação_{11}$	$Situação_{12}$	$Situação_{13}$
B2 (S3)	$Situação_{21}$	$Situação_{22}$	$Situação_{23}$

Balanceamento O balanceamento está relacionado com o número de observações em cada situação experimental (combinação de factores). Um plano diz-se balanceado se existe o mesmo número de observações para cada situação experimental [204]. Caso contrário diz-se não balanceado. O balanceamento pode ser potenciado, mas nunca pode ser garantido à priori porque podem ocorrer eventos imprevistos como, por exemplo, a desistência de participantes. A principal vantagem do balanceamento é facilitar a análise estatística.

A forma de randomização definida para o estudo é simplesmente aleatória. Como o plano escolhido apresenta 6 situações, serão estas situações que terão de ser atribuídas aleatoriamente aos participantes. Existem outras formas de randomização (ver e.g. [233]) que poderiam ter resultado num melhor balanceamento. Dado ser um plano com blocos, também se poderia ter optado pela randomização por bloco. Como os blocos não tinham a ver com especificidades dos participantes, a atribuição aleatória dos participantes às condições dentro de cada bloco não deverá diferir da randomização simples quanto a características dos participantes. No entanto, também poderia ter contribuído para um melhor balanceamento.

5.6.6 Análise Estatística a Realizar

A análise estatística é influenciada pelo plano experimental definido[7] [99]. Ou seja, nem qualquer teste estatístico se adequa a qualquer plano. No caso presente trata-se de um plano aleatorizado com blocagem. A escolha do teste não foi óbvia e nos parágrafos seguintes descrevem-se alguns dos passos mais importantes desse processo.

Os testes estatísticos podem ser divididos em dois tipos:

- Testes paramétricos, ou métodos de teste paramétrico, que requerem que os dados sigam uma determinada distribuição [90];
- Testes não paramétricos, ou livres de distribuição da população [123]. Estes testes não processam directamente os dados obtidos, mas sim as suas gradações.

Os testes do tipo paramétrico, como o $t-test$ são considerados mais potentes que os seus equivalentes não-paramétricos, daí ter sido dada prioridade aos primeiros como se verá nesta secção. Um teste mais vulgar como o $t-test$ permite comparar dois grupos, ou amostras, independentes. Isso quer dizer que se aplica aos casos em que se tem uma única variável independente com 2 níveis.

Para o caso de comparação de mais do que dois grupos independentes é necessário recorrer-se a um outro tipo de teste como o teste ANOVA - análise de variâncias [204]. O teste ANOVA[8] usa-se então quando uma variável possui mais do que 2 níveis e pode

[7] A definição do tipo de plano experimental foi conseguido com o apoio da Professora Barbara Kitchenham, especialista em estudos empíricos.

[8] O teste também se designa por método (de teste) ANOVA, ou modelo ANOVA.

ser usado com planos inter-sujeitos. O teste ANOVA procura dividir a variância de uma amostra em duas partes: na variância que é devida ao factor e na variância que é devida ao residual, que é o erro. Para uma apresentação detalhada do modelo ver e.g. [204].

No entanto, no plano experimental definido para o estudo, os blocos constituem uma variável adicional, o que requer um método adequado para planos com mais do que uma variável independente. No caso de dois factores seria o modelo *two-way* ANOVA [183, 204]. Esta análise de variância permite averiguar se a média varia com uma ou ambas as variáveis [87]. Como existem dois factores e cada factor explica uma parte da variância, há que dividir a variância segundo o que se deve a cada um dos factores e ao residual.

Para mais do que uma observação por cada nível de um tratamento o teste ANOVA diz-se com replicação ou repetição. Segundo [43], um plano com blocos diz-se próprio se todos os blocos contêm o mesmo número de observações. Já um plano com blocos em que o número de replicações é igual para todos os tratamentos diz-se igualmente replicado (*equireplicate*).

Em cada estudo experimental existem três variáveis dependentes, a medida pela Escala, a medida pelo teste Cloze e a medida pelo Tempo de Leitura. Apesar de poder haver uma relação entre si (Escala e Cloze), porque se referem em certa medida ao mesmo conceito, este estudo tem como finalidade analisar o efeito de cada prática nessas variáveis dependentes, mas não o estudo da relação entre as variáveis, ou seja, entre formas de medir a legibilidade. No caso de se pretender incluir no teste estatístico a relação entre as duas variáveis, MANOVA seria o teste a usar na linha de ANOVA, mas para várias variáveis dependentes.

Todos os métodos de teste de hipóteses requerem que as observações sejam independentes [277]. Em rigor, trata-se da independência dos componentes de erro associados a cada observação, ou por outras palavras, que os erros sejam variáveis independentes que não covariam com as variáveis estudadas [99].

No caso concreto do modelo ANOVA, a sua utilização é condicionada por vários pressupostos (e.g. [118, 99, 52]):

- As amostras a estudar terem sido obtidas aleatoriamente;
- População com distribuição normal;

- Homogeneidade das variâncias das populações;
- Variáveis dependentes serem do tipo intervalar.

A natureza aleatória da amostra é importante para garantir que ela é representativa da população. No entanto, em geral, é raro esta condição ser respeitada [277]. Também na engenharia de software é raro encontrar-se um estudo empírico que use amostragem aleatória [232].

O segundo pressuposto é aquele que será violado mais frequentemente sempre que o método ANOVA é usado [118]. Contudo, este problema é minimizado com a utilização de amostras grandes. Em termos práticos, ANOVA pressupõe que os erros (ou residuais) sejam variáveis aleatórias independentes com distribuição normal de média 0 e desvio padrão 1 [99]. A componente de erro representa o efeito de todas as variáveis que não foram incluídas no estudo e que terão algum efeito sobre a variável dependente - a legibilidade. Portanto, os residuais permitem perceber a adequação do modelo ANOVA aos dados obtidos [195].

O teste ANOVA requer que os vários grupos apresentem variâncias iguais ou parecidas [108]. Sabe-se que ANOVA é robusto à violação dessa condição desde que os grupos possuam o mesmo número de participantes [277, 292]. No entanto, sabe-se que violações mais fortes destas condições ocorrem com alguma frequência [292]. Adicionalmente, estudos mais recentes têm mostrado que desvios à normalidade dos dados podem ter consequências importantes nos resultados dos testes [148, 292]. Em [149] pode ser encontrada uma discussão das limitações de vários métodos no contexto da investigação em engenharia de software.

Para o teste ANOVA assume-se que a variável dependente é do tipo intervalar. Em rigor, segundo a teoria da medição, a variável teria de ser pelo menos de uma escala do tipo intervalar em virtude das operações aritméticas necessárias [229, 90]. Relembra-se que a medida Escala é do tipo Likert. Existe alguma discussão acerca da importância do pressuposto e também se uma escala tipo Likert poderá ser considerada uma escala do tipo intervalar (ver e.g. [303]). Uma possível explicação para essa divergência poderá estar na diferença entre item de Likert e escala de Likert, em que o primeiro é do tipo ordinal e a segunda intervalar [267].

No presente estudo têm-se três variáveis dependentes, uma para cada medida, e não uma, o que faz supor uma análise bivariada. Para estes casos existe a variante MANOVA. Um teste multivariado como o MANOVA pretende saber se alguma combinação das variáveis dependentes varia em função dos tratamentos [269]. O MANOVA é superior a usar várias ANOVA, mas com o custo de maior complexidade e, muito possivelmente, de alguma ambiguidade na interpretação dos efeitos. No caso presente, supõe-se que exista alguma relação entre as medidas, mas como dito anteriormente, não faz parte dos objectivos da investigação determinar se existe alguma combinação dessas variáveis dependentes que varie em função do tratamento.

Quanto ao número mínimo de observações por célula para ANOVA em [205] é indicado o valor 10, mas aconselhado um valor de 30 ou superior. Em termos de balanceamento é sugerido um rácio máximo de 1:4 entre células, ou seja não haver nenhuma célula com menos de 1/4 de observações de qualquer outra. No caso de haver, é possível juntar grupos desde que as suas médias sejam idênticas [205].

Como já referido, os métodos estatísticos $t - test$ ou ANOVA requerem que os dados sigam uma determinada distribuição [90].

Na análise dos dados da medida de Cloze e da medida do Tempo de Leitura prevê-se avaliar a normalidade dos dados e se a avaliação for positiva, utilizar-se-á a análise de variâncias. Se for negativa, usar-se-á um método não paramétrico para blocos, usando-se a análise de variância apenas como método complementar essencialmente indicativo. No caso da medida Escala aplica-se a segunda possibilidade.

As hipóteses formuladas anteriormente (secção 5.6.2) são designadas por teóricas ou científicas, mas para o estudo elas têm de ser traduzidas em hipóteses estatísticas [146]. Estas hipóteses estatísticas são aquelas que serão testadas estatisticamente. A hipótese nula do teste ANOVA é distinta da hipótese de um teste não paramétrico. No primeiro, testa-se se as populações de onde as amostras terão sido retiradas possuem a mesma medida de localização, enquanto no segundo, a hipótese nula é que as populações são idênticas em todos os aspectos. As hipóteses segundo o modelo ANOVA permitem comparar as médias das populações. Para o primeiro estudo, existem apenas 2 níveis e consequentemente, duas populações em que na hipótese nula as suas médias deverão ser iguais:

$H1_0 : \theta_1 = \theta_2$

$H1_a : \theta_1 > \theta_2$

onde,

θ_1 é a média da legibilidade do código sentida pela população dos participantes no nível NMC da condição de tratamento, a ser estimada pela média da amostra.

θ_2 é a média da legibilidade do código sentida pela população dos participantes no nível MC, da condição de tratamento, a ser estimada pela média da amostra.

Para o segundo estudo tem-se as seguintes hipóteses:

$H2_0 : \theta_1 = \theta_2 = \theta_3$

$H2_a : \theta_1, \theta_2, \theta_3$ não são todos iguais

A variável de blocagem (*snippet*) tem unicamente a finalidade de ajudar a generalizar os resultados, ou seja, não se pretende estudar o seu efeito. Caso fosse pretendido, haveria que definir as respectivas hipóteses e ainda as hipóteses para a interação entre as duas variáveis independentes.

Com o teste estatístico pretende-se infirmar a hipótese nula e como tal é designado por teste estatístico da hipótese nula (NHST - *Null Hypothesis Statistical Test*). Para isso, o teste usa o valor-p (*p-value*) também conhecido por *probabilidade de significância* ou por *valor de prova* [110]. A hipótese nula é rejeitada quando o valor-p não excede o nível de significância definido previamente.

A decisão de aceitar ou rejeitar a hipótese nula pode estar errada, ou não. No caso de a decisão estar errada podem ser cometidos 2 tipos de erro [90]:

Tipo I - Se a hipótese nula for rejeitada e não existir na realidade um efeito;

Tipo II - Se não se rejeitar a hipótese nula e existir na realidade um efeito.

Se a hipótese nula for rejeitada e não existir na realidade um efeito então comete-se um erro Tipo I, se não se rejeitar a hipótese nula e existir na realidade um efeito então comete-se um erro Tipo II. O erro Tipo II é a probabilidade de não se ter conseguido detectar o efeito quando ele realmente existia. Ver tabela 5.4.

A potência de um teste é a probabilidade de se inferir correctamente que o efeito existe e corresponde ao primeiro quadrante da tabela (ver definição 5.1). Portanto, a potência será igual a $1 - \beta$.

Tabela 5.4: Resultados da inferência.

Hipótese nula	Rejeitada	Não rejeitada
Existe efeito	decisão correcta	β
Não existe efeito	α	decisão correcta

em que:

α - Representa a probabilidade de se cometer um erro Tipo I;

β - Representa a probabilidade de se cometer um erro do Tipo II.

O teste da hipótese nula pode ser acompanhado do respectivo intervalo de confiança para a estatística obtida [283]. Este intervalo de confiança mostra o intervalo de valores plausíveis da estatística de teste na população. Desde há algum tempo (e.g. [148]) que vem sendo recomendado utilizar o designado tamanho (ou magnitude) do efeito (*effect size*). A sua utilização deve complementar o NHST ([294]). Alguns autores defendem a utilização do par tamanho do efeito e intervalo de confiança em lugar da NHST, sempre que tal seja possível (e.g. [196]). Apesar das várias recomendações, a utilização do tamanho do efeito tem ficado aquém do pretendido [148, 274].

No caso de testes não paramétricos é frequente os investigadores não apresentarem a estimativa da magnitude do efeito, ou então utilizarem uma versão paramétrica da estimativa [196]. O número de medidas do tamanho do efeito para testes não-paramétricos é limitado [108], o que poderá ajudar a justificar essa prática. Essa utilização não será a mais adequada sempre que essas medidas sejam afectadas por afastamento dos dados à distribuição normal e heterogenidade das variâncias [196]. Aliás, essas são razões para a utilização de métodos não paramétricos. O tamanho do efeito está associado ao conceito de potência do teste estatístico.

Os dados recolhidos serão inseridos numa folha de cálculo onde serão pré-processados, por exemplo, para análise de dados em falta. Para a análise estatística serão usadas funcionalidades da folha de cálculo Excel e o ambiente R que é simultaneamente uma linguagem e um ambiente para cálculo estatístico. No ambiente R, as funcionalidades são apresentadas sob a forma de funções, com possíveis parâmetros e valores de retorno. Por

sua vez, as funções estão agrupadas em pacotes (*packages*) e estes, poderão estar organizados em bibliotecas (*library*). Os pacotes são disponibilizados no página do software R[9]. O software R está disponível para Linux, Mac OSX e Windows.

As medidas da legibilidade usadas para medir a legibilidade diferem quanto ao tipo de escala. A medida Escala é considerada ordinal, enquanto que a medida obtida a partir do teste Cloze é uma percentagem (razão entre o números de casos preenchidos correctamente e o total de casos a preencher). Uma percentagem é considerada uma medida que pode ser usada como sendo do tipo rácio [307], logo não coloca restrições quanto ao tipo de teste estatístico. Esta última também se aplicará para medida Tempo de Leitura.

O teste das hipóteses, no caso da medida ordinal, será feito com um método não paramétrico próprio para planos com blocagem e com igual número de repetições por situação tratamento-bloco. Uma estatística de teste que se adequa a este tipo de plano é a estatística de Mack-Skillings [123].

Considera-se que cada tratamento possui c observações, que existem k tratamentos e n blocos. N será o número total de participantes, em que $N = nck$:

$$MS = \left[\frac{12}{k(n+N)}\right]\left[\sum_{j=1}^{k} S_j^2\right] - 3(n+N) \tag{5.1}$$

S_j é a soma das médias de cada célula de cada bloco das gradações das c observações do tratamento j e é dada pela expressão seguinte:

$$S_j = \sum_{i=1}^{n}\left[\sum_{q=1}^{c}\frac{r_{ijq}}{c}\right], \qquad \text{para j=1,..., k.} \tag{5.2}$$

A hipótese nula para o teste estatístico não-paramétrico, neste caso é, o de não existirem quaisquer diferenças nos efeitos dos k tratamentos ($\tau_1 ... \tau_k$). A hipótese alternativa é a de existirem pelo menos dois tratamentos cujos efeitos não são iguais. No caso presente, o valor de k é substituído por 2:

$$H_0 = [\tau_1 = ... = \tau_k] \text{ portanto, } [\tau_1 = \tau_2]$$

$$H_a = [\tau_j \neq \tau_{j'}] \text{ para algum } j \neq j' \text{ portanto, } [\tau_1 \neq \tau_2]$$

Para um dado nível de significância α_o a decisão será:

[9] `https://cran.r-project.org/`

Rejeitar H_0 se $MS >= MS_\alpha$, senão, não rejeitar.

A constante MS_α é escolhida para um dado α.

Para a estatística MS foi utilizado o pacote $NSM3$ disponível no sítio da linguagem R. Deste pacote foram usadas duas funções, uma para cálculo do valor crítico MS_α e outra para cálculo do valor MS e respectivo valor-p.

A função $cMackSkil(\alpha, k, n, c)$ determina o valor crítico para um α e um plano específico. A título de exemplo, para $k = 4$, $n = 3$, $c = 3$ e um $\alpha = 0.05$ ter-se-ia $cMackSkil(0.05, 4, 3, 3) = MS_{0.05} = 7.479$.

A função $pMackSkil(dados)$ calcula MS para os dados em causa e determina o respectivo valor-p, isto é, a probabilidade de MS ser igual ou superior ao valor crítico. Por exemplo, para um conjunto fictício de dados na variável *dados*, seria $P0(MS \geq 12.93) = 0.0023$ dado por $pMackSkil(dados)$.

Na perspectiva científica, quanto mais pequeno for o nível de significância com que é detectado um efeito, melhor será, pois menor será a probabilidade de se cometer um erro Tipo I, isto é, de rejeitar H_0 quando na realidade não existe um efeito. Por outras palavras, está-se a ser mais exigente na detecção do efeito. Habitualmente, um estudo inicial usa um valor como 5% e posteriormente estudos para replicação de resultados anteriores usarão valores mais baixos [179]. O custo é um aumento da probabilidade de erro Tipo II, de não rejeitar H_0 quando existe um efeito.

Tal como referido na secção 3.3.3, no estudo em [31], apesar de o código sem encadeamento se ter revelado mais legível, a diferença não é estatisticamente significativa. Como nesse estudo o resultado não foi estatisticamente significativo, o estudo actual não pretende confirmar a existência do efeito, e será encarado como um estudo inicial. Assim, será adoptado um $\alpha = 0.05$. No caso dos comentários, no mesmo estudo foi observado um efeito positivo significativo ($\alpha = 0.01$) para a prática de "bons" comentários. As práticas de "maus" comentários e inexistência de comentários não tiveram efeitos significativos, mas a primeira foi superior à segunda. Pela revisão da literatura não existe consenso quanto ao efeito dos comentários. Dada esta variação e, para se manter em linha com o estudo do encadeamento, será também usado um $\alpha = 0.05$.

Os testes apresentados dizem-se globais, ou *omnibus*, no sentido de apenas indicarem se existe, ou não, pelo menos um grupo que difere dos restantes. Não indicam qual. No

entanto, no caso de se observarem diferenças significativas nos efeitos dos tratamento, importará perceber quais os grupos que diferem entre si. Nesse caso, será feita uma análise que se designa por *post-hoc*. Essa análise pode ser realizada através do teste *Tukey HSD (Honestly Significant Difference)*. Este é um método encontrado frequentemente na literatura, e suportado no ambiente R, mas existem outros que de acordo com os resultados poderão vir a ser considerados como alternativa. Contudo, também é possível definir previamente (de forma planeada) comparações entre tratamentos. Essas comparações designam-se por contrastes. Dada a sua maior complexidade, para este estudo optou-se pela análise *post-hoc*.

5.6.7 Instrumentação

Os instrumentos considerados necessários para o estudo são definidos durante o planeamento de forma a estarem disponíveis para a execução do estudo. Os instrumentos para um estudo experimental podem ser de três tipos [297]:

- Objectos - Os objectos manipulados/usados no estudo e podem ser a especificação ou documentos de código;
- Linhas de orientação - As linhas de orientação incluem, por exemplo, descrição do processo e listas de verificação (*checklist*).
- Instrumentos de medição - Os instrumentos de medição permitem a recolha dos dados.

Os objectos em causa neste estudo são os *snippets*. Para o estudo foi necessário desenvolver um documento que abrangesse os 3 tipos de instrumentos. O documento conterá as instruções necessárias aos participantes acerca de como procederem no estudo, o código a analisar e as questões a serem respondidas as quais correspondem à medição. O documento é o único instrumento necessário. O documento pode ser encontrado nos apêndices C e D. O documento também será designado por questionário.

O documento foi dividido em várias partes. A capa referia o anonimato do participante, a utilização dos resultados, o carácter opcional de participação e que virar a folha significava que pretendia participar. O aluno não poderia voltar atrás no decorrer

do estudo, mas essa indicação, pela sua importância seria transmitida verbalmente pela investigadora durante o estudo.

Como o estudo envolve a leitura e compreensão do código-fonte, foi incluída no instrumento, logo a seguir à capa, a orientação de que os alunos teriam de ler cuidadosamente um bloco de código de modo a ser possível explicar, por palavras suas, esse código aos seus colegas. Esta foi a forma de tentar garantir que os alunos leriam o texto com a preocupação de o compreender. Para a validade do estudo era importante que cada participante avançasse no documento apenas após ter lido e compreendido o código.

A anteceder a parte relativa ao estudo propriamente dito, foram colocadas algumas questões demográficas que os alunos teriam de responder. Habitualmente, é aconselhável colocar este tipo de questões no final de um questionário por poder afectar as respostas [34] mas como neste caso, não se considerou que pudesse influenciar as respostas, incluindo pelo facto de se tratarem de respostas anónimas, preferiu-se colocar as questões no início para que os respondentes não tivessem um período adicional de tempo em que pudessem ser tentados a proceder a alterações às respostas ao estudo.

As questões demográficas são descritas em seguida:

- A primeira pergunta é relativa ao género. Esta questão surgiu basicamente por precaução para o caso de essa informação poder vir a ter importância posteriormente. Foi uma precaução apenas, até porque o número de estudantes de engenharia informática do sexo feminino é habitualmente reduzido.
- Com a segunda questão visa-se saber se o aluno possui alguma dificuldade de leitura, como por exemplo, dislexia. Uma característica da dislexia é a dificuldade de leitura. Logo não seria possível afirmar se um texto é ou não legível se o participante que estivesse a ler esse texto sofresse desse tipo de dificuldade.
- A finalidade da terceira questão é saber qual o nível de competências e conhecimentos do participante. Esta questão só pôde ser finalizada após se conhecer o universo de participantes. Esse universo era constituído pelos alunos da unidade curricular de Paradigmas da Programação que tem como unidade curricular precedente Algoritmia e Programação (APROG). Se os alunos que responderam ao inquérito ainda não tivessem obtido aprovação a esta unidade curricular, seria mais um factor ex-

terno que poderia influenciar a legibilidade do texto a ler. Por isso, foi colocada uma questão para saber se os alunos tinham, ou não, obtido aprovação a essa disciplina.

Para medição do tempo despendido nas tarefas foram colocadas no documento campos para indicação pelo aluno da hora início e fim de cada tarefa. Estes campos foram colocados em todas as páginas do estudo propriamente dito.

A 3ª folha, a primeira sobre a prática, informa que a tarefa do participante consiste em ler com muita atenção um pedaço de código de modo a que seja capaz de o explicar. Teriam de registar o momento de início da leitura e o momento de fim.

Na folha seguinte foram colocadas algumas questões sobre o pedaço de código lido. Mais uma vez, existiam espaços para indicação das horas início e fim. Sobre o código, é pedido ao participante que explique por palavras suas os 3 passos principais da função estudada. Na mesma folha é questionado a legibilidade do código numa escala de 5 pontos.

A última folha sobre a prática volta a apresentar o código, mas agora com partes omissas devidamente assinaladas que o participante terá de preencher. Também aqui o estudante necessita de registar os tempos.

As folhas para a segunda prática são idênticas. A capa e as questões demográficas não são repetidas.

A escolha dos *snippets* foi orientada por dois requisitos principais. Em primeiro lugar, o código teria de ser desconhecido de todos os alunos. Como segundo requisito, os *snippets* a usar teriam de estar escritos na linguagem OO utilizada pelos alunos que iriam participar no estudo. Como os *snippets* usados no estudo original [31] eram supostamente desconhecidos dos alunos, o primeiro requisito não constituiu problema. Quanto à linguagem, esses *snippets* estão todos escritos em Java, que é a linguagem usada pelos alunos participantes. Caso não fosse a linguagem teria de se ter identificado outros *snippets*. Após se verificar no documento do estudo original a localização do código, estes foram descarregados para a realização do estudo.

Foram usados os cinco pedaços de código (*snippets*) usados em [31], mas de uma forma distinta desse estudo. Dos cinco, seleccionaram-se três pedaços para o estudo experimental acerca do encadeamento de métodos e os dois restantes para o estudo acerca dos comentários. Estes dois *snippets* são de maior dimensão que os restantes três o que

requereria à partida maior tempo de leitura. A divisão dos *snippets* reduz o potencial de generalização, mas permite potenciar a quantidade de observações por célula do plano. De qualquer forma, de [31] sabe-se que em cada um dos 2 grupos há *snippets* de maior e menor legibilidade o que favorecerá a generalização dos resultados. Os *snippets* encontram-se no apêndice D.

Os cinco pedaços de código são todos de software de código aberto (*open source*). Os apontadores para a localização de cada um dos *snippets* são dados na tabela 5.5. As três primeiras ligações correspondem aos três *snippets* usados para o estudo do encadeamento de métodos e as duas últimas ligações correspondem aos *snippets* usados para o estudo dos comentários dentro dos métodos.

Tabela 5.5: URL de cada *snippet*.

1	`http://www.web4j.com/web4j/javadoc/src-html/hirondelle/web4j/Controller.html#line.390`
2	`https://github.com/unicase-ls1/unicase/blob/master/core/org.unicase.dashboard/src/org/\&unicase/dashboard/impl/NotificationOperationImpl.java`
3	`https://github.com/unicase-ls1/unicase/blob/master/core/org.unicase.docExport/src/org/\&unicase/docExport/docWriter/ITextWriter.java`
4	`https://github.com/Raptor-Fics-Interface/Raptor/blob/master/raptor/src/raptor/chess/pgn/\&PgnUtils.java`
5	`http://www.docjar.com/html/api/org/eclipse/jface/dialogs/Dialog.java.html`

Metade dos pedaços de código terão de ter aplicadas as práticas a estudar enquanto os pedaços da outra metade não terão as práticas aplicadas. Para o encadeamento de métodos são então usados três *snippets*, cada um com duas possibilidades: com encadeamento (MC) e sem encadeamento (NMC). Para os comentários são usados dois *snippets* cada um com três possibilidades: "bons" comentários (GC), "maus" comentários (BC) e sem comentários (NC). Ter-se-ão 6 *snippets* para cada prática. O total de *snippets* definidos é então 12: 3 *snippets* vezes 2 níveis para o encadeamento, mais 2 *snippets* vezes 3 níveis para os comentários.

Para ambos os estudos, os comentários existentes nos cabeçalhos do código original foram todos devidamente traduzidos para Português. No caso do estudo dos comentários, também foram traduzidos os comentários existentes no meio das linhas de código. Com isto pretendeu-se evitar o efeito de diferentes níveis de conhecimento da língua inglesa entre os participantes.

5.6.8 Participantes

A população-alvo de um estudo deve ser definida previamente e de forma inequívoca [152]. Atendendo ao objectivo do estudo, os participantes deveriam ser alunos de uma disciplina de introdução à programação orientada aos objectos de um curso superior de informática, logo, de alunos que estivessem a ter a sua primeira disciplina de programação orientada a objectos (POO).

A selecção de participantes está associada à capacidade de generalização dos resultados do estudo. De modo a generalizar os resultados para a população desejada, idealmente a amostra seleccionada deverá ser representativa dessa população [90, 297]. Para que a amostra dos alunos seja representativa é necessário que a sua selecção seja feita aleatoriamente. Este processo designa-se por amostragem aleatória. No presente caso, corresponde a recolher uma amostra de entre todos os alunos de programação orientada a objectos de todos os cursos superiores de informática. Este último conjunto será então a população-alvo do estudo.

No entanto, dadas as dificuldades óbvias em realizar a amostragem a partir dessa população, enveredou-se por uma amostra de conveniência. A amostra de conveniência deveria ser composta pelos alunos das unidades curriculares a que se pudesse ter o acesso facilitado. A partir daí haveria que tentar obter a participação do maior número possível de alunos.

Por conveniência, facilidade de acesso, foram consideradas duas instituições de ensino, o ISEP (Instituto Superior de Engenharia do Porto) e a UTAD (Universidade de Trás-os-Montes e Alto-Douro), por serem as instituições da autora e do seu orientador, respectivamente. A possibilidade de escolha da UTAD viu-se gorada porque a unidade curricular de interesse apenas funciona no primeiro semestre do curso. Devido ao tempo que se percebeu ser necessário para preparar o estudo essa possibilidade ficou sem efeito.

No caso do ISEP existia mais do que uma disciplina de introdução à programação orientada a objectos e todas decorriam no segundo semestre. Uma das possibilidades foi a unidade curricular Programação (PROGR) que decorre em simultâneo nas licenciaturas em Engenharia da Computação e Instrumentação Médica (ECIM) e em Engenharia de Instrumentação e Metrologia (EIM). A outra possibilidade foi a unidade curricular Paradigmas da Programação (PPROG) da licenciatura em Engenharia Informática. Esta unidade curricular tem como objectivo ensinar o paradigma da programação OO. Esta unidade era a preferida porque nas outras unidades curriculares consideradas, o nível de conhecimento de programação dos alunos no segundo semestre ainda é bastante elementar e porque se tratam de unidades de cursos não específicos de informática como era pretendido. Isto explica o nível de conhecimentos mais baixo dos alunos destes cursos.

Dada a preferência por PPROG, foi contactado o regente da disciplina para saber da possibilidade de realizar o estudo na sua unidade curricular, o que o regente aceitou. A disciplina tinha 424 alunos inscritos nessa edição. O conjunto de estudantes da disciplina será o universo de estudantes de POO do estudo.

Para frequentar essa unidade curricular os alunos têm de ter obtido nota de frequência positiva a uma unidade de algoritmia e programação do 1º semestre. Isto é, esta precedente possui duas componentes de avaliação, nota de frequência e nota de exame. A precedência a PROGR é determinada apenas pela nota de frequência. Esta precedência em PPROG contribui para homogeneizar os participantes em termos de competências e conhecimentos e reduzir diferenças de capacidade para leitura de código. Uma população mais homógenea permite, como se referiu para a randomização, aumentar a validade interna do estudo.

5.7 Preparação

Antes de o estudo experimental ser executado foi necessário proceder aos preparativos para a execução. A preparação é a primeira fase de operacionalização do planeamento de um estudo experimental, a que se seguirá a execução propriamente dita [90]. Em [90, 297] são fornecidas orientações para esta fase, as quais ajudaram a orientar a preparação do estudo. O primeiro passo da preparação foi agendar a data de realização.

Para isso foi necessário realizar uma nova reunião com o regente da disciplina. Era

preferível que o estudo se realizasse o mais tarde possível no semestre para permitir que os alunos estivessem numa fase o mais adiantada possível de aprendizagem da POO com a linguagem Java e desse modo não prejudicar a realização do exercício de leitura de código. Por comum acordo, o estudo ficou agendado para a última aula teórica do semestre e seria realizado em todas as turmas do regime diurno, ou seja, nas aulas teóricas de 5 turmas. Essas aulas são de uma hora, todas no mesmo dia e em horários consecutivos. Desta forma, seria mais difícil o contacto entre os alunos das várias turmas. Os conhecimentos dos alunos do regime pós-laboral são em geral mais heterogéneos e, em princípio, mais dificilmente representativos da generalidade dos alunos de POO, por se tratar de alunos em geral mais velhos e já com actividade profissional.

É muito importante que os alunos envolvidos na execução do estudo estejam motivados para a sua realização [297]. Nesse sentido, os participantes necessitam saber o que se pretende e concordarem em participar. Na semana anterior ao estudo foi enviada uma mensagem de correio electrónico aos alunos que informava que na última aula teórica do semestre da disciplina iria ser realizado um estudo, para fins de investigação que envolveria a leitura de código-fonte. Também informava que todos os alunos estavam convidados, e dada a sua importância, agradecia-se a adesão de todos. Nessa mensagem também era referido que a participação dos alunos era opcional, que o anonimato das suas respostas seria garantido e que todos os dados seriam utilizados unicamente para efeitos científico-pedagógicos.

Esta fase também envolve preparar o local, o material para a execução do estudo experimental, e definir a distribuição dos alunos. Quanto à preparação do local, foi decidido colocar um relógio digital na sala de modo que cada aluno pudesse registar as horas início e fim e assim assegurar uniformidade na indicação das horas. Para além disso, as salas não requeriam nenhuma preparação especial. Relativamente à distribuição dos participantes neste estudo esse problema não se colocou. Como o estudo estava previsto para decorrer nas aulas da disciplina, não houve necessidade de informar previamente os alunos acerca das salas para que se deveriam dirigir. Sobre o material necessário, tal como, papel e lápis, apenas foi preparado um número muito restrito de canetas para o caso de algum aluno não vir preparado com a sua, ou desta falhar. O questionário concebido antes for produzido e depois policopiado num número conforme com o de alunos da disciplina.

Também foi nesta fase que se procedeu à randomização. Como referido na secção 5.6.5 havia sido decidido usar uma forma automática de randomização. Existem vários programas/sítios para o efeito, mas foi usado o programa indicado em [141] devidamente adaptado para o caso em que existem 2 factores. Trata-se de uma folha de cálculo que permitiu produzir duas listas de pares de números (bloco, tratamento), com cada um dos números de cada par dentro de um intervalo definido. Uma lista para o estudo de cada prática. Após as listas terem sido geradas foram impressas. Os questionários foram depois organizados de acordo com as listas para estarem prontos a serem entregues aos estudantes no momento em que estes entrassem na sala.

5.8 Execução

5.8.1 Recolha dos Dados

Esta secção trata da execução propriamente dita do estudo. O estudo experimental planeado e preparado é agora executado por forma a produzir os resultados que serão analisados posteriormente [297, 90]. Nesta secção são apresentados os passos executados.

Para a execução do estudo esteve presente a investigadora de forma a supervisionar todo o processo e poder responder com prontidão a qualquer questão ou problema que surgisse. A presença do regente da disciplina também era importante por ser responsável pela disciplina e pela sala de aula. Tanto o docente como a investigadora estiveram presentes durante toda a duração do estudo em cada uma das aulas.

Os questionários já preparados e ordenados foram colocados na sala previamente prontos para serem distribuídos. A distribuição dos questionários decorreu da seguinte forma, cada aluno ao entrar na sala pegava no inquérito respectivo indicado pela investigadora e sentava-se onde pretendesse. Dos 424 alunos inscritos na disciplina compareceram 106.

No início de cada sessão, o regente da disciplina apresentou a investigadora e passou-lhe a palavra. Logo de seguida, e antes de os alunos começarem a responder, a investigadora forneceu aos alunos algumas directrizes acerca de como deveriam proceder durante o estudo. Em particular, o objectivo, a garantia de anonimato e a necessidade de seguirem escrupulosamente todas as instruções constantes do questionário, de que se destacou a importância de só avançarem no questionário depois de terem lido e compreendido o código.

Também foi realçada a importância de os alunos responderem com a máxima seriedade de modo a não colocarem em causa a validade das respostas. Finalmente, os alunos foram alertados para o facto de não poderem tomar notas ou copiar os pedaços de código (manual ou eletronicamente) pois nesse caso as respostas seriam inúteis para o estudo e as suas respostas teriam de ser removidas da análise. Foi dito que sempre que uma folha fosse virada não era permitido voltar atrás.

O questionário desenvolvido pedia para que fosse registado o tempo início e fim de cada tarefa. Como definido, para anotação dos tempos pelos alunos foi projetado no quadro da sala um relógio que todos os alunos poderiam ver. A investigadora informou os participantes que deveriam de anotar os tempos, de início e fim de cada tarefa e que para isso teriam de usar o relógio projectado no quadro. Os alunos foram informados que poderiam demorar o tempo que considerassem necessário e que assim que terminassem o preenchimento poderiam abandonar a sala após entregarem o seu questionário.

O documento do questionário tinha uma página de rosto onde era dito ao aluno que o questionário era opcional e anónimo e que assim que virasse a página estaria a aceitar as condições e que também aceitava realizar o inquérito. Nenhum aluno abandonou a sala. Também era informado que os dados seriam confidenciais e que os resultados iriam ser usados unicamente para fins de investigação. Foi também dito aos alunos que este estudo não seria para avaliar o seu desempenho na avaliação do código, até porque o preenchimento do questionário era anónimo.

Na folha seguinte do documento, ainda antes de começarem o estudo, os alunos responderam às questões demográficas. No final deste ponto os alunos teriam de virar novamente a página. Nas páginas seguintes começava o estudo propriamente dito, como explicado na secção 5.6.7.

Assim que cada aluno terminou o estudo, levantou-se e abandonou a sala após entregar o questionário. Todas as 4 sessões decorreram sem incidentes. Os alunos não colocaram dúvidas acerca do preenchimento do questionário.

5.8.2 Resultados

Após a realização do estudo procedeu-se à compilação das respostas para suporte informático de modo a se poder fazer a análise estatística dos resultados e a sua posterior

Tabela 5.6: Resultados dos questionários.

Descrição	**Total**
Nº de alunos	106
Nº de respostas	106
Género Masculino	98
Género Feminino	7
Dificuldade leitura	1
Remoção dificuldade leitura	1
Com aprovação APROG	90
Sem aprovação APROG	14
Nº total de inválidas	38
Sem indicação do tempo	32
Sem resposta escala	5
Sem dados demográficos	1
Respostas válidas primeiro estudo	9
Respostas válidas segundo estudo	11

interpretação. A tabela 5.6 apresenta os valores dessa compilação. Os resultados em si encontram-se nos apêndices E e F.

5.9 Análise e Interpretação dos Resultados

De modo a se obterem conclusões cientificamente válidas de um estudo, é necessário realizar a análise estatística dos dados recolhidos e a sua posterior interpretação [297]. A análise estatística envolve três passos: estatística descritiva, redução do conjunto de dados e teste da hipótese [297]. A redução do conjunto de dados consiste na validação dos dados, por exemplo, pela detecção e eliminação de valores estranhos ("outliers").

Como referido antes, na secção 5.6.6, não será usado o mesmo teste estatístico para as medidas Escala, Cloze e Tempo de Leitura, em virtude de apresentarem tipos de escala distintos. Os dados obtidos com a segunda e a terceira medida serão avaliados quanto aos pressupostos para utilização do modelo ANOVA e caso o resultado seja positivo, será dada preferência a esse modelo. Caso contrário será utilizado um teste não paramétrico. A medida Escala por ser ordinal será avaliada com um teste não-paramétrico.

Um plano com blocos e completo, isto é com uma observação por célula, é sensível a

dados em falta e possui um requisito adicional, que é o da aditividade. No entanto, esses pressupostos não se aplicam quando o plano é com replicações.

5.9.1 Estudo experimental 1

A tabela 5.7 mostra a quantidade de respostas por célula obtidas no estudo. Cada célula da tabela é uma situação. Uma situação é uma combinação bloco[10]/nível[11]. A totalidade dos resultados do estudo podem ser encontrados no apêndice E. Como referido, os resultados foram compilados numa folha de cálculo Excel.

Tabela 5.7: Observações por situação.

Bloco/Tratamento	NMC	MC
S1	16	19
S4	19	19
S5	13	20

Algumas das respostas foram consideradas inválidas, o que fez com que a situação com menor número de observações ficasse com 9 observações.

Para se poder realizar uma análise estatística para planos balanceados é necessário que todas as células possuam o mesmo número de observações. Uma possibilidade consiste em eliminar casos [269]. Assim, eliminaram-se aleatoriamente das células os casos com um número de observações em excesso. Após a eliminação, cada célula ficou então com 9 observações.

Seguidamente, os dados foram agrupados segundo 3 colunas, respectivamente a coluna com indicação do bloco, a do nível do tratamento e a terceira com o valor observado, como mostrado na tabela 5.8.

Posteriormente, estes dados foram gravados num ficheiro de texto, cujo nome seguiu o seguinte formato: *Experimento-Dados-Medida.txt* em que, *Experimento* pode ser *E1* ou *E2* consoante se se tratar do estudo experimental 1 ou do estudo experimental 2, respectivamente. E *Medida* pode ser *escala*, *cloze* ou *tempo*. Tem-se, portanto, 6 ficheiros de texto.

[10] *snippet*

[11] de tratamento (MC ou NMC)

Tabela 5.8: Extracto dos dados na folha de cálculo.

Bloco	Tratamento	Resultado
1	mc	3
...	...	...

Escala Para este estudo, os dados foram gravados com o nome "E1-Dados-escala.txt" que foi depois importado para o ambiente R no formato $data.frame$. Para a leitura dos dados do ficheiro para o formato $dataframe$[12] de R executou-se o comando:

$$escala <- read.table(\text{“}caminho/E1 - Dados - escala.txt\text{”}, header = TRUE)$$

em que *caminho* é o directório onde se encontra o ficheiro de texto com os dados e *escala* a variável que guardará os dados. Os dados ficaram organizados segundo 3 colunas, cada uma com o nome indicado no ficheiro lido. Isso pode ser confirmado com o comando,

$$names(escala)$$

Para verificar o número de observações de cada situação, usa-se o seguinte comando:

$$with(escala, table(BLOCO, Tratamento))$$

Cada coluna da variável *escala* pode ser acedida usando-se o carácter '$'. Por exemplo, para a coluna *Tratamento*, escreve-se *escala$Tratamento*. Usando-se o comando *attach(escala)* o acesso às colunas é directo, ou seja, passa a ser suficiente indicar apenas a coluna $Tratamento$ em qualquer comando do R.

Em R, foram também desenvolvidas algumas funções para auxiliar o processo. Para a leitura dos dados e subsequentemente análise foi desenvolvida a função ler. Esta função pode ser invocada da seguinte forma:

$$e <- ler(ficheiro)$$

em que ficheiro pode ser *escala*, *cloze* ou *tempo*.

Nas tabelas 5.9 e 5.10 são apresentados os intervalos de valores de cada variável, as suas medianas e médias. A mediana para qualquer um dos factores foi sempre 2 (o valor central da escala é 3).

[12] tabela em R

Os intervalos, as medianas e as médias por tratamento, e as medianas e médias por célula podem ser obtidos, respectivamente, através das instruções que se seguem. O termo $factor$ deverá ser substituido por cada um dos 2 factores e *op* por *median* e *mean*, respectivamente

$$a < -aggregate(Resultado \sim factor, data = e, range)$$

em que $factor$ será *Tratamento* ou *BLOCO*.

$$tapply(Resultado, factor, op)$$

$$a < -aggregate(Resultado \sim Tratamento + BLOCO, data = e, op)$$

As instruções para formatar em tabela de médias e medianas por célula foram compiladas numa função de nome *tabela*.

Tabela 5.9: Estatísticas descritivas da variável Encadeamento para a Escala.

Tratamento	Mínimo	Máximo	Mediana	Média
MC	1	4	2	2.444
NMC	1	5	2	2.481

Tabela 5.10: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S1	1	5	2	2.333
S4	1	5	2	2.389
S5	2	4	2	2.667

As medianas e médias por célula encontram-se nas tabelas 5.11a e 5.11b.

Como previsto utilizou-se a estatística Mack-Skilling para testar a hipótese nula. Para o estudo tem-se:

k=2 tratamentos

n=3 blocos

c=9 replicações por situação (célula).

Com esses dados obteve-se o valor crítico $MS_\alpha = 4.05523$.

Tabela 5.11: Médias e medianas por célula.

	MC	NMC
S1	2.111	2.556
S4	2.333	2.444
S5	2.889	2.444

(a) Médias.

	MC	NMC
S1	2	2
S4	2	2
S5	3	2

(b) Medianas.

$$NSM3 :: cMackSkil(0.0500, k, n, c)$$

Com o valor obtido para MS_α rejeita-se H_0 se $MS \geq 4.05523$.

Como a função $pMackSkil$ não aceita o formato $data.frame$ os dados devem ser previamente convertidos para o formato *array* do R.

$$NSM3 :: pMackSkil(z), \text{ em que } z \text{ são os dados no formato } array$$

Para os dados existentes, obteve-se:

$$MS = 0.0786,\ p - value = 0.777$$

Como $MS = 0.0786 < MS_\alpha = 4.05523$, a hipótese nula não pode ser rejeitada, ou seja, a diferença observada de maior facilidade de leitura sem encadeamento não é estatisticamente significativa.

Como forma de verificação, também se aplicou o método *two-way* ANOVA, novamente em R. Em R, o teste ANOVA simples, ou $multi - way$ pode ser realizado pelo menos de duas formas. Ou, via a função *lm* seguida da função *anova*, ou então, usando a função *aov* seguida da função *summary*. De certa forma, ambas as abordagens envolvem a criação do modelo e a sua análise. No estudo foi possível confirmar que produziam os mesmos resultados. Os resultados apresentados na tabela 5.12 são os da primeira forma.

O valor-p resultante para o tratamento (0.8926) foi superior a 5% e, consequentemente, a hipótese nula não pode ser rejeitada e assim aceita-se que não existe diferença significativa entre os dois níveis da variável independente.

Tabela 5.12: Resultado ANOVA para a Escala.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	1	0.019	0.01852	0.0184	0.8926
BLOCO	2	1.148	0.57407	0.5711	0.5685
Residuals	50	50.259	1.00519		

Cloze Os dados na folha de cálculo foram gravados num ficheiro de texto de nome "E1-Dados-cloze.txt" cujo conteúdo foi depois importado para R como *data.frame*.

Os intervalos de variação, as medianas e médias da medida Cloze foram obtidos da mesma forma que para a medida Escala. Nas tabelas 5.13 e 5.14 são apresentados os intervalos de valores de cada variável, as medianas e as médias.

Tabela 5.13: Estatísticas descritivas da variável Encadeamento para o Cloze.

Tratamento	Mínimo	Máximo	Mediana	Média
MC	0	1	0.50	0.722
NMC	0	1	0.75	0.719

Tabela 5.14: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S1	0.0	1.0	0.500	0.565
S4	0.50	1.0	1.000	0.889
S5	0.25	1.0	0.625	0.708

Os valores por célula encontram-se em 5.15a e 5.15b.

Para testar a normalidade dos dados usou-se o teste de normalidade Lilliefors, recomendado em [123]. Como referido anteriormente, idealmente a normalidade deve ser analisada sobre os residuais. Para obtenção dos residuais foi necessário definir em primeiro lugar o modelo linear da ANOVA. Para o cálculo foi usada a função $lillie.test(erros)$ que se encontra na bilioteca *nortest*. O cálculo foi efectuado por amostra de cada tratamento, ou seja, por nível de factor que são 2. Não se fará para o bloco visto o objectivo ser analisar

Tabela 5.15: Médias e medianas por célula.

	MC	NMC
S1	0.611	0.519
S4	0.833	0.944
S5	0.722	0.694

(a) Médias.

	MC	NMC
S1	0.50	0.333
S4	1.00	1.00
S5	0.50	0.75

(b) Medianas.

os efeitos dos tratamentos (se não, seriam 5 grupos). Utilizar-se-á a mesma abordagem para o tempo de leitura e em ambos os estudos.

Para os 2 níveis (grupos) do tratamento obtiveram-se resultados distintos, usando o comando [66]: $with(c\$e, tapply(erros, Tratamento, lillie.test))$

$Para\, mc,\ D = 0.13533,\ p - value = 0.2312$

$Para\, nmc,\ D = 0.20913,\ p - value = 0.003748$

Uma das amostras apresenta um valor-p inferior a 0.05, mesmo inferior a um α de 1% o que fez colocar reservas à utilização de ANOVA. Para a globalidade dos residuais dos dados tem-se $D = 0.30673,\ p - value = 5.701e - 14$

O valor crítico da estatística Mack-Skillings para a medida Cloze é naturalmente o mesmo obtido antes. Já a estatística obtida foi então:

$$MS = 0.01039636,\ p - value = 0.9204$$

o qual não excede o valor crítico $MS_\alpha = 4.05523$ para um $\alpha = 5\%$. A hipótese nula não foi rejeitada.

Testou-se também com ANOVA e o resultado é mostrado na tabela 5.16.

Tabela 5.16: Resultado ANOVA para o Cloze.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	1	0.0001	0.00013	0.0017	0.967241
BLOCO	2	0.9493	0.4R7467	6.2870	0.003668
Residuals	50	3.7750	0.07550		

O valor-p para o tratamento é superior a 0.05 e como tal a hipótese nula não é rejeitada. O factor de blocagem teve um efeito estatisticamente significativo para um α igual a 10%.

Tempo de Leitura A correlação entre a legibilidade e a rapidez de leitura é medida pelo tempo consumido na leitura do código-fonte. O intervalo de tempo é a diferença entre a hora início e fim da leitura indicados pelo aluno. Quanto maior for o valor, significa que o aluno demorou mais tempo a ler o código-fonte. Os valores estão expressos em segundos. Os dados na folha de cálculo foram gravados num ficheiro de texto de nome "E1-Dados-tempo.txt" cujo conteúdo foi novamente importado para o ambiente R.

Tabela 5.17: Estatísticas descritivas da variável Encadeamento para o Tempo de Leitura.

Tratamento	Mínimo	Máximo	Mediana	Média
MC	62	1136	212	315.074
NMC	30	640	241	295.852

Tabela 5.18: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S1	30	640	183.0	273.222
S4	89	1136	254.5	361.500
S5	80	585	238.5	281.667

O processo foi idêntico ao teste Cloze. Em primeiro lugar, determinaram-se as estatísticas descritivas: o intervalo de variação, a mediana e a média. Nas tabelas 5.17 e 5.18 são apresentados os intervalos de valores de cada variável, as medianas e as médias.

Da mesma forma que com as medidas anteriores, Escala e Cloze, determinaram-se os valores por célula. Estes encontram-se nas tabelas 5.19a e 5.19b.

Para o Tempo de Leitura, verifica-se que, para o *snippet* S1, em média, o tempo foi menor e a mediana também foi inferior. Isto para os métodos com encadeamento.

Em seguida, analisou-se a normalidade dos dados, conforme se fez para a medida Cloze. O teste de Lillifors produziu o seguinte resultado:

$$D = 0.16969,\ p - value = 0.0005038$$

Tabela 5.19: Médias e medianas por célula.

	MC	NMC
S1	211.444	335.000
S4	463.444	259.556
S5	270.333	293.000

(a) Médias.

	MC	NMC
S1	178	331
S4	237	268
S5	300	236

(b) Medianas.

E por amostra:

mc : 0.2117048, $p - value = 0.003116684$

nmc : 0.1605397, $p - value = 0.07207767$

Como o valor-p é muito baixo, é mesmo inferior a um α de 1%, a hipótese nula de normalidade foi rejeitada, o que levou à rejeição da normalidade e torna relevante usar o teste não-paramétrico. Será complementado com ANOVA.

Aplicando o teste Mack-Skillings, obteve-se o seguinte resultado:

$$MS = 0.1098,\ p - value = 0.7573$$

Como se pode verificar, o valor não excede o valor crítico $MS_\alpha = 4.05523$ e por isso a hipótese nula não pode ser rejeitada. O $valor - p = 0.7573$, também excede o nível de significância de 0.05.

Mais uma vez, também se procedeu ao teste com ANOVA. O resultado é mostrado na tabela 5.20.

Tabela 5.20: Resultado ANOVA para o Tempo de Leitura.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	1	4988	4988	0.099	0.755
BLOCO	2	85426	42713	0.845	0.435
Residuals	50	2526207	50524		

O valor-p para o tratamento é superior a 0.05 e como tal confirma-se que a hipótese nula não é rejeitada.

Interpretação O julgamento da legibilidade pelos alunos, como é o caso da *Escala*, é uma medida subjectiva. No entanto, num estudo [33] verificou-se que existia uma correlação forte entre essa medida e a análise da compreensão do código através de questões acerca do código. A medida *Cloze* está associada à compreensão. A medida do tempo, refere-se ao tempo de leitura do código antes de responder a qualquer questão sobre o mesmo.

Tanto para uma medida como para a outra os resultados mostraram não ser possível rejeitar a hipótese nula, ou seja, não foi possível rejeitar a hipótese de igualdade dos efeitos dos 2 níveis de encadeamento. A não rejeição da hipótese nula, não implica assumir que a hipótese nula é verdadeira [232], e que se considere indiferente a utilização ou não do encadeamento de métodos. Para ambas as medidas existe uma diferença muito pequena entre as duas médias, para a *Escala* favorável à ausência de encadeamento e para *Cloze* o contrário. Não sendo estatisticamente significativa pode ser interpretada como devido ao acaso, mas também é possível que essa diferença exista efectivamente mas por ser pequena na população, torna improvável encontrar uma diferença significativa nas amostras ou pode, eventualmente, depender de algum outro factor.

Quanto à medida *Escala*, globalmente, cf. tabelas 5.9 e 5.10, as medianas quer por tratamento quer por *snippet* são sempre iguais a 2. Portanto, não é possível distinguir o efeito nem os níveis de tratamento, nem do factor de blocagem na legibilidade com base nas medianas.

Relativamente à média, é ligeiramente superior para os métodos sem encadeamento. Em termos dos *snippets*, apenas o *snippet* $S5$ se destaca com uma média superior. Este resultado fará supor que o *snippet* $S5$ seja o mais legível. Estranhamente, é o único *snippet* que não recebeu qualquer valor máximo, 5.

Faz-se agora a análise por *snippet* e tratamento. Aqui apenas a mediana de $S5$ com encadeamento apresenta um valor distinto, 3, o mais elevado dos seis. É este valor que permite explicar a maior legibilidade do *snippet*. Constata-se também que, para os 2 *snippets* menos legíveis, $S1$ e $S4$, a não utilização de encadeamento tornou o código mais legível. Ocorreu o inverso com o *snippet* mais legível. Aliás, os valores fazem supor que a legibilidade de código com encadeamento varia inversamente à do mesmo código sem encadeamento. Uma hipótese possível é quanto menos legível o código, mais importante

é evitar encadeamento.

Uma explicação possível para não se ter encontrado uma diferença significativa poderá ter a ver com o valor baixo da mediana e da média da legibilidade dos *snippets*. Estes valores são inferiores ao valor central da escala, o que pode indiciar que o grau de dificuldade dos *snippets* escolhidos poderá ter sido excessivo para o nível de competência em programação orientada a objectos (POO) dos alunos envolvidos no estudo, ou seja, para alunos de uma disciplina de introdução ao paradigma orientado a objectos. No caso do *snippet* $S5$, o intervalo é entre 2 e 4, e é o que apresenta a média mais elevada. É importante referir que este *snippet* era o mais simples dos três. Se assim for, será de concluir que a utilização ou não de encadeamento depende do grau de dificuldade do código, em que o código mais complexo deve evitar utilizar encadeamento.

No caso da medida *Cloze* tem-se uma mediana para não encadeamento (NMC) superior à de encadeamento (MC), mas o mesmo não acontece com a média. O *snippet* $S4$ revelou-se o mais compreensível, melhor dizendo mais bem compreendido, com média e mediana mais elevados, o que não está de acordo com os resultados para a *Escala*. Na relação *snippet* e tratamento, $S4$ e $S5$ viram a compreensão piorar sem encadeamento, contrariamente a $S1$ em que houve aumento. A compreensão com encadeamento aumentou de $S1$ para $S4$ e diminui deste para $S5$, isto é, $S1 < S5 < S4$. Sem encadeamento, aumentou de $S1$ para $S4$ e depois diminui deste para $S5$, isto é, $S1 < S5 < S4$. O resultado com $S4$ foi claramente superior para a medida *Cloze*.

Relativamente à medida *Tempo de Leitura*, globalmente, a média do tempo de leitura foi inferior para o código sem encadeamento, mas sem ser uma diferença estatisticamente significativa. O *snippet* $S4$ foi o que requereu em média mais tempo de leitura. Este valor é inverso ao obtido para a medida *Cloze*, em que $S4$ foi considerado o mais compreensível.

Analizando por *snippet* e tratamento, $S1$ e $S5$ tiveram mais tempo de leitura sem encadeamento do que com encadeamento, passando-se o inverso com $S4$. O *snippet* $S4$ requereu bastante mais tempo de leitura com encadeamento. Uma explicação possível, seria os alunos terem consumido mais tempo na leitura de $S4$ e, por isso, terem-no compreendido melhor. Se assim foi, então a falta de compreensão de $S5$ poderá não significar que este é menos compreensível, mas sim que por ser mais simples, ao ser lido mais rapidamente, acabou por fazer com que os alunos revelassem menor compreensão sobre

ele.

Seguem-se algumas explicações possíveis finais e respectivas hipóteses explicativas. $S1$ sem encadeamento foi considerado o mais legível, mas requereu o maior tempo de leitura e a sua compreensão foi a mais baixa. Uma explicação é que não seria assim tão legível e isso pode dever-se ao domínio de aplicação (Http) que era desconhecido dos alunos. Portanto, a ausência de encadeamento, em nada contribuirá para melhorar a leitura e compreensão quando o domínio de aplicação é desconhecido. $S4$ sem encadeamento foi considerado pouco legível e menos do que com encadeamento. Requereu pouco tempo de leitura, mas foi o mais bem compreendido, ainda mais do que com encadeamento. Uma hipótese é que apesar de considerado pouco legível, sem encadeamento seria na realidade simples, legível, de forma que requereu pouco tempo de leitura para ser bastante bem compreendido. Pela explicação, a eliminação do encadeamento provocou uma melhoria da compreensibilidade do código. Aqui pode justificar-se falar em compreensibilidade (e não em compreensão) em virtude de o tempo de leitura ter sido o menor. $S5$ sem encadeamento foi considerado pouco legível e menos do que com encadeamento. Requereu algum tempo de leitura, mas a sua compreensão degradou-se ligeiramente. O resultado faz supor que código mais legível não beneficia com ausência de encadeamento.

Em conclusão, a legibilidade dependerá do conhecimento do domínio de aplicação. Uma segunda conclusão, considerando verdadeiras as explicações anteriores, é que a ausência de encadeamento pode trazer algum benefício para a compreensão do código, desde que o código não seja já de si "demasiado óbvio".

A diferença de valores entre as medidas *Escala* e *Cloze* ajuda a confirmar que medem conceitos distintos, a primeira a legibilidade e a segunda a compreensão. No caso do teste *Cloze* se tivesse sido colocado um limite temporal, poderia afirmar-se que media a compreensibilidade do código. Como não foi imposto esse limite, os participantes tiveram tempo para compreender e, portanto, apenas se pode afirmar que o que a medida mediu foi a compreensão do código por parte dos participantes. Deste modo, quando se fala aqui em código compreensível, na realidade está-se a falar da compreensão dos participantes.

No teste com ANOVA, o factor de blocagem com um valor estatisticamente significativo constitui evidência de que esse factor contribuiu para melhorar a comparação dos tratamentos [52]. Na medida *Escala* não existe uma diferença significativa entre os blo-

cos, mas para a medida *Cloze* existe uma diferença significativa entre os blocos para um α igual a 1%. Esta diferença dá suporte à existência dos blocos. De qualquer maneira, com a medida *Escala* e com a medida *Tempo de Leitura*, o valor-p entre blocos foi inferior ao valor-p entre tratamentos, o que indicia alguma variação entre os blocos.

5.9.2 Estudo experimental 2

A tabela 5.21 mostra a quantidade de respostas por célula obtidas no estudo. Cada célula da tabela é uma situação. A totalidade dos resultados do estudo podem ser encontrados no apêndice F. Como referido, os resultados foram compilados numa folha de cálculo Excel.

Tabela 5.21: Observações por situação.

Bloco/Tratamento	BC	GC	NC
S2	15	18	20
S3	17	15	20

Com este estudo também algumas das respostas foram consideradas inválidas, o que fez com que a situação com menor número de observações ficasse com 11 observações. Da mesma forma que com o primeiro estudo, eliminaram-se aleatoriamente as observações em excesso, resultando 11 em cada.

Resumindo, para o estudo do segundo estudo tem-se:

k=3 tratamentos

n=2 blocos

c=11 replicações por situação.

Tabela 5.22: Extracto dos dados na folha de cálculo.

BLOCO	Tratamento	Resultado
2	bc	1
...	...	...

Escala Mais uma vez, os dados foram gravados num ficheiro de texto, cujo nome seguiu as regras acima indicadas, no estudo experimental 1. Em seguida, o ficheiro "E2-dados-escala.txt" foi importado para o ambiente R.

Nas tabelas 5.23 e 5.24 são apresentadas as estatísticas descritivas, os intervalos de valores de cada variável, as suas medianas e médias.

Tabela 5.23: Estatísticas descritivas da variável Comentário para a Escala.

Tratamento	Mínimo	Máximo	Mediana	Média
BC	1	5	3.0	2.777
GC	1	5	2.0	2.545
NC	1	4	2.5	2.409

Surpreendentemente, a mediana e a média para os "maus" comentários têm valores mais elevados, sendo a mediana de 3 e a média de 2.8, respectivamente. Os *snippets* com "bons" comentários são os que têm a mediana mais baixa.

Tabela 5.24: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S2	1	4	2	2.303
S3	1	5	3	2.848

Os resultados por *snippet* apresentam uma mediana e uma média superiores para o *snippet* S3. Este resultado não é assim tão surpreendente uma vez que o *snippet* S2 é mais difícil que o *snippet* S3.

As medianas e médias por célula encontram-se nas tabelas 5.25a e 5.25b. Aqui, para qualquer das situações, o *snippet* S2 apresenta valores inferiores quer na média quer na mediana.

Como previsto, utilizou-se a estatística Mack-Skilling para testar a hipótese nula. Para o estudo tem-se:

O teste Mack-Skillings produziu o seguinte resultado:

$$MS = 0.744,\ p - value = 0.6752$$

Tabela 5.25: Médias e medianas por célula.

	BC	GC	NC
S2	2.364	2.182	2.364
S3	3.182	2.909	2.455

(a) Médias.

	BC	GC	NC
S2	2	2	2
S3	3	3	3

(b) Medianas.

Esse valor é inferior ao valor crítico $MS_\alpha = 5.80554$ para um $\alpha = 0.05$. A hipótese nula não pode então ser rejeitada.

Mais uma vez, testou-se também com ANOVA e o resultado é mostrado na tabela 5.26.

Tabela 5.26: Resultado ANOVA para a Escala.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	2	1.485	0.7424	0.6243	0.53895
BLOCO	1	4.909	4.9091	4.1282	0.04646 *
Residuals	62	73.727	1.1891		

O valor-p para o tratamento é superior a 0.05 e como tal a hipótese nula não pode ser rejeitada. O factor de blocagem teve um efeito estatisticamente significativo para um α igual a 5%.

Cloze Nas tabelas 5.27 e 5.28 são apresentados os intervalos de valores de cada variável, as suas medianas e médias.

Tabela 5.27: Estatísticas descritivas da variável Comentário para o Cloze.

Tratamento	Mínimo	Máximo	Mediana	Média
BC	0	1	0.500	0.447
GC	0	1	0.550	0.501
NC	0	1	0.775	0.658

As média e mediana para o teste Cloze têm como resultado um valor mais elevado para

o código-fonte sem comentários. O código com "maus" comentários é o que apresenta os valores mais baixos para estas duas estatísticas.

Tabela 5.28: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S2	0	1	0.250	0.417
S3	0	1	0.667	0.654

As medianas e médias por célula encontram-se nas tabelas 5.29a e 5.29b. No teste Cloze, da mesma forma que no teste Escala, os resultados por célula são inferiores para o *snippet* S2, quer em termos da média quer em termos da mediana.

Tabela 5.29: Médias e medianas por célula.

	BC	GC	NC
S2	0.410	0.341	0.500
S3	0.485	0.661	0.815

(a) Médias.

	BC	GC	NC
S2	0.500	0.250	0.500
S3	0.500	0.667	0.833

(b) Medianas.

Em seguida, analisou-se a normalidade dos dados com o teste de Lilliefors como recomendado em [123] O cálculo foi efectuado por amostra de cada tratamento, ou seja, por nível de factor que são 2. Não se fará para o bloco visto o objectivo ser analisar os efeitos dos tratamentos (senão, seriam 5 grupos). Utilizar-se-á a mesma abordagem para o tempo de leitura e em ambos os estudos experimentais.

O teste de Lilliefors produziu o seguinte resultado:

$$D = 0.11326,\ p - value = 0.03509$$

Por amostra:

$bc : 0.1363844,\ p - value = 0.3561041$

$gc : 0.2141587,\ p - value = 0.009911642$

$nc : 0.1737004,\ p - value = 0.08286397$

Globalmente e uma das amostras permitem rejeitar a hipótese nula de normalidade. Faz assim sentido usar o teste não-paramétrico, complementado com o teste ANOVA.

O teste Mack-Skillings produziu o seguinte resultado:

$$MS = 4.685464,\ p - value = 0.0882$$

que não excede o valor crítico $MS_\alpha = 5.80554$ para um $valor - p = 0.0857$. O resultado é estatisticamente significativo apenas para um $\alpha = 10\%$, mas a significância definida inicialmente foi 5%.

Mais uma vez, testou-se também com ANOVA e o resultado é mostrado na tabela 5.30.

Tabela 5.30: Resultado ANOVA para o Cloze.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	2	0.5268	0.26341	2.3742	0.101498
BLOCO	1	0.9258	0.92576	8.3443	0.005323 **
Residuals	62	6.8786	0.11095		

Para a medida Cloze o valor-p para o tratamento é superior a 0.05 e como tal a hipótese nula não pode ser rejeitada. O factor de blocagem teve um efeito estatisticamente significativo, ou seja, existe uma diferença significativa entre os blocos.

Tempo de Leitura Para o segundo estudo, também se analisou o tempo de leitura do *snippet*. Nas tabelas 5.31 e 5.32 são apresentados os intervalos de valores de cada variável, as suas medianas e médias. As medianas e médias por célula encontram-se nas tabelas 5.33a e 5.33b. A verificar os tempos na tabela 5.32, verifica-se que, em média, o *snippet* S3 demorou mais tempo a ler, mas a sua mediana foi inferior à do *snippet* S2.

Em termos de resultados por célula, parecem contraditórios. A média no *snippet* S2 é superior no código com "bons" comentários e sem comentários.

Para analisar a normalidade dos dados usou-se novamente o teste de Lilliefors que produziu o seguinte resultado:

$$D = 0.11989,\ p - value = 0.01963$$

Tabela 5.31: Estatísticas descritivas da variável Comentário para o Tempo de Leitura.

Tratamento	Mínimo	Máximo	Mediana	Média
BC	71	445	174.0	188.636
GC	63	506	228.5	243.773
NC	55	570	216.5	258.091

Tabela 5.32: Estatísticas descritivas de cada *snippet*.

Bloco	Mínimo	Máximo	Mediana	Média
S2	55	527	226	224.939
S3	71	570	196	235.394

Tabela 5.33: Médias e medianas por célula.

	BC	GC	NC
S2	220.636	233.818	220.364
S3	156.636	253.727	295.818

(a) Médias.

	BC	GC	NC
S2	208	231	206
S3	168	216	268

(b) Medianas.

Para cada amostra:

$bc : 0.1531537,\ p - value = 0.1971508$

$gc : 0.140237,\ p - value = 0.3140006$

$nc : 0.1320595,\ p - value = 0.4068555$

O resultado global apresenta um valor-p inferior a 0.05 mas para cada amostra não foi possível rejeitar a hipótese nula de normalidade. Resolveu-se utilizar ambos os testes, não-paramétrico e ANOVA.

O teste Mack-Skillings produziu o seguinte resultado:

$$MS = 3.227759,\ p - value = 0.1928$$

No teste de Mack-Skillings, MS não excede o valor crítico $MS_\alpha = 5.80554$, assumido anteriormente, para um $\alpha = 0.05$, logo a hipótese nula não é rejeitada.

Mais uma vez, testou-se também com ANOVA e o resultado é mostrado na tabela 5.34.

Tabela 5.34: Resultado ANOVA para o Tempo de Leitura.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Tratamento	2	59172	29586.2	2.1970	0.1197
BLOCO	1	1803	1803.4	0.1339	0.7157
Residuals	62	834933	13466.7		

Interpretação Analisa-se a legibilidade dos dois *snippets*. O *snippet* $S3$ obteve uma média e uma mediana para a medida $Escala$ superiores a $S2$. Essa diferença, tanto para a média como para a mediana, também se verificou em cada um dos factores. A situação foi idêntica para a medida $Cloze$ em que $S3$ foi o mais bem compreendido de forma estatisticamente significativa. Os resultados para o efeito do factor de blocagem mostram existir uma diferença estatisticamente significativa para a legibilidade entre os 2 *snippets* para ambas as medidas, $Escala$ e $Cloze$. Em conclusão, o *snippet* $S3$ foi considerado pelos alunos como mais legível que $S2$. Este resultado é coerente com o resultado obtido em [31] na medida em que nesse estudo o *snippet* $S2$ é o mais difícil e o menos legível.

A medida *Escala* é considerada uma medida de legibilidade. A hipótese nula quanto à igualdade de efeitos dos 3 níveis de tratamento não pode ser rejeitada, ou seja, verificou-se não existir, para um α igual a 5% diferença estatisticamente significativa na legibilidade do código-fonte com e sem comentários (NC) no corpo dos métodos, e no caso da existência de comentários fossem estes "bons" (GC) ou "maus" (BC). Como não existiu qualquer diferença estatisticamente significativa não foi necessário tentar encontrar qual dos 3 factores é que teria produzido esse efeito. No caso do primeiro estudo, não se colocou esta questão por haver apenas 2 factores.

A não rejeição da hipótese nula, não implica que a hipótese nula seja verdadeira [232], ou seja, que se considere indiferente colocar ou não comentários no corpo dos métodos, ou a qualidade dos comentários. Poderá significar que a diferença observada não é suficientemente grande para poder ser detectada pelo teste estatístico para o nível de significância estabelecido.

No caso da medida $Escala$, em termos globais, o factor BC foi o que apresentou, tanto para a mediana como para a média, valores mais elevados. NC foi superior a GC para a mediana, mas não para a média. Isto leva a questionar o tipo de comentários "maus" e "bons" colocados nos dois *snippets*.

Para o caso tratamento versus factor de blocagem, e considerando-se as medianas, os valores iguais indiciam ser indiferente a existência ou não de comentários e a sua qualidade, quer para $S2$ quer para $S3$. Já as médias diferem.

A medida $Cloze$ é usada na legibilidade, mas é considerada uma medida de compreensão do código lido. Para esta medida, a hipótese nula de igualdade dos efeitos dos 3 níveis não pode ser rejeitada para um α igual a 5%, mas verifica-se uma diferença estatisticamente significativa apenas para um α igual a 10%. O resultado tanto pode significar um mero acaso quanto às diferenças observadas, como ser o resultado de um efeito pequeno detectável com um risco razoável de um falso positivo (erro do tipo I). Já o teste paramétrico não permitiu detectar o efeito.

A ordem dos efeitos na compreensão foi $BC < GC < NC$. A diferença poderá ter interesse na perspectiva empírica, da prática. Observando-se os valores das médias e medianas na tabela 5.27 constata-se que a maior diferença é para NC, o que dá a indicação de a compreensão do código-fonte ser maior no caso de não possuir comentários.

Quanto à relação tratamento por factor de blocagem, verifica-se que GC tem para $S2$ o valor mais baixo entre os 3 factores e o segundo para $S3$. É possível afirmar que o código é mais compreensível sem comentários e que no caso de código mais legível, $S3$, segundo a medida $Escala$, "bons" comentários são preferíveis a "maus". Mas para código menos legível, os comentários "bons" parecem prejudicar. Mais uma vez, é de questionar os comentários dos métodos. A alternativa seria considerar que comentários em código menos legível podem ter um efeito contraproducente. Só mais investigação poderá esclarecer.

Relativamente à medida *Tempo de Leitura*, não foi possível rejeitar a hipótese nula acerca da igualdade entre os efeitos dos 3 níveis da variável tratamento. O tempo de leitura não foi afectado pelo tipo de *snippet* de forma estatisticamente significativa apesar de, em média, os participantes terem despendido mais tempo na leitura do *snippet* $S3$ ainda que a diferença para $S2$ seja pequena. Poderá supor-se que, como demorou mais tempo a ser lido, permitiu uma melhor compreensão do código e que o tempo de leitura para $S2$, sendo este menos legível, não foi suficiente para a sua compreensão. A conclusão, que parece natural, é que mais tempo de leitura resulta em melhor compreensão e que código menos legível requerá mais tempo de leitura.

O código-fonte sem comentários foi aquele em que os participantes despenderam em média mais tempo. Parece lógico que assim seja, pois a falta de informação requererá um maior tempo de leitura para compreensão. A diferença para comentários "bons" é pequena, mas razoavelmente acentuada para os "maus". Para esta última diferença contribui em grande parte $S3$ que apresenta um tempo médio de leitura muito inferior para o caso BC o que fez baixar a média global de $S3$. Em $S2$ as médias dos 3 factores divergem pouco. Para $S3$, existe uma diferença clara entre os 3 níveis com $BC < GC < NC$.

No caso do *snippet* $S3$ a legibilidade teve a seguinte ordem para as médias $BC > GC > NC$ e a compreensão a seguinte $BC < GC < NC$, ou seja a ordem inversa. O tempo de leitura elevado permite explicar essa diferença pois a sua ordem foi $BC < GC < NC$. Isto é, código menos legível, requereu mais tempo de leitura e no final tornou-se melhor compreendido. No caso do *snippet* $S2$ a legibilidade teve a seguinte ordem $BC = NC > GC$ e a compreensão a seguinte $GC < BC < NC$, ou seja a ordem inversa. a relação entre os tempos de leitura foi $NC \approx BC < GC$. Ou seja, mais tempo de leitura "implicou" ser

menos compreendido.

5.10 Limitações e Ameaças à Validade

Na investigação realizada foi seguido um processo sistemático que permitisse fornecer respostas às questões de investigação. Uma questão fundamental relacionada com os resultados de um estudo é saber o quão válidos eles são [297]. Conseguir que a investigação realizada e os seus resultados sejam válidos não é uma tarefa fácil, como se referiu em 5.2. É importante controlar o processo de investigação de uma forma adequada. Para melhor compreender as deficiências nas situações de investigação e as vantagens de superá-las, vários autores definiram critérios de validação do processo de investigação e dos seus resultados. A análise da validade deve ser realizada segundo quatro vertentes [297]:

1. Validade das conclusões: garantir a validade da análise estatística.
2. Validade interna: garantir coerência entre o tratamento e o resultado.
3. Validade do constructo: garantir até que ponto o estudo avalia a legibilidade (a variável operacionalizada).
4. Validade externa: garantir uma generalização a outros contextos e a outros participantes.

Os critérios de validade ajudam a dar explicações alternativas para os resultados do experiestudo experimentalmento como já referido na secção 5.2. É sobre estes quatro critérios de validade que se realiza a análise de validade descrita nas secções seguintes.

5.10.1 Validade das Conclusões

A partir dos tratamentos realizados e dos resultados obtidos são retiradas conclusões como as descritas na interpretação dos resultados. A validade dessas inferências diz respeito a factores que possam afectar a capacidade de obter conclusões correctas [297]. As conclusões dependem da adequabilidade da análise estatística realizada e por isso num sentido mais estrito se possa falar em validade estatística [210]. Durante a análise estatística dos dados houve necessidade de tomar decisões que poderão ter tido impacto na validade e que por isso importa que sejam descritas. Esta secção cinge-se à validade estatística deixando-se outros factores para a secção da validade interna.

No estudo, apesar de haver dados em falta não houve imputação de valores. Em vez disso, optou-se por considerar essas observações inválidas e reduzir o número de questionários para serem apenas considerados os válidos, isto é, os devidamente preenchidos. Com isso, visou-se reduzir os riscos provocados pela imputação de valores[13] em detrimento do número de dados para análise. A decisão pela não imputação resultou de se considerar que o preenchimento menos correcto do questionário poderia estar associado a alguma falta de empenho e interesse do participante, o que se entendeu poder afectar a validade desse tipo de respostas.

Também se verificou um número desigual de observações em cada célula em resultado da estratégia de randomização que foi utilizada. Para se obter o balanceamento das observações foi eliminado aleatoriamente de cada célula um número de observações correspondente à diferença entre o número de observações na célula e o número de observações da célula com menos observações. A eliminação de observações significa eliminar dados válidos e reduzir a dimensão das amostras.

Em geral, quanto maior o número de participantes maior a potência do teste estatístico. O número de observações por célula em cada estudo, 9 no primeiro e 11 no segundo, poderia ter sido superior se tivesse sido usado um processo distinto de randomização.

A análise estatística teve em conta a natureza ordinal da escala no caso da medida *Escala*. Nesses casos é recomendada a utilização de um teste não-paramétrico tal como foi feito. Para as outras medidas, tipo de escala rácio no caso da medida *Cloze* e intervalar no caso da medida *Tempo de Leitura* foi analisada primeiramente a normalidade dos dados. No entanto, como nunca foi observada normalidade também se utilizou o mesmo teste não-paramétrico. Seja como for, em todos os casos foi aplicado adicionalmente o método ANOVA e verificou-se existir concordância nos resultados com o teste não–paramétrico. Contudo, sabe-se que testes paramétricos têm em geral maior potência que os seus correspondentes não-paramétricos.

A medição do nível de legibilidade fez-se com recurso a uma escala do tipo Likert. Um dos problemas deste tipo de escala é a tendência para o valor central. As respostas dos alunos à pergunta do nível de legibilidade tem como média um valor próximo do valor

[13] As técnicas de imputação recorrem aos dados existentes para daí obterem os dados em falta, podendo, portanto, afectar a validade dos resultados.

central. Não se pode descartar algum viés das respostas. Em ambos estudos, as médias foram sempre muito próximas de 2.5 e as medianas em geral de 2, portanto, abaixo do valor central 3.

5.10.2 Validade Interna

A validade interna diz respeito à existência de uma relação entre o tratamento e os resultados. O objectivo é garantir que algum efeito observado provenha de uma relação causa-efeito entre as variáveis independentes e as variáveis dependentes, e não seja o resultado de um outro factor, estranho, que não tenha sido possível medir ou controlar [297]. Como se percebe, este é o tipo de validade mais fundamental [210] pois depende essencialmente da concepção do estudo. Nos estudos pretendia-se avaliar o efeito do encadeamento e dos comentários sobre a legibilidade.

Um dos objectivos da formação de blocos é a eliminação parcial da influência de variações entre as unidades experimentais na comparação dos tratamentos [43]. Apesar de o estudo ter adoptado um plano com blocagem, a randomização foi simples e não por blocos porque, tal como explicado, a definição dos blocos não se prendia com características dos participantes, mas sim com o código, de forma a tornar os resultados mais generalizáveis. Contudo, na randomização atribuiu-se aleatoriamente cada situação, par bloco/nível, a cada participante e desse modo a presença de cada aluno num bloco foi também aleatória. Ainda acerca da randomização, esta foi realizada tendo por base todos os alunos inscritos na unidade curricular como um todo e não, por exemplo, por turmas. Quer dizer que se considerou que as características dos alunos não diferiam entre turmas, tendo todos a mesma probabilidade de ser atribuídos a cada situação experimental. De qualquer forma, provocou um desequilíbrio no número de participantes em cada situação experimental que obrigou a eliminar observações reduzindo assim o tamanho de cada amostra.

No questionário pedia-se a cada participante que lesse o código com muita atenção de modo a que fosse capaz de o explicar. Posteriormente era pedido ao aluno que classificasse o código quanto à sua legibilidade numa escala de 1 a 5. Estes dois pontos merecem atenção. Primeiro, essa medida de legibilidade é subjectiva. Segundo, não era apresentada explicitamente a definição de legibilidade, mas sim que o aluno fosse capaz de explicar o código, ou seja, a definição estava implícita e que segundo ela considera-se o código legível quando o leitor é capaz de o explicar. Na escolha dos *snippets*, o código escolhido

devia ser supostamente desconhecido de todos os alunos, mas não se pode garantir que isso tenha sido conseguido.

Uma das regras do estudo consistia em os alunos não poderem voltar atrás no questionário. Durante o estudo, a investigadora e o regente estiveram sempre presentes mas não houve a garantia de a regra ter sido cumprida. O teste Cloze será a componente do estudo mais sensível à violação dessa regra e a existir tal violação seria com o código menos legível. É de supor que a necessidade de voltar atrás se sinta mais quando o aluno tem uma dificuldade maior na leitura do código. Se volta atrás nesses casos, então os valores para esses casos serão de certa forma "invertidos". Num estudo futuro haverá que melhorar o cumprimento da regra. Também não se pode excluir ter sido cometido algum erro de medição devido à forma como o teste Cloze foi elaborado.

5.10.3 Validade do Constructo

A validade do constructo refere-se à relação entre a teoria e o que foi observado no estudo [297]. Estando assegurada a validade interna, em geral, importa garantir que o resultado se refere ao constructo do efeito e que o tratamento se refere ao constructo da causa [297].

No presente caso, a teoria está associada ao conceito legibilidade, o conceito abstracto em estudo. Já os tratamentos, encadeamento e comentários, não se podem considerar constructos ainda que sejam difíceis de operacionalizar de forma genérica pois podem ser aplicados de inúmeras formas. A validade do contructo pode ser analisada em dois níveis, validade do instrumento e validade do estudo [210]. O instrumento não levantou dúvidas aos participantes. Mas a sua validade depende também das medidas utilizadas.

Quanto aos extractos e código-fonte, fica a dúvida se o seu grau de dificuldade não seria demasiado alto para o perfil de participantes em virtude de os valores centrais para a *Escala* terem sido em geral abaixo do meio da escala.

Sendo a legibilidade um conceito abstracto, algo não mensurável directamente, houve necessidade de o operacionalizar. Para a operacionalização foi usada a medida *Escala* para a legibilidade percebida pelos participantes, complementada com a medida *Cloze* para a compreensão e com o tempo de leitura. A *Escala* consiste numa escala de 5 pontos. Durante a análise ficou a dúvida se não teria sido preferível uma escala com mais

pontos, por exemplo, 7, para discriminar melhor as percepções dos participantes. Esta afirmação baseia-se no facto de os valores obtidos terem ficado relativamente concentrados abaixo do meio da escala. Talvez mais pontos fornecessem maior sensibilidade. Apesar disso, este tipo de medida tem sido utilizado noutros estudos e foi considerada fiável com uma correlação elevada com a capacidade de responder correctamente a questões sobre o código (cf. secção 5.6.4). No presente estudo não se verificou haver correlação entre *Escala* e *Cloze*.

O teste para a medida *Cloze* seguiu o modelo do estudo em [31]. Este modelo difere dos mais habituais, com uma distância fixa entre espaços em branco (texto removido) a serem preenchidos. A medição foi feita pela proporção de espaços correctamente preenchidos. O número de valores possíveis para o resultado mostrou-se reduzido. Também aqui poderá ter faltado sensibilidade para a escala.

5.10.4 Validade Externa

A validade externa diz respeito à generalização das inferências realizadas com base nos resultados a contextos distintos do contexto do estudo [297, 210]. No presente estudo, a generalização refere-se à generalização das inferências a outros alunos de POO e que poderão ser de outros cursos da escola onde se realizou o estudo experimental, ou de outras escolas. O estudo realizou-se no ambiente real de sala de aula o que favorece a sua validade e consequentemente a sua generalização.

A população-alvo do estudo era composta por todos os alunos de POO de cursos superiores de informática. No entanto, usou-se uma amostra de conveniência, circunscrita a uma única disciplina de uma escola apenas, o que faz com que a generalização dos resultados deva ter isso em consideração. Apesar desta limitação, parece razoável supor que os alunos participantes não difiram significativamente dos alunos de muitas outras instituições, incluindo de outros países. Refira-se que a utilização de amostragem aleatória é, pela dificuldade em obtê-la, pouco comum [232].

A participação dos alunos no estudo foi opcional, o que significa a existência de dois grupos de alunos, o dos que participaram opcionalmente e o dos que entenderam não o fazer. Isto é relevante porque não é possível afirmar que os grupos são idênticos quanto às características dos seus membros. É, portanto, sempre possível afirmar que os alunos

que participaram possuiriam características que os afastavam da população-alvo. Como o estudo se realizou numa aula regular do curso é expectável que a participação tenha sido maior do que se não tivesse sido numa aula regular, o que deverá ter reduzido o número de alunos que optaram por não participar. Aliás, segundo o regente da disciplina e sem ter indicado números, a participação terá sido bastante elevada. Isso leva a crer que o factor dos dois grupos seja mínimo.

Em cada estudo foi usado um número limitado de formas de encadeamento e de comentários. A generalização fica também limitada por estas opções. A utilização de blocagem permitiu usar mais do que um extracto de código em cada estudo, mas mesmo assim não deixa de ser um conjunto limitado. Em conclusão, qualquer generalização deve ser feita com precaução.

5.11 Conclusão

Neste capítulo foram apresentados dois estudos experimentais realizados consecutivamente com o intuito de testar duas práticas de legibilidade de código-fonte. As práticas avaliadas foram respectivamente, a não utilização de encadeamento de métodos e a utilização de comentários no interior dos métodos. Os comentários usados no segundo caso podiam ser de dois tipos, "bons" ou "maus". Para medir a legibilidade usou-se como medida uma escala de 5 pontos. Sendo a escala uma medida subjectiva foi também usado o teste Cloze para medição da compreensão e cada aluno indicou ainda o tempo dedicado à leitura do respectivo extracto de código-fonte. As principais conclusões dos estudos são apresentadas em seguida.

Em primeiro lugar não foi possível obter resultados estatisticamente significativos para um α igual a 5% para qualquer um dos dois estudos experimentais. Esse resultado significa que não foi possível recusar as hipóteses nulas respectivas, ou seja, o encadeamento e não encadeamento, os comentários" bons", "maus" e sua ausência, não tiveram efeitos estatisticamente distintos na legibilidade, na compreensão, ou no tempo de leitura.

No primeiro estudo os resultados sugerem que a legibilidade possa variar inversamente entre código com e sem encadeamento. Colocou-se a hipótese da importância de evitar o aumento de encadeamento com a diminuição da legibilidade. Pelos resultados, pode ser colocada a mesmo hipótese relativamente à compreensão.

Quanto aos três *snippets* verificou-se uma diferença estatisticamente significativa no seu efeito sobre a compreensão. Pelo valor das médias, $S4$ terá sido o mais bem compreendido e o único em que houve uma melhoria da compreensão sem encadeamento. Em alguns dos casos, os tempos de leitura ajudam a explicar a melhor compreensão. Os *snippets* usados poderão ter sido excessivamente difíceis para o nível dos alunos, essencialmente em virtude de serem relativos a domínios de aplicação estranhos aos alunos. A realização de um estudo piloto poderia ter evitado este problema.

No segundo estudo foi avaliado o código sem comentários e com comentários, "bons" e "maus". Os valores da legibilidade foram idênticos, quase não variando com o tipo de comentário, ou com a sua ausência. A compreensão variou com o nível de tratamento para um α igual a 10% no caso do teste não-paramétrico. A compreensão terá sido maior no caso de código sem comentários. Hipotetizou-se se o código mais legível seria prejudicado por comentários. Uma questão a investigar. Para os tempos de leitura, a melhor explicação encontrada foi que, e como seria de esperar, mais tempo de leitura resulta em melhor compreensão e que código menos legível requer mais tempo de leitura. Mas na análise por *snippet* isso não fica claro. Tanto a legibilidade como a compreensão foram afectadas de forma estatisticamente significativa pelos dois *snippets*.

Este estudo pode ser considerado um replicação conceptual do estudo descrito em [31] pois apenas as questões de investigação são equivalentes. Nas questões do estudo actual apenas se refere legibilidade, enquanto no original é referida também a compreensão. Como já explicado, não se fez essa explicitação porque a compreensão está subjacente à medição da legibilidade. O estudo original consistiu num estudo único, mas aqui as questões foram analisadas separadamente.

A unidade curricular em que se desenvolveu o estudo é uma unidade do segundo semestre do primeiro ano de uma licenciatura em engenharia informática que tem por finalidade o ensino da programação orientada a objectos (POO). De notar que, no estudo original os participantes possuiam níveis variados de experiência em programação, havendo mesmo profissionais. Nesse estudo os participantes respondiam remotamente, fora da escola, e sem o controlo do investigador.

A população-alvo do presente estudo é diferente da população-alvo do estudo original. Uma mais valia deste estudo foi ter sido realizado num ambiente real, neste caso na sala de

aula com alunos que estão a aprender POO. Por isso, a amostra deste estudo enquadra-se na população-alvo definida para o trabalho, estudantes de POO de cursos de informática. Apesar disso, qualquer generalização tem de ser feita com precaução em virtude de se tratar de uma amostra de conveniência.

O estudo original produziu um resultado estatisticamente significativo apenas para os comentários "bons". O efeito de "maus" comentários foi superior ao da ausência de comentários, sem ser significativo. Para esse estudo, o efeito da prática ausência de encadeamento foi positivo mas não significativo. Em conclusão, o estudo actual não confirma os resultados do estudo original.

Capítulo 6

Conclusões

Neste trabalho o tema central foi a legibilidade do software, ou seja, do código-fonte. Ao longo do documento foram apresentados os aspectos mais teóricos da legibilidade e a literatura revista. Foi realizado um inquérito a docentes de programação orientada a objectos e foram também realizados dois estudos empíricos sobre as práticas de legibilidade. Neste capítulo são agora apresentadas as conclusões principais.

6.1 Introdução

A legibilidade do código-fonte é um tema com importância, não só na prática da engenharia informática, mas também no ensino desta, tal como houve a preocupação de explicar no capítulo de introdução. Para a melhoria da legibilidade do software já foram sugeridas diversas práticas de programação por variados autores. A investigação da legibilidade do software acontece desde há décadas mas mantém-se necessária.

A legibilidade do software tem por base a procupação com a legibilidade de textos e é um tema de interesse noutras áreas para além do software, como seja o ensino logo desde os primeiros anos, os textos linguísticos, a bibliografia científica, empresarial, médica e em comunicação tecnológica [13]. Como referido na secção 2.2, já em 900 d.C. os Talmudistas realizavam contagens de palavras e ideias e os Hebreus faziam a análise de vocabulário da bíblia.

Como já se viu, a legibilidade do código-fonte está associada à compreensão do software, aliás sem leitura do código não pode ocorrer a compreensão desse código. Por outro

lado, a compreensão do código lido funciona como uma medida do nível de legibilidade. Tanto a compreensão como a legibilidade, sendo actividades humanas, dependem das características do ser humano e, portanto, podem beneficiar do estudo de outras disciplinas, como a psicologia cognitiva, ou as neurociências. Por exemplo, e como se viu, a imagiologia (e.g. ressônancia magnética funcional) tem sido usada para ajudar a compreender o esforço necessário para realização de tarefas de compreensão. Acerca da compreensão existem diversos modelos que tentam explicar esse processo humano. Esse assunto foi tratado no capítulo 2.

Alguns investigadores têm estendido esse esforço, através do desenvolvimento de modelos cognitivos quantitativos que possam ser executados. As arquitecturas cognitivas serão a concretização desses modelos. Por outras palavras, elas são ambientes de simulação (em software) que integram teorias das ciências cognitivas permitindo realizar predições acerca do desempenho de programadores [117]. Outra área de automatização consiste na extracção de conhecimento do código, por exemplo, para criar um resumo da finalidade do programa.

Para melhor delimitação do objecto do estudo, foi necessário distinguir entre estilos de programação e práticas de programação. Apesar de não haver uma separação clara entre estilos de programação e práticas de programação, os estilos podem ser considerados como aspectos de gosto pessoal e/ou tipográfico, enquanto que as práticas, tal como usadas neste texto, são em geral, ou em grande medida, independentes do gosto pessoal do programador e referem-se à programação propriamente dita. As práticas de legibilidade enquadram-se nas práticas de programação.

O presente trabalho teve como principais objectivos, contribuir para a definição de um conjunto de práticas de legibilidade no software, que pudesse ser usado pelos alunos de cursos de informática, nas respectivas disciplinas de introdução à programação orientada a objectos, e aumentar o conhecimento existente sobre a situação da legibilidade do software, em particular nessas disciplinas de introdução à programação orientada a objectos.

Para a concretização dos objectivos, foi elaborado um conjunto preliminar de práticas de legibilidade, que permitiu realizar um inquérito a professores dessas disciplinas de diferentes cursos de Informática do ensino superior, de modo a avaliar a importância dessas práticas e a obter um melhor conhecimento da situação do ensino da legibilidade. Em

seguida, procedeu-se a um levantamento exaustivo das práticas de legibilidade consideradas relevantes por profissionais e investigadores. Posteriormente, foram seleccionadas duas práticas de legibilidade para serem validadas através de um estudo experimental em ambiente real de ensino como o referido antes.

6.2 Contribuições

Dadas as bases da legibilidade do código-fonte, no primeiro capítulo após a introdução, procedeu-se a uma análise da teoria da legibilidade em geral, da sua evolução e foram feitas aí algumas ligações à legibilidade no software. Foi também feita uma apresentação de muitas fórmulas de legibilidade, com o cuidado de incluir as usadas no software. Um dos resultados da análise foi uma compilação das fórmulas de legibilidade mais importantes e mais conhecidas, que pode ser encontrada na secção 2.3.2.

Para além dessa contribuição e como se pode concluir pelo exposto na secção anterior, o trabalho realizado permitiu outras contribuições. São essas contribuições que se listam em seguida:

- Estado actual de utilização de boas práticas, conseguido através de um inquérito a docentes [231];
- Compilação de boas práticas para a legibilidade do código-fonte a usar pelos alunos, conseguido pela revisão da literatura;
- Validação de duas práticas de legibilidade, através de estudos experimentais.

Um dos objectivos deste trabalho consistia em saber-se qual o estado actual de utilização de boas práticas de legibilidade em cursos de informática do ensino superior. Para isso foi realizado um inquérito, com base num questionário, a docentes de programação orientada a objectos dos cursos de informática do ensino superior para saber opinião dos docentes sobre a importância do ensino de práticas de legibilidade e a sua utilização. Este questionário foi disponibilizado através de uma hiperligação. Neste questionário, foi apresentado um conjunto de práticas, preliminar, obtido da literatura. Para cada prática, os docentes tinham de indicar o grau de importância do seu ensino. No apêndice B é possível ver o questionário apresentado aos docentes. Deste inquérito concluiu-se que os

docentes reconhecem a importância do ensino de práticas de legibilidade, em geral. O resultado foi estatisticamente significativo.

Através da revisão da literatura verificou-se existirem muitos estudos contraditórios e estudos em que não foi possível obter resultados estatisticamente significativos de modo a poderem ser retiradas conclusões definitivas. A revisão da literatura teve duas grandes fases: a primeira de *snow-ball* e a segunda, a da revisão sistemática (*mapping review*). Com base na revisão da literatura compilaram-se duas listas de práticas, uma oriunda da literatura científica e outra de livros e com cariz mais profissional. Essas listas foram agregadas num única com identificação da fonte e do tipo da fonte (ver apêndice A). Ficou claro que nem todas as práticas são comuns aos 2 tipos de literatura. A existência de uma lista de práticas permitirá potenciar a criação de código mais legível por parte dos alunos e de forma mais uniforme.

Conforme referido, é importante continuar a realizar estudos empíricos sobre a matéria. Foi ponderada a possibilidade de se validar, através de um estudo experimental, a lista de todas as práticas que resultariam da revisão da literatura científica. Contudo, uma vez que essa lista cresceu muito, a sua validação tornou-se impraticável. O estudo requereria a criação de extractos de código a apresentar aos alunos que obedecessem a todas as práticas e de outros extractos de código que não obedecessem a qualquer prática.

Foram realizados então dois estudos experimentais para testar duas das práticas de legibilidade do código-fonte. O conjunto dos 2 estudos pode ser considerado uma replicação do estudo descrito em [31], a que se chamou estudo original. Em cada um dos experimentos foi avaliada uma prática de legibilidade. No primeiro experimento foi avaliada a utilização de encadeamento de métodos. O experimento tinha a seguinte questão de investigação, que se encontra na secção 5.5:

QI1 Em que medida é que o código-fonte que *não contém encadeamento de métodos* (NMC) é mais legível que o código que *contém encadeamento de métodos* (MC)?

No segundo experimento avaliou-se a utilização de comentários no interior dos métodos. Este experimento possuía a seguinte questão de investigação, que também se encontra na secção 5.5:

QI2 Em que medida é que o código com *comentários* (GC) é mais legível que o código *sem comentários* (NC) ou com *"maus" comentários* (BC)?

Pelos resultados do estudo não foram encontradas diferenças significativas na utilização de cada uma das práticas, nos seus diferentes níveis. Apenas a compreensão parece ser afectada pelo tipo de comentário, mas para um nível de significância de apenas 10%. Os resultados não estão em linha com os do estudo original, onde a ausência de encadeamento tinha obtido um resultado estatisticamente significativo para a sua influência na legibilidade. Portanto, o estudo actual não confirma o original e não se têm assim resultados mais conclusivos acerca da importância dessas duas práticas. O seu estudo continua, portanto, a ser necessário. Para melhor análise dos estudos, as ameaças à sua validade foram apresentadas junto com o estudo (cf. 5.10).

Apesar da inexistência de resultados estatisticamente significativos para os tratamentos, os dados, ainda que não sejam claros, permitem elaborar algumas explicações tentatórias. Por exemplo, pelo primeiro experimento, colocou-se a possibilidade de a importância da prática de não utilização de encadeamento aumentar consoante diminui a legibilidade do código. No segundo experimento colocou-se a hipótese de o código mais legível poder ser afectado negativamente pela introdução de comentários. Em vários casos, um maior tempo de leitura permitiu tornar mais bem compreendidos *snippets* considerados menos legíveis.

Os *snippets* em ambos os experimento tiveram um efeito distinto estatisticamente significativo na compreensão segundo o teste ANOVA. No segundo experimento também tiveram esse efeito sobre a legibilidade. Os valores confirmam, como seria de esperar, que o código tem impacto na sua compreensão pelo leitor. Os valores da legibilidade percepcionada pelos alunos são abaixo da mediana da escala e faz supor que os *snippets* fossem excessivamente difíceis para os alunos. Teria sido preferível realizar um estudo piloto para avaliar a dificuldade do código a usar. Alguns dados também apontam para a relevância do domínio de aplicação, isto é, o interesse dos comentários pode ser afectado pela maior ou menor familiarização do aluno com o domínio de aplicação. A importância do conhecimento do domínio de aplicação está de acordo com a psicologia cognitiva, tema do capítulo 2.

Os experimentos foram realizados num ambiente real de sala de aula de uma disci-

plina de introdução à POO de um curso de informática do ensino superior tal como era objectivo deste trabalho. Trata-se portanto de uma amostra retirada da população-alvo. A população do estudo original não era a mesma e por isso as suas conclusões poderiam não ser válidas para a população-alvo deste trabalho. Essa foi mais uma razão para o estudo desenvolvido. Os resultados obtidos e conclusões elaboradas devem ser no entanto generalizadas com precaução.

6.3 Trabalho Futuro

Dado o interesse demonstrado pelos docentes de POO na legibilidade do código-fonte e comprovado pela revisão da literatura, importa continuar a investigação nesta área. As possibilidade de investigação são várias. Da revisão da literatura resultou uma lista bastante alargada de práticas de legibilidade. Umas com efeito positivo, outras negativo. A investigação da validade dessas práticas seria por si só um programa de investigação de grandes dimensões. Contudo, os experimentos realizados já colocaram diversos problemas e questões. Como trabalho futuro pretende-se em primeiro lugar dar continuidade à investigação realizada e portanto a essas questões e problemas, em lugar de se avançar para o estudo de outras práticas.

Como exposto na secção anterior, os resultados dos experimentos não foram estatisticamente significativos, o que se aconselha a realização de novos estudos com as mesmas questões de investigação, possivelmente com outros estudantes e código distinto. Para avaliar a dificuldade do código deverá realizar-se um estudo piloto com um pequeno número de participantes.

Para além disso, os resultados obtidos nos experimentos também forneceram algumas pistas quanto a trabalho futuro. Na secção anterior foram colocadas algumas questões e hipóteses que podem ser relevantes para investigação futura. Aquelas que se consideram mais importantes ou interessantes para serem investigadas com maior brevidade são apresentadas em seguida sob a forma de questões:

1. Será que o código quanto mais legível for, mais afectado negativamente será pela introdução de comentários?

2. Será que quanto menos legível for o código, mais importante se torna a prática de não utilizar encadeamento?

3. Em que medida a introdução de comentários é afectada pela familiarização, ou falta dela, do aluno com o domínio de aplicação do código fonte?

Um dos problemas apontado na análise das ameaças à validade dos estudos foi o de não se poder garantir que os participantes tenham voltado atrás para reler o código durante o estudo. Em estudos futuros é importante melhorar este aspecto. O questionário também deverá ser revisto, pelo menos quanto ao código-fonte fornecido. De qualquer forma, por maior que seja o esforço no sentido de garantir a validade do estudo, é preciso estar consciente de que existirão sempre limitações ao estudo que impedem a sua total consecução.

Bibliografia

[1] Cor Aarnoutse and Gonny Schellings. Learning reading strategies by triggering reading motivation. *Educational Studies*, 29(4):387–409, 2003.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[3] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. In *Reliability and maintainability symposium, 2002. Proceedings. Annual*, pages 235–241. IEEE, 2002.

[4] Hirohisa Aman, Sousuke Amasaki, Takashi Sasaki, and Minoru Kawahara. Empirical analysis of fault-proneness in methods by focusing on their comment lines. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 2, pages 51–56. IEEE, Dec 2014.

[5] Goodubaigari Amrulla, Murlidher Mourya, Abdul Ahad Afroz, and Syed Kashif Ali. A survey of improving computer program readability to aid modification. *International Journal of Advanced Trends in Computer Science and Engineering*, 3(3):981–992, May 2014.

[6] John R Anderson. *Cognitive Psychology and Its Implications*. Worth Publishers, 8th edition, 2015.

[7] Miranda R Andrus and Mary T Roth. Health literacy: a review. *Pharmacotherapy: The Journal of Human Pharmacology and Drug Therapy*, 22(3):282–302, 2002.

[8] Eran Avidan and Dror G. Feitelson. From obfuscation to comprehension. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 178–181. IEEE Press, 2015.

[9] Earl Babbie. The practice of social research, belmont, ca: Thomson learning, 2007.

[10] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the 10th International Conference on Software Engineering*, pages 356–366. IEEE Computer Society Press, 1988.

[11] K.D. Bailey. *Methods of Social Research*. McGraw-Hill, 1st edition, 1994.

[12] Alan Bailin and Ann Grafstein. The linguistic assumptions underlying readability formulae: A critique. *Language & Communication*, 21(3):285–301, 2001.

[13] Alan Bailin and Ann Grafstein. *Readability: Text and Context*. Springer, 2016.

[14] David W Baker. The meaning and the measure of health literacy. *Journal of General Internal Medicine*, 21(8):878–883, 2006.

[15] David W Baker, Julie A Gazmararian, Joseph Sudano, and Marian Patterson. The association between age and health literacy among elderly persons. *The Journals of Gerontology Series B: Psychological Sciences and Social Sciences*, 55(6):S368–S374, 2000.

[16] T. Baker. Chief programmer team management of production programming. In Edward Yourdon, editor, *Classics in software engineering*, pages 65–82. Yourdon Press, 1979.

[17] Victor R Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[18] Victor R Basili, Richard W Selby, and David H Hutchens. Experimentation in software engineering. *IEEE Transactions on software engineering*, (7):733–743, 1986.

[19] Aisha Batool, Muhammad Habib ur Rehman, Aihab Khan, and Amsa Azeem. Impact and comparison of programming constructs on java and c# source code readability. *International Journal of Software Engineering and Its Applications*, 9(11):79–90, 2015.

[20] John C Begeny and Diana J Greene. Can readability formulas be used to successfully gauge difficulty of reading materials? *Psychology in the Schools*, 51(2):198–215, 2014.

[21] Barbara A Benander, Narasimhaiah Gorla, and Alan C Benander. An empirical study of the use of the goto statement. *Journal of Systems and Software*, 11(3):217–223, 1990.

[22] Rebekah George Benjamin. Reconstructing readability: Recent developments and recommendations in the analysis of text difficulty. *Educational Psychology Review*, 24(1):63–88, 2012.

[23] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[24] Emmett Albert Betts. Readability: its application to the elementary school. *The Journal of Educational Research*, 42(6):438–459, 1949.

[25] AC Bickley, Billie J Ellington, and Rachel T Bickley. The cloze procedure: A conspectus. *Journal of Literacy Research*, 2(3):232–249, 1970.

[26] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[27] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 158–167. IEEE, 2009.

[28] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009. Special Issue on Program Comprehension (ICPC 2008).

[29] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426, 2005.

[30] Barry W Boehm, John R Brown, and Myron Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[31] J. Börstler and B. Paech. The role of method chains and comments in software readability and comprehension: An experiment. *IEEE Transactions on Software Engineering*, 42(9):886–898, Sept 2016.

[32] Jürgen Börstler, Michael E Caspersen, and Marie Nordström. Beauty and the beast–toward a measurement framework for example program quality. 2007.

[33] Jürgen Börstler, Michael E Caspersen, and Marie Nordström. Beauty and the beast: on the readability of object-oriented example programs. *Software Quality Journal*, pages 1–16, 2015.

[34] Norman M Bradburn, Seymour Sudman, and Brian Wansink. *Asking Questions: the Definitive Guide to Questionnaire Design–for Market Research, Political Polls, and Social and Health Questionnaires*. John Wiley & Sons, 2004.

[35] Robert Bringhurst. *The Elements of Typographic Style*, volume 127. Hartley & Marks Point Roberts, 1992.

[36] Andrew Brooks, Marc Roper, Murray Wood, John Daly, and James Miller. Replication of software engineering experiments. *Empirical Foundations of Computer Science Technical Report, EfoCS-51-2003. Department of Computer and Information Sciences, University of Strathclyde*, 2003.

[37] Kim B Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT press, 2002.

[38] Raymond P. L. Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 121–130. ACM, 2008.

[39] Raymond P. L. Buse and Westley R Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, 2010.

[40] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, pages 987–996. IEEE Press, 2012.

[41] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE Press, May 2015.

[42] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 31–35. IEEE, 2009.

[43] T Calinski and S Kageyama. Block designs: a randomization approach. vol. i: Analysis. *Lecture notes in statistics*, 150, 2000.

[44] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[45] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795–799, 1998.

[46] Jeffrey C. Carver, Natalia Juristo, Maria Teresa Baldassarre, and Sira Vegas. Replications of software engineering experiments. *Empirical Software Engineering*, 19(2):267–276, 2014.

[47] Kevin Catalano. On the wire: How six news services are exceeding readability standards. *Journalism & Mass Communication Quarterly*, 67(1):97–103, 1990.

[48] B. D. Chaudhary and H. V. Sahasrabuddhe. Meaningfulness as a factor of program complexity. In *Proceedings of the ACM 1980 Annual Conference*, pages 457–466. ACM, 1980.

[49] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.

[50] Jitender Kumar Chhabra, KK Aggarwal, and Yogesh Singh. Code and data spatial complexity: two important software understandability measures. *Information and Software Technology*, 45(8):539–546, June 2003.

[51] S. R. Chidamber and C. F. Kemerer. *Towards a Metrics Suite for Object Oriented Design*. ACM, 1991.

[52] Alan G. Clewer and David H. Scarisbrick. *Practical Statistics and Experimental Design for Plant and Crop Science*. John Wiley & Sons, 2001.

[53] H. Clifton. A technique for making structured programs more readable. *SIGPLAN Not.*, 13(4):58–63, Apr 1978.

[54] Virginia Clinton. Examining associations between reading motivation and inference generation beyond reading comprehension skill. *Reading Psychology*, 36(6):473–498, 2015.

[55] Pascale Colé, Lynne G. Duncan, and Agnès Blaye. Cognitive flexibility predicts early reading skills. *Frontiers in Psychology*, 5(565), 2014.

[56] Meri Coleman and Ta Lin Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283, 1975.

[57] Emilio Collar Jr and Ricardo Valerdi. Role of software readability on software development cost. In *21st International Forum on COCOMO and Software Cost Modeling*, Nov 2006.

[58] Kevyn Collins-Thompson. Computational assessment of text readability: A survey of current and future research. Working Draft, September 2014.

[59] Bradford R Connatser. Last rites for readability formulas in technical communication. *Journal of technical writing and communication*, 29(3):271–287, 1999.

[60] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *Software Engineering, IEEE Transactions on*, SE-5(2):96–104, March 1979.

[61] Melis Dagpinar and Jens H Jahnke. Predicting maintainability with object-oriented metrics-an empirical comparison. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 155–164, 2003.

[62] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2015.

[63] Edgar Dale and Jeanne S Chall. The concept of readability. *Elementary English*, 26(1):19–26, 1949.

[64] Terry C Davis, Michael A Crouch, Georgia Wills, Sarah Miller, and DM Abdehou. The gap between patient reading comprehension and the readability of patient education materials. *The Journal of family practice*, 1990.

[65] Szabolcs Michael De Gyurky. *The Cognitive Dynamics of Computer Science: Cost-Effective Large Scale Software Development*. John Wiley & Sons, 2006.

[66] Andrie De Vries and Joris Meys. *R for Dummies*. John Wiley & Sons, 2nd edition, 2015.

[67] F. DeiBenbock and M. Pizka. Concise and consistent naming. In *Proceedings of the International Workshop on Program Comprehension*, 2005.

[68] L Deimel and J Naveda. Reading computer programs: Instructor's guide and exercises educational materials cmu. Technical report, SEI-90-EM, 1990.

[69] Lionel E Deimel Jr. The uses of program reading. *ACM SIGCSE Bulletin*, 17(2):5–14, 1985.

[70] F. Deissenboeck and M. Pizka. Concise and consistent naming: Ten years later. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 3–3. IEEE, May 2015.

[71] Joel Denning, Maria Soledad Pera, and Yiu-Kai Ng. A readability level prediction tool for k-12 books. *Journal of the Association for Information Science and Technology*, 67(3):550–565, 2016.

[72] Françoise Détienne. *Software Design–Cognitive Aspects*. Springer-Verlag New York, Inc., 2002.

[73] Gerrit E DeYoung, Garry R Kampen, and James M Topolski. Analyzer-generated and human-judged predictors of computer program readability. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 223–228. ACM, 1982.

[74] Preet Kamal Dhillon and Gurleen Sidhu. Can software faults be analyzed using bad code smells?: An empirical study. *International Journal of Scientific and Research Publications (IJSRP)*, 2(10):1–7, October 2012.

[75] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer-Verlag, 2007.

[76] E. Dijkstra. Programming considered as a human activity. In *Classics in Software Engineering*, pages 1–9. Yourdon Press, 1979.

[77] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *Software Engineering, IEEE Transactions on*, 29(7):665–670, 2003.

[78] Jonathan Dorn. A general software readability model. *University of Virginia, Charlottesville, Virginia*, 2012.

[79] William H DuBay. The principles of readability. *Online Submission*, 2004.

[80] HE Dunsmore. The effect of comments, mnemonic names, and modularity: Some university experiment results. In *Empirical Foundations of Information and Software Science*, pages 189–196. Springer, 1985.

[81] Zoya Durdik, Benjamin Klatt, Heiko Koziolek, Klaus Krogmann, Johannes Stammel, and Roland Weiss. Sustainability guidelines for long-living software systems. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 517–526. IEEE, 2012.

[82] E.S. Edgington and P. Onghena. *Randomization Tests*. Taylor & Francis Group, 4th edition, 2007.

[83] Paul D Ellis. *The Essential Guide to Effect Sizes: Statistical Power, Meta-analysis, and the Interpretation of Research Results*. Cambridge University Press, 2010.

[84] James L. Elshoff. An analysis of some commercial pl/i programs. *IEEE Transactions on Software Engineering*, 2(2):113, 1976.

[85] James L Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8):512–521, 1982.

[86] Warren J Erikson. A pilot study of interactive versus noninteractive debugging. Technical report, System Development Corporation, 1966.

[87] B.S. Everitt and C. Palmer. *Encyclopaedic Companion to Medical Statistics*. John Wiley & Sons, Ltd, 3rd edition, 2011.

[88] Leo C Fay, Weldon G Bradt, and Edward G Summers. Doctoral studies in reading, 1919 through 1960. Technical report, Indiana University, 1964.

[89] N. E Fenton and S. L. Pfleeger. *Software Metrics: a Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[90] Norman Fenton and James Bieman. *Software Metrics: a Rigorous and Practical Approach*. CRC Press, 2014.

[91] Arlene Fink. *The Survey Handbook*, volume 1. Sage, 2003.

[92] Ann Fitzsimmons and Tom Love. A review and evaluation of software science. *ACM Comput. Surv.*, 10(1):3–18, March 1978.

[93] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221, 1948.

[94] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79. IEEE, 2007.

[95] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, Dec 2009.

[96] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[97] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2000.

[98] Françoys Gagné. Academic talent development and the equity issue in gifted education. *Talent Development & Excellence*, 3(1):3–22, 2011.

[99] T. Garcia-Marques. A hipótese de estudo determina a análise estatística: Um exemplo com o modelo anova. *Análise Psicológica*, 15(1):19–28, 1997.

[100] Gareth Gaskell. Language processing. In Nick Braisby and Angus Gellatly, editors, *Cognitive Psychology*, pages 197–230. Oxford University Press, 2005.

[101] Peter Gavora. Text comprehension and text readability: Findings on lower secondary school pupils in slovakia. 2012.

[102] Wolfgang Gellerich and Erhard Ploedereder. The evolution of goto usage and its effects on software quality. In *Informatik'99*, pages 380–389. Springer, 1999.

[103] Arthur C Graesser, Danielle S McNamara, Max M Louwerse, and Zhiqiang Cai. Coh-metrix: Analysis of text on cohesion and language. *Behavior Research Methods, Instruments, & Computers*, 36(2):193–202, 2004.

[104] J. Green and M. d'Oliveira. *Testes Estatísticos em Psicologia.* Editorial Estampa, 1989.

[105] R. Green and H. Ledgard. Coding guidelines: Finding the art in the science. *Communications of the ACM*, 54(12):57–63, Dec 2011.

[106] T . R. G. Green. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50(2):93–109, 1977.

[107] Arnold A Griese. The readability of children's literature published in the united states during the period june 1956 through june 1958. 1960.

[108] R.J. Grissom and J.J. Kim. *Effect Sizes for Research: a Broad Practical Approach.* Lawrence Erlbaum Associates, 2005.

[109] David Groome. *An Introduction to Cognitive Psychology: Processes and Disorders.* Psychology Press, 3rd edition, 2014.

[110] R.C. Guimarães and J.A.S Cabral. *Estatística*. McGraw-Hill, 1997.

[111] Yi Guo, Michael Würsch, Emanuel Giger, and Harald C Gall. An empirical validation of the benefits of adhering to the law of demeter. In *2011 18th Working Conference on Reverse Engineering*, pages 239–243. IEEE, 2011.

[112] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

[113] M. H. Halstead. Toward a theoretical basis for estimating programming effort. In *Proceedings of the 1975 Annual Conference*, pages 222–224. ACM, 1975.

[114] Nuzhat J Haneef. Software documentation and readability: a proposed process improvement. *ACM SIGSOFT Software Engineering Notes*, 23(3):75–77, 1998.

[115] Brenda Hannon. Understanding the relative contributions of lower-level word processes, higher-level processes, and working memory to reading comprehension performance in proficient adult readers. *Reading Research Quarterly*, 47(2):125–152, 2012.

[116] Michael Hansen, Robert L Goldstone, and Andrew Lumsdaine. What makes code hard to understand? *arXiv preprint arXiv:1304.5257*, 2013.

[117] Michael E. Hansen, Andrew Lumsdaine, and Robert L. Goldstone. Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 27–38. ACM, 2012.

[118] Will Hayes and James W Over. The personal software process (pspsm): an empirical study of the impact of psp on individual engineers. Technical report, DTIC Document, 1997.

[119] Kevlin Henney. *97 Things Every Programmer Should Know: Collective Wisdom from the Experts*. O'Reilly Media, Inc., 2010.

[120] Pooneh Heydari and A Mehdi Riazi. Readability of texts: Human evaluation versus computer index. *Mediterranean Journal of Social Sciences*, 3(1):177–190, 2012.

[121] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

[122] CA Hoare. Hints on programming language design. Technical report, DTIC Document, 1973.

[123] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. *Nonparametric Statistical Methods*. Wiley, 3rd edition, 2014.

[124] Virginia M. Holmes. Bottom-up processing and reading comprehension in experienced adult readers. *Journal of Research in Reading*, 32(3):309–326, 2009.

[125] Martin E. Hopkins. A case for the goto. In *Proceedings of the ACM Annual Conference - Volume 2*, pages 787–790. ACM, 1972.

[126] Einar W. Hø/st. *Meaningful Method Names*. PhD thesis, Faculty of Mathematics and Natural Sciences at the University of Oslo, 2010.

[127] John Hunt. *Smalltalk and Object Orientation: An Introduction*. Springer Science & Business Media, 2012.

[128] Stoney Jackson, Premkumar Devanbu, and Kwan-Liu Ma. Stable, flexible, peephole pretty-printing. *Science of Computer Programming*, 72(1):40–51, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[129] J.C. Jesuíno. O método experimental nas ciências sociais. In A. Silva and M. Pinto, editors, *Metodologia das Ciências Sociais*, number 10, pages 215–249. Edições Afrontamento, 1999.

[130] George R Johnson. An objective method of determining reading difficulty. *The Journal of Educational Research*, 21(4):283–287, 1930.

[131] Anker Helms Jørgensen. A methodology for measuring the readability and modifiability of computer programs. *BIT Numerical Mathematics*, 20(4):393–405, 1980.

[132] Frederick P. Brooks Jr. No silver bullet essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[133] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE software*, 21(5):88–92, 2004.

[134] Amela Karahasanović, Annette Kristin Levine, and Richard Thomas. Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software*, 80(9):1541–1559, 2007.

[135] B. Katzmarski and R. Koschke. Program complexity metrics and programmer opinions. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 17–26, June 2012.

[136] Panayiota Kendeou, Paul Broek, Anne Helder, and Josefine Karlsson. A cognitive view of reading comprehension: Implications for reading difficulties. *Learning Disabilities Research & Practice*, 29(1):10–16, 2014.

[137] Brian Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1974.

[138] Brian Kernighan and P. J. Plauger. McGraw-Hill, 2nd edition, 1978.

[139] Yunus Emre Keskin. Fluent interface for more readable code, 3 2014.

[140] Omar EM Khalil and Jon D Clark. The influence of programmers' cognitive complexity on program comprehension and modification. *International Journal of Man-Machine Studies*, 31(2):219–236, 1989.

[141] Jeehyoung Kim and Wonshik Shin. How to do random allocation (randomization). *Clin Orthop Surgery*, 6(1):103–109, 2014.

[142] Suntae Kim and Dongsun Kim. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering*, pages 1–40, 2015.

[143] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. Derivation of new readability formulas (automated readability index, fog count and

flesch reading ease formula) fornnavy enlisted personnel. Technical report, DTIC Document, 1975.

[144] Walter Kintsch. *Comprehension: A Paradigm for Cognition*. Cambridge University Press, 1998.

[145] Walter Kintsch and Katherine A Rawson. *Comprehension*. Blackwell Publishing Ltd., 2005.

[146] Roger E Kirk. *Experimental design: Procedures for the Behavioral Sciences*. Sage Publications Inc., 4th edition, 2012.

[147] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Keele Univ. and Durham Univ. Joint Report.

[148] Barbara Kitchenham. Robust statistical methods: Why, what and how: Keynote. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6. ACM, 2015.

[149] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, pages 1–52, 2016.

[150] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 6: Data analysis. *ACM SIGSOFT Software Engineering Notes*, 28(2):24–27, 2003.

[151] Barbara A Kitchenham and Shad Lawrence Pfleeger. Principles of survey research part 2: Designing a survey. *ACM SIGSOFT Softw. Eng Notes*, 27(1):18–20, 2002.

[152] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug 2002.

[153] Harry Dexter Kitson. *The Mind of the Buyer: A Psychology of Selling*, volume 21549. Macmillan, 1921.

[154] George R Klare. The measurement of readability: Useful information for communicators. *ACM Journal of Computer Documentation (JCD)*, 24(3):107–121, 2000.

[155] Hidetaka Kondoh and Kokichi Futatsugi. To use or not to use the goto statement: Programming styles viewed from hoare logic. *Science of Computer Programming*, 60(1):82–116, 2006.

[156] Jagadeesh Kondru. Using part of speech structure of text in the prediction of its readability, 2006.

[157] Robert Lafore. *Object-Oriented Programming in C++*. Sams, 4th edition, 2001.

[158] Kari Laitinen. Estimating understandability of software documents. *ACM SIGSOFT Software Engineering Notes*, 21(4):81–92, 1996.

[159] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42. ACM, 2001.

[160] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501. ACM, 2006.

[161] Paul J Lavrakas. *Encyclopedia of Survey Research Methods: AM*, volume 1. Sage, 2008.

[162] D. Lawrie, H. Feild, and D. Binkley. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.

[163] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.

[164] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 3–12. IEEE Computer Society, 2006.

[165] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[166] Henry F. Ledgard and William C. Cave. Cobol under control. *Commun. ACM*, 19(11):601–608, Nov 1976.

[167] Taek Lee, Jung Been Lee, and Hoh Peter In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and Its Applications*, 7(5):413–422, Sep 2013.

[168] Meir M Lehman and Laszlo A Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[169] Meir M Lehman and Juan F Ramil. Software evolution—background, theory, practice. *Information Processing Letters*, 88(1):33–44, 2003.

[170] Bennet P Lientz. Issues in software maintenance. *ACM Computing Surveys (CSUR)*, 15(3):271–278, 1983.

[171] Jin-Cherng Lin and Kuo-Chiang Wu. A model for measuring software understandability. In *Computer and Information Technology, 2006. CIT'06. The Sixth IEEE International Conference on*, pages 192–192. IEEE, 2006.

[172] Yangchao Liu, Xiaobing Sun, and Yucong Duan. Analyzing program readability based on wordnet. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 27. ACM, 2015.

[173] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, pages 381–384. IEEE, 2003.

[174] M. V. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[175] M. V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells "humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 399–408. IEEE Computer Society, 2004.

[176] Radu Marinescu and Cristina Marinescu. Are the clients of flawed classes (also) defect prone? In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 65–74. IEEE, 2011.

[177] Micah Martin and Robert C Martin. *Agile Principles, Patterns, and Practices in C#*. Pearson Education, 2006.

[178] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[179] William E Martin and Krista D Bridgmon. *Quantitative and Statistical Research Methods: From Hypothesis to Results*. John Wiley & Sons, 2012.

[180] Ludo Max and Patrick Onghena. Some issues in the statistical analysis of completely randomized and repeated measures designs for speech, language, and hearing research. *Journal of Speech, Language, and Hearing Research*, 42(2):261–270, 1999.

[181] Anneliese Mayrhauser von and A Marie Vans. Program unterstanding: A survey. Technical Report TR-CS-94-120, 1994.

[182] S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[183] John H McDonald. *Handbook of Biological Statistics*, volume 2. Sparky House Publishing, 2008.

[184] G Harry McLaughlin. Smog grading: A new readability formula. *Journal of Reading*, 12(8):639–646, 1969.

[185] Kristen L. McMaster, Christine A. Espin, and Paul van den Broek. Making connections: Linking cognitive psychology and intervention research to improve comprehension of struggling readers. *Learning Disabilities Research & Practice*, 29(1):17–24, 2014.

[186] Emma Medford and Sarah P McGeown. The influence of personality characteristics on children's intrinsic reading motivation. *Learning and Individual Differences*, 22(6):786–791, 2012.

[187] Susan A Mengel and Vinay Yerramilli. A case study of the static analysis of the quality of novice student programs. In *ACM SIGCSE Bulletin*, volume 31, pages 78–82. ACM, 1999.

[188] Simone Teresinha Meurer, CB Luft, TR Benedetti, and Giovana Zarpellon Mazo. Validade de construto e consistência interna da escala de autoestima de rosenberg para uma população de idosos brasileiros praticantes de atividades físicas. *Motricidade*, 8(4):5–15, 2012.

[189] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall Professional Technical Reference, 2nd edition, 1997.

[190] Bertrand Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Publishing Company, 1rt edition, 2009.

[191] Richard J Miara, Joyce A Musselman, Juan A Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, Nov 1983.

[192] John C Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.

[193] Peter B Mosenthal and Irwin S Kirsch. A new measure for assessing document complexity: The pmose/ikirsch document readability formula. *Journal of Adolescent & Adult Literacy*, 41(8):638–657, 1998.

[194] Winnie Mucherah and Alyssa Yoder. Motivation for reading and middle school students' performance on standardized testing in reading. *Reading Psychology*, 29(3):214–235, 2008.

[195] John C Munson. *Software Engineering Measurement*. CRC Press, 2003.

[196] Shinichi Nakagawa and Innes C. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 82(4):591–605, 2007.

[197] Rajendar Namani and J Kumar. A new metric for code readability. *IOSR Journal of Computer Engineering (IOSRJCE)*, 6(6):44–48, Nov.-Dec. 2012.

[198] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Scientific Affairs Division, NATO*. NATO Scientific Affairs Division, Oct 1969.

[199] V. Nguyen. Improved size and effort estimation models for software maintenance. In *2010 IEEE International Conference on Software Maintenance*, pages 1–2. IEEE, Sept 2010.

[200] Alexandre de Pontes Nobre and Jerusa Fumagalli de Salles. O papel do processamento léxico-semântico em modelos de leitura. *Arquivos Brasileiros de Psicologia*, 66(2):128–142, 2014.

[201] Paul W Oman and Curtis R Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 244–250. ACM, 1990.

[202] Laura S Pardo. What every teacher needs to know about comprehension. *The Reading Teacher*, 58(3):272–280, 2004.

[203] RuthM. Parker, DavidW. Baker, MarkV. Williams, and JoanneR. Nurss. The test of functional health literacy in adults. *Journal of General Internal Medicine*, 10(10):537–541, 1995.

[204] Janet Peacock and Philip Peacock. *Oxford handbook of medical statistics*. Oxford University Press, 2011.

[205] J Peat and B Barton. Medical statistics: A guide to data analysis and critical appraisal. *Blackwell Publishing*, 2005.

[206] N. Pennington and B. Grabowski. The tasks of programming. In *Psychology of Programming*, Computer and People Series, pages 45–62. Academic Press Ltd., 1990.

[207] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[208] Ricardo Perez-Castillo and Mario Piattini. Model-driven reverse engineering of open source systems. In H. Rahmn and R. D. Sousa, editors, *Information Systems and Technology for Organizational Agility, Intelligence, and Resilience*, pages 139–160. IGI Global, 2014.

[209] Shari Lawrence Pfleeger and Barbara A Kitchenham. Principles of survey research part 1: Turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001.

[210] Aek Phakiti. *Experimental Research Methods in Language Learning*. Bloomsbury Publishing, 2014.

[211] Venkatesh Podugu. Developing a code readability model to improve software quality. *International Journal of Computer Science & Informatics*, 1(2):106–110, 2011.

[212] 2003-2017 Porto: Porto Editora. legibilidade in dicionário infopédia da língua portuguesa sem acordo ortográfico [em linha], 2014.

[213] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 73–82. ACM, 2011.

[214] Roger S Pressman. *Software Engineering: a Practitioner's Approach*. McGraw Hill Higher Education, 7th edition, 2010.

[215] Ken Pugh. *Prefactoring: Extreme Abstraction; Extreme Separation; Extreme Readability*. O'Reilly, 1rt edition, 2005.

[216] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278. IEEE, June 2002.

[217] Jef Raskin. Comments are more important than code. *Queue*, 3(2):64–65, March 2005.

[218] Darrell R Raymond. Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 3–16. IBM Press, 1991.

[219] Janice Redish. Readability formulas have even more limitations than klare discusses. *ACM J. Comput. Doc.*, 24(3):132–137, Aug 2000.

[220] Phillip A Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, Nov 2005.

[221] J.L.P. Ribeiro. *Investigação e Avaliação em Psicologia e Saúde*. CLIMEPSI, 1999.

[222] Talita Vieira Ribeiro and Guilherme Horta Travassos. On the alignment of source code quality perspectives through experimentation: An industrial case. In *Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry*, pages 26–33. IEEE Press, 2015.

[223] K. V. Roberts. The readability of computer programs. *The Computer Bulletin*, 10(4), 1967.

[224] William F Rosenberger and John M Lachin. *Randomization in Clinical Trials: Theory and Practice*. John Wiley & Sons, 2016.

[225] R. F. Rosin. Proceedings of the pl/i forum sponsored by acm sicplan. *PL/1 Bulletin*, (5):15–30, December 1967.

[226] Andrew Rutherford. Long-term memory: Encoding to retrieval. In Nick Braisby and Angus Gellatly, editors, *Cognitive Psychology*, pages 269–306. Oxford University Press, 2005.

[227] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, Jan 1968.

[228] Felice Salviulo and Giuseppe Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-

informed study with students and professional. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 48. ACM, 2014.

[229] A. Sampaio. *Comparação e Classificação de Métodos de Avaliação do Processo do Software Utilizando um Metodologia Numérica e Exploratória*. PhD thesis, Faculty of Engineering at the Minho University, 2004.

[230] Alberto Sampaio. Improving systematic mapping reviews. *SIGSOFT Softw. Eng. Notes*, 40(6):1–8, Nov 2015.

[231] Isabel Braga Sampaio and Luís Barbosa. Software readability practices and the importance of their teaching. In *Information and Communication Systems (ICICS), 2016 7th International Conference on*, pages 304–309. IEEE, April 2016.

[232] Fabio Sani and John Todman. *Experimental Design and Statistics for Psychology: a First Course*. Blackwell Publishing, 2006.

[233] F. Sardanelli and G.D. Leo. *Biostatistics for Radiologists: Planning, Performing, and Writing a Radiologic Study*. Springer Milan, 2009.

[234] Y. Sasaki, Y. Higo, and S. Kusumoto. Reordering program statements for improving readability. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pages 361–364. IEEE, 2013.

[235] G. Scanniello and M. Risi. Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 190–199. IEEE, 2013.

[236] M. Schatzoff, R. Tsao, and R. Wing. An experimental comparison of time sharing and batch processing. *Commun. ACM*, 10(5):261–265, May 1967.

[237] Andreas Schleicher. *Measuring Student Knowledge and Skills: A New Framework for Assessment*. OECD Publishing, 1999.

[238] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An introduction to program comprehension for computer science edu-

cators. In *Proceedings of the 2010 ITiCSE Working Group Reports*, pages 65–86. ACM, 2010.

[239] Martha Ann Scranton. Smog grading: a readability formula by g. harry mclaughlin. Master's thesis, Kansas Stat,e University, 1970.

[240] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003.

[241] Teresa M. Shaft and Iris Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *J. Manage. Inf. Syst.*, 15(1):51–78, Jun 1998.

[242] Zohreh Sharafi, Zéphyrin Soh, Yann-Gäel Guéhéneuc, and Giuliano Antoniol. Women and men–different but equal: On the impact of identifier style on source code reading. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 27–36. IEEE, 2012.

[243] B. A. Sheil. The psychological study of programming. *ACM Comput. Surv.*, 13(1):101–120, March 1981.

[244] Sylvia B Sheppard, Bill Curtis, Phil Milliman, and Tom Love. Modern coding practices and programmer performance. *Computer*, (12):41–49, 1979.

[245] DJ Sheskin. Handbook of parametric and nonparametric statistical procedures 3 edition rc press. *Boca Raton, FL*, 2004.

[246] Kazuyuki Shima, Yasuhiro Takemura, and Ken-ichi Matsumoto. An approach to experimental evaluation of software understandability. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium on*, pages 48–55. IEEE, 2002.

[247] Ben Shneiderman. Experimental testing in programming languages, stylistic considerations and design techniques. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, pages 653–656. ACM, 1975.

[248] Ben Shneiderman. *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers, 1980.

[249] Forrest Shull, Victor Basili, Jeffrey Carver, José Carlos Maldonado, Guilherme Horta Travassos, Manoel Mendonça, and Sandra Fabbri. Replicating software engineering experiments: Addressing the tacit knowledge problem. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium on*, pages 7–16. IEEE, 2002.

[250] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to Advanced Empirical Software Engineering*, volume 93. Springer, 2008.

[251] Luo Si and Jamie Callan. A statistical model for scientific readability. In *Proceedings of the tenth International Conference on Information and knowledge Management*, pages 574–576. ACM, 2001.

[252] HM Sidek and H Ab Rahim. The role of vocabulary knowledge in reading comprehension: A cross-linguistic study. *Procedia-Social and Behavioral Sciences*, 197:50–56, 2015.

[253] Janet Siegmund and Jana Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, pages 1–34, 2014.

[254] Yogesh Singh and Bindu Goel. A step towards software preventive maintenance. *ACM SIGSOFT Software Engineering Notes*, 32(4):10, 2007.

[255] P Sivaprakasam and V Sangeetha. An accurate model of software code readability. In *International Journal of Engineering Research and Technology*, volume 1. IJERT, August 2012.

[256] Dennis D Smith. *Designing Maintainable Software*. Springer Science & Business Media, 1999.

[257] Catherine Snow. *Reading for Understanding: Toward an R&D Program in Reading Comprehension*. Rand Corporation, 2002.

[258] Ian Sommerville. *Software Engineering*. Pearson Education, 8th edition, 2006.

[259] George Spache. A new readability formula for primary-grade reading materials. *The Elementary School Journal*, pages 410–413, 1953.

[260] Diomidis Spinellis. *Code Reading: the Open Source Perspective*. Addison-Wesley Professional, 2003.

[261] Madelyn Carol Square. The validity of three readability formulas in measuring juvenile fiction. 1968.

[262] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[263] A Jackson Stenner. Measuring reading comprehension with the lexile framework. 1996.

[264] Cheryl Stephens. All about readability. *Plain Language Network*, 2000.

[265] Thomas G Sticht and William B Armstrong. Adult literacy in the united states: A compendium of quantitative data and interpretive comments. 1994.

[266] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: Past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[267] Basu Prasad Subedi. Using likert type data in social science research: Confusion, issues and challenges. *International Journal of Contemporary Applied Sciences*, 3(2):36–49, February 2016.

[268] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22Nd International Conference on Program Comprehension*, pages 271–274. ACM, 2014.

[269] BG Tabachnick and LS Fidell. *Using Multivariate Statistics*. Pearson Education, 6th edition, 2013.

[270] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: Bugs or bad comments?*/. *ACM SIGOPS Operating Systems Review*, 41(6):145–158, Dec 2007.

[271] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. Impact of programming features on code readability. *International Journal of Software Engineering & Its Applications*, 7(6), 2013.

[272] Dennis Van Tassel. *Program Style, Design, Efficiency, DeBugging and Testing*. Prentice Hall PTR, 2nd edition, 1978.

[273] Ted Tenny. Program readability: Procedures versus comments. *Software Engineering, IEEE Transactions on*, 14(9):1271–1279, 1988.

[274] Bruce Thompson. Significance, effect sizes, stepwise methods, and other issues: Strong arguments move the field. *The Journal of Experimental Education*, 70(1):80–93, 2001.

[275] Edward Lee Thorndike. *The Teacher's Word Book*, volume 134. Teachers College, Columbia University New York, 1921.

[276] Simon P. Tiffin-Richards and Sascha Schroeder. The component processes of reading comprehension in adolescents. *Learning and Individual Differences*, 42:1–9, 2015.

[277] John B Todman and Pat Dugard. *Single-Case and Small-n Experimental Designs: a Practical Guide to Randomization Tests*. Psychology Press, 2001.

[278] Paolo Tonella and Surafel Lemma Abebe. Code quality from the programmer's perspective. *Proceedings of Science, XII Advanced Computing and Analysis Techniques in Physics Research, Erice, Italy*, 5:153, 2008.

[279] Priyadarshi Tripathy and Kshirasagar Naik. *Software Evolution and Maintenance*. John Wiley & Sons, 2014.

[280] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design Concepts in Programming Languages*. MIT press, 2008.

[281] Yarmouk Uni. Impact of programming features on code readability. *International Journal of Software Engineering and Its Applications*, 7(6):441–458, 2013.

[282] Paul van den Broek and Christine A. Espin. Connecting cognitive theory and assessment: Measuring individual differences in reading comprehension. *School Psychology Review*, 41(3):315–326, 2012.

[283] Paul van Schaik and Matthew Weston. Magnitude-based inference and its application in user research. *International Journal of Human-Computer Studies*, 88:38–50, 2016.

[284] Anneliese Von Mayrhauser and A. Marie Vans. Program understanding: Models and experiments. *Advances in Computers*, 40:1–38, 1995.

[285] Bruce E Wampler. *The Essence of Object-Oriented Programming with Java and UML*. Addison-Wesley, 2002.

[286] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 35–44. IEEE, Oct 2011.

[287] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process*, 26(1):27–49, 2014.

[288] Anthony I. Wasserman. The design of plain: Support for systematic programming. In *Proceedings of the May 19-22, 1980, National Computer Conference*, pages 731–740. ACM, 1980.

[289] Gerald M Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold New York, 1971.

[290] Larry Weissman. Psychological complexity of computer programs: An experimental methodology. *SIGPLAN Not.*, 9:25–36, Jun 1974.

[291] Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma, and CynthiaL Corritore. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11(3):255–282, 1999.

[292] R.R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Academic Press, 2012.

[293] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE software*, 10(1):75, 1993.

[294] Leland Wilkinson. Statistical methods in psychology journals: Guidelines and explanations. *American psychologist*, 54(8):594, 1999.

[295] Niklaus Wirth. Computing science education: the road not taken. In *ITiCSE - In Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, volume 2, pages 1–3, 2002.

[296] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a repli,cation in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. ACM, 2014.

[297] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[298] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, pages 215–223. IEEE Press, 1981.

[299] Shaochun Xu. A cognitive model for program comprehension. In *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*, pages 392–398. IEEE, 2005.

[300] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–471. ACM, 2014.

[301] Edward Yourdon. Programmer attitudes and reactions towards programming productivity techniques. In *Proceedings of the Thirteenth Annual SIGCPR Conference*, SIGCPR '75, pages 72–84. ACM, 1975.

[302] Edward Nash Yourdon. *Classics in Software Engineering*. Yourdon Press, 1979.

[303] Rohana Yusoff and Roziah Mohd Janor. Generation of an interval metric scale to measure attitude. *SAGE Open*, 4(1), 2014.

[304] Beverly L Zakaluk and S Jay Samuels. *Readability: Its Past, Present, and Future*. ERIC, 1988.

[305] Mostafa Zamanian and Pooneh Heydari. Readability of texts: State of the art. *Theory and Practice in Language Studies*, 2(1):43–53, 2012.

[306] Silvia Zuffi, Carla Brambilla, Giordano Beretta, and Paolo Scala. Human computer interaction: Legibility and contrast. In *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, pages 241–246. IEEE, 2007.

[307] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co., 1997.

Apêndice A

Lista de práticas de Legibilidade

Neste apêndice é apresentada uma lista das práticas compiladas através da revisão da literatura obtidas a partir dos estudos empíricos e da bibliografia de profissionais. Algumas poderão parecer repetidas mas de modo a não perder informação pertinente optou-se pela colocação de todas (mesmo as similares).

Descrição	**Indústria**	**Estudos**
ÂMBITO Ideias relacionadas entre si devem estar verticalmente próximas.	Martin[178]	
Declarações das variáveis junto ao local onde são usadas (âmbito reduzido de uma variáve).	Lafore[157], Martin[178], Kimchi[119]	Elshoff[85], Tashtoush[271], Ribeiro[222]
Declarações de constantes junto ao local onde são usadas.		Elshoff[85], Tashtoush[271], Ribeiro[222]
Evitar local de declaração das variáveis sem critério.	Fowler[97]	
Linhas de código relacionadas devem aparecer juntas.	Martin[178]	
Reorganização das instruções dentro de um módulo de modo a reduzir o âmbito da(s) variável(eis).		Weissman[290], Elshoff [85], Sasaki[234], Tashtoush[271], Ribeiro[222]
Mover as instruções para os blocos mais interiores para reduzir âmbito.		Sasaki[234], Tashtoush[271]
Juntar instruções relacionadas entre si reduzir distância entre definição e referência.		Sasaki[234]
BRANCOS Nas expressões uso de espaços brancos, variáveis temporárias e parêntesis.	Spinellis[260]	

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Espaço branco entre os argumentos/parâmetros nas funções	Martin[178]	
Espaço branco para acentuar as precedências dos operadores numa expressão	Martin[178]	
Utilização de espaços para separação dos tokens nas expressões.	Turbak, Gifford[280]	
No operador atribuição, incluir espaço branco antes e depois do operador	Martin[178]	
Separar os operadores com um espaço branco.	Martin[178]	
Cada ideia no código deve ser separada por uma linha em branco.	Martin[178], Kerninghan&Plaugher[178]	
COMENTÁRIOS		
Utilização de comentários: com critério e consistentes.		Weissman[290], Jorgensen[131], Sheppard[244], Woodfield[298], Baecker[10], Haiduc[112], Tashtoush[271], Wang [287], Elshoff [85], Fluri [94], Tan[270], Ribeiro[222], Tenny[273], Namani [197], Borstler[31]
Usar comentário para explicar propósito de cada função.		Ribeiro[222]
Linhas de comentário só usadas em casos específicos.		Ribeiro[222], Namani [197]
Utilização consistente em termos de estilo dos comentários.		Ribeiro[222]
Evitar comentar código.		Ribeiro[222]
Comentários: só numa lingua, em cada ficheiro *header* um comentário de identificação.		Ribeiro[222]
Utilização criteriosa de comentários.	Martin[178]	
CONTROLO DE FLUXO		
Cada bloco de um *if* deve estar numa linha.	Martin[178]	
Evitar o operador ternário quando tamanho excede limite estabelecido.		Ribeiro [222]
Adição do ramo *Senão* (mesmo vazio).		Elshoff[85]
Evitar o excesso de ramificações das decisões.		Weissman[290], Tashtoush[271], Buse[38]
Uso consistente de chavetas para identificação blocos.		Ribeiro[222]
Optar por blocos de uma entrada e uma saída.		Elshoff[85]
Escolha adequada das estruturas de controlo de fluxo.		Jorgensen[131], Chaudhary[48], DeYoung[73]

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Evitar imbricações profundas.	Meyer[189]	
Usar *switch* para reduzir ramificações.		Elshoff [85]
Evitar grandes blocos lógicos condicionais.	Fowler[97]	
A instrução *for* deve ser usada da forma mais simples. Evitar usar sem expressões.	Lafore[157]	
Inclusão de chavetas nos blocos de instruções.	Lafore[157], McConnell[182]	
Reduzir *if/switch* a favor do polimorfismo.	Martin[178], Pepperdine[178]	
ESTRUTURA		
Maximizar o encapsulamento.	Wampler[285]	
Minimizar o acoplamento.	Wampler[285]	
Evitar código idêntico ou semelhante existente em mais de um local.	Fowler[97]	
Utilização de parêntesis nas expressões.		Buse[38]
Evitar o encadeamento dos métodos.		Borstler[31]
Uniformidade na notação,terminologia e simbologia utilizada.		Shneiderman[248]
Usar variáveis de estado para controlar a execução ao longo de blocos.		Elshoff [85]
Controlar tamanho da solução em termos de código.		Ribeiro[222]
Inserção parágrafos nas listagens.		Jorgensen[131]
Usar modularizações de tipo de dado abstrato.		Woodfield[298]
Usar modularização funcional.		Woodfield[298]
Controlo do número de linhas com instruções.		Weissman[290], DeYoung[73], Stamelos[262], Namani[197], Posnett[213]
Controlo do tamanho do programa.		Chaudhary[48], DeYoung[73], Stamelos[262], Haiduc[112], Posnett[213]
Optar pela programação estruturada.		Pennington[207]
Melhorar a apresentação dos programas.		Baecker[10]
Evitar complexidade ciclomática.		Stamelos[262]
Cuidado em manter a qualidade do código estrutural quando se intervem no código.		Stamelos[262]
Software deverá ter um máximo de 3700 linhas de código.		Chhabra[50]
Incorrecta definição/utilização da herança.		[271]
Evitar expressões com *side-effect*.		Dolado[77]
Evitar densidade proposicional.		Collar[57]

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Ordem nos tipos enumerados: alfabética ou de acordo com o que representam.		Butler[42]
Utilização de constantes simbólicas em substituição de constantes numéricas (Princípio da Constante Simbólica)	Meyer[189], Fowler[97], Martin[178]	Ribeiro[222]
Modularização para eliminação de blocos repetidos.		Elshoff[85]
Incorrecta definição/utilização de herança.		Tashtoush[271]
Evitar a utilização de vectores, substituir por contentores.		Tashtoush[271]
Local de declaração das variáveis sem critério.	Fowler[97]	
Organização criativa do código.	Spinellis[260]	
O código deve obedecer à "Lei de Demeter".	Chidamber [51], Fowler [97], Martin[178]	
Evitar o uso de encadeamento de métodos	Fowler [96]	
Redução do número de variáveis temporárias.	Fowler[97]	
Promover utilização de expressões lambda.	Fowler[97]	
Não comentar *bad code*, reescrevê-lo.	Martin[178], Kerninghan&Plaugher[178]	
Espaçamento consistente.		Ribeiro[222]
Usar parêntesis nos operandos.		Ribeiro[222]
Utilização correcta de prefixos e sufixos.		Ribeiro[222]
Utilização de enumerações em lugar de variáveis de estado.	McConnell[182]	
Minimizar o número de identidades.	Jeffries[178]	
Usar as excepções com muito critério e bom senso.	McConnell[182]	
A lógica deve ser simples de modo a tornar difícil a ocultação dos erros.	Stroustrup[178]	
Dependências mínimas.	Stroustrup[178], Thomas[178]	
Testes unitários e de aceitação.	Thomas[178], Jeffries[178], Martin[178]	
Código literado.	Thomas[178]	
Não possuir código duplicado. Controlar e diminuir.	Jeffries[178], Martin[178]	
Expressar todas as ideias da conceção do sistema.	Jeffries[178]	
Usar a regra: *porque* existe, *o que* faz e *como* é usado.	Martin[178]	
Código relacionado com o domínio do problema deve usar termos desse domínio.	Martin[178]	
Evitar as sentinelas como parâmetros.	Martin[178]	
Cada ficheiro não deve ser demasiado grande.	Martin[178]	
Evitar variáveis globais que possam ser modificáveis.	Kimchi[119]	
Evitar a utilização de *set* e *get* para todos os atributos.	Martin[178]	
Evitar aritmética de apontadores dentro de expressões.		Weissman[290]
Eliminação do desvio incondicional *Goto* e preferência pela programação estruturada: as construções de alto nível, condicionais e de ciclo, são mais claras.	Meyer[189], Sommerville[258], Kimchi[119]	Elshoff [85]

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Evitar *GOTO*.		Jorgensen[131], Ribeiro[222]
Break: usar com critério.	Meyer[189]	
FUNCÕES		
Redução dos tamanhos dos métodos (métodos extensos).	Wampler[285], Ølmheim[119], Fowler [97]	
Deve-se explicitar sempre o tipo retornado nas funções.	Lafore[157]	
Sobrecarga de operadores: a implementação de operadores nas classes em vez de métodos que realizem a mesma funcionalidade nas linguagens que o permitam.	Lafore[157]	
Criar funções curtas. Ølmheim sugere um tamanho máximo de 5 a 10 linhas. Kimchi sugere um máximo de 24 linhas.	McConnell[182], Martin[178], Ølmheim[119], Kimchi[119]	
Cada função/método com uma só tarefa/responsabilidade.	Stroustrup[178], Ron Jeffries[178], Martin[178], Ølmheim[119], Pepperdine[119], Kimchi[119]	
As funções devem encontrar-se pela ordem em que são invocadas, a função que chama deve estar antes da função que é chamada.	Martin[178]	
Controlo na passagem de argumentos/parâmetros das funções.		Weissman[290]
Utilização correcta da redefinição de funções (“overriding”).		Buse[38, 39], Tashtoush[271]
As linguagens devem ser estaticamente tipadas.	Meyer[189]	
Controlo número de métodos.		Weissman[290], Namani [197]
Utilização correcta das funções recursivas.		Buse[38, 39], Tashtoush[271]
Evitar argumentos repetidos.		Collar[57]
Rotinas de acesso à variável em vez de verificar directamente a variável.	McConnell[182]	
Redução do número de parâmetros dos métodos/funções. Martin sugere que o número de argumentos de uma função deve tender para zero, sendo já de evitar 3. Kimchi sugere um máximo de 4.	Fowler[97], Martin[178], Kimchi[119]	
INDENTAÇÃO		
Conetor de linha gerado pelos compiladores e combinado com a indentação.	Clifton[53]	

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Usar indentação. Miara sugere 2 a 4 espaços.	Raymond[218], Meyer[189], Spinellis[260], Martin[178]	Jorgensen[131], Hansen[116],Buse[38, 39], Miara[191], Ribeiro [222]
Indentar as linhas na proporção da sua posição na hierarquia.		[38], Miara[191], Tashtoush[271]
Evitar muitos níveis de indentação.		Stamelos[262]
Usar Indentação em expressões lógicas com dois ou mais operadores diferentes.		Ribeiro [222]
IDENTIFICADORES Escolha adequada dos nomes dos identificadores. Nomes descritivos e distinguíveis.	Raymond[218], Meyer[189], Wampler[285], Spinellis[260], Turbak[280], Gifford[280], Thomas[178], Jeffries[178], Martin[178], Pepperdine[119], Ølmheim[119]	
Usar substantivos para os nomes das classes e iniciar os nomes destas com maiúscula; incluir verbos nos nomes dos métodos/funções.	Martin[178]	
Os nomes das classes e dos parâmetros genéricos formais devem estar todos em maiúsculas.	Meyer[189]	
Princípio da utilização de nomes padrão: sempre que aplicável, a utilização de uma terminologia consistente.	Meyer[189]	
Entidades pré-definidas e expressões e os atributos constantes devem começar com uma letra maiúscula e as restantes letras serem minúsculas.	Meyer[189]	
Todos os outros identificadores devem estar em minúsculas. Fazem parte deste grupo, os atributos não constantes, os argumentos formais das funções e entidades locais.	Meyer[189]	
Tamanho adequado do nome do identificador.		Jorgensen[131], DeYoung[73], Tashtoush[271], Buse [38], Shneiderman[248], Scanniello[235], Butler[42], Borstler[33]

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Escolha criteriosa dos nomes dos identificadores tendo em atenção a visão global.		Weissman[290], Jorgensen[131], Wang[287], Tashtoush[271], Buse[38], Caprille[44], Tonella[278]
Usar nomes compridos para variáveis pouco usadas.		Shneiderman[248]
Usar nomes curtos para variáveis locais ou de ciclo.		Shneiderman[248]
Usar mnemónicas para nomes dos variáveis.		Elshoff [85]
Nomes dos identificadores concisos, consistentes e com significado.		Deissenboeck[67], Haiduc[112], Tashtoush[271], Kim[142]
Evitar nomes de identificadores só com uma letra.		Lawrie[164, 165]
Usar abreviatura ou nome completo para nome do identificador.		Lawrie[164, 165]
Capitalizar os nomes dos identificadores apropriadamente.		Butler[42]
Evitar *Underscores* consecutivos no nome do identificador.		Butler[42]
Evitar *Underscores* externos início e fim.		Butler[42]
Os nomes dos identificadores deverão ser compostos por palavras do dicionário, abreviaturas, e acrónimos que são commummente usados na forma não abreviada.		Butler[42]
Evitar o uso de palavras excessivas no nome do identificador: (mais de quatro palavras ou abreviaturas).		Butler[42]
Não usar identificador numérico.		Butler[42]
Não usar identificador demasiado curto.		Butler[42]
Consistência da variável de acordo com todas as suas utilizações.		[271]
Usar nomes para os identificadores de fácil memorização e numa única língua.		Ribeiro[222]
Evitar a presença de identificadores abreviados no código-fonte.		Scanniello[235]
Usar uma e só uma palavra para um conceito e mantê-la ao longo de todo o programa.	Martin[178]	
Não usar nomes de tipos de dados em nomes de métodos.	Fowler[97]	
Evitar nomes de métodos que não descrevem o que faz.	Fowler[97]	
LINHA Limitar o comprimento da instrução (linha).		[248], Benander[21], Tashtoush[271], Buse[38], Namani [197]

Continua na página seguinte

Tabela A.1 – *Continuação da página anterior*

Descrição	**Indústria**	**Estudos**
Linhas em branco a isolar blocos de instruções.		Jorgensen[131], Ribeiro[222], Wang[286, 287], Namani[197], Tashtoush[271], Buse[38]
Quebra de linha a seguir ao ponto e vírgula.		Namani[197]
Usar linhas em branco a seguir à instrução.		Namani [197]
Optar por linhas curtas.		Buse [38, 39]
Comprimento linha máximo 80 carateres.		Ribeiro [222]
Uma declaração por linha.		Ribeiro[222]
Instruções e expressões: uma instrução por linha.	Meyer [189]	Ribeiro[222]
Usar espaçamento horizontal e vertical de forma a delimitar de forma clara segmentos de código.	Kimchi[119]	Wang[286, 287], Hansen[116]
As linhas devem ser curtas. Sugere 120 carateres como máximo.	Martin[178]	
Quebrar expressões maiores, usando intermédias.	Martin[178]	

Apêndice B

Questionário do Estudo Inicial

Legibilidade do código fonte em Programação Orientada aos Objectos

É sabido que a legibilidade do código fonte é um factor importante para a qualidade do software. Este questionário tem por objectivo avaliar a importância do ensino de um dado conjunto de práticas associadas à legibilidade do código fonte em Programação Orientada aos Objectos (POO) aos alunos de cursos de informática do ensino superior.

Público alvo (destinatários): Docentes de cursos de informática do ensino superior que lecionam/leccionaram unidades curriculares na área de POO.

Todos os dados recolhidos serão utilizados única e exclusivamente para fins de investigação, garantindo-se o seu total anonimato e confidencialidade.

O tempo estimado de preenchimento deste questionário é de cerca de 7 minutos.

* Required

Secção 1. Caracterização do inquirido

1. **1. Qual o ciclo de estudos em que maioritariamente leciona/lecionou POO?** *
   Mark only one oval.

   - ◯ Licenciatura
   - ◯ Mestrado
   - ◯ Pós-graduação

2. **Números de edições em que lecciona(ou) POO?** *
   Não necessariamente na mesma unidade curricular.
   Mark only one oval.

   - ◯ 1
   - ◯ 2
   - ◯ 3
   - ◯ 4
   - ◯ 5
   - ◯ 6
   - ◯ 7
   - ◯ 8
   - ◯ 9
   - ◯ 10 ou mais

3. **2. Na avaliação dos trabalhos dos alunos tem em conta a legibilidade do código fonte?** *

   Mark only one oval.

   - ◯ Sim
   - ◯ Não

4. **3. Como é que avalia em média a legibilidade do código fonte produzido pelos alunos?** *

   Mark only one oval.

	1	2	3	4	5	
Muito fraca	◯	◯	◯	◯	◯	Muito boa

Nas duas secções seguintes são apresentados dois conjuntos de práticas identificadas por diferentes autores como podendo influenciar a legibilidade do código fonte, umas positivamente outras negativamente. Na secção 2 são apresentadas as de efeito positivo e na secção 3 as de efeito negativo.

Secção 2. Práticas que poderão afetar positivamente a legibilidade código fonte

5. **Nível de importância que atribui ao seu ensino:** *

   1 - Muito baixo, 2 - Baixo, 3 - Médio, 4 - Alto, 5 - Muito alto

   Mark only one oval per row.

	1	2	3	4	5
1. Limitar o comprimento da instrução (linha).	◯	◯	◯	◯	◯
2. Indentar as linhas na proporção da sua posição na hierarquia.	◯	◯	◯	◯	◯
3. Utilização de parêntesis nas expressões.	◯	◯	◯	◯	◯
4. Uniformidade na notação, terminologia e simbologia utilizada.	◯	◯	◯	◯	◯
5. Quebra de linha a seguir ao ponto e vírgula.	◯	◯	◯	◯	◯
6. Linhas em branco para isolar blocos de instruções.	◯	◯	◯	◯	◯
7. Linhas de código relacionadas devem aparecer juntas.	◯	◯	◯	◯	◯
8. Declarações das variáveis junto ao local onde são usadas.	◯	◯	◯	◯	◯
9. A função que chama deve estar acima da função que é chamada.	◯	◯	◯	◯	◯
10. No operador atribuição, incluir espaço branco antes e depois.	◯	◯	◯	◯	◯
11. Espaço branco entre os	◯	◯	◯	◯	◯

argumentos/parâmetros nas funções.					
12. Espaço branco para acentuar as precedências dos operadores numa expressão.					
13. Utilização de constantes simbólicas em substituição de constantes numéricas.					
14. Utilização de enumerações em lugar de variáveis.					
15. Utilização de chavetas no bloco de ciclo.					
16. Escolha adequada dos nomes para os identificadores tendo em atenção a visão global.					
17. Tamanho adequado do nome do identificador (aprox. 9 a 16 caracteres).					
18. Controlo do uso dos comentários: consistentes com o código e controlados.					
19. Evitar o excesso de ramificações de decisões.					
20. Reorganização das instruções dentro de um módulo de modo a reduzir o âmbito da(s) variável(eis).					
21. Utilização criteriosa de funções recursivas.					
22. Utilizar a redefinição de funções (”overrriding”) quando apropriado.					
23. Preferir contentores a vetores.					
24. Âmbito reduzido das variáveis.					
25. Consistência da variável de acordo com todas as suas utilizações.					

Secção 3. Práticas que poderão afetar negativamente a legibilidade código fonte

6. **Nível de importância que atribui ao seu ensino (de práticas a evitar):** *
   1 - Muito baixo, 2 - Baixo, 3 - Médio, 4 - Alto, 5 - Muito alto
   Mark only one oval per row.

	1	2	3	4	5
1. Métodos extensos.					
2. Métodos com listas de parâmetros extensas.					
3. Código idêntico ou muito semelhante existente em mais de um local.					
4. Nomes de tipos em nomes de métodos.					
5. Nomes de métodos que não descrevem o que faz.					
6. Incorrecta definição/utiização da herança.					
7. Grandes blocos lógicos condicionais.					
8. Local de declaração das variáveis sem critério.					

7. **Existem outras boas práticas que utiliza no ensino que não estão aqui referenciadas? Em caso afirmativo, por favor indique quais.**

8. **Comentários/Sugestões**

Muito obrigada pela colaboração!

Se tiver interesse em conhecer os resultados do estudo pode deixar o email.

9.

Apêndice C

Capa dos Estudos Experimentais

Na duas páginas que se seguem é apresentado a capa e as considerações iniciais que foram entregues aos alunos que participaram no estudo.

ESTUDO

A participação é opcional, ao virar a página aceita participar.
Os resultados serão usados unicamente para fins de investigação.
O estudo é anónimo.

Este estudo está relacionado com a leitura e compreensão de código fonte.

A sua tarefa consiste em **ler com muita atenção** um pedaço de código de modo a que seja capaz de o explicar.

Em seguida, deverá responder a algumas questões sobre esse pedaço de código.

É muito importante que registe nos locais indicados o tempo que necessitou para ler o código e responder às questões.

Por favor não tome notas ou copie os pedaços de código (manual ou eletronicamente) caso contrário as suas respostas serão inúteis para o estudo.

Para cada uma das alíneas seguintes, por favor escolha só uma opção:

1.Sexo?
Feminino ___
Masculino ___

2. Tem alguma dificuldade de leitura, por exemplo, dislexia?
Sim ___
Não ___

3. Já fez APROG?
Sim ___
Não ___

Apêndice D

Snippets

Nas páginas seguintes são apresentados os *snippets* utilizados no estudo. Os *snippets* são listados sequencialmente. Cada um engloba três páginas: na primeira tem o extracto de código a ler, na segunda página estão as questões, e na terceira página está o teste Cloze. No total são apresentados 12 *snippets* diferentes:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * A seguinte função informa o webmaster de um problema inesperado (Exception "ex")
 * com a aplicação que foi disponibilizada (indicado por "aRequest").
 */
public void logAndEmailSeriousProblem(Throwable ex, HttpServletRequest
aRequest)
{
    TroubleTicket troubleTicket = new TroubleTicket(ex, aRequest);

    fLogger.severe("TOP LEVEL CATCHING Throwable.");
    fLogger.severe(troubleTicket.toString());

    log("SERIOUS PROBLEM OCCURRED.");
    log(troubleTicket.toString());

    aRequest.getSession().getServletContext().
        setAttribute(MOST_RECENT_TROUBLE_TICKET, troubleTicket);
    troubleTicket.mailToWebmaster();
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * A seguinte função informa o webmaster de um problema inesperado (Exception "ex")
 * com a aplicação que foi disponibilizada (indicado por "aRequest").
 */
public void logAndEmailSeriousProblem(Throwable ex, HttpServletRequest
aRequest)
{
    TroubleTicket troubleTicket = new TroubleTicket(ex, aRequest);

    fLogger.severe("TOP LEVEL CATCHING Throwable.");
    fLogger.severe(troubleTicket.toString());

    log("SERIOUS PROBLEM OCCURRED.");
    A:_____;

    aRequest.getSession().B:_____.
        setAttribute(MOST_RECENT_TROUBLE_TICKET, troubleTicket);
    troubleTicket.mailToWebmaster();
}
```

Coloque aqui as suas respostas:

A:

B:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Informa o webmaster de um problema inesperado (Exception "ex")
 * com a aplicação que foi disponibilizada (indicado por "aRequest").
 */
public void logAndEmailSeriousProblem(Throwable ex, HttpServletRequest
aRequest)
{
    TroubleTicket troubleTicket = new TroubleTicket(ex, aRequest);

    fLogger.severe("TOP LEVEL CATCHING Throwable.");
    fLogger.severe(troubleTicket.toString());

    log("SERIOUS PROBLEM OCCURRED.");
    log(troubleTicket.toString());

    HttpSession session = aRequest.getSession();
    ServletContext context = session.getServletContext();
    context.setAttribute(MOST_RECENT_TROUBLE_TICKET, troubleTicket);
    troubleTicket.mailToWebmaster();
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * A seguinte função informa o webmaster de um problema inesperado (Exception "ex")
 * com a aplicação que foi disponibilizada (indicado por "aRequest").
 */
public void logAndEmailSeriousProblem(Throwable ex, HttpServletRequest
aRequest)
{
    TroubleTicket troubleTicket = new TroubleTicket(ex, aRequest);

    fLogger.severe("TOP LEVEL CATCHING Throwable.");
    fLogger.severe(troubleTicket.toString());

    log("SERIOUS PROBLEM OCCURRED.");
    A:_____;

    HttpSession session = aRequest.getSession();
    B:_____ = session.C:_____;
    context.setAttribute(MOST_RECENT_TROUBLE_TICKET, troubleTicket);
    troubleTicket.mailToWebmaster();
}
```

Coloque aqui as suas respostas:

A:

B:

C:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar.** Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Aplica as notificações transmitidas ("nStore") para o projeto ("project")
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    /* If project space has not been initialized, there is nothing to do. */
    if (projectSpace != null) {

        /* Get project properties and check if there are notifications. */
        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        NotificationList notifications;

        if (property != null) {
            Value value = property.getValue();

            /* If the project already has notifications                    */
            /* and if transmitted notifications are acknowledged,          */
            /* then remove the transmitted notifications from the project. */
            /* Otherwise, add the transmitted notifications to the project. */
            if (value != null && value instanceof NotificationComposite) {

                NotificationComposite nComposite = value;
                notifications = nComposite.getNotifications();

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
        } else {

            /* If the project did not have notifications yet          */
            /* and if transmitted notifications are not acknowledged, */
            /* then add the transmitted notifications to the project  */
            /* and store them in the NOTIFICATION_COMPOSITE property. */
            if (!nStore.isAcknowledged()) {

                NotificationComposite nComposite =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
        }
    }
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Aplica as notificações transmitidas ("nStore") para o projeto ("project")
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    /* If project space has not been initialized, there is nothing to do. */
    if (projectSpace != null) {

        /* Get project properties and check if there are notifications. */
        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        A:_____;

        if (property != null)) {
            Value value = property.getValue();

            /* If the project already has notifications                    */
            /* and if transmitted notifications are acknowledged,          */
            /* then remove the transmitted notifications from the project. */
            /* Otherwise, add the transmitted notifications to the project. */
            if (value != null && value instanceof NotificationComposite) {

                NotificationComposite nComposite = value;
                B:_____ = C:_____;

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
        } else {

            /* If the project did not have notifications yet          */
            /* and if transmitted notifications are not acknowledged, */
            /* then add the transmitted notifications to the project  */
            /* and store them in the NOTIFICATION_COMPOSITE property. */
            if (!nStore.isAcknowledged()) {

                D:_____ =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
        }
    }
}
```

Coloque aqui as suas respostas:

A: **B:**

C: **D:**

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Aplica as notificações transmitidas ("nStore") para o projeto ("project")
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    /* If projectSpace is not null */
    if (projectSpace != null) {

        /* Define local variables. */
        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        NotificationList notifications;

        if (property != null) {
            Value value = property.getValue();

            /* If value is not null and if nstore is acknowledged,   */
            /* then remove nstore-notifications from notifications.  */
            /* Otherwise, add nstore-notifications to notifications. */
            if (value != null && value instanceof NotificationComposite) {

                NotificationComposite nComposite = value;
                notifications = nComposite.getNotifications();

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
        } else {

            /* If property is not null                  */
            /* and if nStore is not acknowledged,       */
            /* then add nStore-notifications,           */
            /* and set NOTIFICATION_COMPOSITE property. */
            if (!nStore.isAcknowledged()) {

                NotificationComposite nComposite =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
        }
    }
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
//**
 * Aplica as notificações transmitidas ("nStore") para o projeto (“project”)
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    /* If projectSpace is not null */
    if (projectSpace != null) {

        /* Define local variables. */
        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        A:_____;

        if (property != null)) {
            Value value = property.getValue();

            /* If value is not null and if nstore is acknowledged,   */
            /* then remove nstore-notifications from notifications.  */
            /* Otherwise, add nstore-notifications to notifications. */
            if (value != null && value instanceof NotificationComposite) {

                NotificationComposite nComposite = value;
                B:_____ = C:_____;

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
        } else {

            /* If property is not null                  */
            /* and if nStore is not acknowledged,       */
            /* then add nStore-notifications,           */
            /* and set NOTIFICATION_COMPOSITE property. */
            if (!nStore.isAcknowledged()) {

                D:_____ =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
        }
    }
}
```

Coloque aqui as suas respostas:

A: **B:**

C: **D:**

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar.** Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Aplica as notificações transmitidas ("nStore") para o projeto ("project")
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    if (projectSpace != null) {

        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        NotificationList notifications;

        if (property != null) {
            Value value = property.getValue();

            if (value != null && value instanceof NotificationComposite){
                NotificationComposite nComposite = value;
                notifications = nComposite.getNotifications();

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
        } else {

            if (!nStore.isAcknowledged()) {
                NotificationComposite nComposite =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
        }
    }
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Aplica as notificações transmitidas ("nStore") para o projeto ("project")
 * de modo que as notificações conhecidas são excluídas e as outras
 * são adicionadas.
 */
public void apply(Project project, NotificationStore nStore)
{
    ProjectSpace projectSpace = project.eContainer();

    if (projectSpace != null) {

        PropertyManager manager = projectSpace.getPropertyManager();
        StoreProperty property =
            manager.getLocalProperty(NOTIFICATION_COMPOSITE);
        A:_____;

        if (property != null) {
            Value value = property.getValue();

            if (value != null && value instanceof NotificationComposite) {
                NotificationComposite nComposite = value;
                B:_____ = C:_____;

                if (nStore.isAcknowledged()) {
                    notifications.removeAll(nStore.getNotifications());
                } else {
                    notifications.addAll(nStore.getNotifications());
                }
            }
         } else {

            if (!nStore.isAcknowledged()) {
                D:_____ =
                    Factory.createNotificationComposite();
                notifications = nComposite.getNotifications();

                notifications.addAll(nStore.getNotifications());
                manager.setLocalProperty(NOTIFICATION_COMPOSITE, nComposite);
            }
         }
     }
  }
```

Coloque aqui as suas respostas:

A:

B:

C:

D:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    /* Create a new table with the same size and color as uTable. */
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    UColor color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    /* Go through all entries of uTable and copy image information. */
    Cell cell;
    for (UTableCell uCell : uTable.getEntries()) {

        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        /* Check if there is image information to copy. */
        if (children.size() > 0 && children.get(0) instanceof UImage) {

            UImage image = children.get(0);
            Path path = image.getPath();
            USegment segment = path.lastSegment();


            /* Copy the last segment of the image. False and true */
            /* are transformed to "no" and "yes", respectively.   */
            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (segment.startsWith("true")) {
                    cell = new Cell("yes");
                } else {
                     cell = new Cell(segment);
                }
            }

            /* Copy background color of uCell. */
            option = uCell.getBoxModel();
            color = option.getBackgroundColor();
            if (color != null){
                cell.setBackgroundColor(color);
            }
        }
         table.addCell(cell);

    }
    return table;
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    /* Create a new table with the same size and color as uTable. */
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    Ocular color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    /* Go through all entries of uTable and copy image information. */
    A:_______;
    for (UTableCell uCell : uTable.getEntries()) {

        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        /* Check if there is image information to copy. */
        if (children.size() > 0 && children.get(0) instanceof UImage) {

            B:____ = C:_____________________;
            Path path = image.getPath();
            USegment segment = path.lastSegment();

            /* Copy the last segment of the image. False and true */
            /* are transformed to "no" and "yes", respectively.   */
            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (D:_____________________________) {
                    cell = new Cell("yes");
                } else {
                    cell = new Cell(segment);
                }
            }

            /* Copy background color of uCell. */
            option = uCell.getBoxModel();
            color = option.getBackgroundColor();
            if (E:_____ != F:____){
                cell.setBackgroundColor(color);
            }
        }
        table.addCell(cell);
    }
    return table;
}
```

Coloque aqui as suas respostas:

A: **B:**

C: **D:**

E: **F:**

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    /* Define local variables. */
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    UColor color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    Cell cell;
    for (UTableCell uCell : uTable.getEntries()) {

        /* Define local variable. */
        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        /* If children size larger than 0 and first children is an image. */
       if (children.size() > 0 && children.get(0) instanceof UImage) {

            UImage image = children.get(0);
            Path path = image.getPath();
            USegment segment = path.lastSegment();


            /* If segment starts with false, then create a cell with "no".*/
            /* Otherwise, if segment starts with true, then a create cell */
            /* with "yes". Otherwise create cell with segment.            */
            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (segment.startsWith("true")) {
                    cell = new Cell("yes");
                } else {
                   cell = new Cell(segment);
                }
            }

            /* Copy background color. */
            option = uCell.getBoxModel();
            color = option.getBackgroundColor();

           if (color != null) {
                  cell.setBackgroundColor(color);
           }
        }
        table.addCell(cell);
    }
    return table;
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    /* Define local variables. */
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    UColor color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    A:_______;
    for (UTableCell uCell : uTable.getEntries()) {

        /* Define local variable. */
        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        /* If children size larger than 0 and first children is an image. */
        if (children.size() > 0 && children.get(0) instanceof UImage) {

            B:____ = C:_________________;
            Path path = image.getPath();
            USegment segment = path.lastSegment();

            /* If segment starts with false, then create a cell with "no". */
            /* Otherwise, if segment starts with true, then a create cell  */
            /* with "yes". Otherwise create cell with segment.             */
            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (D:_____________________) {
                    cell = new Cell("yes");
                } else {
                    cell = new Cell(segment);
                }
            }

            /* Copy background color. */
            option = uCell.getBoxModel();
            color = option.getBackgroundColor();
            if (E:_____ != F:____){
                cell.setBackgroundColor(color);
            }
        }
        table.addCell(cell);
    }
    return table;
}
```

Coloque aqui as suas respostas:

A: **B:**

C: **D:**

E: **F:**

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue de modo que seja capaz de o explicar. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    UColor color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    Cell cell;
    for (UTableCell uCell : uTable.getEntries()) {

        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        if (children.size() > 0 && children.get(0) instanceof UImage) {

            UImage image = children.get(0);
            Path path = image.getPath();
            USegment segment = path.lastSegment();


            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (segment.startsWith("true")) {
                    cell = new Cell("yes");
                } else {
                    cell = new Cell(segment);
                }
            }

            option = uCell.getBoxModel();
            color = option.getBackgroundColor();
            if (color != null) {
               cell.setBackgroundColor(color);
            }
         }
        table.addCell(cell);
    }
    return table;
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * As imagens de uma tabela unificada ("uTable") são copiadas para uma tabela
 * simplificada do tipo "yes/no".
 */
protected Table writeUTable(UTable uTable)
{
    Table table = new Table(uTable.getColumnsCount());
    BoxModelOption option = uTable.getBoxModel();
    UColor color = option.getBorderColor();
    table.setBorderColor(new Color
        (color.getRed(),color.getGreen(),color.getBlue()));

    A:_______;
    for (UTableCell uCell : uTable.getEntries()) {

        UParagraph paragraph = uCell.getContent();
        UChildren children = paragraph.getChildren();

        if (children.size() > 0 && children.get(0) instanceof UImage) {

            B:____ = C:_________________;
            Path path = image.getPath();
            USegment segment = path.lastSegment();

            if (segment.startsWith("false")) {
                cell = new Cell("no");
            } else {
                if (D:_____________________) {
                    cell = new Cell("yes");
                } else {
                    cell = new Cell(segment);
                }
            }

            option = uCell.getBoxModel();
            color = option.getBackgroundColor();
            if (E:_____ != F:____){
                cell.setBackgroundColor(color);
            }
        }
        table.addCell(cell);
    }
    return table;
}
```

Coloque aqui as suas respostas:

A: **B:**

C: **D:**

E: **F:**

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Gera uma string ("builder") para um dado movimento ("move") de xadrez no
 * PGN (Portable Game Notation). O movimento inclui o número do movimento e todas
 * as notações NAG (Numeric Annotation Glyph) usadas em PGN.
 */
public static boolean getMove(StringBuilder builder, Move move) {

    boolean result = false;
    int moveNumber = move.getFullMoveCount();

    builder.append(moveNumber).append(move.isWhitesMove() ? ". " : "... ").
        append(move.toString());

    for (SublineNode subline : move.getSublines()) {
        result = true;
        builder.append(" (");
        appendSubline(builder, subline);
        builder.append(")");
    }

    for (Comment comment : move.getComments()) {
        builder.append(" {").append(comment.getText()).append("}");
    }

    for (Nag nag : move.getNags()) {
        builder.append(" ").append(nag.getNagString());
    }

    builder.append(" {").append(move.getTimeTakenForMove().getText()).append("}");

    return result;
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos/partes principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ____
Difícil ____
Neutro ____
Fácil ____
Muito fácil ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Gera uma string ("builder") para um dado movimento ("move") de xadrez no
 * PGN (Portable Game Notation). O movimento inclui o número do movimento e todas
 * as notações NAG (Numeric Annotation Glyph) usadas em PGN.
 */
public static boolean getMove(StringBuilder builder, Move move) {

    boolean result = false;
    int moveNumber = move.getFullMoveCount();

    builder.append(moveNumber).A:_____ ? ". " : "... ").
        append(move.toString());

    for (SublineNode subline : move.getSublines()) {
        result = true;
        builder.append(" (");
        appendSubline(builder, subline);
        builder.append(")");
    }

    for (Comment comment : move.getComments()) {
        builder.append(" {").append(comment.getText()).append("}");
    }

    for (B:_____) {
        builder.append(" ").append(nag.getNagString());
    }

    builder.append(" {").append(move.getTimeTakenForMove().getText()).append("}");

    return result;

 }
```

Coloque aqui as suas respostas:

A:

B:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Gera uma string ("builder") para um dado movimento ("move") de xadrez no
 * PGN (Portable Game Notation). O movimento inclui o número do movimento e todas
 * as notações NAG (Numeric Annotation Glyph) usadas em PGN.
 */
public static boolean getMove(StringBuilder builder, Move move) {

    boolean result = false;
    int moveNumber = move.getFullMoveCount();

    builder.append(moveNumber);
    builder.append(move.isWhitesMove() ? ". " : "... ");
    builder.append(move.toString());

    for (SublineNode subline : move.getSublines()) {
        result = true;
        builder.append(" (");
        appendSubline(builder, subline);
        builder.append(")");
    }

    for (Comment comment : move.getComments()) {
        builder.append(" {");
        builder.append(comment.getText());
        builder.append("}");
    }

    for (Nag nag : move.getNags()) {
        builder.append(" ");
        builder.append(nag.getNagString());
    }

    builder.append(" {");
    TimeTakenForMove timeTaken = move.getTimeTakenForMove();
    builder.append(timeTaken.getText());
    builder.append("}");

    return result;
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Gera uma string (“builder”) para um dado movimento (“move”) de xadrez no
 * PGN (Portable Game Notation). O movimento inclui o número do movimento e todas
 * as notações NAG (Numeric Annotation Glyph) usadas em PGN.
 */
public static boolean getMove(StringBuilder builder, Move move) {

    boolean result = false;
    int moveNumber = move.getFullMoveCount();

    builder.append(moveNumber);
    builder.A:_____ ? ". " : "... ");
    builder.append(move.toString());

    for (SublineNode subline : move.getSublines()) {
        result = true;
        builder.append(" (");
        appendSubline(builder, subline);
        builder.append(")");
    }

    for (Comment comment : move.getComments()) {
        builder.append(" {");
        builder.append(comment.getText());
        builder.append("}");
    }

    for (B:_____) {
        builder.append(" ");
        builder.append(nag.getNagString());
    }

    builder.append(" {");
    TimeTakenForMove timeTaken = move.getTimeTakenForMove();
    builder.append(timeTaken.getText());
    builder.append(“}");

    return result;
}
```

Coloque aqui as suas respostas:

A:

B:

REGISTE A HORA (hora : m

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue **de modo que seja capaz de o explicar**. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Encurta o texto ("textValue") para que sua largura em pixels não exceda a
 * largura do controlo ("control"). Para isso, o método shortenText
 * substitui tantos caracteres quantos os necessários no centro do texto
 * original com o carácter reticências que está na constante ("ELLIPSIS").
 */
public static String shortenText(String textValue, Control control)
{
    GraphicsContext gc = new GraphicsContext(control);
    int maxExtent = gc.textExtent(textValue).x;
    int maxWidth = control.getBounds().width - 5;

    int length = textValue.length();
    int start = length/2;
    int end = length/2 + 1;

    while (start >= 0 && end < length) {
        String s1 = textValue.substring(0, start);
        String s2 = textValue.substring(end, length);
        String s = s1 + ELLIPSIS + s2;

        int l = gc.textExtent(s).x;

        if (l < maxWidth) {
            gc.dispose();
            return s;
        }
        start--;
        end++;
    }
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Encurta o texto ("textValue") para que sua largura em pixels não exceda a
 * largura do controlo ("control"). Para isso, o método shortenText
 * substitui tantos caracteres quantos os necessários no centro do texto
 * original com o carácter reticências que está na constante ("ELLIPSIS").
 */
public static String shortenText(String textValue, Control control)
{
    GraphicsContext gc = new GraphicsContext(control);
    int maxExtent = gc.textExtent(textValue).x;
    int maxWidth = A:_____;

    int length = textValue.length();
    int start = length/2;
    int end = length/2 + 1;

    while (B:_____) {
        String s1 = textValue.substring(0, start);
        String s2 = textValue.substring(end, length);
        String s = s1 + ELLIPSIS + s2;

        int l = gc.textExtent(s).x;
        if (l < maxWidth) {
            gc.dispose();
            return s;
        }
        start--;
        end++;
    }
}
```

Coloque aqui as suas respostas:

A:

B:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

3. Por favor leia cuidadosamente a secção de código que se segue de modo que seja capaz de o explicar. Gaste o tempo que necessitar antes de passar ao ponto seguinte.

```
/**
 * Encurta o texto TextValue para que sua largura em pixels não exceda a
 * largura do controlo ("control"). Para isso, o método shortenText
 * substitui tantos caracteres quantos os necessários no centro do texto
 * original com o carácter reticências na constante ("ELLIPSIS" = "...").
 *
public static String shortenText(String textValue, Control control)
{
    GraphicsContext gc = new GraphicsContext(control);
    Extent extent = gc.textExtent(textValue);
    int maxExtent = extent.x;
    Bounds bounds = control.getBounds();
    int maxWidth = bounds.width - 5;

    int length = textValue.length();
    int start = length/2;
    int end = length/2 + 1;

    while (start >= 0 && end < length) {
        String s1 = textValue.substring(0, start);
        String s2 = textValue.substring(end, length);
        String s = s1 + ELLIPSIS + s2;

        extent = gc.textExtent(s);
        int l = extent.x;

        if (l < maxWidth) {
            gc.dispose();
            return s;
        }
        start--;
        end++;
    }
}
```

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

4. Por favor, sumarie muito sucintamente, por palavras suas, os 3 passos principais de execução da função estudada. Se considerar que existem mais ou menos, então descreva esses.

5. Baseado na sua experiência de programação, como classifica a legibilidade do pedaço de código anterior? Por favor escolha só uma das opções seguintes:

Muito difícil ___
Difícil ___
Neutro ___
Fácil ___
Muito fácil ___

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

6. Por favor reconstrua as partes de código que faltam nos locais marcados da melhor forma que conseguir. Coloque as suas respostas a seguir ao código no local indicado.

```
/**
 * Encurta o texto ("textValue") para que sua largura em pixels não exceda a
 * largura do controlo ("control"). Para isso, o método shortenText
 * substitui tantos caracteres quantos os necessários no centro do texto
 * original com o carácter reticências que está na constante ("ELLIPSIS").
 */
public static String shortenText(String textValue, Control control)
{
    GraphicsContext gc = new GraphicsContext(control);
    Extent extent = gc.textExtent(textValue);
    int maxExtent = extent.x;
    Bounds A:_____ = B:__________________;
    int maxWidth = C:_______________;

    int length = textValue.length();
    int start = length/2;
    int end = length/2 + 1;

    while (D:_____________________________) {
        String s1 = textValue.substring(0, start);
        String s2 = textValue.substring(end, length);
        String s = s1 + ELLIPSIS + s2;

        extent = gc.textExtent(s);
        int l = extent.x;

        if (l < maxWidth) {
            gc.dispose();
            return s;
        }
        start--;
        end++;
    }
}
```

Coloque aqui as suas respostas:

A:

B:

C:

D:

REGISTE A HORA (hora : minutos : segundos) ____ : ____ : ____

Apêndice E

Resultados do Estudo Exp. 1

Neste apêndice é apresentada uma tabela que inclui os resultados válidos do estudo experimental 1. A tabela inclui os *snippets* S1, S4 e S5 para os dois tratamentos, com encadeamento de métodos (MC) e sem encadeamento de métodos (NMC).

Duração				Cloze				Escala		
BLOCO	Tratamento	Resultado		BLOCO	Tratamento	Resultado		BLOCO	Tratamento	Resultado
S1	mc	206		S1	mc	0.5		S1	mc	4
S1	mc	400		S1	mc	1		S1	mc	1
S1	mc	537		S1	mc	0.5		S1	mc	1
S1	mc	178		S1	mc	1		S1	mc	1
S1	mc	188		S1	mc	0.5		S1	mc	2
S1	mc	62		S1	mc	0		S1	mc	2
S1	mc	71		S1	mc	0.5		S1	mc	3
S1	mc	114		S1	mc	0.5		S1	mc	2
S1	mc	147		S1	mc	1		S1	mc	3
S1	nmc	167		S1	nmc	0		S1	nmc	2
S1	nmc	470		S1	nmc	1		S1	nmc	2
S1	nmc	91		S1	nmc	1		S1	nmc	5
S1	nmc	640		S1	nmc	0.7		S1	nmc	2
S1	nmc	159		S1	nmc	0.3		S1	nmc	4
S1	nmc	331		S1	nmc	0.3		S1	nmc	2
S1	nmc	580		S1	nmc	0.3		S1	nmc	2
S1	nmc	547		S1	nmc	0.3		S1	nmc	1
S1	nmc	30		S1	nmc	0.7		S1	nmc	3
S4	mc	213		S4	mc	0.5		S4	mc	3
S4	mc	161		S4	mc	0.5		S4	mc	2
S4	mc	188		S4	mc	0.5		S4	mc	2
S4	mc	668		S4	mc	1		S4	mc	2
S4	mc	1136		S4	mc	1		S4	mc	2
S4	mc	237		S4	mc	1		S4	mc	4
S4	mc	1020		S4	mc	1		S4	mc	1
S4	mc	163		S4	mc	1		S4	mc	2
S4	mc	385		S4	mc	1		S4	mc	3
S4	nmc	268		S4	nmc	1		S4	nmc	1
S4	nmc	89		S4	nmc	1		S4	nmc	2
S4	nmc	120		S4	nmc	1		S4	nmc	3
S4	nmc	372		S4	nmc	1		S4	nmc	2
S4	nmc	441		S4	nmc	1		S4	nmc	2
S4	nmc	112		S4	nmc	1		S4	nmc	5
S4	nmc	298		S4	nmc	1		S4	nmc	2
S4	nmc	241		S4	nmc	1		S4	nmc	3
S4	nmc	395		S4	nmc	0.5		S4	nmc	2
S5	mc	371		S5	mc	0.5		S5	mc	3
S5	mc	128		S5	mc	1		S5	mc	2
S5	mc	533		S5	mc	0.5		S5	mc	2
S5	mc	315		S5	mc	0.5		S5	mc	2

S5	mc	347		S5	mc	1		S5	mc	4
S5	mc	212		S5	mc	1		S5	mc	3
S5	mc	300		S5	mc	0.5		S5	mc	4
S5	mc	147		S5	mc	0.5		S5	mc	3
S5	mc	80		S5	mc	1		S5	mc	3
S5	nmc	171		S5	nmc	0.25		S5	nmc	2
S5	nmc	143		S5	nmc	0.5		S5	nmc	2
S5	nmc	455		S5	nmc	0.5		S5	nmc	4
S5	nmc	229		S5	nmc	0.75		S5	nmc	2
S5	nmc	232		S5	nmc	1		S5	nmc	2
S5	nmc	585		S5	nmc	0.75		S5	nmc	2
S5	nmc	345		S5	nmc	0.5		S5	nmc	2
S5	nmc	236		S5	nmc	1		S5	nmc	4
S5	nmc	241		S5	nmc	1		S5	nmc	2

Apêndice F

Resultados do Estudo Exp. 2

Neste apêndice é apresentada uma tabela que inclui os resultados válidos do estudo experimental 2. A tabela inclui os *snippets* S2 e S3 para os três tratamentos, "bons" comentários (GC), "maus" comentários (BC) e sem comentários (NC).

Duração				Cloze				Escala		
BLOCO	Tratamento	Resultado		BLOCO	Tratamento	Resultado		BLOCO	Tratamento	Resultado
S2	bc	309		S2	bc	0.5		S2	bc	1
S2	bc	445		S2	bc	0		S2	bc	2
S2	bc	144		S2	bc	0		S2	bc	1
S2	bc	82		S2	bc	0.25		S2	bc	4
S2	bc	287		S2	bc	1		S2	bc	2
S2	bc	125		S2	bc	0.5		S2	bc	1
S2	bc	275		S2	bc	0.25		S2	bc	2
S2	bc	161		S2	bc	0		S2	bc	2
S2	bc	256		S2	bc	1		S2	bc	4
S2	bc	208		S2	bc	0.5		S2	bc	4
S2	bc	135		S2	bc	0.5		S2	bc	3
S2	gc	250		S2	gc	0		S2	gc	4
S2	gc	226		S2	gc	0		S2	gc	2
S2	gc	240		S2	gc	0.75		S2	gc	2
S2	gc	150		S2	gc	1		S2	gc	2
S2	gc	149		S2	gc	0		S2	gc	2
S2	gc	345		S2	gc	0		S2	gc	2
S2	gc	263		S2	gc	0.75		S2	gc	2
S2	gc	231		S2	gc	0.25		S2	gc	2
S2	gc	63		S2	gc	0.75		S2	gc	3
S2	gc	430		S2	gc	0.25		S2	gc	2
S2	gc	225		S2	gc	0		S2	gc	1
S2	nc	206		S2	nc	0.25		S2	nc	1
S2	nc	128		S2	nc	0.25		S2	nc	1
S2	nc	338		S2	nc	0.25		S2	nc	3
S2	nc	227		S2	nc	0.75		S2	nc	1
S2	nc	527		S2	nc	0.25		S2	nc	2
S2	nc	253		S2	nc	0.5		S2	nc	4
S2	nc	55		S2	nc	1		S2	nc	4
S2	nc	140		S2	nc	0		S2	nc	2
S2	nc	132		S2	nc	0.75		S2	nc	3
S2	nc	280		S2	nc	1		S2	nc	2
S2	nc	138		S2	nc	0.5		S2	nc	3
S3	bc	229		S3	bc	0.66667		S3	bc	3
S3	bc	92		S3	bc	0.5		S3	bc	4
S3	bc	71		S3	bc	0		S3	bc	3

S3	bc	183		S3	bc	0		S3	bc	3
S3	bc	165		S3	bc	0.5		S3	bc	5
S3	bc	190		S3	bc	0.5		S3	bc	2
S3	bc	244		S3	bc	1		S3	bc	1
S3	bc	124		S3	bc	0.66667		S3	bc	5
S3	bc	168		S3	bc	0.16667		S3	bc	2
S3	bc	77		S3	bc	0.83333		S3	bc	3
S3	bc	180		S3	bc	0.5		S3	bc	4
S3	gc	240		S3	gc	0.5		S3	gc	2
S3	gc	216		S3	gc	0.5		S3	gc	3
S3	gc	376		S3	gc	1		S3	gc	4
S3	gc	121		S3	gc	1		S3	gc	2
S3	gc	295		S3	gc	0.6		S3	gc	3
S3	gc	197		S3	gc	0.33333		S3	gc	2
S3	gc	391		S3	gc	1		S3	gc	2
S3	gc	506		S3	gc	0		S3	gc	4
S3	gc	147		S3	gc	0.66667		S3	gc	5
S3	gc	115		S3	gc	1		S3	gc	3
S3	gc	187		S3	gc	0.66667		S3	gc	2
S3	nc	200		S3	nc	1		S3	nc	3
S3	nc	196		S3	nc	0.83333		S3	nc	4
S3	nc	446		S3	nc	1		S3	nc	2
S3	nc	570		S3	nc	0.33333		S3	nc	1
S3	nc	304		S3	nc	1		S3	nc	3
S3	nc	268		S3	nc	0.83333		S3	nc	3
S3	nc	423		S3	nc	0.33333		S3	nc	4
S3	nc	110		S3	nc	1		S3	nc	1
S3	nc	401		S3	nc	0.83333		S3	nc	1
S3	nc	156		S3	nc	1		S3	nc	2
S3	nc	180		S3	nc	0.8		S3	nc	3