Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise do impacto na compreensão de programas Java com a introdução de expressões lambda

Walter Lucas Monteiro de Mendonça

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise do impacto na compreensão de programas Java com a introdução de expressões lambda

Walter Lucas Monteiro de Mendonça

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Gustavo Pinto
FC/UFPA

Prof. Dr. Eduardo Aranha
DIMAP/UFRN

Prof.a Dr.a Edna Dias Canedo
CIC/UnB

Prof. Dr. Bruno Macchiavello
Coordenador do Programa de Pós-graduação em Informática

Brasília, 15 de Abril de 2019

Dedicatória

Dedico este trabalho à minha família e, em especial, à meu avô *Luiz Alves Monteiro* (*in memorian*), minha mãe *Jaidete Monteiro de Mendonça* (*in memorian*) e ao meu pai *Walter Pereira de Mendonça* por todos os ensinamentos, amor e carinho nas horas mais difíceis.

Agradecimentos

A presente dissertação de mestrado não poderia chegar a bom porto sem o precioso apoio de várias pessoas. Em primeiro lugar, não posso deixar de agradecer ao meu orientador, Professor Rodrigo Bonifácio, por toda a paciência, empenho e sentido prático com que sempre me orientou neste trabalho e em todos aqueles que realizei durante o mestrado. Muito obrigado por me corrigir quando necessário sem nunca ter me desmotivado. Desejo agradecer também a professora *Edna Dias Canedo* e ao professor *Eduardo Monteiro* por toda ajuda e paciência.

Desejo igualmente agradecer a todos os meus colegas do Mestrado em Informática e Departamento de Ciência da Computação, especialmente a *Nilson Guerin, Jeremias Gomes, Guilherme Branco, Lucas Souza, Vitor Lopes, Leomar Camargo, Ranyelson Neres, Luís Amaral, João Sousa, Dário Santos, Matheus Sobrinho, Daniel Tavares, Guilherme Vergara, Vinicius Coutinho*, cujo apoio e amizade estiveram presentes em todos os momentos.

Desejo agradecer a todos os professores pela paciência durante as matérias cursadas no Mestrado. Agradeço aos funcionários do Departamento de Ciência da Computação que foram sempre prestáveis e que me ajudaram a ultrapassar um grande obstáculo. Por último, quero agradecer à minha família e amigos pelo apoio incondicional, especialmente ao meu pai e ao meu avô *Luiz Alves Monteiro* (*in memorian*) pelo carinho e ensinamentos com muita sabedoria.

Resumo

As expressões lambda foram introduzidas na linguagem Java com o intuito de facilitar o estilo de programação funcional e com o passar do tempo o número de desenvolvedores que utilizam os novos recursos vêm crescendo. Trabalhos recentes sem o uso de avaliações rigorosas sugerem que a adoção de expressões lambda leva a um ganho direto na legibilidade de código. Todavia, existem alguns fatores impeditivos para a aplicação de ferramentas de transformação, o que incluem sugestões inadequadas que não levam a resultados satisfatórios.

Esta pesquisa teve como objetivo realizar uma investigação empírica para avaliar se a adoção de expressões lambda ocasiona melhorias na compreensão do programa, um dos benefícios esperados pelo uso da nova construção em Java e em quais situações deve ser aplicada. Foi realizado um estudo de métodos mistos, no qual foi elaborado um survey, para captar a percepção dos desenvolvedores em relação ao impacto na legibilidade de trechos de código com a adoção de expressões lambda, e foram realizados cálculos de métricas extraídas diretamente do código fonte, com o intuito de comparar a interpretação dos dados extraídos do survey.

O estudo empírico realizado mostra cenários onde os desenvolvedores percebem uma melhora na compreensão do código e cenários onde a transformação não ocasiona melhorias. Acredita-se que através desses resultados os desenvolvedores de bibliotecas de refatoração poderão melhorar suas ferramentas. Além de melhorias a legibilidade foi possível inferir que adoção de expressões lambda podem reduzir a complexidade e o tamanho dos programas.

Palavras-chave: Compreensão de Programas, Expressões Lambda, Linguagem de Programação Java

Abstract

Lambda expressions were introduced in the Java language in order to facilitate the functional programming style and with the passage of time the number of developers using the new features has been growing. Recent works without the use of rigorous evaluations suggest that the adoption of lambda expressions leads to a direct gain in code readability. However, there are some hindering factors for the application of transformation tools, which include inadequate suggestions that do not lead to satisfactory results.

The aim of this research was to evaluate whether the adoption of lambda expressions leads to improvements in the understanding of the program, one of the benefits expected from the use of the new Java construct and in which situations it should be applied. A study of mixed methods was carried out, in which a survey was elaborated to capture the perception of the developers in relation to the impact on the readability of code snippets with the adoption of lambda expressions, and calculations were made of metrics extracted directly from the source code, with the purpose of comparing the interpretation of the data extracted from the survey.

The empirical study shows scenarios where developers perceive an improvement in code comprehension and scenarios where transformation does not lead to improvements. It is believed that through these results, developers of refactoring libraries will be able to improve their tools. In addition to readability improvements it was possible to infer that adopting lambda expressions can reduce the complexity and size of programs.

Keywords: Program Comprehension, Lambda Expressions, Java Programming Language

Sumário

Lista de Figuras

Lista de Tabelas

Lista de códigos fonte

Capítulo 1

Introdução

A manutenção de um *software* consiste em alterações que podem ser motivadas por diferentes fatores, tais como adaptações no negócio, mudanças nos requisitos, transições arquiteturais (como o uso de ambientes em nuvem), necessidades de melhorias de qualidade (como refatoramento), entre outros [46].

Aspectos técnicos pontuais também podem ser relevantes, uma vez que cenários de evolução de *softwares* podem ser motivados a partir da evolução das linguagens de programação utilizadas, que passam a contar com novas construções. Eventualmente, algumas construções recentes de uma linguagem de programação passam a coexistir com recursos e idiomas obsoletos e que raramente são removidos dos programas [40]. É importante destacar que, além da inclusão de novas funcionalidades (ou alteração de funcionalidades existentes), as atividades de manutenção são fundamentais para aumentar o tempo de vida útil do *software*. A evolução dos sistemas através da manutenção, pode ser influenciada pela complexidade em que um código pode ser lido e compreendido, podendo afetar a qualidade, a manutenção e outros fatores que são características essenciais para permitir que um software seja mantido—inclusive por uma equipe que não foi responsável pelo desenvolvimento das primeiras versões do *software* [17].

É possível realizar algumas melhorias de código utilizando suporte ferramental de *software*. Nesse caso, os desenvolvedores podem utilizar ferramentas disponibilizadas pelas *integrated development environment (IDEs)* como Eclipse e Itellj e bibliotecas de refatoração que realizam as transformações de código legado, inclusive para rejuvenescer um software [36]. Esse tipo de transformação faz com que um programa passe a utilizar novos recursos disponibilizados por uma versão mais recente de uma linguagem de programação. Essas ferramentas são fundamentais para que os *softwares* evoluam com facilidade através da realização de transformações de programas de forma automatizada [36, 20].

1.1 Estilo Funcional de Programação em Java

A programação funcional é um paradigma de programação que reforça o uso de funções sem efeito colateral como principal unidade de decomposição—com o intuito de evitar estados globais e dados mutáveis nos programas [27]. Apesar de existirem linguagens essencialmente funcionais como Haskell e ML, algumas linguagens orientadas a objeto incluíram suporte ao estilo funcional como por exemplo, Scala, C#3 e C++11. Mais recententemente, o estilo funcional de programação foi introduzido na linguagem Java em sua versão 8. A intenção dos designers da linguagem Java com a adição de expressões lambda foi facilitar o estilo de programação funcional e a parametrização de comportamento como, sendo atualmente possível passar funções anônimas (expressões lambda) como argumentos para outras funções e diminuir a quantidade de declaração de classes anônimas nos programas que podem também ser substituídas por expressões lambda [55].

1.2 Problema de Pesquisa

Considerando estudos anteriores sobre a adoção de expressões lambda em Java [34, 52, 53], foi possível perceber que o número de desenvolvedores que estão adotando as novas construções da linguagem Java relacionadas ao estilo funcional vem crescendo com o passar do tempo. Por outro lado, pesquisas recentes indicam que existem situações em que a mera transformação de código legado para usar *expressões lambda* em Java não leva a resultados satisfatórios [36], mesmo considerando (a) as restrições descritas na literatura e (b) estudos recentes que, mesmo sem o uso de avaliações mais rigorosas, sugerem que a adoção de *expressões lambda* leva a um ganho direto na legibilidade do código [34]. Por outro lado, existe uma lacuna na literatura reportando estudos empíricos que avaliem de forma rigorosa tal benefício. Portando, *a falta de estudos empíricos avaliando os benefícios relacionados à legibilidade de código com a adoção de expressões lambda em Java dificulta o entendimento das situações em que transformar um código legado leva a uma melhoria do código.*

1.3 Objetivos

O principal objetivo desse trabalho é avaliar o impacto na compreensão de código em decorrência da adoção de expressões lambda em programas escritos na linguagem *Java*. Mais especificamente, os seguintes objetivos devem ser alcançados.

(OBJ1:) Realizar uma revisão da literatura sobre compreensão de programas e sobre como avaliar empiricamente a compreensão de programas.

(OBJ2:) Planejar e conduzir um estudo empírico para avaliar o impacto da adoção de expressões lambda em programas Java.

(OBJ3:) Configurar um ambiente para a execução do estudo empírico. Por se tratar de um estudo com características bem específicas, o desenvolvimento de um ambiente pode ser necessário.

(OBJ4:) Identificar e minerar cenários reais de rejuvenescimento de programas Java relacionados ao uso de *expressões lambda*. Tais cenários serão usados na execução do estudo empírico.

(OBJ5:) Analisar e reportar os resultados do estudo empírico. Tais resultados podem ser úteis para refinar a implementação de ferramentas existentes que realizam transformações de programas Java para usar expressões lambda.

1.4 Questões Gerais de Pesquisa

Neste contexto, alguns questionamentos sobre a adoção de expressões lambda em sistemas legado Java ganham relevância. São eles:

RQ1: A introdução de *expressões lambda* após o rejuvenescimento de código melhora a compreensão pelos desenvolvedores?

RQ2: Quais situações são adequadas, na perspectiva dos desenvolvedores, para incluir *expressões lambda* em programas Java?

Tendo em vista tais questões de pesquisa, este trabalho realizou um estudo empírico para cobrir essa ausência de conhecimento sobre os benefícios da adoção de *expressões lambda* em relação à facilidade de compreensão de código.

1.5 Metodologia

A realização deste trabalho envolveu a atividade de revisão da literatura com o intuito de entender sobre a qualidade, manutenção e evolução de *software*. A pesquisa envolveu ainda o estudo sobre a adoção de *expressões lambda* em Java e de modelos propostos anteriormente para mensurar a compreensibilidade de software [7, 14, 43, 48]. Para realizar esta pesquisa, foi utilizada uma abordagem de *métodos mistos*. Em estudos de métodos mistos são utilizados metodologias quantitativas e qualitativas de pesquisa em uma mesma investigação [10]. A utilização de métodos mistos possibilita a compreensão sobre um problema que não se obteria com a utilização de uma única abordagem. Como base deste

estudo, foi adotado a *Estratégia de Triangulação Concomitante*. Os principais aspectos metodológicos do estudo estão sintetizados na Tabela 1.1.

Tabela 1.1: Síntese dos aspectos metodológicos do estudo

Desenho do Estudo	***Pesquisa de métodos mistos com estratégia de triangulação concomitante de dados***	
	Quantitativo	**Qualitativo**
	Estudo Transversal	Pesquisa exploratória
Objetivo específico	Analisar trechos de código de transformações com expressões lambda em projetos reais através de métricas associadas a legibilidade	Avaliar o impacto da adoção de expressões lambda na compreensão de programas Java
Participantes	Desenvolvedores	Desenvolvedores
Coleta de dados	- Métricas de código fonte	- Questionário - Caracterização pessoal, profissional
Análise de dados	- Estatística descritiva - Estatística inferencial	- Análise de texto - Estátistica Descritiva

Os dados qualitativos e quantitativos foram combinados pela *estratégia de triangulação concomitante* com o objetivo de determinar se há convergência, diferenças e combinações entre os dois bancos de dados (dados armazenados separadamente). Essa combinação foi realizada na interpretação dos dados, portanto, mantendo os dois bancos de dados separados. De acordo com Creswell and Clark [11] nessa estratégia, em condições ideais, os dados são coletados simultaneamente, mas na prática um ou outro pode ser priorizado.

Esta pesquisa realizou a coleta dos dados a serem analisados em um determinado espaço de tempo (corte temporal), caracterizado pelo *Estudo transversal* [38]. Nessa pesquisa, também foi utilizado o método de *Pesquisa exploratória* que é uma metodologia de pesquisa qualitativa bastante utilizada em engenharia de software [56]. Optou-se pela exploratória, pois pretendemos proporcionar uma visão geral do assunto estudado. O intuito da utilização dessa abordagem é descrever o impacto causado pela adoção de *expressões lambda* em programas Java na compreensão dos desenvolvedores. Para investigar a percepção dos desenvolvedores *Java* sobre a introdução de *expressões lambda* através de transformações de código, realizamos uma pesquisa online. Os desenvolvedores foram submetidos a avaliar um grupo de pares de trechos de código e responder algumas questões em forma de *escala likert*. Foram calculadas algumas métricas para cada par de trechos (códigos anteriores e após a transformação) e para reforçar o escopo do trabalho, utilizamos a abordagem *Goal,Question and Metrics (Objetivo, Questões e Métricas)* para planejar medições baseadas em objetivos específicos [8]. Por fim, foi realizada a integração dos resultados quantitativos e qualitativos do estudo e a condução da investigação empírica de forma similar a outros trabalhos existentes [15].

1.6 Contribuições

A pesquisa teve como objetivo principal realizar uma avaliação empírica sobre o impacto causado a legibilidade de código Java após a introdução de *expressões lambda*. Neste cenário, as principais contribuições foram:

- Foi possível inferir que desenvolvedores e estudantes percebem uma melhoria na legibilidade em algumas situações. Com identificação dessas situações, acredita-se que será possível contribuir com informações para auxiliar no desenvolvimento/melhoria das ferramentas de rejuvenescimento e evolução de softwares para desenvolvedores e IDE's;
- Ao avaliar alguns modelos disponibilizados na literatura [7, 43] foi possível observar que os modelos possuem limitações quanto as novas construções da linguagem Java e que podem não ser eficientes para avaliar esses cenários;
- As métricas de complexidade ciclomática e quantidade de linhas de código são consideradas importantes para avaliar a qualidade de uma transformação e que a maioria das transformações avaliadas nesse trabalho ocasionaram uma redução dessas métricas. Pode-se concluir que, essas métricas podem indicar uma melhora na legibilidade do código após a introdução de *expressões lambda*.

1.7 Estrutura do Trabalho

Este trabalho está organizado em mais seis capítulos. No Capítulo 2 é apresentado a fundamentação teórica visando apoiar o leitor na compreensão dos temas importantes da dissertação como qualidade, manutenção, evolução de software e a adoção de *expressões lambda* em Java. Ainda no Capítulo 2, são discutidos modelos para mensurar a compreensibilidade de *software* elaborados em trabalhos anteriores e investigações sobre o uso de expressões lambda em java. No Capítulo 3 será discutido como foi conduzido o estudo empírico para avaliar tanto os atributos de código quanto a legibilidade percebida pelos desenvolvedores. No Capítulo 4 será apresentado a análise dos dados e os resultados obtidos. Logo depois, no Capítulo 5 é apresentado a conclusão seguida das limitações e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta a fundamentação necessária para uma melhor compreensão dos resultados desta pesquisa e que possui o objetivo de avaliar os potenciais benefícios na compreensão de software com a adoção de expressões lambda em Java. A literatura sugere que compreensão de software é essencial para as atividades de manutenção, um dos atributos de qualidade de software. A Seção 2.1 introduz qualidade de software de forma mais ampla, antes de uma discussão sobre manutenção, evolução e compreensão de software na Seção 2.2. Alguns usos típicos sobre a adoção de expressões lambda em Java são apresentados na Seção 2.3. Finalmente, alguns trabalhos relacionados, que analisam o uso de expressões lambda, são apresentados na Seção 2.4.

2.1 Avaliação de Qualidade de Software

A qualidade de *software* pode ser compreendida como a capacidade de um produto em atender as necessidades de seus usuários [13, 28]. Existe um conjunto de características para avaliar a qualidade de um produto de *software*. De acordo com a literatura, caso essas características sejam continuamente avaliadas, existem maiores chances de se produzir um *software* de qualidade [45].

De acordo com o modelo descrito pela norma ISO/IEC 9126 [19], a avaliação de qualidade de um *software* pode ser organizada em duas partes: *qualidade interna e externa* e *qualidade em uso*. Na primeira parte do modelo são definidas seis características para *qualidade interna e externa*. De acordo com esse modelo, as seis características para mensurar a qualidade interna e externa de um produto de *software* são [19, 29]:

- *Funcionalidade*: Capacidade que o *software* possui para realizar com eficácia as funcionalidades que lhe foram solicitadas;
- *Confiabilidade*: Nível de confiança em relação a sua maturidade e tolerância a falhas;

- *Usabilidade*: Facilidade de uso que o *software* possui em aprendizado e operabilidade;
- *Eficiência*: Comportamento de um *software* em relação ao tempo de processamento e consumo de recursos (como memória e processamento);
- *Manutenibilidade*: Facilidade que um *software* possui para ser alterado;
- *Portabilidade*: Facilidade com que um *software* pode ser disponibilizado em outros ambientes.

Em relação a segunda parte do modelo, são definidas quatro características de *qualidade em uso*, sendo elas: *Eficácia*, *Produtividade, Segurança* e *Satisfação*. A *qualidade em uso* pode ser entendida como a percepção do usuário sobre um produto de *software* ao ser executado em um ambiente e contexto específico de uso [19]. A norma defende que as características definidas são aplicáveis em qualquer tipo de *software* e que essas características fornecem um conjunto de termos consistentes para a avaliação da qualidade do produto [19].

Para avaliar a qualidade de um produto de *software*, podem ser mensurados atributos internos (medidas estáticas de produtos intermediários), atributos externos (medindo o comportamento do código quando executado) ou atributos de qualidade em uso [19]. Os atributos estáticos de um produto de *software*, relacionados à arquitetura, estrutura e seus componentes, representam os atributos internos de qualidade de *software* [28]. Garantir que o código de um sistema esteja bem escrito e de fácil entendimento por outros desenvolvedores é fundamental para a realização de atividades de manutenção de *software* [29], sendo possível avaliar alguns aspectos relacionados à manutenibilidade considerando a legibilidade do código fonte [37].

2.2 Manutenção e Evolução de Software

A manutenção é considerada como uma das fases mais importantes do ciclo de vida de um *software*, pois é um fator com alto índice de influência no custo total de um sistema [2]. A manutenção de *software*, que antes era relacionada ao desenvolvimento sem qualidade, passou a ser considerada como desenvolvimento evolutivo [31]. De acordo com a norma ISO/IEC 12207 [50], as atividades de manutenção de *software* podem ser caracterizadas como *Manutenção Corretiva*, *Manutenção Adaptativa* e *Manutenção Perfectiva*. A manutenção corretiva objetiva à correção de erros identificados por qualquer natureza. A manutenção adaptativa corresponde às alterações em partes do *software* quando ocorre a mudança de contexto ao qual é preciso ser adaptado. Finalmente, a manutenção perfectiva são aquelas mudanças efetuadas quando o *software* precisa evoluir, com o objetivo de atender as novas necessidades do usuário.

De acordo com Pigoski [41], mais de 50% dos esforços de manutenção de *software* são motivados pela inclusão de novas funcionalidades aos sistemas. Portanto, como a maior parte das mudanças que ocorrem durante a manutenção são direcionadas a adição de novas funcionalidades, os sistemas precisam ter uma evolução constante. Além disso, após os sistemas serem implantados, as alterações são necessárias para se manterem úteis, ocasionando uma constante execução de manutenções perfectivas [51]. Lehman [32] examinou a evolução de um número considerável de sistemas de grande porte, e propôs as chamadas *Leis de Lehman*. A primeira lei indica que a manutenção de um sistema de *software* é um processo constante e inevitável, pois novas necessidades, novos requisitos e mudanças de ambiente ocorrem com relativa frequência, necessitando de alterações para atendê-las. A segunda lei sugere que a complexidade do sistema tende a aumentar durante os processos de evolução; enquanto que a terceira indica que um sistema de *software* pode ter um número limitado de mudanças, devido a cultura organizacional e os fatores estruturais e arquiteturais utilizados durante o processo de desenvolvimento. A quarta lei estabelece que as mudanças em recursos do sistema possuem efeitos imperceptíveis durante a evolução do *software*. Finalmente, a quinta lei sugere que a cada adição de uma nova funcionalidade ao sistema pode ocasionar novos defeitos.

A evolução dos sistemas através da manutenção perfectiva está relacionada às questões acerca da manutenibilidade. A complexidade de um código é uma propriedade que influencia a facilidade com que um determinado código pode ser lido e compreendido e que pode afetar a qualidade, a manutenção, entre outros fatores. Caso não exista uma preocupação em se manter bons níveis de manutenibilidade, as atividades de manutenção serão cada vez mais difíceis de serem realizadas [12].

2.2.1 Manutenibilidade e Compreensão de Software

Manutenibilidade é um dos atributos de qualidade interna e externa descritos na Seção 2.1, sendo definida como a facilidade em que um produto de *software* pode ser modificado [29]. A manutenibilidade corresponde a um atributo de qualidade que objetiva mensurar o esforço de manutenção de um *software* [29]. Ou seja, quanto maior a manutenibilidade, menores os custos para evoluir um software. Os custos de manutenção tendem a aumentar quando é necessário alterar diversas partes do *software*, em um cenário de evolução específico. De acordo com Pressman [44], alguns fatores podem influenciar a manutenibilidade, incluindo o desenho arquitetural, as escolhas tecnológicas, a documentação do sistema disponível e a compreensibilidade do software.

A compreensibilidade, fator mais relevante para esta pesquisa, é considerada um elemento fundamental para que a manutenção seja realizada de maneira simples. Essa característica envolve a legibilidade de código [43], que corresponde à facilidade em que um

trecho de código pode ser lido e entendido [6]. A legibilidade é considerada crucial durante o ciclo de vida de um produto de software, uma vez que os desenvolvedores necessitam ler e entender o código durante todo o desenvolvimento do software.

Um código considerado legível, na maioria das vezes, diminui o tempo necessário para ser entendido, facilitando sua alteração em um tarefa de manutenção. Dessa forma, a legibilidade e compreensibilidade são fatores que possuem um forte relacionamento, pois a legibilidade diz respeito ao grau de dificuldade que um desenvolvedor percebe em relação a um trecho de código antes de altera-lo [6, 43]. Para se medir a legibilidade de um código, deve-se levar em conta as percepções subjetivas que podem ser variadas, pois requerem análise de aspectos humanos em conjunto com aplicações estatísticas [6]. A próxima seção apresenta alguns modelos usados para avaliar a compreensibilidade de um software.

2.2.2 Avaliação de Compreensibilidade de Software

Trabalhos avaliando a legibilidade comparando trechos de código já foram realizados pela academia, mas com objetivos diferentes deste trabalho. Magalhães e Gerosa [15] realizaram uma pesquisa com desenvolvedores de *software* avaliando padrões de convenção de codificação e mostraram que grande parte desses padrões influencia positivamente na legibilidade percebida pelos desenvolvedores. Outros trabalhos objetivam quantificar relações entre atributos de código e a legibilidade percebida pelos desenvolvedores, com o objetivo de criar modelos para mensurar legibilidade [7, 14, 43, 48].

No estudo realizado por Buse e Weimer [7], a legibilidade foi definida como o entendimento humano sobre a facilidade de compreender um texto e que a legibilidade de um programa está relacionada a sua capacidade de manutenção. A hipótese do estudo é que programadores possuem alguma noção intuitiva para apontar características e recursos do programa que serão bons indicadores para a legibilidade. Com isso, foi apresentado um modelo (referenciado a partir de agora por BW) descritivo de legibilidade de *software* baseado em opiniões de programadores e noções de qualidade de *software*.

O modelo BW é baseado em características simples que podem ser extraídas automaticamente do código fonte dos programas. Tais características são parecidas com as avaliadas em linguagens naturais, considerando fatores léxicos (e.g., média de sílabas por palavra). Todavia, apesar de possuírem características similares, a legibilidade do código é muito diferente das linguagens naturais, pois o código além de ser altamente estruturado, compõe-se de elementos que atendem a propósitos diferentes, incluindo design, documentação e lógica [7].

Para construção do modelo BW, Buse e Weimer conduziram um estudo com 120 alunos de diferentes níveis de experiência em codificação, pedindo aos participantes que fornecessem pontuações subjetivas de classificação da leitura dos trechos de código [7].

Adicionalmente, os autores construíram uma ferramenta automatizada que realiza a extração de trechos de código em programas Java. A extração tinha como alvo pequenos trechos de código, com o objetivo de eliminar o contexto e a complexidade para que os desenvolvedores focassem sua atenção nos detalhes de "baixo nível" de legibilidade [7]. Com isso, os trechos de código foram restritos, uma vez que os trechos deveriam possuir apenas três instruções e declarações simples. Além disso, os trechos deveriam incluir linhas que não são instruções, como comentários ou linhas em branco. Os trechos não deveriam possuir vários métodos ou iniciar dentro de uma instrução composta. Finalmente, a distribuição de tamanho (em número de caracteres por trecho) foi em média de 278,94, mínima de 92 e máxima de 577. Foram selecionados 5 projetos para a extração de 100 trechos de código.

Para capturar as pontuações dos participantes, foi realizado um *Survey* onde cada participante recebeu o mesmo conjunto de trechos. Os participantes podiam selecionar um número próximo de cinco para trechos "mais legíveis" ou próximo de um para trechos "menos legíveis", com uma pontuação de três indicando neutralidade. Estas pontuações foram validadas e agregadas para produzir pontuações de opinião média. O resultado desse estudo foi um conjunto de trechos de código acompanhados por 12.000 avaliações sobre legibilidade. Em seguida, foi realizada uma coleta de medidas a nível de *token* automaticamente dos trechos de código. Os autores então construíram um modelo baseado em algumas características que possuem certa influência sobre a facilidade de leitura e compreensão de código [7]. A Tabela 2.1 apresenta as características do modelo proposto. A primeira coluna indica o número médio de ocorrências daquele recurso por linha. Na segunda coluna, é indicado o número máximo de ocorrências do recurso para todas as linhas. As setas indicam se o recurso possui correlação com a legibilidade (em relação a percepção dos desenvolvedores), sendo positivo (▲) indicando um alto índice ou negativo (▼) indicando o contrário. Como resultado, Buse e Weimer [7] mostraram que o comprimento dos nomes dos identificadores não possuem nenhuma influência na legibilidade. Todavia, foi observado que o número de identificadores e caracteres por linha tem forte influência na métrica de legibilidade. A quantidade de linhas em branco também possui correlação positiva com a métrica. Apesar de considerarem que os comentários em um trecho de código aumentam a legibilidade, foi pouco correlacionado com a noção de legibilidade na percepção dos desenvolvedores [7]. As cores indicam a força preditiva (verde = "alto", amarelo = "médio", vermelho = "baixo").

Tabela 2.1: Tabela de características adaptadas do modelo de Buse [7].

Média	Máximo	Característica
▼	▼	*Comprimento de linha (Quantidade de caracteres)*
▼	▼	*Quantidade de Identificadores*
▼	▼	*Comprimento dos identificadores (Quantidade de caracteres)*
▼	▼	*Identações (hierarquia dentre blocos de código)*
▼	▼	*Quantidade de Palavras chave*
▼	▼	*Quantidade de Números*
▲		*Quantidade de Comentários*
▼		*Quantidade de Períodos*
▼		*Quantidade de Vírgulas*
▼		*Quantidade de Espaços*
▼		*Quantidade de Parêntese*
▲		*Quantidade de Operadores Aritméticos*
▼		*Quantidade de Operadores de Comparação*
▼		*Quantidade de Atribuições (=)*
▼		*Quantidade de Ramificações (if)*
▼		*Quantidade de Laços (for, while)*
▲		*Quantidade de Linhas em branco*
	▼	*Quantidade de caracteres únicos*
	▼	*Quantidade de identificadores únicos*

Com base nas características consideradas pelo modelo BW, foi possível identificar as características a serem analisados quando se pretende verificar a legibilidade de um trecho de código. Essa contribuição foi importante pois abriu a possibilidade para criação de ferramentas automáticas que podem fornecer pontuações de legibilidade como *feedback* aos desenvolvedores. Esse *feedback* contínuo pode melhorar potencialmente a legibilidade e, portanto, a capacidade de manutenção do código a longo prazo.

Posnett et al. [43] realizaram um trabalho com o intuito de aprimorar o modelo BW, e apresentaram uma teoria de legibilidade simples e intuitiva, baseada no tamanho e na entropia de código [16]. Esse modelo usa métricas de tamanho e métricas de Halstead [24]. Posnett et al. defendem que sua abordagem é teoricamente mais fundamentada e utilizável para a medição da legibilidade. Eles também apontam alguns detalhes que podem ter influenciado negativamente o modelo BW. Em um primeiro relato, os autores abordam sobre o tamanho da amostra utilizada e que uma análise de componentes principais revelou forte sobreposição. Isso sugere que a definição de um modelo mais simples pode levar a uma teoria generalizada para legibilidade [43].

Outra possível limitação apontada ao modelo BW está relacionado ao tamanho dos trechos de código, onde é defendido que o tamanho de um determinado trecho de código

não possui influência na legibilidade. No entanto, eles argumentam que é preciso incluir a noção de tamanho no modelo, para diferenciar a dependência em relação a fatores não dimensionais. Os autores defendem essa posição apresentando a correlação de *Spearman* entre várias características utilizadas no modelo BW. Em seguida os autores apontam algumas métricas de Halstead e argumentam que elas adicionam poder explanatório ao modelo em relação ao tamanho [43].

Posnett et al. classificaram os trechos usando o mesmo limiar de pontuação usado na avaliação do modelo BW. Também calcularam métricas de tamanho, métricas de Halstead e entropia considerando tanto *tokens* quanto *bytes* para cada um dos trechos. Esses dados foram utilizados para treinar classificadores. Para construir o modelo eles utilizaram uma técnica na qual um classificador pode escolher entre vários recursos, passando por um refinamento com o intuito de remover recursos que não apresentam melhora ao modelo. Para isso foi utilizado um conjunto de medidas de desempenho para avaliar os modelos.

Foi realizada uma extração de trechos de código com mais de 200 linhas e fizeram uma comparação entre os dois modelos abordados. De acordo com Posnett e seus colegas, o modelo BW classifica todas as funções com mais de 200 linhas como "menos legíveis", porém o modelo aprimorado de Halstead continua a classificar algumas funções como legíveis e outras como não. Como resultado, foi possível mostrar que o modelo proposto supera o modelo BW como um classificador de legibilidade em pequenos trechos de código.

J. Dorn [14] apresentou um outro modelo para mensurar a legibilidade de *software*. Este modelo foca a contagem de símbolos em pequenos trechos de código e é baseado em aspectos visuais, espaciais e linguísticos, como por exemplo, padrões estruturais, tamanhos de blocos de código e palavras em linguagem natural(inglês). O autor aponta que os modelos anteriores não são baseados em padrões de codificação [1], mas sim baseados em combinações de características sintáticas extraídas do código como a quantidade de operadores ou comprimento de linhas. O modelo de J. Dorn agrega aspectos e características geométricas, baseadas em padrões linguísticos e em uma métrica de legibilidade [14].

Para execução do estudo, foram avaliados diferentes idiomas em três linguagens de programação como: Java, Python e CUDA (derivada de C e *assembly* voltada para programação de GPUs) com noções potencialmente distintas de legibilidade. Para isso, foram apresentados aos participantes do estudo tanto trechos pequenos de código (como nas abordagens anteriores) quanto trechos de código maiores, totalizando 360 amostras de código—uma amostra maior que a usada nos estudos anteriores. Esses trechos foram apresentados a 5.000 desenvolvedores diferentes, dos quais 1.800 possuíam experiência na indústria. Ele aponta que o modelo BW não consegue capturar aspectos visuais e linguísticos [14].

O modelo proposto por J. Dorn leva em consideração a estrutura espacial e o conteúdo

de linguagem natural de vários *tokens*[1]. Os idiomas em que os tipos de *tokens* podem depender do contexto, como por exemplo, "identificador"e "nome do tipo"em linguagens semelhantes ao C, são todos tratados como identificadores [14]. Esse comportamento permite que a métrica capture mais riquezas semânticas, mas sem exigir análise formal. J. Dorn defende que essa abordagem permite uma captura mais rica em semânticas do que a contagem pura de caracteres como nos modelos anteriores [7, 43]. Dorn define seis recursos para o modelo, sendo eles:

(a) *Características do Padrão Estrutural*: Podem ser reconstruídas a partir de componentes da transformada discreta de Fourier [5];

(b) *Recursos de Percepção Visual*: Modela os efeitos da sintaxe atribuindo diferentes cores simbólicas a caracteres que compõem diferentes tipos de *tokens*;

(c) *Características de Alinhamento*: Analisa o código-fonte como uma matriz de caracteres para encontrar regiões onde símbolos iguais que ocorrem em linhas consecutivas são alinhados à mesma coluna;

(d) *Características da Linguagem Natural*: Realiza uma busca por identificadores onde o prefixo ou sufixo é uma palavra inglesa;

(e) *Características da Sintaxe Rasa*: Incorpora recursos utilizados nos modelos anteriores, como medidas do comprimento de linha, a contagem de identificadores distintos, frequência de palavras-chave, literais numéricos e operadores;

(f) *Modelo Descritivo Formal*: Avalia os recursos de interação que podem ter uma associação positiva ou negativa à legibilidade e seleciona um subconjunto de recursos para basear o modelo. A métrica foi construída utilizando uma regressão logística.

O estudo também realizou um *Survey*, onde cada participante recebeu vinte amostras e pediu para classificá-las em uma *escala Likert* de 1 a 5, de muito ilegível a muito legível [14]. J. Dorn relata que apesar da atenção ser restrita apenas a participantes com no mínimo cinco anos de experiência industrial, a pesquisa conteve 1091 participantes a mais em relação as anteriores.

Como contribuição desse trabalho, foi possível argumentar que a métrica de legibilidade se correlaciona com a densidade de defeitos, uma noção externa de qualidade de *software*. Também foi possível analisar as semelhanças e diferenças nas opiniões sobre legibilidade de código entre idiomas, níveis de experiência e contextos de programação, como por exemplo, a percepção de alunos é focada no número de operadores e literais numéricos, já profissionais da industria dão maior prioridade aos espaços em branco.

[1]Tokens são elementos de texto significativos para o compilado. Ex: caracteres de pontuação como colchetes, chaves, parênteses e vírgulas.

2.3 Uso de Expressões lambda em Java

As *expressões lambda* em Java correspondem a funções anônimas que podem ser passadas como argumento para funções ou usadas como valor de retorno de uma função, permitindo a parametrização do comportamento [55]. Na linguagem Java, as expressões Lambda permitem que o programador forneça a implementação do método abstrato de uma interface funcional[2] de uma maneira mais concisa, tratando toda a expressão como uma instância de uma interface funcional. Também é possível conseguir o mesmo com uma classe interna anônima (*Anonymous Inner Class*). Entre as várias opções de contextos para se adotar *expressões lambda* em sistemas legado em Java, duas situações são mais frequentes e comumente citadas: o uso de expressões lambda no lugar de classes anônimas e o uso de expressões lambda para implementar padrões recursivos em coleções [36].

Para ilustrar a utilização de *expressões lambda*, implementamos alguns trechos de código baseados em um modelo de sistema simples desenvolvido em um trabalho realizado anteriormente chamado *101 companies* [18]. O objetivo deste trabalho foi desenvolver um recurso de conhecimento livre através de um projeto comunitário em Ciência da Computação (mais particularmente para o ensino de linguagens de programação). O projeto criou o modelo de um sistema para gerenciamento de recursos humanos, onde desenvolvedores e estudantes, entre outros, pudessem realizar contribuições de versões diferentes do sistema com poucos recursos e aspectos das linguagens utilizadas. O projeto é interessante pois proporciona desafios e comparação entre tecnologias e linguagens de programação [18].

A Figura 2.1, ilustra um diagrama de classes UML que modela o domínio do sistema. De acordo com o diagrama, o sistema possui uma empresa que é composta por departamentos, que podem ser divididos em sub-departamentos. Cada departamento pode possuir vários funcionários e um gerente. Os recursos do sistema podem ser variados, como por exemplo, possuir um método que totaliza os salários de todos os funcionários.

[2]Uma interface funcional possui apenas um método abstrato, permitindo que *expressões lambda* sejam passadas para substituir o comportamento desse método e pode possuir outros métodos como *default*.

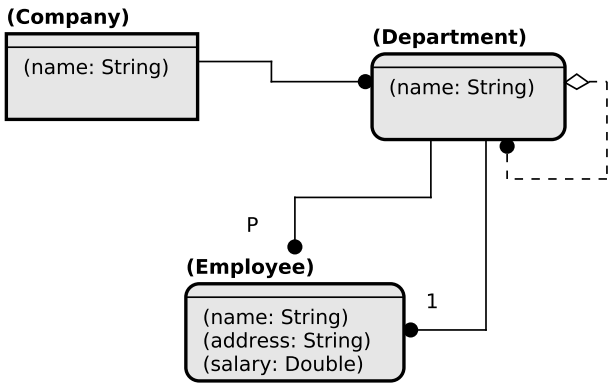

Figura 2.1: Diagrama UML para ilustração do modelo adaptado de [18]

2.3.1 *Annonymous Inner Classes* e *Expressões lambda*

Antes do suporte a expressões lambda em Java, para passar um método (comportamento) como parâmetro de algum outro método era necessário a utilização de uma *Anonymous Inner Classes*, que permite definir e instanciar uma classe anônima que implementa os métodos de uma interface existente. Apesar de ser trivial implementar uma classe anônima, esse recurso é considerado verboso, o que pode afetar a legibilidade e compreensão do código por outros desenvolvedores [52]. Com a evolução da linguagem e a adição das expressões lambda, pode-se notar que, em algumas ocasiões em que era necessária a definição de uma classe anônima, ocorre uma redução da quantidade de linhas e da complexidade ciclomática [55].

O Listagem 2.1 a seguir, ilustra como fazer a comparação de dois funcionários da empresa. A classe anônima instância o método de uma interface funcional que faz a comparação entre dois funcionários. A solução tradicional faz a verificação se os funcionários são iguais e caso a condição seja atendida é retornado um *booleano true*.

```
Compare verify = new Compare() {
  @Override
  public boolean compareEmployees(Employee e1, Employee e2) {
    if (e1.equals(e2)) {
      return true;
    }
    return false;
  }
};
```

Código 2.1: Solução tradicional utilizando classe anônima

A Listagem 2.2 ilustra o código correspondente utilizando expressão lambda. Podemos observar que a quantidade de linhas necessárias para realizar essa operação teve uma redução significativa.

```
Compare verify = (e1, e2) -> e1.equals(e2);
```

Código 2.2: Solução utilizando expressão lambda

2.3.2 *Collections* e *Expressões lambda*

As coleções (*Collections*) são *API's* fundamentais para um *software*, pois são usadas para agrupar e processar dados. Com as novas construções do Java 8, os designers desenvolveram uma nova forma de iteradores *Streams*, com o intuito de melhorar o desempenho no processamento de dados, aproveitando as arquiteturas *multicore*. O intuito é viabilizar a paralelização das instruções de uma forma transparente, sem a necessidade do programador se preocupar em escrever código *multithread* [55]. Os métodos *Streams* podem receber como parâmetro uma *expressão lambda*, o que corresponde à aplicação de um comportamento em coleções [52].

Os padrões recursivos descritos no restante dessa seção, ilustram alguns recursos que fazem parte do contexto das implementações que serão analisadas durante a realização desta pesquisa. O código completo do exemplo implementado pode ser encontrado em um repositório[3]. Estão sendo considerados casos específicos relacionados a padrões recursivos, como *filter* e *map* ilustrados por um conjunto de transformações que convertem laços *for* em *expressões lambda* usando *Streams*.

Padrão Recursivo *ForEach*

O padrão recursivo `forEach` executa um comando para cada elemento de uma coleção. Como os demais padrões recursivos, o método `forEach` espera uma expressão lambda que contem a operação a ser realizada [42] em cada um dos elementos da coleção. O código na Listagem 2.3 ilustra como implementar uma operação de corte de salários em um departamento (e seus sub-departamentos) usando uma solução tradicional de laço de iteração, onde cada um dos laços `for` itera ou na lista de funcionários ou na de sub-departamentos. Para cada elemento contido nas listas, é feita uma chamada a uma operação que permite cortar um percentual do salário dos funcionários.

```
public void cutSalaries(double percent) {
  for(Employee e: employees) {
    e.cutSalaries(percent);
```

[3]https://github.com/waltim/101Companies

```
  }

  for(Department d: subdepts) {
    d.cutSalaries(percent);
  }
}
```

Código 2.3: Solução tradicional utilizando laços de repetição para `forEach`

A Listagem 2.4 ilustra o código correspondente utilizando o método `forEach` da interface *Stream* e expressões lambda. É possível observar que a quantidade de linhas necessárias para realizar essa operação teve uma redução significativa.

```
public void cutSalaries(double percent) {
  employees.stream().forEach(e -> e.cutSalaries(percent));
  subdepts.stream().forEach(d -> cutSalaries(percent));
}
```

Código 2.4: Solução utilizando a interfsace *Stream* para `forEach`

Padrão Recursivo *Filter*

O padrão recursivo `filter` recebe uma expressão lambda contendo um predicado (uma função que mapeia um objeto em um valor booleano), e filtra os elementos da coleção que não satisfazem o predicado [42]. O código ilustrado na Listagem 2.5 filtra os objetos (funcionários) da coleção quando o salário do funcionário não satisfaz à condição estabelecida, usando uma solução tradicional que realiza a iteração usando um laço de repetição nos funcionários verificando se o salário do funcionário atende ou não à condição (populando em uma outra lista os objetos que satisfazem à condição).

```
public List<Employee> employeeWithHighSalaries(double salary) {
  List<Employee> res = new ArrayList<>();
  for(Employee e: employees) {
    if(e.getSalary() > salary)
      res.add(e);
  }
  return res;
}
```

Código 2.5: Solução tradicional de `filter` utilizando laços de repetição

A Listagem 2.6 utiliza a interface *Stream* e expressões lambda para filtrar os funcionários que possuem altos salários. A condição `getSalary() > salary` no corpo da expressão lambda filtra os objetos que possuem o salário abaixo do especificado. O método `collect` transforma os elementos da *Stream* em elementos de uma lista.

```
public List<Employee> employeeWithHighSalaries(double salary) {
  return employees.stream()
                  .filter(e -> e.getSalary() > salary)
                  .collect(Collectors.toList());
}
```

Código 2.6: Solução de `filter` utilizando a interface *Stream*

Padrões Recursivos *Map* e *Reduce*

O padrão recursivo `map` mapeia os objetos de uma coleção em objetos de outro tipo, mantendo a estrutura de dados original. O padrão recursivo `reduce` realiza uma computação que consolida os valores em uma estrutura de dados para um valor. Por exemplo, supondo que se deseja computar todos os salários de um departamento, torna-se necessário computar os salários dos funcionários e aplicar uma totalização. O `map` mapeia um funcionário no seu salário, enquanto que o `reduce` totaliza os salários pagos.

O código ilustrado na Listagem 2.7, utiliza laços de repetição nos elementos das coleções de funcionários e sub-departamentos, com o intuito de totalizar os salários de todos os funcionários vinculados ao departamento.

```
public double totalSalary() {
  double total = 0;
  for(Employee e: employees) {
    total += e.totalSalary();
  }

  for(Department d: subdepts) {
    total += d.totalSalary();
  }
  return total;
}
```

Código 2.7: Solução tradicional utilizando laços de repetição para os padrões `map` e `reduce`

A operação `totalSalary()` na Listagem 2.8 totaliza os salários pagos em um departamento utilizando as operações `map` e `reduce` da interface *Stream*. Esse exemplo também ilustra a composição dos padrões recursivos, onde inicialmente o `map` transforma os elementos das coleções (ou funcionários ou departamentos) em objetos do tipo `Double`. Logo em seguida, o método `reduce()` executa a consolidação dos elementos, em termos de um somatório que usa `0` como valor de identidade da soma.

```
public double totalSalary() {
  double total = employees.stream()
```

```
                            .map(e -> e.totalSalary())
                            .reduce(0,Double::sum);

  total += subdepts.stream()
                   .map(d -> d.totalSalary())
                   .reduce(0,Double::sum);
  return total;
}
```

Código 2.8: Solução utilizando a interface *Stream* para os padrões `map` e `reduce`

Considerando os exemplos anteriores, é possível observar que o código fica mais conciso ao utilizar a interface *Stream* em conjunto com os padrões recursivos e expressões lambda. Também foi possível ver como a evolução da linguagem Java incluiu novas construções para facilitar a escrita de código mais conciso e enxuto pelos desenvolvedores, que necessitam garantir a qualidade de seus *softwares*. Por outro lado, ainda existe uma falta de estudos sistemáticos para avaliar os ganhos de legibilidade com a adoção de expressões lambda.

2.4 Trabalhos Relacionados à Adoção de Expressões Lambda

Estudos sobre a adoção de expressões lambda foram encontrados na literatura. Alguns desses trabalhos buscam aplicar técnicas de refatoramento usando expressões lambda. Por exemplo, Tsantalis et al. investigaram a utilidade das expressões Lambda para parametrizar diferenças comportamentais em clones, propondo uma técnica que analisa a aplicabilidade das expressões lambda para a refatoração de clones através de uma investigação em um grande conjunto de dados de clones [54].

Outro estudo realizou mineração de dados para coletar perguntas e respostas no *StackOverflow* [53]. Nesse trabalho, foram analisadas questões sobre expressões lambda, onde foram extraídas as 100 questões mais populares e compreendido que a taxa de satisfação sobre questões envolvendo lambda em Java é acima de 80%. Foram encontradas também pesquisas que desenvolveram ferramentas e bibliotecas de refatoração [20, 23, 36]. Apesar das ferramentas facilitarem a evolução dos softwares através de transformações em trechos de código, a maioria das transformações tendem a ser feitas manualmente, pois as ferramentas nem sempre atendem às expectativas dos desenvolvedores [34].

Através dos estudos mencionados, foi possível identificar que algumas transformações em trechos de código envolvendo expressões lambda não são aceitas pelos desenvolvedores dos projetos as quais eram submetidas (usando o mecanismo de *pull-requests* do Github).

Alguns desenvolvedores mostraram insatisfação sobre transformações em classes de teste [36]. Apesar de analisarem trechos de código com expressões lambda, nenhum destes trabalhos realizou uma investigação empírica sobre a compreensão dos desenvolvedores em relação a adoção de expressões lambda em programas Java.

Capítulo 3

Planejamento Experimental

Este capítulo apresenta os procedimentos realizados durante o trabalho. Inicialmente o escopo da pesquisa é caracterizado utilizando a abordagem *Objetivo, Questões e Métricas* [47] (Seção 3.1). Em seguida, são detalhados os procedimentos e ferramentas que foram utilizados durante a condução do estudo (Seção 3.2).

3.1 Objetivo, Questões e Métricas

O objetivo dessa pesquisa é investigar os benefícios da introdução de *expressões lambda* na compreensão de código. Para responder as questões de pesquisa mencionadas na Seção 1.4, duas abordagens foram seguidas. A primeira, consiste no cálculo de algumas métricas discutidas na literatura para estimar a complexidade e a legibilidade do código. Foram calculadas a quantidade de linhas por trecho de código e a complexidade ciclomática (métrica que quantifica os caminhos de execução independentes a partir de um código fonte para os trechos de código [33]). Adicionalmente, dois modelos para estimar legibilidade foram usadas. A Seção 3.1.2 descreve o modelo de Buse e Weimer [7], enquanto que a Seção 3.1.3 descreve o modelo de Postnett et al. [43]. As métricas foram calculadas para todos os pares de trecho de código, onde cada par contém o trecho de código na versão anterior a inclusão de expressões lambda e o código posterior à introdução de *expressões lambda*. A segunda abordagem de pesquisa corresponde a um estudo qualitativo, cujos procedimentos refletem a abordagem seguida por Santos e Gerosa [15], onde pares de trechos de código são apresentados aos participantes do estudo, que devem responder a um grupo de questões para cada par de trechos avaliado. As seguintes questões do *survey* foram exploradas na dissertação:

(SQ1:) A introdução de expressões lambda melhora a legibilidade?;

(SQ2:) Qual dos dois trechos de código você prefere, antes ou depois da introdução de expressões lambda?;

(SQ3:) Você gostaria de realizar algum comentário adicional às suas respostas, justificando sua decisão?

As métricas e questões do *Survey* foram combinadas para melhorar o entendimento e refinar as respostas das questões de pesquisa. Por exemplo:

- Os resultados de ambas as métricas de legibilidade (dados quantitativos) foram combinados com os resultados da *SQ1* do *Survey* (dados qualitativos) para responder a **RQ1**;
- Quantidade de linhas de código e complexidade ciclomática (dados quantitativos) foram combinadas com as questões *SQ2 e SQ3* do *Survey* (dados qualitativos) para responder a **RQ1**;
- As respostas das questões *SQ1 e SQ3* do *Survey* (dados qualitativos) para responder a **RQ2**.

3.1.1 Métricas de complexidade e distribuição de código

Para estimar a complexidade, as seguintes métricas foram usadas.

(M1:) SLOC (linhas por trecho de código): é uma métrica utilizada para estimar o tamanho de um programa ou trecho de código medindo a quantidade de linhas;

(M2:) Complexidade Ciclomática [35] é uma métrica de software que mede a quantidade de caminhos de execução independentes a partir de um trecho de código, para estimar a complexidade de um programa.

3.1.2 Modelo de legibilidade de Buse e Weimer.

O modelo de Buse e Weimer [7] utiliza técnicas de aprendizado de máquina para estimar a legibilidade do código, considerando julgamentos humanos. O modelo aprende da seguinte forma: dado um conjunto de recursos de legibilidade de código como entrada $\{x_1, x_2, x_3, ..., x_n\}$ (como o comprimento das linhas de cada trecho de código, comprimento dos identificadores, quantidade de linhas em branco, etc.) e um conjunto de classificações humanas de saída $\{y_1, y_2, y_3, ..., y_m\}$, o modelo estima a legibilidade de instâncias de trechos de código. Como resultado, o modelo de Buse e Weimer classifica em um intervalo entre 0 a 1 o nível de legibilidade para qualquer código, onde 0 indica a pior legibilidade e 1 indicando alta legibilidade.

3.1.3 Modelo de legibilidade de Posnett et al.

Postnett et al. apresentam um modelo para estimar legibilidade [43], que utiliza como base o modelo Buse e Weimer, mas considerando um conjunto mais reduzido de características dos trechos de código. A legibilidade Z de um trecho de código X pode ser calculada pelo modelo a partir da Eq. (4.1).

$$Z(X) \quad = \quad 8.87 - 0.033V(x) + 0.40L(X) - 1.5H(X) \tag{3.1}$$

onde $V(X)$ corresponde ao Volume de Halstead [24]. O volume é computado considerando duas medidas de tamanho (comprimento e vocabulário), usando a fórmula $V(X) = N(X)log_2 n(x)$. O comprimento $N(X)$ de um trecho de código é dado por $N(X) = N1(X) + N2(X)$, i.e., a soma do número total de operadores (N1) e operandos (N2). O vocabulário $n(X)$ de um trecho de código é calculado por $n(X) = n1(X) + n2(X)$, i.e., a soma do número de operadores únicos ($n1$) e operandos únicos ($n2$). O termo $L(X)$ na Eq. (4.1) representa o número de linhas contidas no código; e, por fim, a o termo $H(X)$ corresponde a *Entropia* de um trecho de código [43]—que em nosso contexto estima o grau de desordem do código fonte. A entropia de um documento X é dada pela Eq. (4.2), onde x_i é um *token* em X, $count(x_i)$ é o número de ocorrências de x_i no documento X, e $p_(x_i)$ é calculada pela Eq. (4.3). Após calcular o valor de $Z(X)$ é utilizada a função $\frac{1}{1+e^{-z}}$ para obter o resultado do modelo simples [43] ao qual resultados mais próximos de 0 indicam alta legibilidade para qualquer trecho avaliado.

$$H(X) = -\sum_{i=1}^{n} p_(x_i)\, log_2\, p_(x_i) \tag{3.2}$$

$$p_(x_i) = \frac{count(x)}{\sum_{j=1}^{n} count(x_j)} \tag{3.3}$$

3.2 Procedimentos para a Realização da Pesquisa

Esta seção descreve os procedimentos seguidos na condução da pesquisa, incluindo a seleção dos projetos, a coleta de exemplos reais de transformação de código para uso de *expressões lambda*, o cálculo das métricas e a aplicação do *Survey*—para coletar as opiniões dos desenvolvedores quanto à introdução de *expressões lambda*.

3.2.1 Seleção dos Projetos

Um estudo anterior realizado por Mazinanian et al [34] investiga a introdução de *expressões lambda* em Java desenvolveram uma ferramenta web (MineWebApp) com a finalidade de monitorar projetos Java de código aberto no GitHub. Esta ferramenta tem o intuito de minerar e analisar trechos de código com *expressões lambda*, facilitando a coleta de exemplos reais, os quais foram usados nesta pesquisa. A ferramenta monitora as ocorrências de uso de *lambda* nos projetos selecionados. As ocorrências encontradas são agrupadas em três categorias:

- *New method*: Quando um método novo contendo expressões lambda é adicionado a uma classe existente do projeto;
- *New class*: Quando uma nova classe é adicionada ao projeto, envolvendo métodos com *expressões lambda*;
- *Existing method*: transformações de trechos existentes onde são introduzidas *expressões lambda*.

A partir da possibilidade de coletar os exemplos diretamente de um *commit* existente, foi feita a seleção de alguns projetos onde foram considerados apenas os que possuíam *expressões lambda* analisadas pela ferramenta MineWebApp. Dos projetos selecionados, foram extraídos 59 trechos de código. A extração considerou exclusivamente os trechos de código da terceira categoria (*Existing method*) que caracterizavam a introdução de *expressões lambda* em código legado. Também foram extraídos 29 trechos disponibilizados por uma ferramenta de rejuvenescimento de código Java legado, onde eram detectados trechos de código aptos para transformações com a adição de *expressões lambda* através de refatorações de código automatizadas [36]. No total, foram selecionados 88 trechos de 22 projetos, conforme apresentado na Tabela 3.1. Através dos *commits* dessas transformações, selecionamos os trechos de código a serem usados nesta pesquisa.

Tabela 3.1: Projetos selecionados

ID	**Nome do Projeto**	**N. de expressões lambda extraídas MW**	**N. de expressões lambda extraídas RJTL**
1	seleniumQuery	0	10
2	elasticsearch	2	4
3	CoreNLP	0	15
4	SeeWeather	1	0
5	vertx-examples	3	0
6	Swagger2Markup	3	0
7	spring-framework	1	0
8	SpongeAPI	4	0
9	tailor	6	0
10	Agrona	1	0
11	RxAndroidBle	6	0
12	optaplanner	10	0
13	RxJava-Android-Samples	3	0
14	kaa	3	0
15	jersey	5	0
16	uhabits	2	0
17	graylog2-server	1	0
18	FluentLenium	1	0
19	vert.x	2	0
20	VirtualApp	1	0
21	qualitymatters	1	0
22	spring-integration	3	0

3.2.2 Caracterização dos Cenários de Uso de Expressões Lambda

A Figura 3.1 apresenta uma visão geral do procedimento utilizado para coletar e caracterizar os trechos de código candidatos. Foram criados alguns scripts para automatizar esse procedimento. A principal entrada consiste em um arquivo `csv` preenchido pelo desenvolvedor (manualmente) que guia a execução de um *web crawler* para extrair as informações a partir da ferramenta MinerWebApp. O arquivo `csv` possui as seguintes colunas:

- Url do *commit*: A primeira coluna do arquivo é preenchida com a url do commit, junto com o #diff (Tipo de chave usada pelo GitHub para identificar os arquivos e os commits);

- Linha de início: Número da linha onde inicia o trecho de código, por exemplo, L10 (a letra maiúscula no início representa *LEFT*, que são as linhas do código anterior);
- Linha final: Número da linha onde termina o trecho de código, por exemplo, R20 (a letra maiúscula no inicio representa *RIGHT*, que são as linhas do código após o *commit*);
- Tipo da refatoração: São informados os identificados dos tipos cadastrados na plataforma para indicar o tipo da transformação a ser executada;
- Linguagem do trecho de código: São informados os identificadores das linguagens na plataforma para indicar qual linguagem de programação aquele trecho de código representa.

Para capturar os cenários de uso de *expressões lambda*, foi desenvolvida uma ferramenta chamada Crawler que executa a extração dos trechos de código diretamente do GitHub. O Crawler é o componente de extração da plataforma, que ao receber um arquivo `csv` como entrada, acessa a *url* do commit, procura as linhas de código e extrai os trechos com a mesma formatação apresentada pelo GitHub. Os trechos de código extraídos são armazenados em um banco de dados de forma que é possível caracterizar o código legado e o código após a introdução de*expressões lambda*.

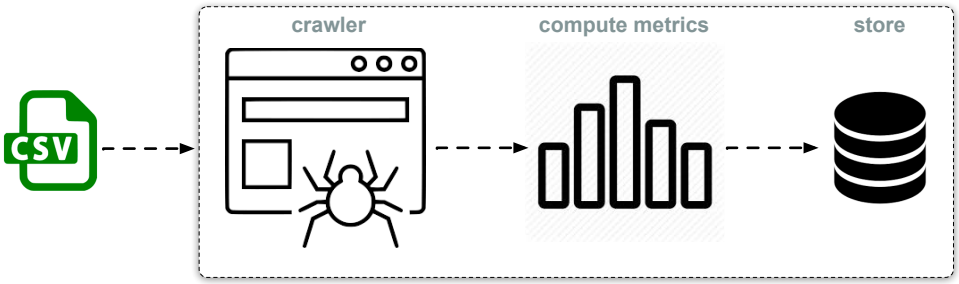

Figura 3.1: Fluxograma de coleta dos trechos de código e cálculo de métricas

Após os cenários de uso de *expressões lambda* serem armazenados em um banco de dados, um outro *script* é usado para aplicar o calculo das métricas relacionadas à legibilidade (Seção 3.1.2 e 3.1.3). Finalmente, os resultados das métricas foram armazenados em banco de dados para serem analisados. Todos os passos executados nesse processo foram automatizados, exceto o preenchimento do arquivo de entrada.

3.2.3 Condução do Survey

Para que fosse possível alcançar os objetivos desta pesquisa, foi necessário implementar uma aplicação *Web* para coletar as percepções dos desenvolvedores em relação aos trechos

de código. A Figura 3.2 ilustra uma parte da página do questionário que é gerada para cada par de trechos de código a serem visualizados e avaliados pelos desenvolvedores. No lado esquerdo é apresentado o código antes da introdução de *expressões lambda* enquanto que o lado direito ilustra o resultado da transformação de código para uso de *expressões lambda*.

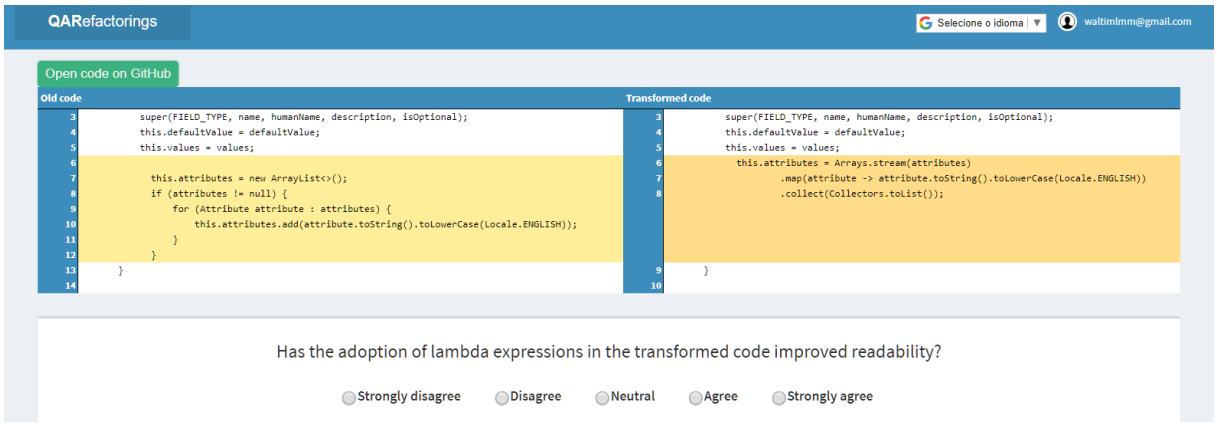

Figura 3.2: Tela de apresentação dos trechos de código no questionário

Logo abaixo dos trechos de código são apresentadas as perguntas a serem respondidas pelos participantes em *Escala Likert*. As questões podem ser respondidas pelos entrevistados selecionando uma das cinco opções: Concordo plenamente (5), Concordo (4), Neutro (3), Discordo (2) e Discordo plenamente (1). Além disso, existe uma questão onde os participantes informam qual código eles preferem contendo apenas duas opções (sim ou não) e uma questão discursiva que permite aos participantes (opcionalmente) comentarem sobre suas escolhas ao final de cada questionário.

A Figura 3.3 ilustra como o *survey* foi conduzido. Os pesquisadores envolvidos selecionaram e cadastraram um grupo de questões que foram apresentadas para cada par de trechos de código. Em um primeiro momento, foi conduzido um piloto com uma quantidade pequena de estudantes (5) para avaliar se a ferramenta capturava de forma esperada a opinião dos desenvolvedores. Após a condução do piloto foram feitos vários ajustes no layout e funcionalidades da ferramenta validando para as próximas execuções do *survey*.

A condução do *survey* consistiu primeiro na geração de uma amostra dos exemplos de transformação de código que devem compor o *survey* individual para cada respondente. Tal amostra pode considerar um *cluster* das transformações. Ou seja, os trechos de código estão separados por tipos de transformações como por exemplo: saindo de uma classe anônima para um lambda, *forEach* para um lambda e assim sucessivamente. Da base de dados contendo 88 trechos, foi feita uma segunda classificação para remover trechos que

não estariam aptos a serem avaliados durante a execução do *survey*. Está classificação possui os seguintes critérios:

(a) trechos de código que já possuíam *expressões lambda* antes da transformação foram classificados como não aptos;

(b) trechos de código que apresentaram um comportamento diferente após a transformação foram classificados como não aptos.

Após a segunda classificação restaram 66 trechos avaliados como aptos sendo 37 extraídos da ferramenta de análise WebMinerApp e 29 extraídos da ferramenta de rejuvenescimento RJTL [36]. Logo em seguida, foi extraída aleatoriamente uma amostra de 9 trechos, gerando para cada participante um survey online com base nas questões previamente cadastradas, contendo um conjunto de transformações e questões que devem ser respondidas para cada transformação. Além de responder as questões sobre as transformações, os participantes selecionados também informaram alguns dados pessoais (como gênero, formação acadêmica e experiência com programação funcional). As respostas dos entrevistados foram armazenadas em um banco para serem posteriormente analisadas.

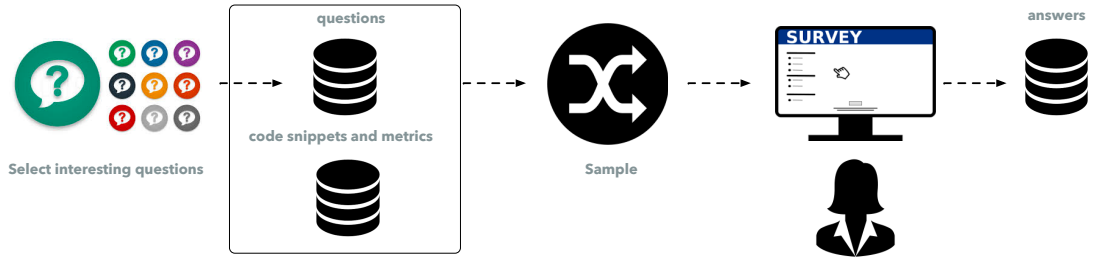

Figura 3.3: Fluxograma de coleta das respostas dos entrevistados

O *Survey* foi conduzido em duas fases. A primeira considerou respostas de profissionais com maior experiência em programação, atuando em empresas desenvolvimento de software. A segunda fase coletou apenas a percepção de estudantes. Para convidar os estudantes e desenvolvedores. Os pesquisadores envolvidos elaboraram e enviaram um e-mail para alguns conhecidos, solicitando que o *survey* fosse respondido e que compartilhassem com outros desenvolvedores/estudantes em seus grupos de amigos, caracterizando uma amostragem de Snowball [21]. A Tabela 3.2 apresenta os dados dos participantes da primeira fase. E por fim, a Tabela 3.3 apresenta os dados dos participantes da segunda fase.

Tabela 3.2: Caracterização dos participantes da pesquisa (primeira fase)

ID	Formação	Experiência com Lambda	Experiência com programação funcional	Experiência com Java
6	Estudante	Não	1 a 4 anos	4 anos
7	Graduado	Sim	1 a 4 anos	2 anos
12	Mestre	Sim	Mais de 5 anos	11 anos
13	Graduado	Sim	1 a 4 anos	4 anos
14	Mestre	Sim	1 a 4 anos	10 anos
15	Graduado	Não	Mais de 5 anos	11 anos
16	Mestre	Sim	1 a 4 anos	11 anos
17	Mestre	Sim	Mais de 5 anos	11 anos
19	Mestre	Não	Sem experiência	7 anos
20	Graduado	Sim	1 a 4 anos	5 anos
22	Graduado	Sim	Mais de 5 anos	5 anos
25	PhD	Sim	Sem experiência	10 anos
26	Graduado	Sim	1 ano	11 anos
27	Mestre	Não	Sem experiência	5 anos
28	Mestre	Sim	Sem experiência	7 anos
30	PhD	Não	4 a 5 anos	5 anos
31	Mestre	Sim	1 ano	4 anos
32	Graduado	Sim	1 a 4 anos	2 anos
33	Estudante	Não	1 ano	1 ano
34	Graduado	Sim	Sem experiência	7 anos
36	Graduado	Sim	Mais de 5 anos	11 anos
38	Estudante	Sim	Sem experiência	1 ano
39	Graduado	Sim	1 ano	1 ano
41	Estudante	Sim	Sem experiência	1 ano
43	Estudante	Sim	1 ano	4 anos
45	Graduado	Sim	4 a 5 anos	5 anos
46	Graduado	Não	Sem experiência	1 ano
47	Graduado	Sim	Sem experiência	11 anos

Tabela 3.3: Caracterização dos participantes da pesquisa (segunda fase)

ID	**Formação**	**Experiência com lambda**	**Experiência com programação funcional**	**Experiência com Java**
7	Estudante	Sim	1 ano	7 anos
10	Estudante	Sim	5 anos	11 anos
12	Estudante	Sim	1 ano	1 ano
13	Estudante	Sim	1 a 4 anos	2 anos
14	Estudante	Sim	1 a 4 anos	4 anos
15	Estudante	Sim	1 ano	5 anos
19	Estudante	Sim	1 ano	3 anos
20	Estudante	Sim	1 ano	3 anos
21	Estudante	Sim	1 a 4 anos	4 anos
23	Estudante	Sim	1 ano	3 anos
27	Estudante	Sim	1 a 4 anos	1 ano
30	Estudante	Sim	1 ano	1 ano
32	Estudante	Sim	1 a 4 anos	3 anos
33	Estudante	Sim	1 a 4 anos	2 anos
43	Estudante	Sim	1 ano	1 ano

Capítulo 4

Resultados

Neste Capítulo é apresentado os resultados do estudo realizado. Inicialmente, na Seção 4.1 é discutido os resultados das métricas de legibilidade e uma comparação entre os modelos. Em seguida, na Seção 4.2 será discutido os resultados para a primeira fase do survey com desenvolvedores da industria. Na seção 4.3 será discutido os resultados para a segunda fase do survey avaliando a opinião de estudantes. Logo em seguida, na seção 4.4 são combinados os resultados quantitativos e qualitativos com o intuito de validar os resultados das métricas. Na seção 4.5 é apresentado alguns testes não paramétricos para avaliar se existem diferenças entre a percepção dos desenvolvedores em relação a sua experiência e seu nível acadêmico. Por fim, na seção 4.6 é realizada uma meta-análise entre os resultados dos estudos qualitativos.

4.1 Aplicando as métricas de legibilidade para avaliar as transformações

Em uma primeira análise quantitativa, foram aplicadas as métricas de legibilidade [7, 43] para cada par de trechos de código (código legado e após a transformação com introdução de *expressões lambda*). Um total de 66 pares de trechos foram anaĺisados e para cada modelo obteve-se os seguintes resultados:

- Para a métrica de Buse e Weimer [7] 44 (66,5%) transformações apresentaram piora na legibilidade após a introdução de *expressões lambda*, apenas 13 (19,5%) transformações apresentaram melhora na legibilidade e por fim, 9 (14%) transformações mantiveram a legibilidade. As transformações ao qual a classificação de legibilidade foi mantida, receberam 0 antes e após a introdução das *expressões lambda*;

- Para a métrica proposta por Posnett et al [43] 31 (47%) transformações obtiveram uma melhora após a introdução de *expressões lambda* e 35 (53%) pioraram a legibilidade após a transformação.

Ao analisar os resultados dos modelos, também foi possível observar se as métricas concordavam ou discordavam entre si para indicar uma melhora ou piora na legibilidade após a introdução de *expressões lambda*. As métricas concordaram em 34 (51,5%) dos trechos avaliados e discordaram em 32 (49,5%). De acordo com o modelo de Posnett et al [43] 93% dos trechos após a transformação são avaliados com alta legibilidade e 95% são avaliados com baixa legibilidade pelo modelo de Buse e Weimer [7]. As Tabelas 4.1 e 4.2 apresentam a comparação entre os resultados dos modelos para as transformações avaliadas pelo estudo empírico 4.2 e 4.3.

Tabela 4.1: Resultados das métricas de legibilidade para as transformações avaliadas no estudo empírico 1.

ID	Modelo BW	Modelo Posnett
1027	Piorou	Piorou
1035	Piorou	Piorou
1052	Melhorou	Melhorou
1062	Piorou	Piorou
1166	Piorou	Melhorou
1180	Melhorou	Piorou
1182	Manteve	Melhorou
1183	Piorou	Melhorou
1192	Piorou	Piorou

Tabela 4.2: Resultados das métricas de legibilidade para as transformações avaliadas no estudo empírico 2.

ID	Modelo BW	Modelo Posnett
1022	Manteve	Melhorou
1026	Piorou	Piorou
1042	Melhorou	Piorou
1060	Melhorou	Piorou
1061	Piorou	Piorou
1070	Piorou	Piorou
1178	Manteve	Piorou
1180	Melhorou	Piorou
1183	Piorou	Melhorou
1185	Piorou	Piorou
1188	Piorou	Melhorou
1192	Piorou	Piorou

Com base nos resultados apresentados, foi possível observar que as avaliações entre as métricas concordam em boa parte dos casos para indicar se houve uma piora ou melhora após a introdução de *expressões lambda* e que grande parte das transformações possuem baixa legibilidade de acordo com a métrica de Buse e Weimer.

4.2 Estudo Empírico 1

Considerando o primeiro estudo empírico, 28 participantes avaliariam de 3 a 6 trechos de código. Para cada par de trecho de código, os participantes responderam a 3 questões. O *Survey* ficou disponível durante 16 dias e cada participante gastou em média 02:30 minutos para avaliar cada trecho de código. Foram utilizadas duas formas de análise de dados para as respostas do *survey*. Primeiro, as distribuições de respostas foram apresentadas em forma de gráfico para construir uma visão ampla de cada questão fechada. Na segunda análise, seguimos os procedimentos de codificação aberta e codificação axial como descrito em Strauss e Corbin [9]. Foram avaliadas as respostas dos desenvolvedores para a questão aberta do survey. Ainda assim, embora algumas citações tenham erros de digitação, foi decidido mantê-las para tornar as respostas mais fidedignas. Foram excluídas as respostas para três trechos que continham erros de código o que resultou nessa variação da quantidade de trechos avaliados pelos participantes.

4.2.1 Impacto na Legibilidade

O objetivo da **SQ1** é avaliar se, na percepção dos desenvolvedores, a introdução de *expressões lambda* ocasiona melhoria na legibilidade do código. De acordo com a Tabela 4.3 (considerando todos os trechos avaliados), 11.1% dos participantes concordaram plenamente que o código após a transformação melhorou a sua legibilidade, enquanto que 39.7% concordaram que ocorreu uma melhora, 21.4% discordaram e apenas 3.2% discordaram plenamente. Isso sugere que introdução de *expressões lambda* apresentou melhora a legibilidade.

A Figura 4.1 apresenta a distribuição das respostas agrupadas por cada transformação. É possível observar que as transformações 1035, 1052, 1180, obtiveram mais de 60% das respostas indicando uma melhora na legibilidade após a transformação de código. Por outro lado, a transformação 1182 obteve 43% de respostas indicando que não houve melhora após a transformação.

Tabela 4.3: Resultado consolidado para a questão A adoção de expressões lambda no código transformado melhorou a legibilidade?

SQ1	Frequência	Porcentagem	Porcentagem Acomulada
Strongly disagree	4	3.2	3.2%
Disagree	27	21.4	24.6%
Neutral	31	24.6	49.2%
Agree	50	39.7	88.9%
Strongly Agree	14	11.1	100.0%
Missing	0	0.0	
Total	126	100.0	

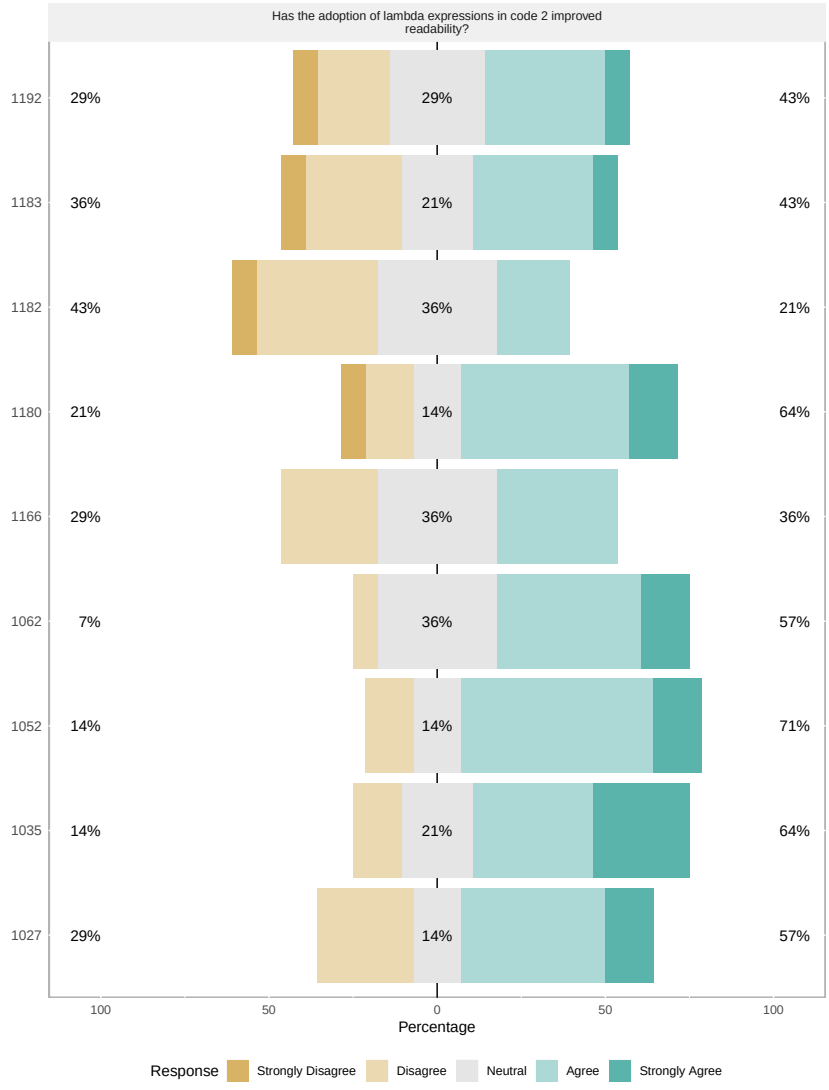

Figura 4.1: Taxas de respostas para a questão 1 do *survey*.

De acordo com Strauss e Corbin [9] para realizar a codificação aberta, os dados são

divididos em partes. Logo em seguida são examinados, comparados por similaridades e diferenças. Por fim, são feitas algumas questões sobre seu reflexo nos dados. Descrever os dados é a primeira etapa na análise com o intuito de que na finalização se tenha uma quantidade de códigos definida que permita sua categorização. A Tabela 4.4 apresenta a análise das respostas das questões abertas e mostra que alguns entrevistados consideram que a sintaxe do código utilizando *expressões lambda* é mais compreensível em alguns cenários.

Tabela 4.4: Análise das respostas dos participantes e codificação dos dados.

TRECHOS DAS RESPOSTAS	CÓDIGOS
"Aqui a transformação fez sentido por eliminar o uso de classe anônima (anonymous inner class) com método trivial (muito utilizado para implementar o padrão Command em Java"	Eliminação de classe anônima que implementa método trivial
"Acho que a substituição do for each normal para o obj.forEach() não se justifica quando não há uso de chamadas map/filter ou alguma outra função que facilita o processamento de listas."	Não utilizar a função foreach sem o auxílio de funções que facilitam o processamento de listas
"Acredito que daria para fazer assim também. ao invés de usar o foreach usar o map para criar o objeto novo .map(obj ->new Objeto(obj)).collect(Collectors.toList());"	utilizar a função map para criação de novos objetos ao invés de foreach
"O código transformado é muito mais claro pela mera remoção de código boilerplate proporcionada pelas expressões lambda."	remoção de código boilerplate

Com base nas respostas, em alguns cenários nas quais uma *transformação de código remove o uso de classes anônimas que utilizam métodos triviais ocasiona melhoria na legibilidade do código*. A transformação 1052 executa o mapeamento dos objetos *CharSequence...* e adição dos novos elementos a uma nova lista pelo método *collect*, apresentando uma sintaxe mais clara para 71% dos desenvolvedores participantes. Para a transformação 1182, que não apresentou melhora na legibilidade, foi possível perceber que cenários nos quais *o rejuvenescimento de um código legado passando de um* For *para* Foreach utilizando expressões lambda *sem o uso de chamadas* Map *ou* Filter *podem não apresentar melhorias a legibilidade*.

A transformação 1183 substitui o método *for* interativo por uma *expressão lambda* que mapeia os objetos removendo um valor do objeto e adicionando outro caso o predicado seja atendido. Para essa transformação, 43% dos entrevistados reportaram que o código melhorou após a legibilidade. Um participante sugeriu que utilização do método *Map* ao invés do *ForEach* traria ainda mais benefícios para a legibilidade. A transformação 1027 remove parte do código considerado como *boilerplate* (trechos de código que podem

ser incluídos em muitos lugares com pouca ou nenhuma alteração) melhorando assim a legibilidade do código.

Com base nas respostas, foi possível identificar cenários onde a introdução de *expressões lambda* melhoram a compreensão e também cenários que não ocasionam melhorias na perspectiva dos desenvolvedores. A próxima etapa consiste em agrupar os códigos permitindo ao pesquisador agrupar códigos semelhantes para formar categorias iniciais que serão trabalhadas posteriormente como descrito por Strauss e Corbin [9]. A Tabela 4.5 apresenta as categorias identificadas a partir dos códigos.

Tabela 4.5: Formando categorias a partir do código.

CÓDIGOS ABERTOS	**CATEGORIAS**
eliminar classe anônima com método trivial	eliminação de classes anônimas
foreach sem funções auxiliares	foreach sem funções axiliares
map ao invés de foreach para novos objetvos	foreach para mapear novos objetos
remoção de código boilerplate	eliminação de boilerplate

com a criação das categórias através dos códigos, é possível iniciar o processo de codificação axial de acordo com a metodologia. É realizado novamente o agrupamento com o intuito de fazer conexões entre as categorias para identificar categorias mais abrangentes. A codificação axial, de acordo com Strauss e Corbin [9] faz comparação constante com o intuito de permitir ao pesquisador identificar semelhanças e diferenças entre as situações, ações, eventos e/ou unidades sociais que formaram as categorias abertas. A Tabela 4.6 apresenta as categorias e suas subcategorias separando as situações identificadas durante o estudo.

Tabela 4.6: Formando categorias a partir das subcategorias.

CÓDIGOS ABERTOS	**SUBCATEGORIAS**	**CATEGORIAS**
eliminar classe anônima com método trivial	eliminação de classes anônimas	Situações desejadas
remoção de código boilerplate	eliminação de boilerplate	
foreach sem funções auxiliares	foreach sem funções axiliares	Situações indesejadas
map ao invés de foreach para novos objetvos	foreach para mapear novos objetos	

Como resposta para a questão de pesquisa RQ1, foi possível inferir que os desenvolvedores percebem uma melhora na legibilidade do código após a introdução de *expressões lambda* em alguns cenários específicos.

Como resposta para a questão de pesquisa RQ2, alguns cenários onde a introdução de *expressões lambda* melhoram a legibilidade puderam ser identificados: transformações ao qual é eliminado código considerado boilerplate, remoção de classes anônimas que implementam métodos triviais. Também foi identificado um cenário onde a transformação não leva a melhorias no código: uso do método recursivo *ForEach* sem o auxilio de outros métodos que melhoram o processamento de listas como *Map, Filter*.

4.2.2 Preferência de sintaxe

Para responder a questão de pesquisa **RQ1** também foi seguida a mesma abordagem proposta por do Santos e Gerosa [15]. Isso possibilitou avaliar a escolha dos participantes entre o código legado e código resultante da introdução de *expressões lambda*. A Tabela 4.7 mostra que os participantes preferem o código após a introdução de *expressões lambda* na maioria dos tipos avaliados. Podemos observar também que o tipo *(for) to ForEach ->Lambda* obteve um empate em relação as escolhas dos participantes, podendo ser ocasionado pela situação relatada para transformação 1182.

Tabela 4.7: Análise das escolhas dos desenvolvedores por tipo de transformação.

Tipo da transformação	**Código legado**	**Código com introdução de lambda**
(for) to Filter ->Lambda	28%	72%
(for) to Filter/ForEach ->Lambda	40.5%	59.5%
(for) to ForEach ->Lambda	50%	50%
(for) to Map ->Lambda	7%	93%
(for) to Map/Collect ->Lambda	7%	93%
Anonymous inner classes ->Lambda	14%	86%

A Tabela 4.8 faz a comparação da métrica SLOC com a escolha dos desenvolvedores entre o código legado e após a introdução de *expressões lambda*. Os dados mostram que mais da metade dos desenvolvedores preferem o código após a introfução de *expressões lambda* em situações que houve redução da quantidade de linhas. Ao observar as transformações 1027, 1035 e 1052, percebe-se que houve uma redução significativa em relação a quantidade de linhas após a introdução das *expressões lambda* e foram escolhidas por

92,8% dos participantes. Por outro lado, as transformações 1182 e 1183 ocasionaram em um aumento da quantidade de linhas e foram escolhidas por 58% dos desenvolvedores.

Tabela 4.8: Comparação da quantidade de linhas com as escolhas dos desenvolvedores.

Transformação	Qtd. Linhas (Antes)	Qtd. Linhas (Depois)	Código legado	Código com introdução de lambda
1027	8	5	7%	93%
1035	22	11	7%	93%
1052	9	6	7%	93%
1062	9	3	28%	72%
1166	3	4	58%	48%
1180	10	6	14%	86%
1182	8	12	42%	58%
1183	10	13	42%	58%
1192	15	11	35%	65%

A transformação 1166 é um exemplo claro onde a transformação de um código envolvendo *expressões lambda* não apresenta melhorias a legibilidade na percepção dos desenvolvedores. Também é possível observar que essa transformação ocasionou o aumento de uma linha ao trecho de código em relação ao código legado, mas que não foi um fator impactante na legibilidade como avaliado por alguns desenvolvedores. Ao observar o código legado da transformação 1062 pode-se notar que a introdução de *expressões lambda* reduziu em 66% a quantidade de linhas de código e foi selecionada por 72% dos entrevistados. Ao observar a transformação 1192, nota-se que não houve uma redução significativa de linhas de código. O código recebe uma lista de elementos e popula outra lista apenas com elementos diferentes de nulo.

Além das analises comparando os dados da tabela, foi realizado um teste de correlação de Spearman [39]. O teste de correlação de Spearman avalia a correlação entre variáveis ao qual não pressupõe que elas sigam distribuição normal. Ao realizar o teste, entre a quantidade de linhas que reduziram após a introdução de *expressões lambda* e a quantidade de participantes que escolheram o código após a transformação, foi possível observar que existe uma correlação ($\mathbf{p = 0.04462}$), ou seja, quanto maior a redução de linhas maior o número de desenvolvedores que preferem o código após a adição de *expressões lambda*. Com base nas respostas, foi possível inferir que a redução da quantidade de linhas é importante para avaliar a qualidade de uma transformação e que a quantidade de linhas reduzidas após a introdução de *expressões lambda* pode indicar uma melhora na legibilidade de acordo com a percepção dos desenvolvedores entrevistados. A Tabela 4.9 compara os resultados da métrica de complexidade ciclomática com o código selecionado pelos desenvolvedores. Com essa análise, pode-se notar que todos os códigos onde se obteve uma

redução da métrica de complexidade ciclomática após a introdução de *expressões lambda* foram selecionados pelos entrevistados. Ao observar a transformação 1166, nota-se que apesar de manter a mesma complexidade, 58% parte dos desenvolvedores selecionaram o código legado.

Além da comparação dos dados apresentados na Tabela 4.9, foi realizado um teste de correlação de Spearman [39]. O teste foi utilizado para avaliar se a redução da complexidade ciclomática possui correlação com o quantidade de participantes que escolheram o código após a introdução de *expressões lambda*. Ao realizar o teste não foram encontradas evidências de correlação entre as variáveis.

Tabela 4.9: Comparação da métrica de complexidade ciclomática com as escolhas dos desenvolvedores.

Transformação	Complexidade Ciclomática (Antes)	Complexidade Ciclomática (Depois)	Código legado	Código com introdução de lambda
1027	2	2	7%	93%
1035	1	1	7%	93%
1052	2	1	7%	93%
1062	3	1	28%	72%
1166	2	2	58%	42%
1180	1	1	14%	86%
1182	3	3	42%	58%
1183	3	2	42%	58%
1192	3	2	35%	65%

Com base nesses resultados, apesar da métrica de complexidade ciclomática ser considerada relevante para avaliar a qualidade de uma transformação, foi possível inferir que a substituição de código legado pela adição de *expressões lambda* pode reduzir a complexidade do software.

Como resposta para a questão de pesquisa RQ1, foi possível inferir que as métricas de SLOC e complexidade ciclomática são importantes para avaliar a qualidade de uma transformação e que podem ser fortes indicadores se uma transformação com a introdução de *expressões lambda* melhora a legibilidade. Também foi possível inferir que a introdução de expressões lambda pode reduzir a complexidade e tamanho do programa.

4.3 Estudo Empírico 2

Considerando o segundo estudo empírico, uma nova execução do *Survey* foi realizada com o intuito de capturar a opiniões de alunos sobre o impacto na legibilidade de código após

a introdução de *expressões lambda*. A seguinte questão foi apresentada:

(Q1:) A adoção de *expressões lambda* no código transformado melhorou a legibilidade?;

Para esse estudo, 15 participantes responderam a questão e avaliaram 12 pares de trechos de código. Foram selecionados apenas 12 trechos para não tornar o *survey* tão cansativo e evitar que os alunos desistissem e não completasse as avaliações. Os participantes foram divididos em dois grupos, onde cada um avaliou 6 transformações sendo 3 delas extraídas através da ferramenta de monitoramento WebMinerApp [34] e 3 extraídas transformações realizadas pela ferramenta RJTL [36]. Para as respostas do *survey*, as distribuições serão apresentadas em forma de gráfico para construir uma visão ampla da questão fechada considerando todos os trechos avaliados.

4.3.1 Impacto em Legibilidade

O objetivo da **Q1** é avaliar se, na percepção dos alunos, a transformação de um código utilizando *expressões lambda* representa melhoria na legibilidade do código. Para essa questão, 28,9% dos participantes concordaram plenamente que o código após a transformação melhorou a legibilidade, enquanto que 36.7% concordaram que houve uma melhora, 7,8% discordaram e apenas 2.2% discordaram plenamente que introdução de *expressões lambda* indicando que a transformação não melhorou a legibilidade. De acordo com a Figura 4.2 66% dos alunos acreditam que a adição de *expressões lambda* melhora a legibilidade do código.

> Com base nesses resultados, é possível inferir que os alunos percebem uma melhora na legibilidade do código após a introdução de *expressões lambda*.

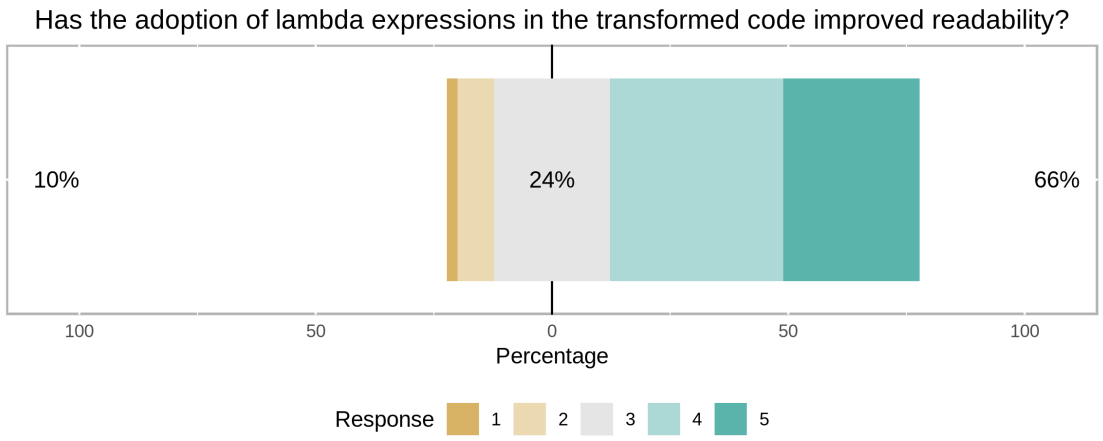

Figura 4.2: Taxas de respostas para a questão 1 do *survey*.

4.4 Comparação de resultados das métricas de legibilidade com as avaliações dos desenvolvedores

Foi realizada uma análise para avaliar se os resultados das métricas de legibilidade para o trecho após a transformação com introdução de *expressões lambda* correspondiam com a avaliação dos entrevistados. Os resultados das métricas são apresentados da seguinte forma: A primeira barra representa o resultado para a métrica de Posnett, A segunda barra representa o resultado para a métrica de Buse e Weimer e por fim, a terceira e ultima barra representa a avaliação dos participantes. Apesar dos resultados das métricas serem semelhantes, o direcionamento para apontar uma melhora ou piora pelas métricas se diferem. A métrica de Buse e Weimer [7] avalia a legibilidade de um trecho de código em um intervalo de $(0, 1)$ onde 0 indica a pior legibilidade e 1 indicando alta legibilidade. Para o modelo de Posnett [43], quanto mais próximo de 0 mais alta é a legibilidade do trecho avaliado. Para a métrica de Posnett, foi realizado um pequeno ajustes nos resultados, invertendo de 0 para 1, ou seja quanto mais proximo de 1, mais legivel com intuito de facilitar a comparação com os outros resultados avaliados.

A Figura 4.3 compara os resultados das métricas de legibilidade nesta pesquisa [7, 43] com a avaliação dos participantes. O modelo BW classifica a transformação 1180 com uma pontuação alta (**0,858**), indicando que houve melhora após a introdução de *expressões lambda*. Acredita-se que essa classificação é resultante das linhas em branco contidas no código. A transformação 1182 recebeu a pior classificação pela métrica, que por sua vez se aproxima da análise dos participantes onde apenas 21% indicaram perceber uma melhora após a introdução de *expressões lambda*. Ao avaliar os resultados da métrica de legibilidade proposta por Posnett [43], foi possível observar que a métrica apresenta uma redução na legibilidade após a introdução de *expressões lambda* na maioria dos casos e classifica a transformação 1192 com uma baixa legibilidade, tendo seu resultado próximo da avaliação dos participantes. A Figura 4.4 compara os resultados para as transformações avaliadas durante o segundo estudo empírico. A transformação 1026 recebe a pior avaliação pelo modelo BW após a introdução de *expressões lambda*, onde o resultado se difere da avaliação dos participantes (83%) que perceberam uma melhora após a transformação.A transformação 1070 recebe uma classificação alta pela métrica de Posnett que por sua vez se diverge da avaliação dos estudantes. Os dados mostram que as avaliações entre as métricas se divergem e que por sua vez, também divergem da avaliação dos participantes em boa parte dos casos.

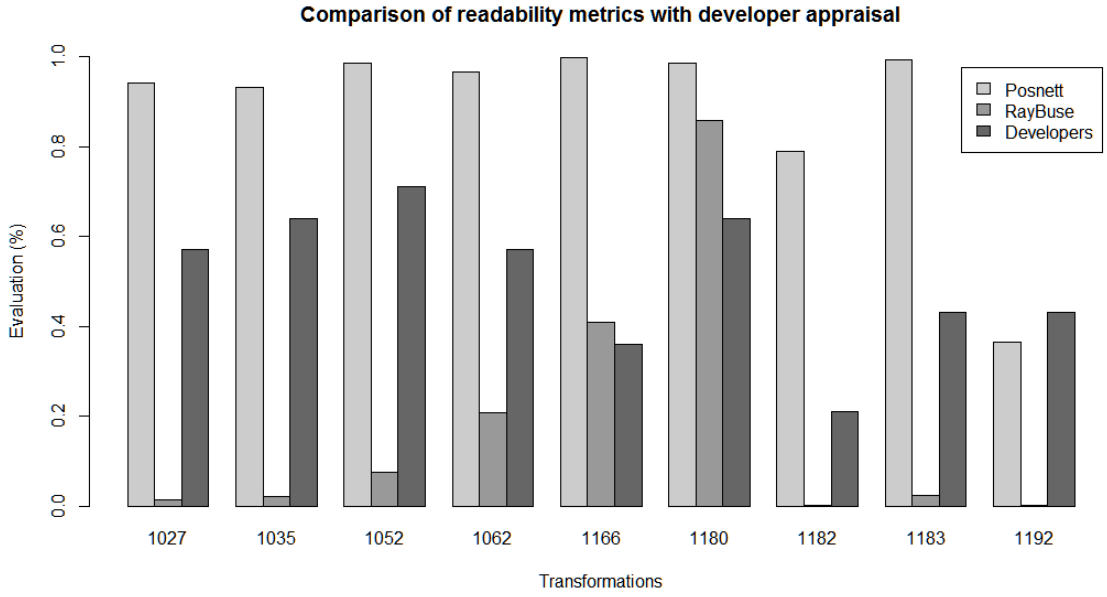

Figura 4.3: Comparação das métricas de legibilidade com a avaliação dos desenvolvedores.

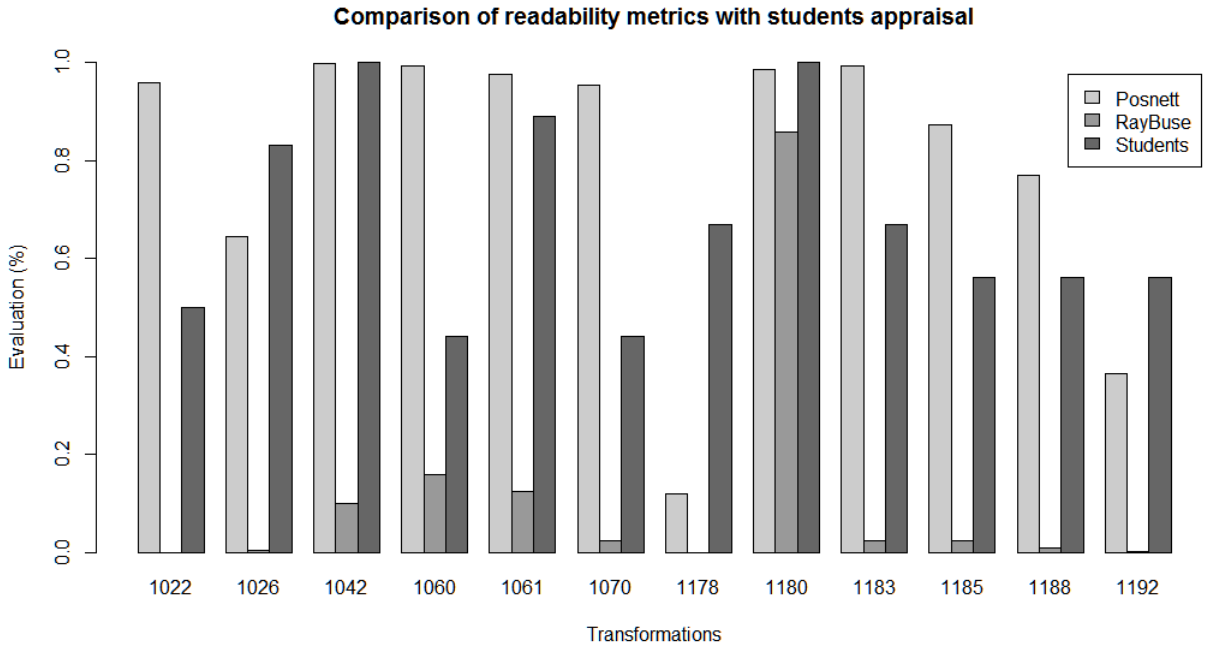

Figura 4.4: Comparação das métricas de legibilidade com a avaliação dos estudantes.

Com base nesses resultados, foi possível observar que as métricas de legibilidade avaliadas possuem resultados que divergem da avaliação dos participantes e podem não ser adequadas para avaliar a legibilidade em casos onde uma transformação de código leva a adição de *expressões lambda*. Todavia, apesar de grande parte dos trechos avaliados terem indicado uma redução na legibilidade pelas métricas, os desenvolvedores demonstraram perceber uma melhora na legibilidade contrastando os resultados das métricas avaliadas.

4.5 A percepção dos desenvolvedores se difere em relação a sua experiência ou nível acadêmico?

Foi realizado um teste não paramétrico com o intuito de avaliar se existe diferença entre a percepção dos desenvolvedores agrupados em diferentes categorias. O teste aplicado foi de *permutação de independência* que pode ser usado para dados unidirecionais com uma variável dependente ordinal. O teste é uma alternativa para os testes não paramétricos tradicionais (Mann-Whitney ou Kruskal-Wallis) em que há uma variável de bloco (adequado para este caso, onde os participantes avaliaram trechos diferentes) de acordo com Hothorn et al. [26].

Para essa avaliação, foi estabelecida as seguintes hipóteses:

- **Hipótese nula**: *A percepção entre os grupos é semelhante.*
- **Hipótese alternativa**: *Existe pelo menos um grupo que possui percepção diferente dos demais.*

O teste de permutação teve como parâmetro as respostas da primeira questão do *survey* e os desenvolvedores agrupados e separados em três categorias: Formação, Experiência com programação funcional e Experiência com Java. As Figuras 4.5, 4.6 e 4.7 mostram como os desenvolvedores foram divididos entre grupos contidos nas categorias ao qual foram submetidos os testes. A Figura 4.5 mostra que os desenvolvedores com nível de Estudante e Graduado tiveram uma taxa de resposta maior para o item ***Concordo*** representado pelo número 4 na Figura 4.5 indicando uma melhora na legibilidade após a introdução de *expressões lambda* em sua percepção. Por outro lado, desenvolvedores com Mestrado mostraram-se divididos em relação a sua opinião e doutores reportaram que não houve uma melhora na legibilidade, possuindo a maior taxa de respostas para o item ***Discordo*** representado pelo número 3 na Figura
reffig:formation.

Com os resultados apresentados pela Figura 4.5 foi possível observar que desenvolvedores com maior nível acadêmico possuem uma avaliação mais rigorosa sobre o impacto na legibilidade com a adição de *expressões lambda* em trechos de código Java. Ao aplicar o teste de permutação para esta categoria, foi obtido um valor de $\mathbf{p = 0{,}01675}$. Como o valor de p é menor que o nível de significância de *0,05*, pode-se concluir que existem diferenças significativas nas percepções entre os desenvolvedores em relação ao nível acadêmico.

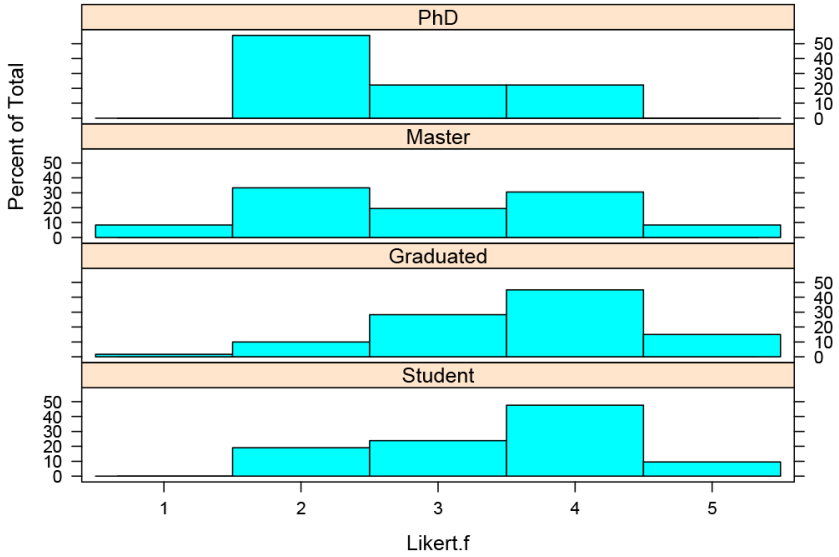

Figura 4.5: Percepção dos desenvolvedores agrupados pelo nível acadêmico.

Como o teste de permutação de independência apresentou diferença entre os grupos, uma análise (*post-hoc*) pode ser realizada para distinguir quais grupos se diferem de acordo com Hothorn et al[26]. A Tabela 4.10 apresenta as comparações entre pares de grupos realizadas pelo teste de permutação pareada. Os grupos que apresentaram diferença possuem o valor de $p < 0{,}05$ com 95% de confiança.

Tabela 4.10: Comparações entre pares de grupos realizadas pelo teste de permutação pareada.

Comparações	**p.value**	**p.adjust**
Graduado - Mestre	***0.003888***	***0.01757***
Graduado - PhD	***0.005857***	***0.01757***
Graduado - Estudante	0.5471	0.54710
Mestre - PhD	0.4578	0.54710
Mestre - Estudante	0.09477	0.14220
PhD - Estudante	***0.03647***	***0.07294***

A Figura 4.6 mostra que desenvolvedores com 1 ano e entre 1 a 4 anos de experiência percebem uma melhora na legibilidade. Os desenvolvedores com 1 ano apresentaram maior taxa para o item de resposta ***Concordo*** representado pelo número 4 na Figura 4.6 em quanto os desenvolvedores sem experiência com programação funcional apresentaram a maior taxa de ***Discordo*** representado pelo número 2.

Ao aplicar o teste de permutação para esta categoria, foi obtido um valor de $\mathbf{p = 0.06}$. Como o valor de p é maior que o nível de significância de *0,05*, pode-se concluir que não existem diferenças significativas nas percepções entre desenvolvedores em relação a sua experiência com programação funcional.

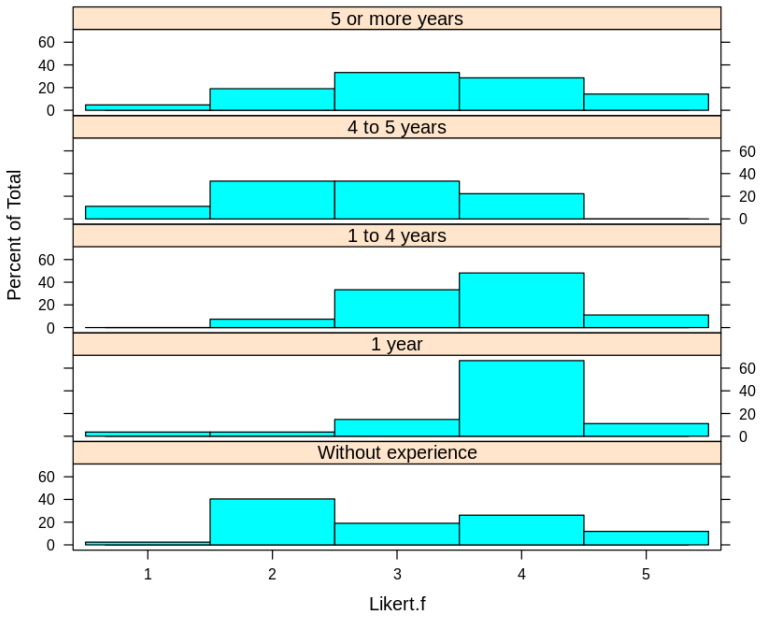

Figura 4.6: Percepção dos desenvolvedores agrupados pela experiência com programação funcional.

A Figura 4.7 mostra que desenvolvedores com maior nível de experiência com Java percebem uma melhora na legibilidade. Os desenvolvedores com até 3 anos experiência apresentaram maior taxa para o item de resposta ***Concordo*** representado pelo número 4 na Figura 4.7 em quanto os desenvolvedores entre 6 a 7 anos apresentaram a maior taxa de ***Discordo*** representado pelo número 2. Os desenvolvedores entre 2 a 3 anos e 4 a 5 anos apresentaram a mesma taxa de percepção para o item ***Neutro*** e ***Concordo*** o que pode significar uma dúvida sobre o impacto na legibilidade causado pela introdução de *expressões lambda*. Ao aplicar o teste de permutação para esta categoria, obteve um valor de $\mathbf{p = 0.96}$. Como o valor de p é maior que o nível de significância de *0,05*, pode-se

concluir que não existem diferenças significativas nas percepções entre desenvolvedores com relação a sua experiência com Java.

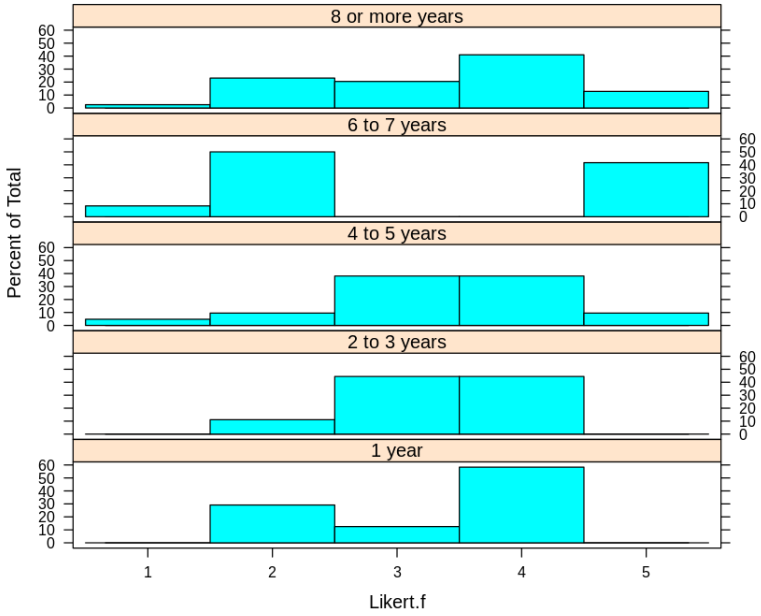

Figura 4.7: Percepção dos desenvolvedores agrupados pela experiência com programação Java.

Com base nos resultados, foi possível identificar que desenvolvedores com maior nível acadêmico discordam que a introdução de *expressões lambda* melhoram a legibilidade do código e que a percepção entre os níveis de formação se diferem. Também foi possível observar que desenvolvedores com menos tempo de experiência com programação funcional e java percebem uma melhora na legibilidade com a introdução de *expressões lambda*. Acredita-se que os desenvolvedores com menor tempo de experiência, podem ter iniciado seu aprendizado com a linguagem Java após a introdução das novas construções e que estejam mais familiarizados com o uso de lambdas.

4.6 Meta-análise

Foi realizada uma meta-análise que combina os resultados das duas fases do survey executadas durante a pesquisa. A meta-análise é considerada um procedimento quantitativo utilizado para aumentar a confiança das revisões sistemáticas, de acordo Horthon e Everitt [25]. A meta-análise é um procedimento popular no campo da medicina baseada em evidências. Embora as experiências descritas neste trabalho tenham sido idealizadas e

direcionadas ao propósito desta pesquisa, foi seguido o procedimento de meta-análise com o intuito de combinar seus resultados computando seus tamanhos de efeito e avaliando seu efeito sumário e heterogeneidade de acordo com Borenstein et al, [4]. Para avaliar se a (introdução de *expressões lambda*) possuem vantagens em relação a legibilidade na percepção dos desenvolvedores comparado a utilização de códigos legados (solução tradicional), foi seguido o modelo de efeitos fixos, principalmente porque o número de estudos investigados é pequeno de acordo com Konstantopoulos et al [30]. Inicialmente foi calculado o tamanho do efeito e o fator de peso para cada estudo. Os resultados da meta-análise são apresentados usando gráficos de floresta e serão discutidos o tamanho do efeito e da heterogeneidade dos estudos.

Para calcular o tamanho de efeito de cada estudo, foi utilizado o modelo odds ratio (OR). O calculo do odds ratio para cada estudo é dado pela seguinte formula de acordo com Greenland et al [22]:

$$OR = \frac{\mathcal{P}e/(1-\mathcal{P}e)}{\mathcal{P}c/(1-\mathcal{P}c)}$$

Onde ($\mathcal{P}e$) e ($\mathcal{P}c$) representam a probabilidade do evento ocorrer no grupo experimental e controle. Como entrada para a função são informados a quantidade de eventos que ocorreram tanto no grupo experimental quanto controle e o tamanho total da amostra dos dois grupos de acordo com Schwarzer et al [49]. Para estimar as diferenças entre o tamanho do efeito (heterogeneidade), existem várias estatísticas para testar a heterogeneidade entre os diferentes estudos segundo [4]. Para esta pesquisa, será utilizada a análise de heterogeneidade baseada na estatística I^2—recomendada por não ser afetada pelo número de estudos.

Existem diferentes métodos e abordagens para apresentar os resultados de uma meta-análise. Todavia, o método mais utilizado consiste na apresentação dos resultados através do gráfico de floresta (*forest plot*). Tal gráfico é comumente utilizado por agrupar as informações individuais de cada estudo incluindo os resultados da meta-análise e, por este motivo, sumariza em uma única figura todas as informações de efeito e contribuição de cada estudo para a análise de acordo com Berwanger et al [3]. A Figura 4.8 apresenta o gráfico de floresta com os resultados da meta-análise. As duas primeiras linhas do gráfico representam os dois estudos realizados em quanto a terceira e quarta linha fazem a comparação dos resultados para o modelo de efeito fixo e randômico. A primeira coluna mostra informações sobre os estudos (breve descrição). Em seguida, o tamanho do efeito para cada estudo é apresentado pelo gráfico. As três últimas colunas do gráfico mostram a odds ratio, o intervalo de confiança (considerando 95% como referência) e os pesos para o modelo fixo e randômico. Finalmente, o diamante (losango) na última linha do gráfico da floresta revela o tamanho geral do efeito de resumo. O centro do diamante representa

o tamanho do efeito de resumo e sua largura representa os limites, considerando um intervalo de confiança de 95%.

De acordo com os resultados apresentados pelo gráfico de floresta na Figura 4.8, o tamanho do efeito para os estudos é significativo. O valor de $p = 0.65$ e o valor de $I^2 = 0\%$ indica que os estudos não possuem heterogeneidade. Como a medida meta-analítica se encontra totalmente à direita da linha vertical em 1, existe um efeito estatisticamente significativo. Ambos os estudos apresentam um efeito de tratamento que favorece a introdução de *expressões lambda* porque, como podemos observar, os efeitos se encontram à direita da linha vertical. Com isso, é possível aferir que a introdução de *expressões lambda* possui maior preferência do que a utilização de código legado como mostrado na Figura 4.8.

Com base nos resultados obtidos pela meta-análise, é possível inferir que o resultado combinado é significativo e o OR de 17.71 sugere uma vantagem do uso de *expressões lambda* sobre a utilização de códigos legados.

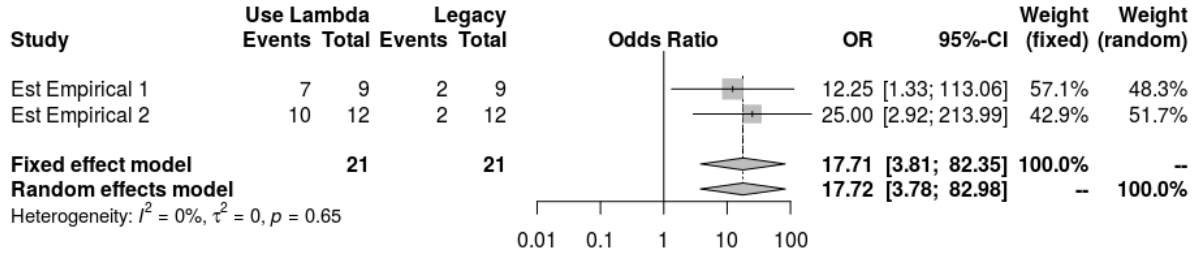

Figura 4.8: Gráfico de floresta apresentando resultados da meta-análise que avalia os efeitos da introdução de expressões lambda em código legado Java.

Capítulo 5

Conclusão

Garantir um código considerado legível facilita as tarefas de manutenção, tanto para adicionar novas funcionalidades quanto para mudanças de código nos softwares desenvolvidos. ao qual necessitam de uma etapa anterior de leitura do código antes que uma transformação possa ser realizada.

O objetivo deste trabalho foi realizar uma investigação empírica para avaliar se a introdução de *expressões lambda* melhora a legibilidade do código. Para alcançar tal objetivo foram identificados e coletados cenários reais de transformações com a adição de *expressões lambda* substituindo trechos de código legado.

O Capítulo 2 apresentou alguns conceitos inerentes ao trabalho como a avaliação de compreensibilidade e cenários de uso de expressões lambda em java que são os dois pontos chave do trabalho. Ainda no capítulo 2 foram apresentadas métricas propostas na literatura para mensurar a legibilidade do código [7, 43].

O Capítulo 3 apresentou os procedimentos seguidos para a realização do estudo empírico. Apresenta como foram conduzidas as duas fases de execução do estudo qualitativo através da aplicação do survey e a forma de captura das percepções dos desenvolvedores.

O Capítulo 4 apresentou as análises realizadas e a discussão a respeito do que foi obtido. Através das respostas do survey, o estudo possibilitou aferir que desenvolvedores e alunos percebem uma melhora na legibilidade do código após a introdução de *expressões lambda*, podendo assim responder a questão de pesquisa **RQ1**.

De acordo com a percepção dos desenvolvedores e alunos, o estudo possibilitou identificar uma melhora na legibilidade em algumas situações como resposta para a questão de pesquisa **RQ2** onde a transformação com a adição de *expressões lambda* ocasionam: remoção de código boilerplaite, remoção de classes anônimas com métodos triviais. Por outro lado, também foram reportados situações onde as transformações não apresentaram melhoras na opinião dos desenvolvedores: a substituição do método *for* por um *forEach* sem

a utilização de funções que facilitam o processamento de listas ou utilização de *forEach* ao invés de *Map* para mapear novos objetos.

Ao avaliar os modelos para mensurar a legibilidade de código, foi possível observar que os resultados das métricas se diferem da opinião dos desenvolvedores em boa parte dos casos. Com esses resultados foi possível identificar que as métricas avaliadas possuem limitações quanto as novas construções da linguagem Java e que podem não ser eficientes para mensurar a legibilidade de código em cenários onde transformações possuem adição de *expressões lambda*.

Para avaliar as transformações e as percepções dos desenvolvedores, várias análises estatísticas foram realizadas e alguns resultados foram estimados. O estudo possibilitou identificar que a introdução de *expressões lambda* podem diminuir a complexidade de programas ao qual transformações envolvendo as novas construções levam a uma redução da quantidade de linhas e complexidade ciclomática. Também foi possível identificar que a percepção dos desenvolvedores se diferem em relação ao seu nível acadêmico, tendo em vista que quanto maior o nível de formação do desenvolvedor, mais rigorosas são suas avaliações quanto o impacto sobre a introdução de *expressões lambda* na legibilidade.

Por fim, a realização de uma meta-análise entre as duas fases de execução do *survey*, possibilitou inferir que o uso de *expressões lambda* possui vantagem sobre a utilização de códigos legados.

5.1 Limitações

Nesta seção, será discutido as ameaças à validade interna e externa, que foram descobertas durante a fase de planejamento e em seguida na fase de análise dos resultados.

5.1.1 Validade Interna

Para a realização dessa pesquisa, foi desenvolvido uma ferramenta para extrair os trechos de código antes e após a sua transformação envolvendo introdução de *expressões lambda* diretamente das páginas de *commits* do repositório GitHub. Contudo as páginas do repositório estão em constante mudança e atualização de *layout* o que dificultou o processo de extração e limitando a ferramenta em algumas situações.

Apesar dos mecanismos implementados na ferramenta e a ausência de evidências de múltiplas participações de um mesmo indivíduo, a ferramenta é incapazes de prevenir toda e qualquer tentativa nesse sentido. Portanto, deve-se considerar a possibilidade de alguns participantes terem opinado mais de uma vez, algo que poderia prejudicar nossa amostra e impactando os resultados da pesquisa.

5.1.2 Validade Externa

Os participantes da pesquisa pertencem a um grupo relativamente restrito de programadores profissionais com experiência em Java. Embora pertençam a um grupo de grande interesse para o mercado de desenvolvimento de software, não foi possível conseguir uma amostra de participantes grande o suficiente para generalização das conclusões obtidas para grupos gerais de programadores.

5.2 Trabalhos Futuros

Como sugestão para trabalhos futuros, existem outros modelos para mesurar legibilidade e que podem ser avaliados para comparação com a percepção de desenvolvedores. Através dos resultados desta pesquisa, acredita-se que é possível implementar novos modelos para mensurar legibilidade e comparar com os modelos propostos na literatura. Também é interessante conduzir novos estudos empíricos utilizando estratégias distintas às propostas nessa dissertação.

Referências

[1] S. Ambler. Java coding standards. *Software Development*, 5(8):67–71, 1997. 14

[2] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000. 9

[3] O. Berwanger, E. A. Suzumura, A. M. Buehler, and J. B. Oliveira. Como avaliar criticamente revisões sistemáticas e metanálises. *Rev Bras Ter Intensiva*, 19(4):475–80, 2007. 49

[4] M. Borenstein, L. Hedges, J. Higgins, and H. Rothstein. Introduction to meta analysis (statistics in practice) 2009. 49

[5] E. O. Brigham and E. O. Brigham. *The fast Fourier transform and its applications*, volume 448. prentice Hall Englewood Cliffs, NJ, 1988. 15

[6] R. P. Buse and W. R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008. 11

[7] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. xi, 5, 7, 11, 12, 13, 15, 23, 24, 33, 34, 43, 51

[8] V. R. B.-G. Caldiera and H. D. Rombach. Goal question metric paradigm. *Encyclopedia of software engineering*, 1:528–532, 1994. 6

[9] J. Corbin, A. Strauss, et al. Basics of qualitative research: Techniques and procedures for developing grounded theory. 2008. 35, 36, 38

[10] J. W. Creswell. Projeto de pesquisa métodos qualitativo, quantitativo e misto. In *Projeto de pesquisa métodos qualitativo, quantitativo e misto*. Penso Editora, 2010. 5

[11] J. W. Creswell and V. L. P. Clark. *Pesquisa de Métodos Mistos-: Série Métodos de Pesquisa*. Penso Editora, 2015. 6

[12] M. Criscuolo. *Qualidade de Produto de Software: uma abordagem baseada no controle da complexidade*. PhD thesis, Universidade de São Paulo, 2008. 10

[13] O. I. de Normalización. *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011. 8

[14] J. Dorn. A general software readability model. *MCS Thesis available from (http://www. cs. virginia. edu/˜ weimer/students/dorn-mcs-paper. pdf)*, 2012. 5, 11, 14, 15

[15] R. M. dos Santos and M. A. Gerosa. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension*, pages 277–285. ACM, 2018. 6, 11, 23, 39

[16] L. H. Etzkorn, S. Gholston, and W. E. Hughes Jr. A semantic entropy metric. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):293–310, 2002. 13

[17] J.-M. Favre. Languages evolve too! changing the software time scale. In *Principles of Software Evolution, Eighth International Workshop on*, pages 33–42. IEEE, 2005. 3

[18] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: a community project on software technologies and software languages. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 58–74. Springer, 2012. x, 16, 17

[19] I. O. for Standardization/International Electrotechnical Commission et al. International standard iso/iec 9126-1: Software engineering–product quality–part 1: Quality model. *ISO/IEC, Tech. Rep*, 2001. 8, 9

[20] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. Lambdaficator: from imperative to functional programming through automated refactoring. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1287–1290. IEEE Press, 2013. 3, 21

[21] L. A. Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961. 30

[22] S. Greenland, J. Pearl, J. M. Robins, et al. Causal diagrams for epidemiologic research. *Epidemiology*, 10:37–48, 1999. 49

[23] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013. 21

[24] M. H. Halstead et al. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, 1977. 13, 25

[25] T. Hothorn and B. S. Everitt. *A handbook of statistical analyses using R*. Chapman and Hall/CRC, 2014. 48

[26] T. Hothorn, K. Hornik, M. van de Wiel, and A. Zeileis. Implementing a class of permutation tests: The coin package. *Journal of Statistical Software, Articles*, 28(8):1–23, 2008. 45, 46

[27] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989. 4

[28] I. ISO. Ieee, systems and software engineering–vocabulary. *IEEE computer society, Piscataway, NJ*, 2010. 8, 9

[29] H.-W. Jung, S.-G. Kim, and C.-S. Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE software*, 21(5):88–92, 2004. 8, 9, 10

[30] S. Konstantopoulos and L. V. Hedges. Analyzing effect sizes: Fixed-effects models. 2009. 49

[31] P. J. Layzell and L. A. Macaulay. An investigation into software maintenance—perception and practices. *Journal of Software Maintenance: research and practice*, 6(3):105–120, 1994. 9

[32] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 10

[33] H. Liu, X. Gong, L. Liao, and B. Li. Evaluate how cyclomatic complexity changes in the context of software evolution. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 756–761, July 2018. 23

[34] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in java. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017. 4, 21, 26, 42

[35] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976. 24

[36] R. Medeiros, A. J. Carvalho, D. Marcílio, L. Fantin, U. Silva, W. Lucas, and R. Bonifácio. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In •, 2018. 3, 4, 16, 21, 22, 26, 30, 42

[37] P. R. M. Meirelles. *Monitoramento de métricas de código-fonte em projetos de software livre*. PhD thesis, Universidade de São Paulo, 2013. 9

[38] P. MORETTIN and W. BUSSAB. Estatística básica, editora saraiva. *São Paulo*, 2002. 6

[39] L. Myers and M. J. Sirois. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences*, 12, 2004. 40, 41

[40] J. L. Overbey and R. E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *ACM SIGPLAN Notices*, volume 44, pages 493–502. ACM, 2009. 3

[41] T. M. Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996. 10

[42] J. Platform. Java platform, standard edition 8 api specification. https://docs.oracle.com/javase/8/docs/api/index.html. [Online; acessado 22-Novembro-2018]. 18, 19

[43] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82. ACM, 2011. 5, 7, 10, 11, 13, 14, 15, 23, 25, 33, 34, 43, 51

[44] R. S. Pressman. *Engenharia de software*, volume 6. Makron books São Paulo, 1995. 10

[45] R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005. 8

[46] V. Rajlich. Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering*, pages 133–144. ACM, 2014. 3

[47] H. Rajput and L. K. Singh. Improvement of software quality attributes in object oriented analysis and design phase using goal-question-metric paradigm. *JSEA*, 4(6):345–349, 2011. 23

[48] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016. 5, 11

[49] G. Schwarzer et al. meta: An r package for meta-analysis. *R news*, 7(3):40–45, 2007. 49

[50] R. Singh. International standard iso/iec 12207 software life cycle processes. *Software Process: Improvement and Practice*, 2(1):35–50, 1996. 9

[51] I. Sommerville et al. *Software engineering*. Addison-wesley, 2007. 10

[52] A. L. G. Tavares and F. C. Caldas. Caracterizando a adoção de expressões lambda em código java legado. *Universidade de Brasilía*, 2017. 4, 17, 18

[53] T. P. Torreão. Mineração de questões sobre o uso de expressões lambda em java 8. *Universidade de Brasilía*, 2018. 4, 21

[54] N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 60–70. IEEE Press, 2017. 21

[55] R.-G. Urma, M. Fusco, and A. Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning Publications Co., 2014. 4, 16, 17, 18

[56] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. 6

Apêndice A

Transformações avaliadas

A.1 1027

```
static List transform(final Resource parent, final List list) {
 return Lists.transform(list, new Function() {
     @Override
     public ResourceMethod apply(Data data) {
         return (data == null) ? null : new ResourceMethod(parent, data)
            ;
     }
 });
}
```

Código A.1: Código 1027 utilizando solução trandicional

```
static List transform (final Resource parent, final List list) {
return list.stream()
           .map(data1 -> (data1 == null) ? null : new ResourceMethod(
              parent, data1))
           .collect(Collectors.toList());
}
```

Código A.2: Código 1027 com adição de expressão lambda

A.2 1035

```
_subscriptions.add(//
        Observable.interval(INITIAL_DELAY, POLLING_INTERVAL, TimeUnit.
            MILLISECONDS)
                    ...
            .doOnSubscribe(new Action0() {
                @Override
                public void call() {
                    ...
                }
            })
            .subscribe(new Action1() {
                @Override
                public void call(String taskName) {
                    ...
                }
               })
  );
```

Código A.3: Código 1035 utilizando solução tradicional

```
_subscriptions.add(//
        Observable.interval(INITIAL_DELAY, POLLING_INTERVAL, TimeUnit.
            MILLISECONDS)
          .map(this::_doNetworkCallAndGetStringResult)//
          .take(pollCount)
          .doOnSubscribe(() ->
                _log(String.format("Start simple polling - %s", _counter
                    )))
          .subscribe(taskName -> {
              _log(String.format(Locale.US, "Executing polled task [%s]
                  now time : [xx:%02d]",
                      taskName, _getSecondHand()));
              })
  );
```

Código A.4: Código 1035 com a adição de expressão lambda

A.3 1052

```
private static Collection wrap(final Map sources)
{
      final Collection collection = new ArrayList<>(sources.size());
      for (final Map.Entry entry : sources.entrySet())
      {
          collection.add(new CharSequenceJavaFileObject(entry.getKey(),
              entry.getValue()));
      }
      return collection;
}
```

Código A.5: Código 1052 utilizando solução tradicional

```
private static Collection wrap(final Map sources)
{
      return sources.entrySet()
          .stream()
          .map((e) -> new CharSequenceJavaFileObject(e.getKey(), e.
              getValue())).collect(toList());
}
```

Código A.6: Código 1052 com a adição de expressão lambda

A.4 1062

```
private Optional getUnsafeView(DataQuery path) {
      Optional val = get(path);
      if (val.isPresent()) {
          if (val.get() instanceof DataView) {
              return Optional.of((DataView) val.get());
          }
      }
      return Optional.empty();
}
```

Código A.7: Código 1062 utilizando solução tradicional

```
private Optional getUnsafeView(DataQuery path) {
      return get(path).filter(obj -> obj instanceof DataView).map(obj ->
          (DataView) obj);
}
```

Código A.8: Código 1062 com adição de expressões lambda

A.5 1166

```
for (CRFThread thread : threads) {
    thread.start();
  }
```

Código A.9: Código 1166 utilizando solução tradicional

```
threads.forEach(
          thread -> {
            thread.start();
          });
```

Código A.10: Código 1166 com adição de expressão lambda

A.6 1180

```
private String expectedAnswer2 =
  "<doc> <el arg=\"funny&amp;&apos;&gt;&quot;stuff\"> yo! C&amp;C!yo! C&
     amp;C! </el> </doc>";


   private Function duplicate = new Function() {
     public String apply(String in) {
       return in + in;
     }
   };
```

Código A.11: Código 1180 utilizando solução tradicional

```
private String expectedAnswer2 =
  "<doc> <el arg=\"funny&amp;&apos;&gt;&quot;stuff\"> yo! C&amp;C!yo! C&
     amp;C! </el> </doc>";


  private Function duplicate = (String in) -> { return in + in;};
```

Código A.12: Código 1180 com a adição de expressão lambda

A.7 1182

```
  assertEquals(numRequests, responses.size());
    for (TestResponse testResponse : responses) {
        Response response = testResponse.getResponse();
        assertEquals(testResponse.method, response.getRequestLine().
           getMethod());
        assertEquals(testResponse.statusCode, response.getStatusLine().
           getStatusCode());
        assertEquals((pathPrefix.length() > 0 ? pathPrefix : "") + "/" +
            testResponse.statusCode,
                response.getRequestLine().getUri());
    }
}
```

Código A.13: Código 1182 utilizando solução tradicional

```
  assertEquals(numRequests, responses.size());
    responses.forEach(
            testResponse -> {
                Response response = testResponse.getResponse();
                assertEquals(testResponse.method, response.
                   getRequestLine().getMethod());
                assertEquals(testResponse.statusCode, response.
                   getStatusLine().getStatusCode());
                assertEquals(
                        (pathPrefix.length() -> 0 ? pathPrefix : "")
                             "/"
                             testResponse.statusCode,
                        response.getRequestLine().getUri());
```

Código A.14: Código 1182 com a adição de expressões lambda

A.8 1183

```
private List buildSimpleSnapshotInfos(final Set toResolve,
                                      final RepositoryData
                                         repositoryData,
                                      final List currentSnapshots)
                                          {
    List snapshotInfos = new ArrayList<>();
          for (SnapshotInfo snapshotInfo : currentSnapshots) {
              if (toResolve.remove(snapshotInfo.snapshotId())) {
                  snapshotInfos.add(snapshotInfo.basic());
              }
          }
```

Código A.15: Código 1183 utilizando solução tradicional

```
private List buildSimpleSnapshotInfos(
                          final Set toResolve,
                          final RepositoryData repositoryData,
                          final List currentSnapshots) {
                List snapshotInfos = new ArrayList<>();
                      currentSnapshots
                              .stream()
                              .filter(snapshotInfo -> toResolve.remove(
                                 snapshotInfo.snapshotId()))
                              .forEach(
                                      snapshotInfo -> {
                                          snapshotInfos.add(snapshotInfo
                                              .basic());
                                      });
```

Código A.16: Código 1183 com a adição de expressões lambda

A.9 1192

```
public static SeleniumQueryObject closest(SeleniumQueryObject caller,
    String selector) {
  List elements = caller.get();
  WebDriver driver = caller.getWebDriver();

  List closests = new ArrayList<>(elements.size());
  for (WebElement element : elements) {
    WebElement closestElement = closest(driver, element, selector);
    if (closestElement != null) {
      closests.add(closestElement);
    }
  }

  return SqObjectFactory.instance().createWithInvalidSelector(caller.
      getWebDriver(), closests, caller);
}
```

Código A.17: Código 1192 utilizando solução tradicional

```
public static SeleniumQueryObject closest(SeleniumQueryObject caller,
    String selector) {
  List elements = caller.get();
  WebDriver driver = caller.getWebDriver();

  List closests = new ArrayList<>(elements.size());
  elements.stream().map(closestElement -> closest(driver, element,
      selector))
      .filter(closestElement -> closestElement != null).forEach(
          closestElement -> {
        closests.add(closestElement);
      });

  return SqObjectFactory.instance().createWithInvalidSelector(caller.
      getWebDriver(), closests, caller);
}
```

Código A.18: Código 1192 com a adição de expressões lambda

A.10 1022

```
public ListField(String name, String humanName, List<String>
    defaultValue, Map<String, String> values, String description,
    Optional isOptional, Attribute... attributes) {
     super(FIELD_TYPE, name, humanName, description, isOptional);
     this.defaultValue = defaultValue;
     this.values = values;

     this.attributes = new ArrayList<>();
     if (attributes != null) {
         for (Attribute attribute : attributes) {
             this.attributes.add(attribute.toString().toLowerCase(
                 Locale.ENGLISH));
         }
     }
}
```

Código A.19: Código 1022 utilizando solução tradicional

```
public ListField(String name, String humanName, List<String>
    defaultValue, Map<String, String> values, String description,
    Optional isOptional, Attribute... attributes) {
     super(FIELD_TYPE, name, humanName, description, isOptional);
     this.defaultValue = defaultValue;
     this.values = values;
     this.attributes = Arrays.stream(attributes)
             .map(attribute -> attribute.toString().toLowerCase(
                 Locale.ENGLISH))
             .collect(Collectors.toList());
}
```

Código A.20: Código 1022 com a adição de expressões lambda

A.11 1026

```
// Calculations. (depend on recommended destinations)
final List<Calculation> calculations = new ArrayList<>(
   recommended.size());
for (final Destination dest : recommended) {
    calculations.add(calculation.resolveTemplate("from", "Moon")
       .resolveTemplate("to", dest.getDestination())
            .request().get(Calculation.class));
}
```

Código A.21: Código 1026 utilizando solução tradicional

```
// Calculations. (depend on recommended destinations)
final Map<String, Calculation> calculations = new HashMap<>();
recommended.stream().forEach(destination -> {
    try {
        calculations.put(destination.getDestination(),
           calculation.resolveTemplate("from", "Moon")
                .resolveTemplate("to", destination.
                   getDestination())
                .request().get(Calculation.class));
    } catch (final Throwable throwable) {
        errors.offer("Calculation: " + throwable.getMessage());
    }
});
```

Código A.22: Código 1026 com a adição de expressões lambda

A.12 1042

```
public static SingletonInverseVariableSupply
    mockSingletonInverseVariableSupply(
         final TestdataChainedEntity[] allEntities) {
     return new SingletonInverseVariableSupply() {
         @Override
         public Object getInverseSingleton(Object planningValue) {
             for (TestdataChainedEntity entity : allEntities) {
                 if (entity.getChainedObject().equals(planningValue))
                     {
                     return entity;
                 }
             }
             return null;
         }
     };
}
```

Código A.23: Código 1042 utilizando solução tradicional

```
public static SingletonInverseVariableSupply
    mockSingletonInverseVariableSupply(
         final TestdataChainedEntity[] allEntities) {
     return planningValue -> {
         for (TestdataChainedEntity entity : allEntities) {
             if (entity.getChainedObject().equals(planningValue)) {
                 return entity;
             }
         }
         return null;
     };
}
```

Código A.24: Código 1042 com a adição de expressões lambda

A.13 1178

```
public class GoldenSectionLineSearchTest extends TestCase {
    public void testEasy() {
      GoldenSectionLineSearch min = new GoldenSectionLineSearch(false,
          0.00001, 0.0, 1.0, false);
            Function f2 = new Function() {
              public Double apply(Double x) {
                // this function used to fail in Galen's version; min
                   should be 0.2
                // return - x * (2 * x - 1) * (x - 0.8);
                // this function fails if you don't find an initial
                   bracketing
                return x < 0.1 ? 0.0: (x > 0.2 ? 0.0: (x - 0.1) * (x -
                    0.2));
                // return - Math.sin(x * Math.PI);
                // return -(3 + 6 * x - 4 * x * x);
              }
            };
      assertEquals(0.15,min.minimize(f2),1E-4);
  }
}
```

Código A.25: Código 1178 utilizando solução tradicional

```
public class GoldenSectionLineSearchTest extends TestCase {
    public void testEasy() {
      GoldenSectionLineSearch min = new GoldenSectionLineSearch(false,
          0.00001, 0.0, 1.0, false);
            Function f2 = (Double x)->{ return x < 0.1 ? 0.0: (x > 0.2
                ? 0.0: (x - 0.1) * (x - 0.2));};
      assertEquals(0.15,min.minimize(f2),1E-4);
    }
}
```

Código A.26: Código 1178 com a adição de expressões lambda

A.14 1060

```
    @Override
    public DataContainer copy() {
        final DataContainer container = new MemoryDataContainer();
        for (DataQuery query : getKeys(false)) {
            container.set(query, get(query).get());
        }
        return container;
    }
```

Código A.27: Código 1060 utilizando solução tradicional

```
  @Override
    public DataContainer copy() {
        final DataContainer container = new MemoryDataContainer();
        getKeys(false).stream()
                .forEach(query ->
                        get(query).ifPresent(obj ->
                                container.set(query, obj)
                        )
        );
        return container;
    }
```

Código A.28: Código 1060 com a adição de expressões lambda

A.15 1061

```
@Override
public Optional<DataView> getView(DataQuery path) {
    Optional<Object> val = get(path);
    if (val.isPresent()) {
        if (val.get() instanceof DataView) {
            return Optional.of((DataView) val.get());
        }
    }
    return Optional.empty();
}
```

Código A.29: Código 1061 utilizando solução tradicional

```
@Override
public Optional<DataView> getView(DataQuery path) {
    return get(path).filter(obj -> obj instanceof DataView).map(obj
       -> (DataView) obj);
}
```

Código A.30: Código 1061 com a adição de expressões lambda

A.16 1070

```
// Start the front end server using the Jax-RS controller
vertx.createHttpServer()
    .requestHandler(new VertxRequestHandler(vertx, deployment))
    .listen(8080);
System.out.println("started");
```

Código A.31: Código 1070 utilizando solução tradicional

```
// Start the front end server using the Jax-RS controller
vertx.createHttpServer()
    .requestHandler(new VertxRequestHandler(vertx, deployment))
    .listen(8080, ar -> {
      System.out.println("Server started on port "+ ar.result().
          actualPort());
    });
```

Código A.32: Código 1070 com a adição de expressões lambda

A.17 1185

```
public List findWebElements(SearchContext context) {
    Set elements = new LinkedHashSet<>();
       for (TagComponent tagComponent : tagComponents) {
           List elementsFound = tagComponent.findWebElements(context
              );
           elements.addAll(elementsFound);
       }
    return new ArrayList<>(elements);
}
```

Código A.33: Código 1185 utilizando solução tradicional

```
public List findWebElements(SearchContext context) {
    Set elements = new LinkedHashSet<>();
       tagComponents.stream().map(elementsFound -> tagComponent.
          findWebElements(context)).forEach(elementsFound -> {
                      elements.addAll(elementsFound);
                  });
    return new ArrayList<>(elements);
}
```

Código A.34: Código 1185 com a adição de expressões lambda

A.18 1188

```
    private String toChainedNotSelectors(WebDriver webDriver,
        SqCssFunctionalPseudoClassArgument functionalPseudoClassArgument)
         {
         CssSelectorList parsedNotPseudoClassArgument = ParseTreeBuilder.
            parse(functionalPseudoClassArgument.getArgumentAsString());
         StringBuilder chainedNotSelectors = new StringBuilder();
                 for (CssSelector cssSelector :
                     parsedNotPseudoClassArgument) {
             chainedNotSelectors.append(":").append(PSEUDO_PURE_NOT).
                append("(").append(cssSelector.toElementFinder(webDriver)
                .toCssString()).append(")");
                 }
         return chainedNotSelectors.toString();
    }
```

Código A.35: Código 1188 utilizando solução tradicional

```
    private String toChainedNotSelectors(WebDriver webDriver,
        SqCssFunctionalPseudoClassArgument functionalPseudoClassArgument)
         {
         CssSelectorList parsedNotPseudoClassArgument = ParseTreeBuilder.
            parse(functionalPseudoClassArgument.getArgumentAsString());
         StringBuilder chainedNotSelectors = new StringBuilder();
                 parsedNotPseudoClassArgument.forEach(cssSelector -> {
             chainedNotSelectors.append(":").append(PSEUDO_PURE_NOT).
                append("(").append(cssSelector.toElementFinder(webDriver)
                .toCssString()).append(")");
                 });
         return chainedNotSelectors.toString();
    }
```

Código A.36: Código 1188 com a adição de expressões lambda