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

# Caracterizando a Adoção de Expressões Lambda em Código Java Legado

Aline Laís Gomes Tavares
Filipe Cardoso Caldas

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2017

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

# Caracterizando a Adoção de Expressões Lambda em Código Java Legado

Aline Laís Gomes Tavares
Filipe Cardoso Caldas

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof.ª Dr.ª Edna Dias Canedo
FGA/UnB

Prof. Dr. Vinicius Ruela Pereira Borges
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 8 de Agosto de 2017

# Dedicatória

Dedicamos este trabalho às nossas famílias que sempre nós apoiaram, a todos os nossos amigos, colegas e professores da universidade que nos ajudaram durante a nossa graduação e colaboraram de alguma forma para nossa formatura. Dedicamos também ao nosso orientador, professor doutor Rodrigo Bonifácio, que nos ajudou durante o desenvolvimento deste trabalho.

# Agradecimentos

Agradecemos aos nossos familiares, que sempre nos apoiam incondicionalmente durante toda a nossa vida e especialmente durante o nosso período na Universidade, diante de todos os desafios enfrentados, acreditando sempre no nosso sucesso.

Ao nosso orientador, professor doutor Rodrigo de Almeida Bonifácio, por trabalhar conosco no desafio de desenvolver este projeto.

E por fim, a todos os nossos amigos que ingressaram no curso de ciência de computação na UnB conosco no segundo semestre de 2010.

# Resumo

Este trabalho apresenta um estudo sobre como as expressões lambda, introduzidas na linguagem Java desde 2014, estão sendo utilizadas em projetos Java *open-source*. Através de uma amostragem de 99 projetos populares existentes na plataforma *Github*, observou-se como e quando esta característica foi introduzida nestes projetos, identificando padrões adotados e métodos de refatoração de código, caso estes existam. Foi realizado um estudo geral para identificar padrões gerais, além de estudos de caso com o objetivo de obter uma análise mais aprofundada sobre o assunto. Foram escolhidos cinco projetos previamente classificados de acordo com a abordagem distinta de introdução de expressões lambda para realizar os estudos de caso.

**Palavras-chave:** Mineração em Repositórios de Software, git, Java, Expressões Lambda, Anonymous Inner Class

# Abstract

This work presents a study about how lambda expressions are being used in open-soruce Java projects. Using a sample of 99 popular projects' repositories in Github, it was analyzed how and when this feature was introduced within those projects, identifying patterns used to implement such feature and possible code refactoring approaches, in case any code was refactored. It was conducted a general study to identify similarities among projects and the way they use lambda expressions, and also case studies that intended to realize a richer analysis about the subject. For the case studies, it was chosen five projects previously classified according to the distinct approach used when introducing lambda expressions.

**Keywords:** Mining Software Repositories, git, Java, Lambda Expressions, Anonymous Inner Class

# Sumário


**1 Introdução** **1**

1.1 Contexto . . . . . 1

1.2 Objetivos . . . . . 2

1.2.1 Objetivos Gerais . . . . . 2

1.2.2 Objetivos Específicos . . . . . 2

1.3 Metodologia . . . . . 3

1.4 Organização do Trabalho . . . . . 3

**2 Evolução e Mineração de Repositórios de Software** **5**

2.1 Evolução de Software . . . . . 6

2.2 MSR: Mineração de dados e a Engenharia de Software . . . . . 8

2.2.1 Classificação . . . . . 8

2.3 Trabalhos Relacionados . . . . . 11

**3 Java SE 8 e JDK 8** **13**

3.1 Histórico . . . . . 13

3.2 Princípios . . . . . 14

3.3 Evolução . . . . . 15

3.4 Expressões Lambda . . . . . 18

3.4.1 Sintaxe . . . . . 18

3.4.2 Principais formas de uso . . . . . 19

**4 Estudo e o Processo de Mineração** **22**

4.1 Questões de Pesquisa . . . . . 23

4.2 Fonte de Dados e Objetos de Estudo . . . . . 24

4.3 Processo de Mineração . . . . . 25

4.3.1 *static-analysis* . . . . . 26

4.3.2 *BDAnalisador* . . . . . 27

4.4 Classificação e Análise . . . . . 29

4.4.1 Média de Lambda . . . . . 30

4.4.2 Análise Temporal . . . . . . . . . . 30
4.4.3 Refatoração de Código . . . . . . . . . . 32
4.4.4 Casos de Uso . . . . . . . . . . 32

**5 Resultados Obtidos e Análise** **33**
5.1 Visão Geral . . . . . . . . . . 33
5.1.1 Projetos Selecionados . . . . . . . . . . 34
5.1.2 *agera* . . . . . . . . . . 34
5.1.3 *vert.x* . . . . . . . . . . 35
5.1.4 *Android-CleanArchitecture* . . . . . . . . . . 36
5.1.5 *RxJava* . . . . . . . . . . 37
5.1.6 *guava* . . . . . . . . . . 38
5.2 Refatoração de Código . . . . . . . . . . 39
5.3 Estudos de Caso . . . . . . . . . . 41

**6 Considerações Finais** **46**

**Referências** **48**

**Anexo** **51**

**I Projetos utilizados** **52**

# Lista de Figuras

2.1 Taxonomia em camadas adaptada do trabalho de Kagdi et al [1] . . . . . . 9

4.1 Processo utilizado para realização do estudo. . . . . . . . . . . . . . . . . . . 22
4.2 Metodologia implementada para coleta e tratamento de dados. . . . . . . . 26
4.3 Diagrama de classes UML do *BDAnalisador* . . . . . . . . . . . . . . . . . 28
4.4 Modelo relacional do banco de dados populado pelo *BDAnalisador*. Imagem gerada com software *DBeaver* [2] . . . . . . . . . . . . . . . . . . . . . . . . 29
4.5 Classificação dos projetos. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.1 Gráficos dos projetos separados por ano de introdução de expressões lambda e média de lambda por *snapshot*. . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Gráficos de análise temporal do projeto *agera* . . . . . . . . . . . . . . . . 35
5.3 Gráficos de análise temporal do projeto *Vert.x* . . . . . . . . . . . . . . . . 36
5.4 Gráficos de análise temporal do projeto *Android-CleanArchitecture* . . . . . 37
5.5 Gráficos de análise temporal do projeto *RxJava* . . . . . . . . . . . . . . . 38
5.6 Gráficos de análise temporal do projeto *Guava* . . . . . . . . . . . . . . . . 39
5.7 Comparação entre todos os projetos com expressão lambda sobre refatoração de código no mês de introdução da característica. . . . . . . . . . . . . 40
5.8 *Commit* de refatoração de código do projeto Vert.x [3] . . . . . . . . . . . 42
5.9 Exemplo de um arquivo do *commit* de adição de código novo em Java 8 do projeto *Guava* [4] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.10 *Commit* de adição da biblioteca *Retrolambda* no projeto *Agera* [5] . . . . . 44
5.11 *Commit* de remoção de Lambdas e volta para o uso de AICs no projeto *RXJava* [6] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

# Lista de Tabelas

3.1 As versões anteriores de java e principais características . . . . . . . . . . . . 14

4.1 Informações mensais sobre o projeto *agera*. . . . . . . . . . . . . . . . . . . . . 30

# Capítulo 1

# Introdução

## 1.1 Contexto

A evolução e manutenção de software é uma característica intrínseca do desenvolvimento de sistemas, considerando que, a partir do momento em que um software começa a ser desenvolvido, são realizadas constantes manutenções [7]. Apesar de o conceito de evolução de software representar uma grande área de pesquisa na engenharia de software, nem sempre pesquisadores reconhecem a importância da evolução das linguagens de programação como um grande fator influenciador na evolução de software [8], uma vez que um software que não evolui junto com a sua linguagem, tem a sua capacidade de evolução limitada [9].

Surge então a necessidade de alinhar estudos empíricos com o intuito de investigar este processo de evolução do software juntamente com a evolução de sua linguagem, através de investigação e análise de seu código fonte que pode ser feita por meio de análise de repositórios de software em grande escala, o que deu origem a abordagem de Mineração em Repositório de Software (MSR). O MSR tem como base o estudo sistemático e empírico de repositórios de software com o objetivo de analisá-los utilizando informações estatisticamente úteis coletadas para observar diversos padrões, como a evolução do software, padrões de projetos, dentre outros [10].

A complexidade e dificuldade de realizar tal estudo empírico é alta, pois atualmente existem diversos tipos de plataformas para coleta de dados, linguagens e focos variados. Assim, para escolher o foco do objeto de estudo, é necessário levar em consideração estas dificuldades e especificações e, dessa forma, este trabalho tem como foco a análise de sistemas de software na linguagem Java.

Até a data deste trabalho, de acordo com os dados do TIOBE [11], Java é a linguagem de programação com mais linhas de codigo catalogadas no mundo. Isso faz com que seja extremamente interessante o estudo empírico da evolução da linguagem e software

desenvolvidos nela, desde sua primeira versão antes de se tornar uma linguagem popular e ainda com poucos pacotes, até a sua atual versão estável, com uma grande quantidade de novas características.

A versão mais recente do Java, a JDK 8, possui diversas adições interessantes para a organização do código e facilidade do programador. Dessas novas implementações, conceitos de linguagem funcional como expressões lambda e iteradores externos que recebem uma expressão lambda como argumento, como o `map()`, `filter()` e `forEach()` tornaram a escrita de código paralelo mais simples e permitiram ganhos de performance [12].

Diversas linguagens de programação também populares como Javascript e Ruby já possuíam expressões lambda desde suas versões iniciais, sendo o Java 8, em 2014, mais uma linguagem a introduzir esse novo conceito. Podemos descrever as expressões como pequenas funções, normalmente feitas em apenas uma linha, sem um nome identificador. Elas podem ser retornadas de outras funções ou passadas como parâmetro. Um exemplo de expressão lambda pode ser visualizado pela função de soma de dois inteiros escrita da forma: $(int\ x, int\ y) \rightarrow x + y$ [13].

Antes da inclusão de expressões lambda em Java, o mesmo comportamento era possível utilizando classes anônimas, ou AICs (*Anonymous inner classes*). Contudo, é necessário a criação de uma interface com apenas um método e instanciá-la como uma AIC. A utilização de AICs gera inúmeras linhas de código adicionais que poderiam ser encapsuladas em apenas uma linha com a utilização de expressões lambda. Como é interessante para o programador ter um código sempre mais legível e sucinto, torna-se interessante a refatoração de código que utiliza uma tecnologia mais antiga para a nova.

Assim, este trabalho visa conduzir um estudo de mineração em repositórios de software *open-source* desenvolvidos na linguagem Java para entender como desenvolvedores estão se adaptando às novas mudanças introduzidas no Java 8.

## 1.2 Objetivos

### 1.2.1 Objetivos Gerais

Este trabalho tem como objetivo geral caracterizar o uso de expressões lambda entre os projetos Java mais populares desenvolvidos pela comunidade do *GitHub*.

### 1.2.2 Objetivos Específicos

Sabendo do benefício da refatoração de código para o uso de expressões lambda e da pouca quantidade de estudos acadêmicos sobre o assunto, surge então uma oportunidade de pesquisa para melhor entender a forma como essa característica está sendo adotada por

projetos com foco em refatoração de código, o que originou a necessidade deste estudo. Assim, para alcançar o objetivo deste trabalho, foram traçados, em forma de questões de pesquisa, os objetivos específicos a serem alcançados por este trabalho:

- Quando os projetos Java de código aberto começaram a introduzir o uso de expressões lambda em seus códigos e quão significativo é esse uso?
- Como as equipes de desenvolvimento estão utilizando expressões lambda: através de refatoração de código ou introduzindo apenas em código novo?
- Quais são os padrões tipicamente adotados para o uso de expressões lambda?

## 1.3 Metodologia

Este trabalho foi desenvolvido utilizando uma abordagem de MSR que, dentro da classificação de Kagdi et al. [1], pode ser divido em quatro grandes etapas. Primeiramente, definiu-se a fonte de dados a ser utilizada e dentre as possibilidades encontradas na literatura [14], optou-se por utilizar repositórios de controle de versão, mais especificamente o *Github*, tendo como critério de seleção os 99 repositórios Java mais populares na plataforma. Em seguida, definiu-se o propósito do estudo, no caso, os objetivos em forma de perguntas de pesquisa a serem respondidas.

A terceira grande etapa foi a definição do método em si, a forma como seria realizado o estudo e este, por sua vez, constitui-se de: *download* de todos os 99 repositórios mais populares, realização de uma análise estática para coletar informações necessárias de cada projeto por versão e armazenamento de informações pertinentes em uma base de dados.

A etapa final foi a realização da avaliação dos dados obtidos, através de métricas e análises quantitativas para melhor responder as questões de pesquisa, além de estudos de caso feitos com cinco projetos escolhos de acordo com a maneira distinta em que estes fazem o uso de expressões lambda.

## 1.4 Organização do Trabalho

Este trabalho se divide em seis capítulos:

- **Capítulo 2** Apresenta um referencial teórico sobre evolução de software demonstrando as estratégias utilizadas e a importância da evolução em um projeto de código aberto, além de enunciar as leis de evolução de software de Lehman. O capítulo também apresenta o processo de mineração de dados explicando cada uma das camadas da classificação e demonstra trabalhos relacionados a esse tópico que serviram de base para este trabalho.

- **Capítulo 3** Traz um contexto histórico e técnico da linguagem Java necessários para entender a importância que as novas características adicionadas na versão 8 fornecem para os desenvolvedores de projetos determinados a manter o projeto sempre evoluído. Também é apresentada uma seção para demonstrar os detalhes e formas de utilização das expressões Lambda, que são o foco dos objetivos deste trabalho.
- **Capítulo 4** Descreve a metodologia e processo utilizados para o desenvolvimento e obtenção dos resultados deste trabalho. O capítulo desenvolve o contexto das questões de pesquisa e explica os motivos e processos de desenvolvimento das ferramentas utilizadas para minerar os dados, além de determinar a classificação dos resultados obtidos em grupos para facilitar a organização dos dados de forma que as perguntas sejam respondidas claramente.
- **Capítulo 5** Apresenta os resultados obtidos e de que forma eles ajudaram a responder as questões de pesquisa. É descrita uma análise de um projeto de cada categoria especificada no capítulo anterior a fim de contextualizar cada tipo de resultado para que o leitor entenda a forma que os projetos Java estão lidando com as expressões Lambda.
- **Capítulo 6** Descreve as dificuldades e problemas obtidos durante o trabalho e de que forma eles influenciaram nos resultados. Também é apresentada uma seção contendo as ideias surgidas durante o desenvolvimento deste trabalho que ficaram para ser feitas no futuro devido à gama de informações que a mineração dos dados trouxe.

# Capítulo 2

# Evolução e Mineração de Repositórios de Software

As linguagens de programação estão sempre evoluindo, aumentando sua complexidade e expressividade. Da mesma forma, além de se preocupar com dificuldades de estabelecer compatibilidade do software em relação a tecnologias antigas, desenvolvedores também precisam acompanhar essa evolução das linguagens, para que suas ferramentes não se tornem obsoletas [9]. Isso agrega um grande desafio no desenvolvimento e evolução de software.

O conceito de evolução de software amadureceu e se tornou um grande foco de pesquisa. Porém, nem sempre estudos de evolução de software reconhecem a importância da evolução da linguagem de programação no contexto do desenvolvimento de software [8]. Quando um código não evolui junto com a linguagem, com o objetivo de manter uma compatibilidade com versões anteriores, sua capacidade de evolução se torna limitada [9].

A refatoração de código, que consiste em realizar mudanças no código fonte sem modificar o comportamento observado do software [15], permite que a evolução de linguagens e ferramentas aconteçam mais naturalmente. No entanto, novos desafios surgem, como, por exemplo, saber quando e como realizar a refatoração de código de forma eficiente e coesa. Dentre as técnicas de pesquisa em engenharia de software para entender esse problema, torna-se interessante explorar as técnicas de mineração de dados aplicados a repositórios de software, com o objetivo de entender o relacionamento entre evolução e refatoração.

Assim, este capítulo busca esclarecer conceitos e estratégias de evolução de software, mineração de repositórios de software e o estado da arte sobre o assunto.

## 2.1 Evolução de Software

Durante o processo de produção e manutenção de software, há a necessidade de diversas tomadas de decisões sobre seu desenvolvimento, como, por exemplo, decidir como melhorar a qualidade do software. Tais decisões precisam ser tomadas levando em consideração o ambiente em que se está inserido, com novas tecnologias surgindo todos os dias, *feedback* de usuários exigentes e concorrentes produzindo um software similar com ideias inovadoras.

Tudo isso acontece em um ritmo de constante mudança e, assim, como diria Lehman, é imprescindível que software de sistemas sejam capazes de evoluir ou os mesmos correm o risco de sofrer uma morte prematura [16], fazendo, assim, da evolução de software um conceito importante dentro do seu processo de desenvolvimento. Essa evolução deve ocorrer considerando o ambiente em que o software está inserido, o qual possui aspectos técnicos e sociais: sua infraestrutura que pertence ao seu ambiente técnico, o qual sempre necessita manutenção, e ao mesmo tempo, usuários que utilizam o software e estão constantemente formando opiniões, oferecendo *feedback* e na expectativa por novidades [7]. Assim, a medida que novas características são desenvolvidas, novas tecnologias de infraestrutura são necessárias, novos requisitos são adicionados e o sistema de software deve se adaptar a essas mudanças.

Com relação aos aspectos sociais que envolvem a evolução do software, é importante ressaltar os conceitos básicos de como essa evolução acontece. Durante um estudo empírico pioneiro na área de evolução de software para sistemas de grande escala, Lehman e seus colaboradores formularam um conjunto de observações que hoje são conhecidas como as leis da evolução de software, ou simplesmente Leis de Lehman [16]. No geral, as leis de evolução de software podem ser descritas da seguinte forma [16]:

1. **Mudança contínua**: um sistema se tornará progressivamente menos satisfatório para seus usuários com o tempo, a menos que se adapte continuamente às necessidades que surgirão.

2. **Complexidade Crescente**: um sistema se tornará cada vez mais complexo, a menos que se trabalhe para explicitamente reduzir sua complexidade.

3. **Auto-regulação**: o processo de evolução de software é auto-regulatório no que diz respeito às distribuições dos artefatos e produtos produzidos.

4. **Conservação da estabilidade organizacional**: a taxa média de atividade global efetiva em um sistema em evolução não muda com o tempo, ou seja, a média de trabalho e esforço colocado em cada entrega do sistema permanece a mesma.

5. **Conservação de familiaridade**: a quantidade de conteúdo novo em cada entrega do sistema tende a se manter constante com o tempo.

6. **Crescimento contínuo**: A quantidade de funcionalidades em um sistema crescerá com o tempo, com o objetivo de satisfazer seus usuários.

7. **Declínio de qualidade**: Um sistema será percebido com qualidade inferior com o passar do tempo, a menos que seu design tenha uma manutenção minuciosa e se adapte a novas restrições.

8. **Sistema de *Feedback***: Evoluir um sistema com sucesso requer reconhecer que o processo de desenvolvimento acontece em um sistema de *feedback* de vários ciclos, agentes e níveis.

Os aspectos técnicos que envolvem a evolução de software, levam em consideração as tecnologias utilizadas para o seu desenvolvimento, desde a linguagem de programação às máquinas onde o software será hospedado, que sofrem atualizações, recebem novas versões com novas características implementadas.

Um estudo realizado por Lavre mostra como a evolução das linguagens de programação, no contexto da evolução de software, muitas vezes são subestimadas, uma vez que muitos podem considerar em seus estudos a linguagem de computação como um artefato imutável [8]. Por exemplo, coloca-se muita ênfase e cuidado para realizar o versionamento de código fonte de um software, mas este pode ser tornar inutilizável futuramente se ninguém souber qual versão de compilador determinada versão de código foi desenvolvida.

Para não se deixar estagnar, linguagens de programação precisam se adaptar e evoluir. Quando uma linguagem de programação atualiza, softwares que eram antes considerados atualizados, podem se tornar rapidamente obsoletos. E, ao mesmo tempo que novas características são adicionadas, a linguagem precisa fornecer compatibilidade com versões anteriores, aumentando sua complexidade. Desenvolvedores então encontram-se também em um empasse, buscando um equilíbrio entre manter a compatibilidade da ferramenta com versões anteriores da linguagem e atualizá-la para manter o software atualizado [9].

A estratégia de refatoração de código pode ser uma opção que permite a linguagem e o software co-evoluírem com um menor grau de dificuldade [9]. Refatoração (do inglês, *refactoring*) é o processo de modificação de um software com o objetivo de melhorar a estrutura interna do mesmo sem alterar sua percepção externa. É uma maneira disciplinada de melhorar o design do código após ele já ter sido implementado. Por exemplo, dividir uma funcionalidade grande feita em apenas um bloco de código em vários módulos [15].

A refatoração de código pode ser um meio utilizado para evoluir o software, adaptando o código com características novas introduzidas pela linguagem, modificando o design do sistema, sem comprometer sua integridade. No entanto, novos desafios surgem, visto que

refatorar o código fonte de um sistema de grande porte não é uma tarefa fácil, havendo a necessidade de ferramentas que possam automatizar o processo ou realizar um estudo para entender como as novas características da linguagem podem ser utilizadas para substituir ou melhorar funcionalidades obsoletas [9].

## 2.2 MSR: Mineração de dados e a Engenharia de Software

Com o objetivo de compreender, prever e planejar os vários aspectos do desenvolvimento de um projeto, em especial tudo que possa impactar na evolução de software, a técnica de mineração de dados tem sido bastante utilizada em engenharia de software [17].

O termo Mineração de Repositórios de Software (MSR) tem sido utilizado para descrever várias maneiras de se examinar repositórios de software. Um repositório de software pode ser considerado um artefato que é produzido e arquivado durante a evolução de software [1]. Estes responsórios armazenam informações que podem fornecer todos os detalhes de desenvolvimento do software, provendo meios necessários para uma análise aprofundada de evolução de software.

Repositórios de software são comumente utilizados para armazenar e acompanhar o histórico de desenvolvimento do software, mas raramente usados para tomada de decisões em relação ao projeto. Assim, pesquisadores de MSR buscam modificar a impressão de que tais repositórios são estáticos que servem apenas para manutenção de registros, transformando-os em repositórios ativos que podem auxiliar na tomada de decisão em projetos de software modernos, através da identificação de padrões para realizar a propagação de mudanças [18].

### 2.2.1 Classificação

De acordo com Kagdi et al, dentre as abordagens de MSR utilizadas atualmente, é possível descrevê-las, para então compará-las e possivelmente classificá-las, levando em consideração uma taxonomia de quatro dimensões: os repositórios de software utilizados (fontes de dados), o propósito do MSR, a metodologia e a avaliação da abordagem [1], como mostra a Figura 2.1, onde as taxonomias destacadas em amarelo são as que prevaleceram, porém não exclusivamente, na metodologia adotada por este trabalho.

Figura 2.1: Taxonomia em camadas adaptada do trabalho de Kagdi et al [1]

### Fontes de Dados

As fontes de informação variam de acordo com a abordagem utilizada e, dentre as informações mineradas em repositórios de software, inclui-se as seguintes categorias: os artefatos de software e suas versões, as diferenças entre os artefatos de software e suas versões e os metadados sobre as mudanças no software.

As fontes de dados podem ser melhor classificadas de acordo com Hassan et al. [17]:

- Repositórios de controle de Versão: estes repositórios guardam o histórico de desenvolvimento de um projeto, armazenando todas as mudanças no código fonte, assim como os metadados associados com cada mudança. Exemplos: *git*, *SVN*.
- *Bug repositories*: Estes repositórios rastreiam o histórico de resolução de bugs que são reportados por usuários e desenvolvedores. Exemplos: *Bugzilla* e *Jira*.

- Histórico de Comunicação (*Archived communications*): estes repositórios rastreiam discussões sobre os vários aspectos de um projeto de software por sua vida útil. Por exemplo: listas de e-mails.
- *Deployment logs*: *logs* com informações do software gerados por ferramentas de desenvolvimento.
- Repositórios de software: Repositórios onde o código fonte de vários projetos são armazenados, como por exemplo, *Sourceforge.net*.

No contexto deste trabalho, estes repositórios consistem em fontes de código armazenadas em sistemas de controle de versão, especificamente, códigos na plataforma *Github*.

### O propósito

O propósito da mineração de repositórios de software se resume nas perguntas de pesquisa que podem ser respondidas utilizando mineração. Assim, existem duas classes de perguntas de pesquisa para MSR [1].

A primeira é a pergunta de análise *market-basket*[1]: "Se $A$ acontece, o que mais pode acontecer?". A resposta para este tipo de pergunta é um conjunto de regras descrevendo tendências de relacionamentos, como por exemplo, se $A$ acontecer, então $B$ e $C$ acontecem durante um período $X$.

A segunda classe diz respeito às perguntas de prevalência (*Prevalence Questions - PQ*), como por exemplo, perguntas do tipo "A funcionalidade $Y$ foi modificada?", que possuem uma resposta *booleana* ou "Quantas vezes esta funcionalidade $Y$ foi modificada?", contendo uma resposta quantitativa.

As perguntas de pesquisa deste trabalho se encaixam majoritariamente na segunda classe, onde o objetivo traçado é respondido de forma quantitativa, com algumas tentativas de classificação de tipos de repositórios, como será descrito nos capítulos seguintes.

### A metodologia

A metodologia corresponde a maneira como as questões de MSR irão ser respondidas e várias abordagens estão disponíveis. Porém, Kagdi indica duas estratégias básicas que podem ser utilizadas.

Primeiramente, pode-se utilizar a medição indireta e análise da evolução do software. Esta abordagem pode ser realizada da seguinte forma: extrair e computar informações de cada versão de um software separadamente, e, em seguida, individualmente comparar

[1] Análise *Market-basket* é uma técnica de modelagem baseada na teoria de que se você comprar certos tipos de produtos, você há uma probabilidade maior (ou menor) de comprar outros tipos de produtos. [19]

propriedades encontradas, como por exemplo, comparar duas versões distintas de um software através de um repositório *github*, realizando uma investigação de alto nível no que diz respeito a evolução do software [1].

A segunda perspectiva de metodologia que pode ser utilizada corresponde à medição direta e análise da evolução do software. Ela diz respeito às investigações que estudam os mecanismos que leva um software a ser modificado de uma versão para outra, focando nas diferenças específicas entre estas versões, como por exemplo, examinando mudanças específicas das classes, arquivos, funções de cada parte do código [1].

Assim, é possível descrever explicitamente que este trabalho utiliza, em sua maioria, uma metodologia com medições diretas, através da análise estática de informações do código fonte de diferente versões para analisar os diferentes padrões de desenvolvimento em que expressões lambda (definidas no capítulo seguinte) são utilizadas.

#### A Avaliação

Por fim, existem diversas formas para se avaliar o estudo realizado, sempre levando em consideração a validade dos resultados obtidos. Pode-se utilizar de métricas para avaliar os dados, considerando o quanto das informações obtidas são relevantes. Outra forma de se avaliar, seria através da utilização de modelos probabilísticos, seguindo uma abordagem de teoria da informação. Assim, foram utilizadas determinadas métricas quantitativas para avaliar os resultados obtidos neste trabalho.

## 2.3 Trabalhos Relacionados

Como ressaltado na introdução, o foco deste trabalho encontra-se em utilizar técnicas de MSR para caracterizar a adoção de expressões lambda em repositórios Java. Ao realizar um estudo sobre o estado da arte, encontrou-se diversos trabalhos que utilizam de MSR para obter informações relevantes sobre repositórios de software em diversos focos e, em especial, trabalhos que investiguem repositórios na linguagem Java. Já existem também alguns trabalhos que apresentam ferramentas de refatoração de código para implementação de expressões lambda com foco na evolução do software. Dentre os trabalhos encontrados mencionados, destacam-se:

- Em um trabalho sobre utilização de refatoração de código para realizar a transição de código Java para as novas características implementadas no Java 8, Gyori et al. apresenta uma ferramenta que automatiza, com sucesso, o trabalho de conversão de código, como *Annonymous Inner Class*, para expressões lambda [12]. O capítulo a seguir descreve em detalhes estes conceitos.

- *Regrowing a Language* de Overbey apresenta um estudo sobre a importância da co-evolução entre a linguagem de programação e os projetos de software, focando nas linguagens Fortran e Java e provando como ferramentas de refatoração de código auxiliam para que o software consiga evoluir juntamente com sua linguagem de programação [9].
- O trabalho sobre a adoção de *generics* em código Java feito por Parnin utiliza de mineração de dados para realizar uma investigação empírica, avaliando projetos de código aberto com o objetivo de observar como desenvolvedores destes projetos estão utilizando *generics* em Java [20].

Os estudos mencionados focam em assuntos particulares pertinentes a este trabalho em que, de um lado tem-se trabalhos que mostram a necessidade de refatoração de código para evolução de software, e do outro, tem-se trabalhos que mostram que é possível utilizar de MSR para entender melhor como a evolução da linguagem de programação Java tem afetado projetos reais de código aberto e como estes projetos estão se adaptando a estas evoluções.

# Capítulo 3

# Java SE 8 e JDK 8

Este trabalho tem como objetivo caracterizar a adoção de expressões Lambda em sistemas Java que podem ser considerados legados, pois foram concebidos em um momento anterior a introdução dessa nova característica na linguagem Java. Com o intuito de facilitar o entendimento do leitor sobre tais construções, esse capítulo apresenta informações históricas referentes à linguagem Java, desde sua criação, até a sua ascensão como uma das linguagens de programação mais populares. Além disso, serão apresentadas as novas características presentes na versão Java SE 8, com destaque nas expressões Lambda e de que forma vieram para auxiliar no desenvolvimento de sistemas.

## 3.1 Histórico

Java é uma linguagem de programação orientada a objetos para propósitos gerais [21]. A linguagem nasceu com o nome de *Oak* no ano de 1991 com o foco em televisões digitais a cabo, mas, devido à complexidade e poder da linguagem, logo expandiu-se para os outros domínios, principalmente a *Internet* [22]. Posteriormente, a linguagem recebeu o nome de *Greentalk* devido à marca *Oak* já ser registrada por outra empresa e o principal projeto da equipe se chamar *Green*. Por fim, o nome Java foi o mais votado pela equipe da *Sun* em uma reunião para decidir o mais rápido possível um nome definitivo para a linguagem [23].

Em 1995, em sua primeira versão pública 1.0, a linguagem Java já começava a ter um crescimento em sua popularidade como *Web Applet* nos navegadores mais populares, fornecendo segurança e rapidez nas plataformas mais utilizadas [24]. A cada nova versão publicada, novas funcionalidades e plataformas eram adicionadas e tornavam a linguagem mais difundida no meio computacional, fazendo com que em 2006 a linguagem fosse dividida em *Java EE* (foco em rede e serviços web), *Java ME* (foco em sistemas embarcados) e *Java SE* (foco em *desktops* e servidores) [25].

Após anos de evolução e aprimoramento através de diversas versões, como visto na Tabela 3.1, hoje o Java se tornou uma das linguagens de programação mais utilizadas no mundo, estando em aproximadamente 15% das linhas de código disponíveis pela *internet* [11]. A quantidade de plataformas suportadas pela linguagem e com o advento do sistema operacional *Android* sendo rodado na maior parte dos *smartphones* do mundo, além da simplicidade em codificar no paradigma orientado a objetos, foram os fatores relevantes para que a linguagem se popularizasse no mundo digital [26].

Tabela 3.1: As versões anteriores de java e principais características

| **Versão** | **Principais características** |
|---|---|
| Java 1.0 | JVM e JDK |
| Java 2 | *Event Listeners* |
| Kestrel (Java 3) | *HotSpot JVM* |
| Merlin (Java 4) | Diversas novas APIs como *Assertion* e expressões regulares |
| Tiger (Java 5) | *Generics* |
| Mustang (Java 6) | *Web Services* |
| Dolphin (Java 7) | Suporte a linguagens dinamicas |

## 3.2 Princípios

Com o foco em alcançar a *Web* e participar mais ativamente do mercado virtual e *e-Commerce*, foram projetados princípios que permitissem com que a linguagem pudesse alcançar seus objetivos principais ao mesmo tempo que mantinha a facilidade e familiarização da linguagem a seus desenvolvedores [27]. Dessa forma, a equipe da Oracle determinou cinco princípios que caracterizariam o foco da linguagem Java:

- **simples, orientada a objeto e familiar**: projetada para ser orientada a objeto e permitir que seus desenvolvedores comecem a codificar rapidamente sem a necessidade de uma curva de aprendizagem acentuada.
- **segura e robusta**: projetada para prover software confiável por meio de checagens a tempo de compilação e execução. Há uma grande preocupação com segurança devido à larga utilização da linguagem em sistemas distribuídos.
- **de arquitetura neutra e portável**: projetada para ser independente de qualquer arquitetura, bastando rodar a JVM, dessa forma, garantindo a portabilidade para vários ambientes e dispositivos.

- **executada em alta performance**: projetada para fornecer a maior performance possível em tempo de execução com a utilização do *garbage collector* em uma *thread* de baixa prioridade.
- **interpretada, dinâmica e *threaded***: projetada para ser interpretada por qualquer dispositivo que possua JVM desenvolvida garantindo um desenvolvimento rápido e em ciclos. O uso da classe `Thread` permite que a linguagem acompanhe a tendência do paralelismo. Além disso, a ligação de classes é feita em tempo real e somente quando necessária, tornando-a dinâmica.

## 3.3 Evolução

A primeira versão do Java 8 foi lançada em 2014 [28] e apresentou diversas atualizações espalhadas em categorias como linguagem, segurança, `Collections`, ferramentas, *deploy*, entre outras. Dentre a grande quantidade de mudanças, os destaques se encontraram nas novas melhorias presentes na linguagem, que afetaram de forma mais direta a maneira com que os desenvolvedores passaram a tratar e organizar o código fonte. As atualizações referentes a linguagem descritas oficialmente pela Oracle [29] são:

1. ***Lambda Expressions*** ou apenas Expressões Lambda, correspondem a uma nova característica que permite que uma pequena funcionalidade possa ser utilizada como um argumento de um método, facilitando a escrita de ações que serão feitas repetidamente em uma `Collection` ou quando um processo se completa ou quando encontra um erro. É também uma forma compacta de se escrever código que previamente era escrito com *anonymous inner classes* [29]. A Listagem 3.1 mostra uma classe hipotética `Calculator` cujo método `operateBinary()` recebe como parâmetro dois inteiros e um objeto da classe `IntegerMath` que representa a operação que deverá ser realizada, ou seja, uma funcionalidade dinâmica.

```
1  public class Calculator {
2
3    interface IntegerMath {
4        int operation(int a, int b);
5    }
6
7    public int operateBinary(int a, int b, IntegerMath op) {
8        return op.operation(a, b);
9    }
10
11   public static void main(String... args) {
12
```

```
13          Calculator myApp = new Calculator();
14          IntegerMath addition = (a, b) -> a + b;
15          IntegerMath subtraction = (a, b) -> a - b;
16          System.out.println("40 + 2 = " +
17              myApp.operateBinary(40, 2, addition));
18          System.out.println("20 - 10 = " +
19              myApp.operateBinary(20, 10, subtraction));
20      }
21  }
22
```

Listagem 3.1: Exemplo de código com o uso de Expressão Lambda [30]

2. ***Method References***. Frequentemente, expressões lambda criam métodos anônimos. Algumas vezes, porém, são utilizadas para chamar um método já existente na classe. Ao invés de escrever a expressão Lambda para esse caso, pode-se optar por utilizar um *Method Reference*, que facilita a escrita e leitura desse uso de expressão Lambda [29]. A Listagem 3.2 mostra um exemplo de um método estático que compara duas pessoas pelas suas idades e a Listagem 3.3 a utilização de *Method Reference* para aplicar a utilização desse método em um *array* e reduzir a escrita de uma expressão lambda a uma forma mais legível.

```
1  public static int compareByAge(Person a, Person b) {
2          return a.birthday.compareTo(b.birthday);
3      }}
4
```

Listagem 3.2: Exemplo de método de uma classe Pessoa [31].

```
1  Arrays.sort(rosterAsArray, Person::compareByAge);
2
```

Listagem 3.3: Exemplo de uso de *Method Reference* [31].

3. ***Default Methods*** são métodos com a palavra-chave *default* que servem para adicionar implementações de métodos a uma interface de alguma biblioteca mantendo a compatibilidade binária com versões mais antigas dessa interface. Também possibilita a implementação de métodos estáticos em interfaces [29]. A Listagem 3.4 apresenta a inclusão de um método *default* na interface `TimeClient`.

```
1      public interface TimeClient {
2          void setTime(int hour, int minute, int second);
3          void setDate(int day, int month, int year);
4          void setDateAndTime(int day, int month, int year,
5                                       int hour, int minute, int second);
```

```
6        LocalDateTime getLocalDateTime();
7
8        static ZoneId getZoneId (String zoneString) {
9            try {
10               return ZoneId.of(zoneString);
11           } catch (DateTimeException e) {
12               System.err.println("Invalid time zone: " + zoneString
   +
13                   "; using default time zone instead.");
14               return ZoneId.systemDefault();
15           }
16       }
17
18       default ZonedDateTime getZonedDateTime(String zoneString) {
19           return ZonedDateTime.of(getLocalDateTime(), getZoneId(
   zoneString));
20       }
21   }
22
```

Listagem 3.4: Exemplo de interface com um método *default* [32].

4. ***Repeating Annotations*** permitem a implementação e utilização de anotações de um mesmo tipo que podem ser utilizadas mais de uma vez para a mesma declaração [29]. A Listagem 3.5 apresenta duas anotações iguais com parâmetros diferentes, permitindo uma customização maior na aplicação de um tipo de anotação.

```
1 @Schedule(dayOfMonth="last")
2 @Schedule(dayOfWeek="Fri", hour="23")
3 public void doPeriodicCleanup() { ... }
4
```

Listagem 3.5: Exemplo de *Repeating Annotations* [33].

5. ***Type Annotations*** permitem que anotações possam ser utilizadas em tipos além de declarações, dessa forma, pode-se garantir que um campo sempre atenda às suas expectativas e evite testes desnecessários [29]. A Listagem 3.6 mostra um exemplo de restrição a uma `String` que nunca poderá ser nula devido à anotação de `@NonNull`.

```
1  @NonNull String str;
```

Listagem 3.6: Exemplo de *Type Annotation* garantindo que o campo nunca seja `null` [34].

## 3.4 Expressões Lambda

Uma das mudanças mais interessantes implementadas no Java 8 foi a adição de expressões lambda. As expressões lambda permitem que a linguagem Java possa atuar parcialmente como uma linguagem de programação funcional, onde comportamentos são avaliados e aplicados diferentemente da programação imperativa que foca na mudança de estados. Na prática, um comportamento pode ser passado como parâmetro de um método através de uma escrita mais reduzida e legível. Os benefícios de utilizar expressões lambda incluem uma melhora na legibilidade e organização de código que antes era escrito com *Anonymous Inner Class*, além de permitir uma forma simples de utilizar comportamento paralelo com o uso de *Streams*.

### 3.4.1 Sintaxe

A sintaxe de uma expressão lambda é dividida nas seguintes partes [30]:

- Uma lista de parâmetros separados por vírgulas.
- Um *token* representado por uma seta ->.
- Um corpo que pode ser representado por uma expressão única ou um bloco de código dentro de chaves.

As Listagens 3.7 e 3.8 mostram duas formas diferentes de se aplicar expressões Lambda com uma sintaxe similar. A primeira aplica a expressão diretamente após a seta, utilizada normalmente para trechos de código pequenos e simples, e a segunda utiliza-se de chaves para incorporar o código e da palavra-chave `return` para especificar o retorno, forma mais utilizada para uma quantidade maior de código.

```
1 p -> p.getGender() == Person.Sex.MALE
2 && p.getAge() >= 18
3 && p.getAge() <= 25
```

Listagem 3.7: Exemplo de expressão lambda com uma expressão [30].

```
1 p -> {
2     return p.getGender() == Person.Sex.MALE
3         && p.getAge() >= 18
4         && p.getAge() <= 25;
5 }
```

Listagem 3.8: Exemplo de expressão lambda com um bloco de código [30].

### 3.4.2 Principais formas de uso

***Anonymous Inner Class***

Antes do Java 8, a necessidade de passar funções ou expressões como parâmetro de algum método era suprida através de *Anonymous Inner Classes*. A utilização desse tipo de declaração junto com a anotação de `@Override` permite sobrescrever um método de uma interface funcional e ao mesmo tempo passa-la como argumento para um método. Uma expressão lambda se comporta da mesma forma, porém, a quantidade de código escrito pela *Anonymous Inner Class* é maior e pode dificultar a leitura de código, pois contém muita informação não utilizada para efetivo entendimento do que está sendo feito naquele trecho.

A Listagem 3.9 apresenta a instanciação da classe `EventHandler` internamente ao método `setOnAction()`, de forma que o desenvolvedor possa escrever o comportamento que deseja passar como parâmetro do método através da substituição do método `handle()`, que já existe, pelo método `handle()`, criado e escrito pelo desenvolvedor.

```
1         btn.setOnAction(new EventHandler<ActionEvent>() {
2             @Override
3             public void handle(ActionEvent event) {
4                 System.out.println("Hello World!");
5             }
6         });
```

Listagem 3.9: Exemplo de uso de *Anonymous Inner Class* [35]

**Interface funcional**

Ao projetar um código novo, o desenvolvedor que optar por utilizar expressões lambda, deve iniciar com a criação de uma interface funcional. Esse tipo de interface possui um único método abstrato permitindo que expressões lambda sejam usadas para sobrescrever o comportamento do método abstrato. Uma interface funcional pode possuir outros métodos desde que sejam *default*. Uma nova anotação `@FunctionalInterface` foi adicionada com a atualização do Java 8 para conferir em tempo de compilação se a interface satisfaz os requisitos desse modelo de interface.

A Listagem 3.10 apresenta um exemplo de interface funcional com o objetivo de permitir o uso de expressões lambda que utilizarão de dois `Double` como parâmetro e já utiliza da nova anotação `@FunctionalInterface` introduzida no Java 8.

```
1 @FunctionalInterface
2 public interface DoubleOp {
3     public Double apply(Double a, Double b);
```

```
4 }
```

Listagem 3.10: Exemplo de interface funcional

Dessa forma, conforme as Listagens 3.11 e 3.12, o método `apply()` pode tomar forma de qualquer operação entre dois `Double` que será especificada quando a expressão lambda for montada, dando o aspecto de linguagem funcional à linguagem e mais facilidade e liberdade ao desenvolvedor.

```
1 public class Calculator {
2     public static Double calc(Double op1, Double op2, DoubleOp operator) {
3         return operator.apply(op1, op2);
4     }
5 }
```

Listagem 3.11: Exemplo de aplicação da interface funcional

```
1 Double result1 = Calculator.calc(10d, 50d, (a, b) -> a + b);
```

Listagem 3.12: Uso de expressão lambda para o exemplo anterior

#### ***Collections***

Dentre as atualizações de pacotes do Java 8, foi adicionado um método *stream*() na interface `Collection`. Esse método trata as `Collection` como *streams*, o que significa que apesar de não ser possível acessar elementos diretamente, algumas novas manipulações são possíveis. Essas manipulações são, na verdade, métodos da interface `Stream` que podem receber como parâmetro uma expressão lambda, ou seja, um comportamento a ser aplicado à `Collection`. Dentre os métodos, temos:

- **filter()**: O método `filter()` recebe uma expressão lambda contendo uma condição para filtragem da *collection*.
- **map()**: O método `map()` mapeia os elementos de uma `Collection` em outro objeto como uma `String`. A expressão lambda passada por parâmetro deve retornar o tipo de objeto ao qual será mapeado.
- **forEach()**: O método `forEach()` aplica uma operação específica para cada elemento da `Collection`. A expressão lambda a ser passada deve conter a operação a ser realizada.
- **min() e max()**: Os métodos `min()` e `max()` retornam o elemento menor ou maior da `Collection` dependendo da expressão lambda passada como parâmetro.

A Listagem 3.13 demonstra como várias operações com uma `Collection` podem ser aplicadas de uma vez, de forma que lendo a expressão lambda já é possível saber o resultado da aplicação do método à `Collection`.

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Listagem 3.13: Exemplo de utilização de métodos de *stream* com expressões lambda [30].

# Capítulo 4

# Estudo e o Processo de Mineração

A definição da metodologia utilizada para realização deste trabalho é importante para que exista uma padronização e melhor entendimento de como os resultados foram obtidos, para que a replicação do mesmo possa ser realizada em trabalhos futuros [36].

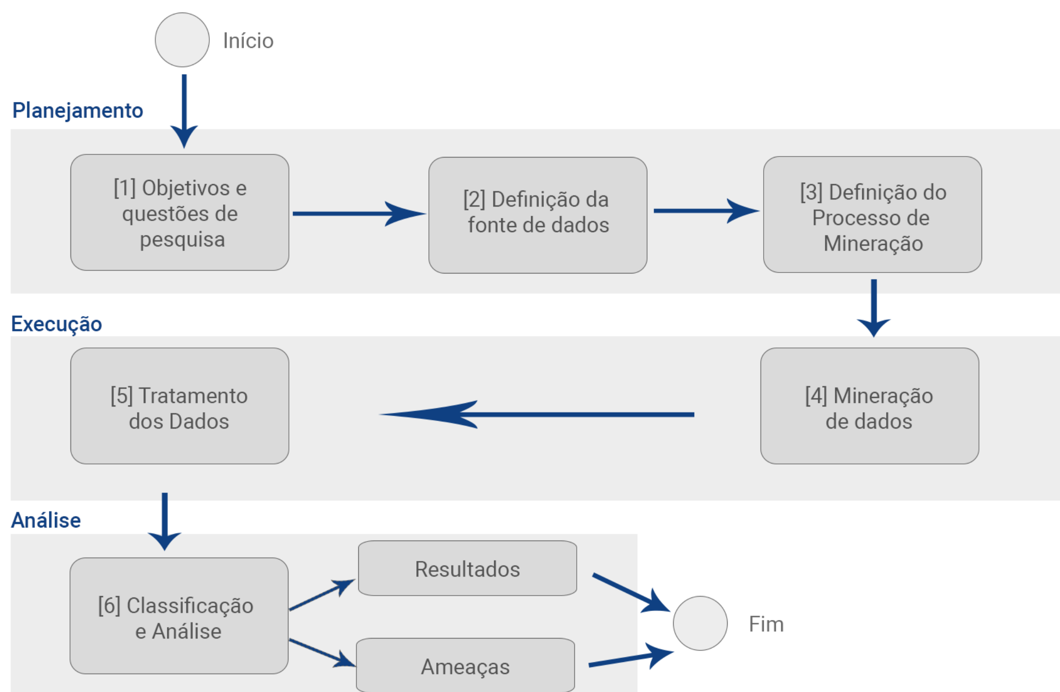


Figura 4.1: Processo utilizado para realização do estudo.

O esquema representado na Figura 4.1 descreve o processo seguido para a realização deste trabalho, que foi divido em três grandes etapas: a de planejamento, definindo os objetivos, questões de pesquisa e meios de investigação deste projeto, a etapa de execução, realizando todo o procedimento metodológico definido para a coleta e refinamento de

dados, e a etapa de avaliação, onde foram classificados e avaliados os dados coletados para responder às questões de pesquisa.

A seguir, na primeira seção, serão apresentadas, em detalhes, as questões de pesquisa a serem respondidas. Em seguida, será feito uma breve descrição da fonte de dados utilizada - especificamente, os projetos utilizados como objetos de estudo deste trabalho e, por fim, todo o restante da metodologia que foi de fato desenvolvida para obter os resultados necessários.

## 4.1 Questões de Pesquisa

Com o objetivo de entender e caracterizar melhor o uso de expressões lambda, foram definidas as seguintes questões de pesquisa a serem respondidas:

- **PQ-1**: Quando os projetos começaram a introduzir o uso de expressões lambda em seus códigos e qual a média de expressões lambda por *snapshots*, caso existam?
- **PQ-2**: Como as equipes de desenvolvimento estão utilizando expressões lambda: através de refatoração de código ou introduzindo apenas em código novo?
- **PQ-3**: Quais são os padrões tipicamente adotados para o uso de expressões lambda?

A primeira pergunta de pesquisa gira em torno de como projetos populares *open-source* estão se adaptando à introdução de expressões lambda em Java: muitos projetos já estão utilizando as expressões lambda? Se sim, a partir de quando começaram a utilizá-las: desde o início da introdução das mesmas no Java 8 ou em um período mais tardio? Além disso, é interessante descobrir se a utilização dessa nova característica tem acontecido de forma expressiva ou não.

Em seguida, para melhor entender a forma como as expressões lambda estão sendo introduzidas em projetos, é interessante investigar se as equipes de desenvolvimento estão se preocupando em refatorar código com o objetivo de evoluir o software ou se estão experimentando utilizar expressões lambda apenas em código novo introduzido.

Como a introdução de novas funcionalidades em projetos *open-source* de grande escala depende do objetivo do projeto e os padrões de projeto utilizados [37], espera-se observar diversas abordagens quanto a utilização de expressões lambda para, então, poder classificar estes softwares quanto a abordagem utilizada.

É de grande importância para este estudo conseguir identificar, na prática, como expressões lambda estão sendo tipicamente adotadas nestes projetos. Assim, a terceira pergunta de pesquisa busca identificar alguns exemplos de utilização, através da realização de alguns estudos de caso.

É importante ressaltar que, apesar de as expressões lambda serem uma alternativa ao uso de AIC, elas não os substituem completamente. Existem precondições para que uma AIC possa ser substituída por expressões lambda [12], descritas a seguir:

- AIC deve instanciar-se de uma interface.
- AIC não deve possuir nenhum campo e declarar apenas um método.
- AIC não deve possuir uma referência para `this` ou `super`.
- AIC não deve declarar um método recursivo.

Para responder à primeira pergunta, apenas dados gerais sobre todos os projetos serão necessários, ao passo que, pelas perguntas 2 e 3 possuírem uma natureza mais complexa, será necessário, além de uma análise geral estatística, a realização de alguns estudos de caso para obter resultados mais completos.

## 4.2 Fonte de Dados e Objetos de Estudo

Para responder às perguntas de pesquisa estabelecidas, foram selecionados os 99 projetos mais populares na linguagem Java dentro da plataforma *github* até a data da coleta dos dados, feita em Abril de 2017. São considerados populares, os repositórios que possuem a maior quantidade possível de estrelas (*stars*). A decisão de escolher os 99 projetos mais populares foi por sua natureza imparcial e, ao mesmo tempo, para que se possa analisar projetos que são de familiaridade e conhecimento da comunidade do *github* e de domínio público. Os 99 projetos estão listados no Anexo I.

Para entender a transição feita por esses projetos entre as versões anteriores do Java e o Java 8, com as expressões lambda, é necessário analisar o código de várias versões a fim de observar alguma mudança, para cada projeto. Como são 99 projetos e vários desses projetos possuem milhares de linhas de código e *commits*, seria impossível avaliar todos os *commits*. Optou-se então por coletar *snapshots* mensais de cada projeto, conforme procedimento também seguido anteriormente [38]. Cada *snapshots* representa o estado do projeto em um determinado mês e, como critério de escolha padrão, todos os *snapshots* equivalem ao último *commit* realizado no determinado mês que este representa.

O próximo passo é determinar o período de tempo em que esses *snapshots* seriam coletados. Como Java 8 (e, consequentemente, as expressões lambda) foi introduzido apenas em meados de 2014 [28], coletar dados de projetos a partir de um período muito antes deste ano não agregaria valor ao estudo em questão. Definiu-se então um período para a coleta mensal desses *snapshots* através dos anos: Janeiro de 2013, um ano antes da introdução do Java 8 para que se pudesse identificar, para todos os projetos, o mês

exato em que expressões lambda foram introduzidas, até o mês da coleta de dados, Abril de 2017.

É importante ressaltar que a definição dos meses foi definida do dia primeiro de um mês ao dia primeiro do próximo mês e, por consequência, alguns *snapshots* que deveriam representar o final de determinado mês correspondem, na verdade, a *commits* realizados no dia primeiro do mês seguinte. Por exemplo, pode existir casos em que o *commit* que represente o mês de Março de 2016 seja, na verdade, o *commit* realizado no dia primeiro de Abril de 2016. Isso se deve a natureza da forma como os resultados de log de *commits* são obtidos pelo git [39]. Porém, isso não interfere nas análises ou no propósito deste trabalho, uma vez que foram realizadas verificações e validações para que não existam datas repetidas (e consequentemente, *commits* repetidos) por repositório.

Neste trabalho, o termo *projeto* e *repositório* foram utilizados de forma intercalada, para representar a mesma coisa: o software desenvolvido que possui um repositório na plataforma *github*. Os termos *commit*, *snapshops* e *versões* também representam a mesma coisa, uma vez que foi definido que seriam utilizados apenas um *commit* por mês para representar um determinado *snapshot* ou versão do repositório em questão.

Todos os 99 projetos serão utilizados para a realização de uma análise geral de caracterização do uso de expressões lambda. Para as análises aprofundadas, foi feito um critério de classificação para que apenas alguns projetos possam ser analisados em nível de código como estudos de caso.

Resumindo, foram escolhidos os 99 projetos mais populares de Java no *github*, coletando dados mensais de seus *commits* a partir do ano de 2013 até a data da coleta dos dados, lembrando que nem todos os projetos existem desde 2013 (projetos mais recentes).

## 4.3 Processo de Mineração

A abordagem utilizada para coletar e analisar os 99 projetos escolhidos, como mostra a Figura 4.2, foi realizada de forma automatizada, utilizando *scripts python* e dois projetos desenvolvidos em Java para a coleta e tratamento de dados, como descrito a seguir.

Primeiramente, foi realizado uma cópia dos repositórios atuais de todos os 99 projetos mais populares do *github* em Java. Para automatizar o processo, foi desenvolvido um *script* em *python* que lista todos os projetos escolhidos, clona todos os repositórios em uma pasta na máquina local e cria um arquivo temporário com as informações do nome de cada repositório e seu caminho absoluto na máquina local.

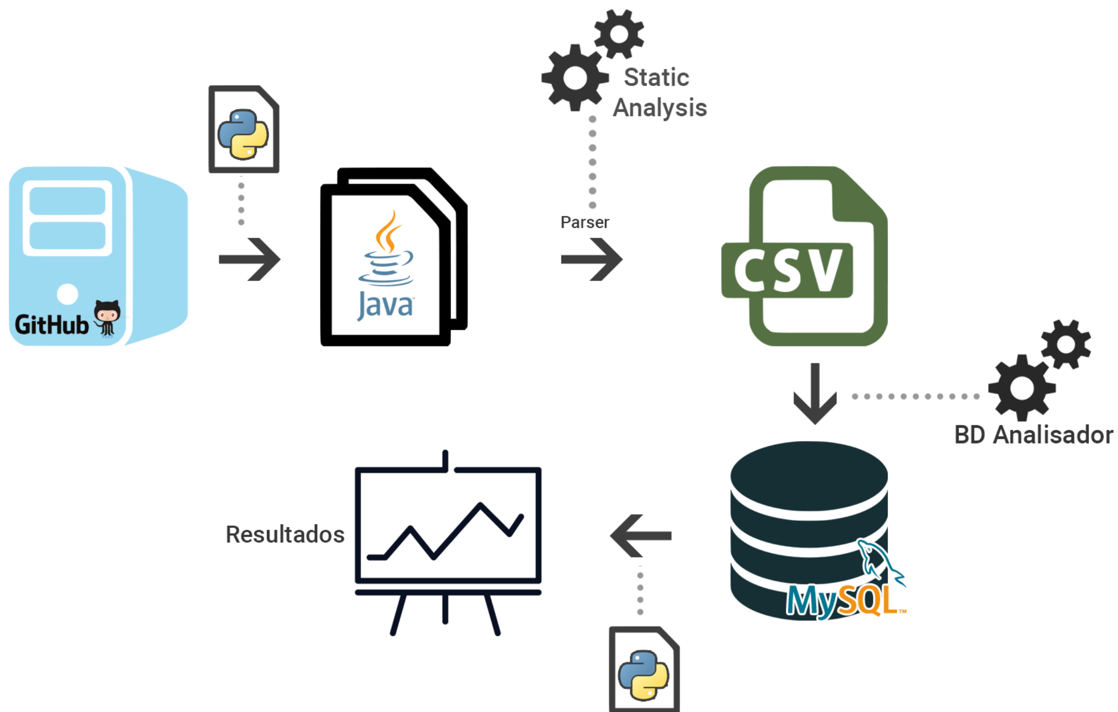


Figura 4.2: Metodologia implementada para coleta e tratamento de dados.

Em seguida, foi necessário realizar o *checkout* de todas as versões entre Janeiro de 2013 a Abril de 2017 de cada projeto. Por motivos de otimização de espaço da máquina local e automatização no processo de *checkout*, utilizou-se de um segundo *script python* que realizava as seguintes tarefas: para cada mês, em cada projeto, verificar se dentre o mês estipulado houve algum *commit* para aquele projeto; se sim, pega-se o *hash* do último *commit* do mês, realiza o *checkout*, conta-se a quantidade de linhas de código em Java que o projeto tem, utilizando a biblioteca *cloc*, e armazena todas as informações pertinentes referentes a todos os projetos que naquele mês tiveram *commits* para que o projeto *static-analysis* possa então ser executado sobre esses projetos naquele determinado momento. As informações coletadas que são utilizadas pelo *static-analysis* correspondem ao nome do projeto, o *hash* do *commit* para aquela versão, a pasta onde o projeto está armazenado na máquina local, a quantidade de linhas de código Java do projeto e data referente ao commit. Essas informações são pertinentes para que o projeto *static-analysis* consiga funcionar de forma correta, projeto este descrito em detalhes a seguir.

### 4.3.1 *static-analysis*

O *static-analysis* foi um projeto desenvolvido anteriormente a este trabalho, pelos alunos Thiago Cavalcanti e Vinicius de Almeida, cujo objetivo é realizar a análise estática de

códigos-fonte Java e gerar arquivos de saída contendo informações de classes e métodos com suas devidas ocorrências de características da linguagem [40].

A análise estática de código consiste em analisar um código-fonte sem a necessidade de executá-lo. Esse tipo de análise é de extrema importância para manter um código de qualidade, evitar futuras refatorações e obter informações estatísticas de forma rápida e prática [41].

Seguindo esse princípio, o *static-analysis* foi projetado para receber um arquivo de entrada *.csv* seguindo o padrão:

```
Tipo da Aplicação; Inicio do projeto Antes ou Após Java 5; Nome do Projeto;
Versão; Caminho absoluto;Quantidade de linhas de código;
```

Através da informação do caminho absoluto do projeto, o *static-analysis* varre os diretórios em busca dos arquivos fonte .java, realiza o parse desses arquivos e utiliza de `Visitors` para extrair as informações desejadas e exporta-las em arquivos *.csv*. O uso de `Visitors` permite que sejam escolhidas as informações desejadas para a consulta, como expressões Lambda e *AIC*s, além de possibilitar que sejam implementados novos `Visitors` conforme a linguagem evolua ou as necessidades mudem.

Para este trabalho foram utilizados os `Visitors` de expressões Lambda, `map()`, `filter()`, *AICs* e declarações de métodos.

### 4.3.2 *BDAnalisador*

Após obter todas as informações necessárias geradas pelo *static-analysis*, era preciso manipular os dados de forma que facilitasse a coleta de dados para o estudo proposto. Como vários arquivos em formato *.csv* são de difícil manipulação, foi desenvolvido então uma ferramenta chamada *BDAnalisador*. Esta ferramenta é responsável por processar os arquivos em *.csv* gerados pelo *static-analysis* e popular uma base de dados relacional *MySQL* com as informações pertinentes a este estudo.

O *BDAnalisador* foi desenvolvido com a intenção de tornar mais simples o trabalho com os dados resultantes do *static-analysis* de forma que consultas em *SQL* pudessem ser feitas a ponto de permitir a obtenção de resultados mais complexos. Seguindo esse objetivo, o *framework Hibernate* foi escolhido como ferramenta para persistência devido a ser um *framework* já consolidado no mercado e que de forma simples relaciona os objetos Java ao banco de dados *MySQL* [42]. A divisão de classes foi projetada em quatro entidades: `Projeto`, `Versao`, `Classe` e `Metodo`, cada uma com seus respectivos campos contendo as informações necessárias para a pesquisa, como pode ser visto na Figura 4.4. É importante ressaltar que a tabela de `Versao` guarda *hashs* de *commits*, devido à escolha de utilizar *commits* para obter mais informação de evolução de código em um menor

espaço de tempo. Além disso, para este trabalho, optou-se por persistir todos os métodos e classes de cada versão sem se importar com a repetição desses dados, pois a hipótese de haver grandes refatorações entre versões poderia comprometer a confiabilidade das informações.

Figura 4.3: Diagrama de classes UML do *BDAnalisador*

Como visto na Figura 4.3, seguindo parcialmente a arquitetura *MVC*, cada entidade possui sua respectiva classe *DAO(Data Access Object)* para persistência dos dados, e apenas uma classe `Controlador` no projeto, responsável pela manipulação e parse dos dados dos arquivos *.csv* gerados pelo *static analysis* para serem enviados às classes *DAO*. O sistema inicia chamando o método `inicializar()` da classe `Controlador`. Esse método é responsável por ler um arquivo de entrada *.csv* que segue o mesmo modelo do arquivo do *static-analysis* citado na seção anterior. O arquivo contém os dados de todos os projetos e suas respectivas versões, dessa forma, o método instancia um objeto da classe `Projeto` e persiste o projeto no banco de dados. Para cada versão encontrada no arquivo de entrada, o `inicializar()` aciona os métodos responsáveis por instanciar e preencher as listas da classe `Classe` e `Metodo`. Os métodos responsáveis por preencher as listas varrem os arquivos *.csv* gerados pelo *static-analysis* e fazem o parse das informações relevantes, como nomes dos métodos e quantidades de lambdas, para construir os objetos que serão gravados. Ao final dos preenchimentos, o objeto `Versão` é instanciado e gravado no banco de dados através de sua classe *DAO*, ao mesmo tempo que grava suas respectivas `Classes` e `Metodos`.

Após vários testes e correções de bugs, o banco de dados se encontrava finalmente com as informações de 99 projetos Java *open-source* separados por um *commit* por mês desde 2013, cada um com suas respectivas informações de classes, métodos e a quantidade de expressões lambda, *filters* e *maps* em cada método. *Filter*, ou *Filter pattern*, é o padrão de projeto que tem como objetivo filtrar objetos de uma coleção (*Collection*) de acordo com algum critério. Já o *map*, ou *map pattern*, é um padrão que "mapeia os elementos de uma coleção, aplicando uma função a eles para uma nova coleção [43]. A coleta de informações sobre esses padrões podem ser úteis para possíveis análises de oportunidades de uso de expressões lambda para refatoração de código.

A Figura 4.4 mostra o esquema de como o banco de dados resultante foi gerado.

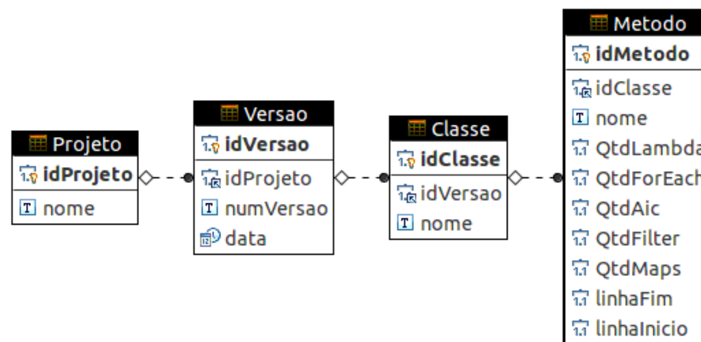


Figura 4.4: Modelo relacional do banco de dados populado pelo *BDAnalisador*. Imagem gerada com software *DBeaver* [2]

## 4.4 Classificação e Análise

Com o banco de dados populado, restam apenas as etapas de classificação e análise para responder as questões de pesquisa propostas. Para isso, inicialmente, houve uma tentativa de se utilizar apenas *queries SQL* para analisar os dados de acordo com as informações relevantes. Porém, devido ao fato de o banco de dados conter todas as informações de todos os projetos e todas as suas versões, *queries* que possuem uma alta complexidade demoravam muito tempo para retornar os dados.

Assim, com o objetivo de facilitar o trabalho, alguns scripts *python* foram desenvolvidos para rodar as *queries* separadamente e individualmente para cada projeto, de forma automatizada, a fim de melhorar o tempo de resposta para obtenção dos resultados.

### 4.4.1 Média de Lambda

O primeiro *script* retorna a quantidade de expressões lambda (ou AICs) por $n$ *snapshots* que contenha expressões lambda. O objetivo é ter uma visão geral da média de expressões lambda e AICs utilizadas por cada projeto e, como nem todos os *snapshots* coletados de cada projeto possuem expressões lambda e AICs, apenas aqueles com alguma expressão lambda ou AIC são incluídos no cálculo da média.

$$M_{media} = \frac{\sum^{n} qtdLambda_i}{n}$$

### 4.4.2 Análise Temporal

O segundo *script* retorna, para cada projeto, a soma total da quantidade de expressões lambda, AIC, `map()` e `filter()` no projeto inteiro para cada versão. Ou seja, tem-se o total de cada uma das quatro propriedades selecionadas para cada mês do projeto, obtendo, assim, uma visão temporal de evolução do uso das características descritas. A Tabela 4.1 ilustra parcialmente o resultado obtido para o projeto *agera*, onde `idVersao` é o identificador no banco de dados referente ao *hash* do *commit* mensal coletado, *qtd* é a quantidade de cada uma das características escolhidas e a data do *commit* encontra-se no formato americano.

Tabela 4.1: Informações mensais sobre o projeto *agera*.

| **idVersao** | **Data do Commit** | **qtd lambda** | **qtd AIC** | **qtd maps** | **qtd filter** |
|---|---|---|---|---|---|
| 823 | 5/1/2016 | 0 | 29 | 4 | 0 |
| 755 | 6/1/2016 | 13 | 28 | 4 | 0 |
| 686 | 6/15/2016 | 13 | 29 | 4 | 0 |
| 621 | 8/1/2016 | 13 | 29 | 4 | 0 |
| 555 | 8/16/2016 | 13 | 30 | 4 | 0 |
| 485 | 9/2/2016 | 13 | 30 | 4 | 0 |
| 422 | 10/13/2016 | 14 | 30 | 4 | 0 |
| 356 | 11/28/2016 | 14 | 30 | 4 | 0 |
| 295 | 12/21/2016 | 21 | 30 | 5 | 0 |
| 232 | 2/1/2017 | 21 | 30 | 5 | 0 |
| 175 | 3/1/2017 | 21 | 30 | 5 | 0 |
| 112 | 3/31/2017 | 21 | 36 | 5 | 0 |
| 48 | 4/24/2017 | 21 | 48 | 5 | 0 |

Os resultados obtidos com esta análise temporal permitiu a identificação de padrões na utilização de expressões lambda em relação a utilização de AIC por todos os projetos. Foi

possível classificar todos os projetos em 6 grupos, ilustrado na Figura 4.5. Primeiramente, separou-se os projetos entre os que possuem expressões lambda e os que não possuem. Dentre os que possuem, identificou-se cinco categorias:

1. **Grupo 1**: Quantidade de AICs e expressões lambda aumentam com o tempo proporcionalmente.
2. **Grupo 2**: Quantidade de AICs diminui à medida que a quantidade de expressões lambda aumentam e a quantidade de expressões lambda ultrapassa a quantidade de AIC em um determinado momento.
3. **Grupo 3**: Quantidade de AICs diminui à medida que a quantidade de expressões lambda aumentam, mas a quantidade de AIC permanece superior à quantidade de lambda.
4. **Grupo 4**: Expressões lambda são introduzidas e existem por um pequeno período de tempo, mas são completamente removidas posteriormente.
5. **Grupo 5**: Quantidade de expressões lambda é constante e muito inferior comparado com a quantidade de AICs no projeto. Foi determinado que, para pertencer a esta classificação, o projeto precisa ter uma razão de média de expressões lambda/*snapshot* por média de AIC/*snapshot* inferior a 0.20. Esse parâmetro foi estabelecido com o objetivo de definir um padrão para o que pode ser considerado um projeto com poucas expressões lambda.

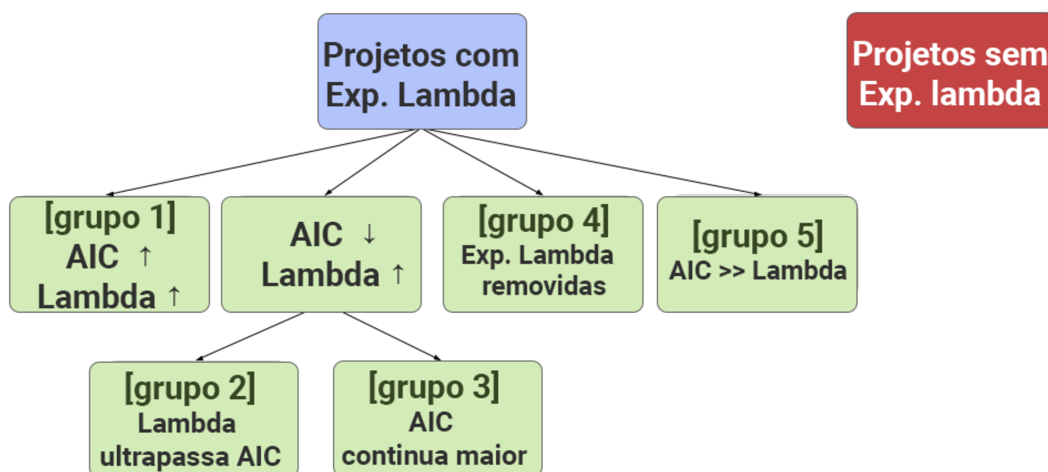


Figura 4.5: Classificação dos projetos.

### 4.4.3 Refatoração de Código

O último *script* teve como objetivo realizar uma tentativa inicial de identificar refatoração de código da seguinte maneira: para todos os projetos que possuem expressões lambda, foram identificados o mês imediatamente antes da introdução de expressões lambda e o mês seguinte, onde houve a primeira ocorrência de lambda no projeto. Assim, realizou-se uma análise do tamanho, em quantidade de linhas de código, de todos os métodos do projeto que no mês seguinte tiveram introdução de lambda. Em seguida, fez-se uma comparação com os mesmos métodos em relação ao mês anterior e, assim, foi possível detectar quais métodos tiveram a quantidade de linhas de código reduzidas, o que seria um indicativo, não necessariamente exclusivo, de que uma refatoração de código possa ter ocorrido.

Sabe-se que este método de detecção de refatoração de código possui suas limitações, uma vez que as comparações para saber se um método existe em ambas versões é feito puramente por comparação de `String` (nome do método) e a classe que este pertence.

### 4.4.4 Casos de Uso

Os métodos descritos acima ajudaram a identificar alguns padrões e facilitar a análise dos dados. Porém, não são suficientes para que se possa responder às questões de pesquisa. Dessa forma, após a identificação dos grupos de classificação para os projetos, foram escolhidos um projeto de cada grupo (com exceção do grupo dos projetos sem expressões lambda) para realizar um breve estudo de caso, com o objetivo de identificar e caracterizar melhor a adoção de expressões lambda. Os projetos escolhidos foram: *agera*, *vert.x*, *Android-CleanArchitecture*, *RxJava* e *guava*.

# Capítulo 5

# Resultados Obtidos e Análise

Neste capítulo serão apresentados os dados obtidos e análise com o objetivo de responder às questões de pesquisa propostas. Como mencionado anteriormente, as perguntas serão respondidas de duas maneiras: algumas com informações gerais sobre todos os projetos, e outras com breve estudos de caso envolvendo cinco projetos.

## 5.1 Visão Geral

Dentre os repositórios coletados para a análise, 74 (74.74%) não utilizaram nenhuma expressão lambda no projeto durante todo o período analisado (Janeiro de 2013 a Abril de 2017), restando apenas 25 repositórios (25.25%) com expressões lambda. É interessante destacar que, apesar de a quantidade de repositórios que não possuem expressões lambda ser relativamente expressiva, muitos destes repositórios são de projetos em Java que um dia foram populares e atualmente não estão sendo mais desenvolvidos (por exemplo, um aplicativo que só funciona para *Android* 2.3, atualmente descontinuado).

Levando em consideração apenas os projetos que possuem expressões lambda, existem dois tipos de dados que podem ser analisados para responder à primeira questão de pesquisa: através dos dados obtidos pelo método anteriormente mencionado como análise temporal e a média de lambda por *snapshot*, mencionados no capítulo anterior.

Com relação ao ano em que a primeira ocorrência foi encontrada, 6 projetos introduziram expressões lambda já em 2014, 8 começaram a utilizar em 2015, 7 em 2016 e apenas 4 tiveram a primeira ocorrência de expressões lambda em 2017. Dessa forma, percebe-se que os projetos estão relativamente bem distribuídos em grupos quanto ao ano em que se introduziu expressões lambda.

Pela média de lambda por *snapshot* (lambda/*snapshot*), a maioria (11 projetos) possui uma quantidade expressiva: acima de 100 lambda/*snapshot* (com 6 projetos entre 20 e 100 lambda/*snapshot* e 8 projetos com menos de 20 lambda/*snapshot*), como apresentados no

gráfico da Figura 5.1. Isso mostra que essa nova característica já está sendo bem utilizada nos projetos que começaram a migrar para a nova versão do Java.

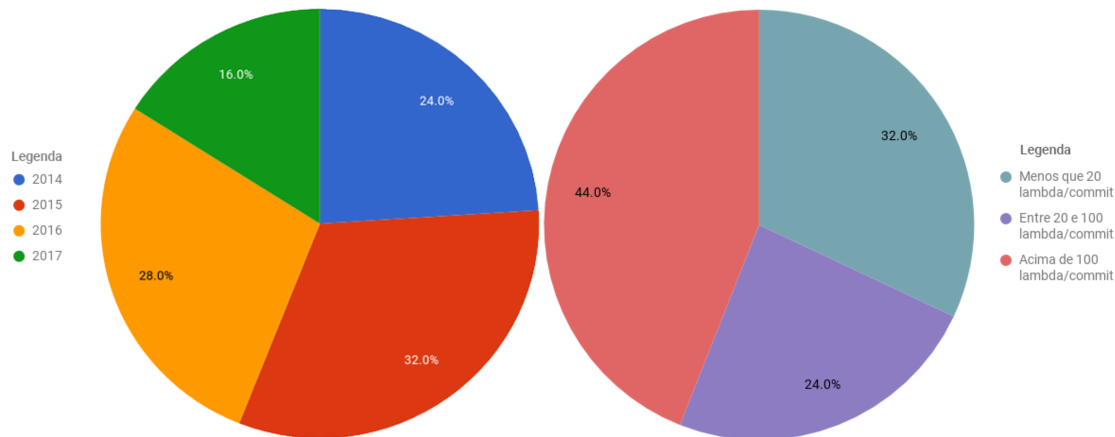


Figura 5.1: Gráficos dos projetos separados por ano de introdução de expressões lambda e média de lambda por *snapshot*.

Com isso, pode-se observar que apesar de a quantidade de projetos que possuem expressões lambda não ser tão alta, é possível obter resultados expressivos, uma vez que a quantidade dos projetos com lambda estão bem dispersos quanto ao período em que começaram a adotar essa característica e a quantidade de expressões lambda por *snapshot*.

### 5.1.1 Projetos Selecionados

Através dos grupos de classificação identificados pela análise temporal, descritos na Figura 4.5, cinco projetos foram selecionados para uma análise detalhada levando em consideração o grupo em que eles pertencem. Estes projetos serão apresentados a seguir.

### 5.1.2 *agera*

*Agera* é uma biblioteca *Android* que ajuda a introduzir programação funcional reativa (*functional reactive programming*) em projetos *Android*. É um projeto recente, lançado em 2016 pela Google [44], e encontra-se no grupo dos projetos que introduziram expressões lambda em 2016, no mesmo ano de seu lançamento, contendo uma média de expressões lambda de 16 lambda/*snapshot*.

Na classificação estabelecida, *agera* pertence ao Grupo 1, onde a quantidade de expressões lambda no projeto, com o tempo, aumentam juntamente com a quantidade de AICs, como mostra o gráfico de análise temporal na Figura 5.2.

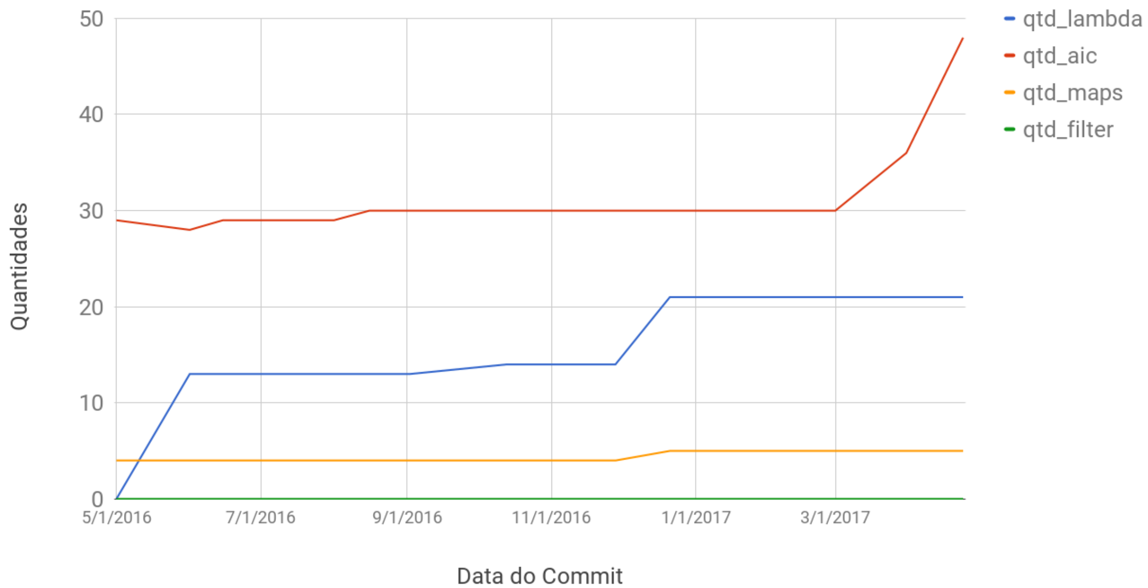


Figura 5.2: Gráficos de análise temporal do projeto *agera*

### 5.1.3 *vert.x*

*Eclipse Vert.x* é um conjunto de ferramentas para criação de *reactive applications* na máquina virtual Java (JVM) em grande escala [45]. Seu primeiro lançamento aconteceu em 2011 e a primeira ocorrência de expressões lambda no projeto aconteceu em 2014. Este projeto possui uma média de 2867 lambda/*snapshot* e, dentre os projetos considerados neste trabalho, é o que possui a maior quantidade de expressões lambda.

Este projeto pertence ao Grupo 2, onde a quantidade de expressões lambda aumentou consideravelmente pelos meses, ultrapassando a quantidade de AICs que diminuiu drasticamente, como mostra a Figura 5.3.

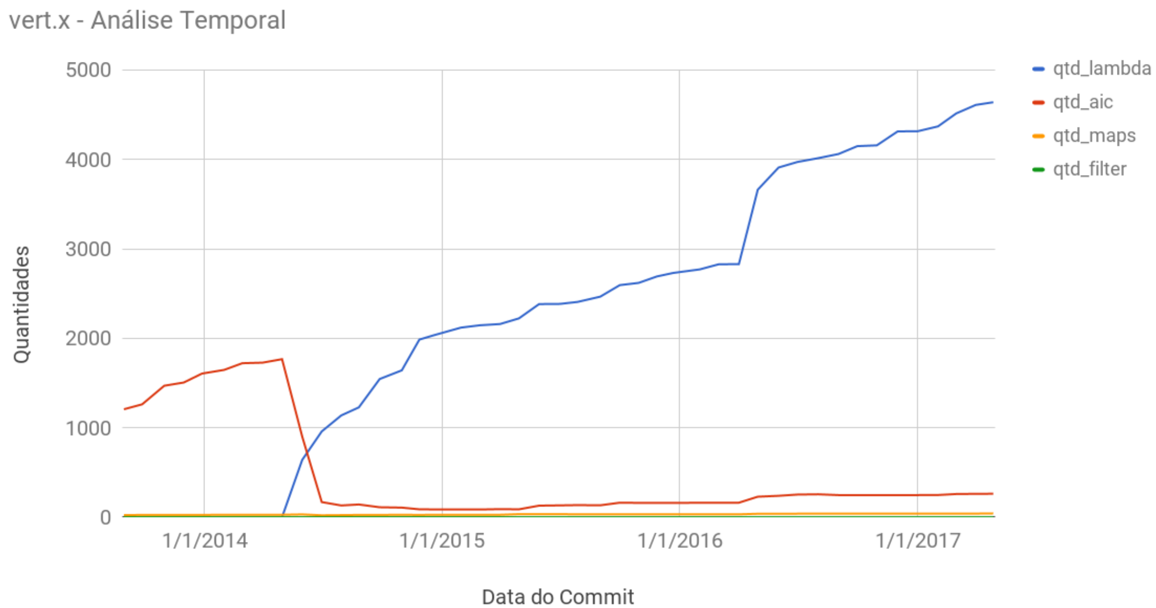


Figura 5.3: Gráficos de análise temporal do projeto *Vert.x*

### 5.1.4 *Android-CleanArchitecture*

Apesar de ser uma amostra de projeto, Android-*CleanArchitecture*, que tem como objetivo construir aplicações *Android* utilizando *clean architecture*, foi lançado em 2014, mas se mantém sempre atualizado [46].

A introdução de expressões lambda, que aconteceu no ano de 2015, se deu de forma tímida, com uma média de 3.5 lambda/*snapshot*. De qualquer forma, este projeto consegue representar, com seu gráfico de análise temporal, o grupo 3, composto por projetos em que a quantidade de expressões lambda aumentou mas sem ultrapassar AICs, que diminuíram, demonstrado no gráfico da Figura 5.4.

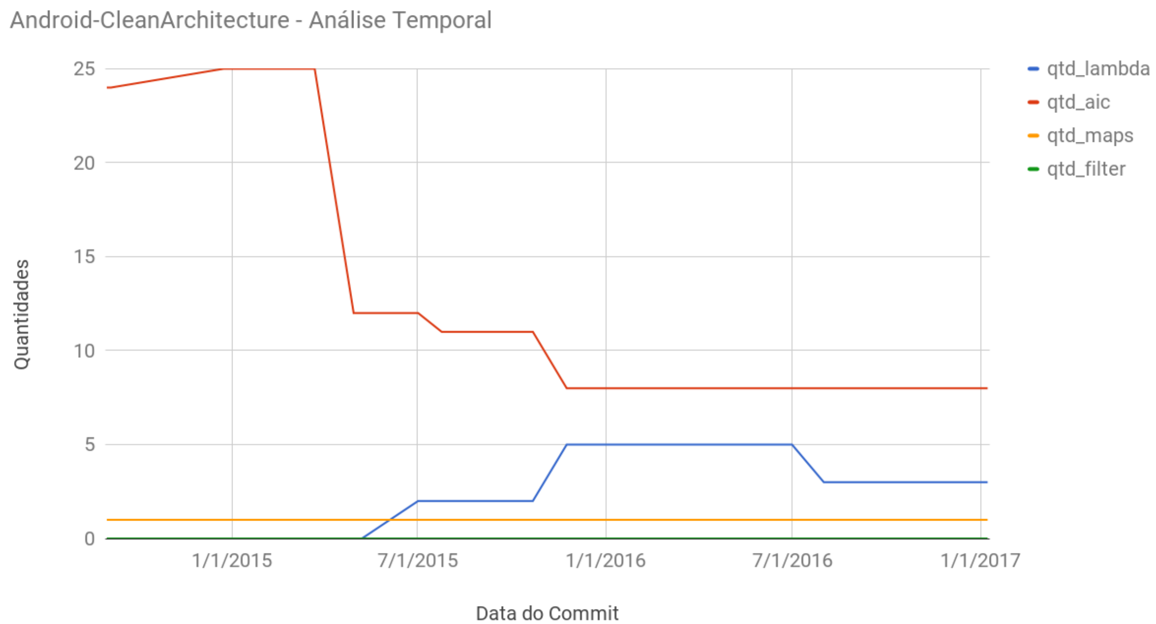


Figura 5.4: Gráficos de análise temporal do projeto *Android-CleanArchitecture*

### 5.1.5 *RxJava*

Dentre os repositórios estudados, *RxJava* é considerado o mais popular. Como uma implementação de *Reactive Extensions* para a JVM, *RxJava* teve seu lançamento dado em 2013 [47]. Expressões lambda apareceram pela primeira vez no ano de 2015, com uma média de 766 lambda/*snapshot*.

Apesar de uma média relativamente alta, o tempo de vida das expressões lambda neste projeto foi bem curto, caracterizando, assim, projetos do grupo 4: houve a introdução das expressões lambda no projeto, porém, houve a remoção completa de todas as expressões lambda previamente introduzidas, desaparecendo completamente do projeto até a data da coleta de dados (Figura 5.5).

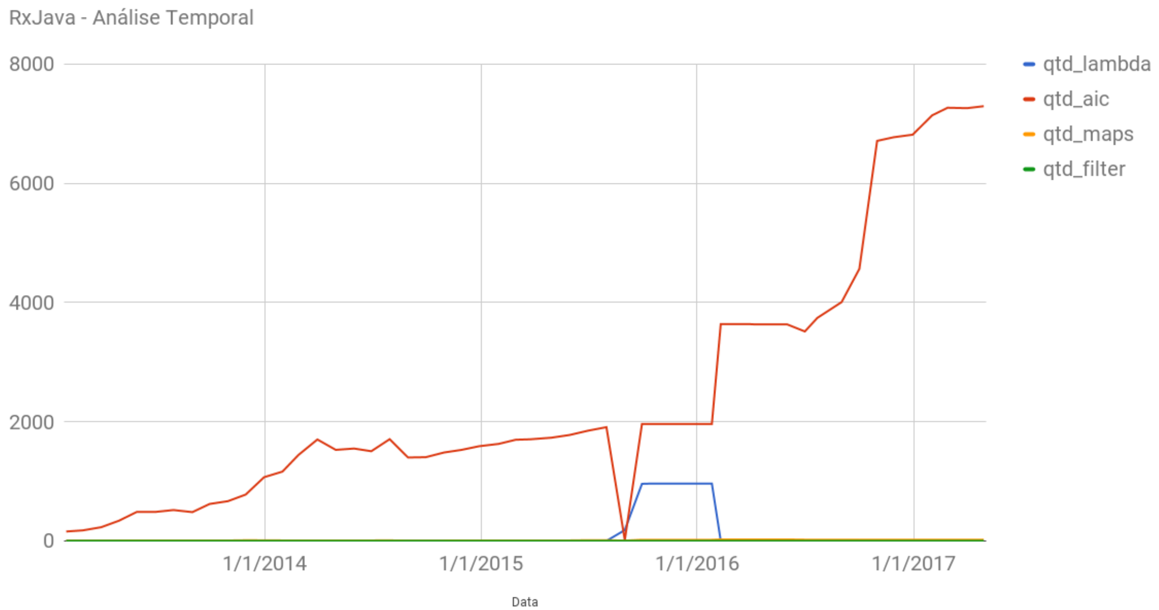


Figura 5.5: Gráficos de análise temporal do projeto *RxJava*

### 5.1.6 *guava*

*Guava* é um conjunto de bibliotecas para Java da Google lançado em 2011. A atualização do código do projeto para Java 8 aconteceu somente em 2016 e, apesar do projeto conter uma média de 281 lambda/*snapshot*, ele foi escolhido como representante dos projetos do Grupo 5 (Figura 5.6), que possuem expressões lambda mas de forma não expressiva o suficiente quando comparado com a quantidade de AIC, pelo fato de sua razão de média de lambda/*snapshot* por média de AIC/*snapshot* ser 0.04. Ou seja, embora o projeto tenha expressões lambda, estas são consideradas muito poucas quando inseridas no contexto geral deste projeto, que é considerado um projeto grande.

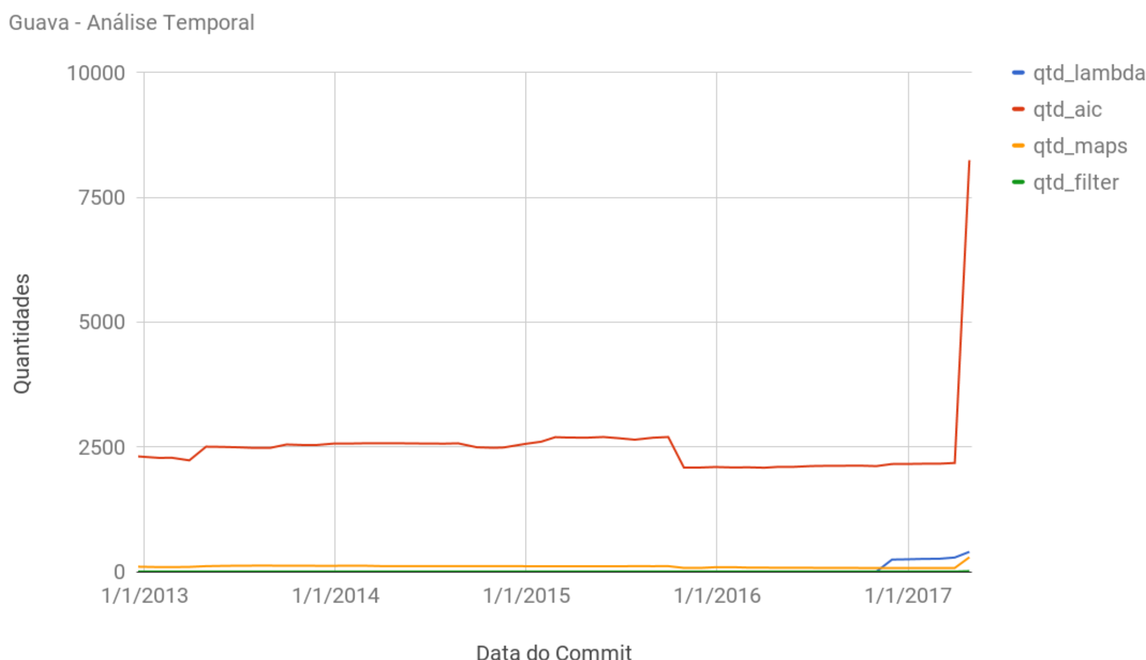


Figura 5.6: Gráficos de análise temporal do projeto *Guava*

## 5.2 Refatoração de Código

Com o objetivo de caracterizar melhor como as expressões lambda estão sendo utilizadas, é interessante identificar se estas estão sendo introduzidas através de refatoração de código ou em código novo.

Os projetos classificados como pertencentes ao Grupo 2 possuem em comum o fato de expressões lambda aumentarem a medida que AICs diminuem, ultrapassando-as, e essa característica em comum pode indicar que nestes projetos possivelmente houve refatoração de código, visto que AIC sendo substituído por expressões lambda, dentro das condições determinadas, é a maneira mais comum de se identificar refatoração de código para introdução de expressões lambda [12]. Os projetos que compõem o grupo 2 representam 44% de todos os projetos estudados que possuem expressões lambda, o que pode ser um indicativo, caso a hipótese de refatoração de código seja positiva, de que muitos projetos que estão adotando o Java 8 estão se preocupando, de alguma forma, com refatoração se código. Fazem parte desse grupo os seguintes projetos: *elasticsearch*, *java-design-patterns*, *PocketHub*, *presto*, *RxJava*-Android-*Samples*, *spark*, *vert.x*, *zipkin*, *java8-tutorial*, *dropwizard* e *material-dialogs*.

Outra maneira de verificar possíveis indicadores de refatoração de código foi através da análise dos tamanhos (em quantidade de linhas de código) dos métodos com expressões lambda no mês em que se introduziu expressões lambda e um mês imediatamente antes deste. Essa análise mostra que dos 25 projetos com expressões lambda, 12 (48%) projetos tiveram ao menos um método em que houve uma diminuição no tamanho, o que pode ser considerado um indicativo de refatoração de código. A Figura 5.7 mostra todos os projetos em relação à quantidade total de métodos que possuem lambda no primeiro mês de introdução do mesmo em vermelho, comparado com a quantidade desses métodos que tiveram uma diminuição no tamanho quando comparados com o código do mês anterior, em azul.

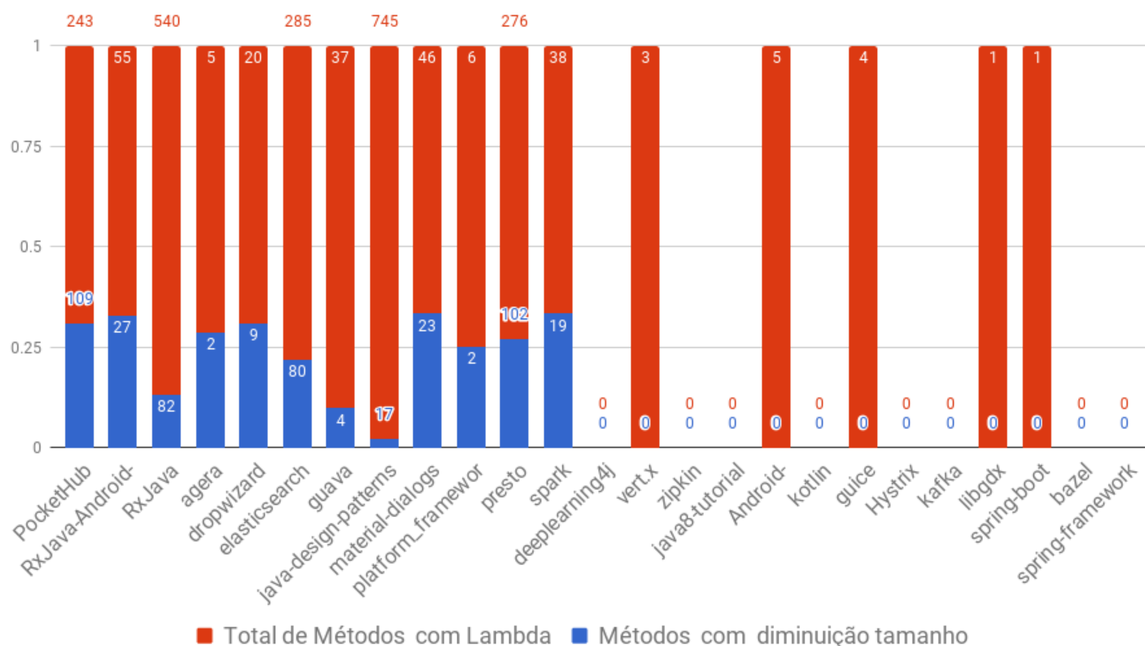


Figura 5.7: Comparação entre todos os projetos com expressão lambda sobre refatoração de código no mês de introdução da característica.

Ainda observando a figura 5.7, com relação aos projetos que não tiveram nenhum método com diminuição de tamanho (barra azul), percebe-se que todos os 13 projetos sequer tiveram uma quantidade significativa de métodos que possuem expressões lambda (com o maior tendo apenas 5 métodos) no mês de introdução da característica no projeto e, ao mesmo tempo, já existiam no mês em que não existia lambda no projeto. Essa é uma observação importante porque projetos como *vert.x*, sendo este o que possui a maior média de lambda/*snapshot* de todos os projetos e ao mesmo tempo tendo apenas três métodos que continham expressões lambda no mês da introdução e que existiam no mês

anterior, permitem inferir que todas as outras expressões lambda existentes no projeto para este determinado mês de introdução encontram-se em código novo introduzido.

É importante ressaltar que somente essa análise não é exclusivamente suficiente para determinar se estes projetos estão realmente introduzindo lambda em código novo ou realizando refatoração, uma vez que, como a comparação aconteceu apenas no mês de introdução com o mês imediatamente anterior, não reflete todo o ciclo de vida do projeto. Por exemplo, algumas equipes podem escolher refatorar o código depois de um tempo de adaptação após a transição para a nova versão da linguagem. Assim, para obter uma melhor análise sobre refatoração de código ou não, feita nestes projetos, além de outras análises referentes aos padrões tipicamente adotados para implementação de lambda nestes projetos, serão apresentados a seguir, os estudos de caso dos cinco projetos previamente selecionados.

Outro ponto importante é o fato de que, apesar das informações quanto às quantidades de *filter* e *map* terem sido incluídas inicialmente no trabalho para observar possíveis padrões de refatoração de código, percebeu-se que estas não são utilizadas de forma considerável pelos projetos estudados e para fins de análises generalizadas, não apresentaram tanta influência na utilização de expressões lambda quanto a quantidade de AICs no projeto.

## 5.3 Estudos de Caso

Dentre os projetos analisados, verificou-se que os projetos de pequeno e médio porte optaram por refatorar de uma vez seus códigos para incluir expressões lambda em situações em que se utilizava AICs anteriormente ou, em alguns poucos casos, de utilização de `Collection`. Os projetos de grande porte não apresentaram nenhuma refatoração de código ou uma refatoração parcial gradativa, provavelmente devido à grande quantidade de arquivos e código que possuem, o que pode tornar o trabalho de refatoração extenso e cansativo.

O projeto *vert.x*, que pertence ao grupo 2, dos projetos em que a quantidade de expressões lambda aumentaram e as de AICs diminuíram com lambda ultrapassando AICs, executou inicialmente *commits* em maio de 2014 com adição de código novo utilizando expressões lambda e, após um mês, realizou um *commit* maior com a refatoração de todas as AICs para expressões lambda. Após a refatoração, o projeto se preocupou em manter a utilização de expressões lambda ao mesmo tempo que o uso de AICs se manteve apenas aos casos de classes com interfaces não funcionais. A Figura 5.8 apresenta uma parte do *commit* de refatoração de código que mostra em vermelho a parte eliminada do código (AIC) e em verde o código novo, representando a substituição por uma expressão lambda.

**Java8-ify code**

master (#1) 3.4.2 ... 3.0.0-dev_preview1

**purplefox** committed on 2 Jun 2014 — 1 parent d36bdaa — commit 93f95e9c77419f442a42fe711b6eeaef88192fd3

Browse files

Showing **22 changed files** with **407 additions** and **618 deletions**. Unified Split

9 ...x-core/src/main/java/org/vertx/java/core/datagram/impl/DatagramChannelFutureListener.java View

```
@@ -51,20 +51,15 @@ public void operationComplete(final ChannelFuture future) throws Exception {
51 51           context.reportException(t);
52 52         }
53 53       } else {
54    -      context.execute(new Runnable() {
55    -        public void run() {
56    -          notifyHandler(future);
57    -        }
58    -      });
   54 +      context.execute(() -> notifyHandler(future));
59 55       }
60    -
61 56     }
62 57
63 58     private void notifyHandler(ChannelFuture future) {
64 59       if (future.isSuccess()) {
65 60         handler.handle(new DefaultFutureResult<>(result));
66 61       } else {
67    -      handler.handle(new DefaultFutureResult<T>(future.cause()));
   62 +      handler.handle(new DefaultFutureResult<>(future.cause()));
68 63       }
69 64     }
70 65   }
```

Figura 5.8: *Commit* de refatoração de código do projeto Vert.x [3]

Alguns projetos ainda não se preocuparam em refatorar o código para o uso de expressões lambda até a data deste trabalho, que é o caso do projeto *guava*, que apresentou um *commit* em 03/11/2016 com a atualização do projeto para Java 8. Mas, a partir da atualização, se preocupou em utilizar as expressões lambda apenas na implementação de código novo. O cenário em que a refatoração não é prioridade ainda é a realidade de vários projetos de grande porte visto que os dados de projetos sem nenhuma expressão lambda são o maior número neste trabalho. Porém, projetos que já adotaram a utilização do Java 8 tendem a mudar isso com o passar do tempo. A Figura 5.9 apresenta em verde uma adição de código novo com expressão lambda sem a eliminação de algum código anterior a esse, que estaria em vermelho.

```
63  68       }
64  69
    70   +   @Beta
    71   +   public static <T, K, V> Collector<T, ?, ImmutableSortedMap<K, V>> toImmutableSortedMap(
    72   +       Comparator<? super K> comparator,
    73   +       Function<? super T, ? extends K> keyFunction,
    74   +       Function<? super T, ? extends V> valueFunction) {
    75   +     return CollectCollectors.toImmutableSortedMap(comparator, keyFunction, valueFunction);
    76   +   }
    77   +
    78   +   @Beta
    79   +   public static <T, K, V> Collector<T, ?, ImmutableSortedMap<K, V>> toImmutableSortedMap(
    80   +       Comparator<? super K> comparator,
    81   +       Function<? super T, ? extends K> keyFunction,
    82   +       Function<? super T, ? extends V> valueFunction,
    83   +       BinaryOperator<V> mergeFunction) {
    84   +     checkNotNull(comparator);
    85   +     checkNotNull(keyFunction);
    86   +     checkNotNull(valueFunction);
    87   +     checkNotNull(mergeFunction);
    88   +     return Collectors.collectingAndThen(
    89   +         Collectors.toMap(
    90   +             keyFunction, valueFunction, mergeFunction, () -> new TreeMap<K, V>(comparator)),
    91   +         ImmutableSortedMap::copyOfSorted);
    92   +   }
    93   +
65  94       // Casting to any type is safe because the set will never hold any elements.
66  95       @SuppressWarnings("unchecked")
```

Figura 5.9: Exemplo de um arquivo do *commit* de adição de código novo em Java 8 do projeto *Guava* [4]

Na categoria de projetos em que os AICs e expressões lambda aumentam proporcionalmente (grupo 1), o projeto *agera* realizou um *commit* em 19/05/2016 com a adição da biblioteca `retrolambda` como dependência do projeto. Essa biblioteca utilizada principalmente por projetos Android, permite executar código Java 8 contendo expressões lambda em Java 5, 6 e 7 [48]. O *commit* também apresenta refatoração dos AICs advindos de interfaces funcionais para expressões lambda. Apesar de o projeto continuar a utilizar as expressões lambda em seus *commits* posteriores, o número de AICs também continuou a aumentar, visualizando os *commits* posteriores notou-se que esses AICs são relativos a interfaces não funcionais, ou seja, que possuem mais de um método. O projeto Android-*CleanArchitecture* é similar ao *agera*, pois também optou pela instalação da biblioteca *retrolambda* para uso de expressões lambda em seus códigos, e, como também é um projeto de pequeno porte, houve uma pequena refatoração dos AICs. A Figura 5.10 mostra a parte do *commit* de refatoração de código em que o desenvolvedor adiciona a biblioteca *retrolambda* como dependência.

Collapsed testapp with retrolambda
master (#1) v1.4.0-alpha4 ... v1.1.0-beta1
Browse files

ernstsson committed on 19 May 2016 1 parent 42c7440 commit 5e518b7b05f9e7a34a314945f68deff7b2c3e334

Showing 19 changed files with 111 additions and 428 deletions. Unified Split

7 agera/build.gradle View

```
@@ -18,6 +18,13 @@ apply plugin: 'com.github.dcendents.android-maven'
18 18  apply plugin: 'com.novoda.bintray-release'
19 19  apply plugin: 'jacoco'
20 20
   21 +android {
   22 +    compileOptions {
   23 +        sourceCompatibility gradle.javaVersion
   24 +        targetCompatibility gradle.javaVersion
   25 +    }
   26 +}
   27 +
21 28  publish {
22 29      userOrg = gradle.bintrayUser
23 30      uploadName = gradle.bintrayName
```

Figura 5.10: *Commit* de adição da biblioteca *Retrolambda* no projeto *Agera* [5]

Outro caso interessante é o do projeto *RxJava*, que em 2015 optou pela refatoração do código para a utilização de expressões lambda, quase eliminando por completo o uso de AICs, mas que no ano seguinte reverteu os *commits* de forma que eliminasse completamente o uso de expressões lambda para manter a retrocompatibilidade com o Java 6. A Figura 5.11 demonstra uma parte do *commit* de reversão do código para Java 6 em que pode ser visto em vermelho o código anterior com expressões lambda e o verde com o novo código utilizando AIC novamente.

```
@@ -31,38 +33,50 @@ public OperatorDistinct(Function<? super T, K> keySelector, Supplier<? extends P
31  33          this.keySelector = keySelector;
32  34      }
33  35
34     -    public static <T, K> OperatorDistinct<T, K> withCollection(Function<? super T, K> keySelector, Supplier<? extends C
35     -        Supplier<? extends Predicate<? super K>> p = () -> {
36     -            Collection<? super K> coll = collectionSupplier.get();
37     -
38     -            return t -> {
39     -                if (t == null) {
40     -                    coll.clear();
41     -                    return true;
42     -                }
43     -                return coll.add(t);
44     -            };
    36 +    public static <T, K> OperatorDistinct<T, K> withCollection(Function<? super T, K> keySelector, final Supplier<? ext
    37 +        Supplier<? extends Predicate<? super K>> p = new Supplier<Predicate<K>>() {
    38 +            @Override
    39 +            public Predicate<K> get() {
    40 +                final Collection<? super K> coll = collectionSupplier.get();
    41 +
    42 +                return new Predicate<K>() {
    43 +                    @Override
    44 +                    public boolean test(K t) {
    45 +                        if (t == null) {
    46 +                            coll.clear();
    47 +                            return true;
    48 +                        }
    49 +                        return coll.add(t);
    50 +                    }
    51 +                };
    52 +            }
45  53          };
```

Figura 5.11: *Commit* de remoção de Lambdas e volta para o uso de AICs no projeto *RXJava* [6]

Essa preocupação com retrocompatibilidade de código pode ser um fator crucial na decisão das equipes de desenvolvimento na hora de decidir como fazer o software co-evoluir com a linguagem, sem perder a compatibilidade com versões anteriores.

# Capítulo 6

# Considerações Finais

Este estudo permitiu entender melhor como projetos estão realizando a migração para o Java 8, em especial, utilizando uma nova característica introduzida: expressão lambda. Apesar de a maioria dos projetos populares considerados não possuírem nenhuma ocorrência de expressões lambda, os projetos que tinham expressões lambda foram suficientes para realizar um primeiro estudo de caracterização no uso desta característica e, com os projetos efetivamente utilizados para estudo, foi possível agrupá-los quanto a maneira em que as expressões lambda estavam sendo utilizadas quando comparadas com AICs em 5 grandes grupos: quando AIC e lambda aumentam proporcionalmente, quando AIC diminui e lambda aumenta ultrapassando AIC, quando AIC diminui mas continua superior ao lambda crescente, expressões lambda que foram introduzidas e retiradas completamente do código e expressões lambda que não existem em quantidades expressivas.

Foi possível, então, realizar um estudo aprofundado, levando em consideração as diferentes maneiras com que as expressões lambda foram adotadas nos projetos através da escolha de um projeto que representasse cada grupo para análise. Dentre os cinco projetos estudados, foi possível observar alguns padrões tipicamente adotados por desenvolvedores para implementar expressões lambda em seus projetos, como a utilização da bibliotecas (*retrolambda*) que permitissem uma flexibilização na utilização de expressões lambda, ou como lambda é primariamente utilizada para substituir determinados AICs ou operações em `Collection`.

Além disso, foi possível observar como a refatoração de código acontece em cada um desses projetos. Observou-se como projetos menores tendem a refatorar código mais facilmente, ao passo que projetos maiores e mais complexos fazem menos refatoração ou demoram mais entre o período em que houve a primeira migração de código para a nova versão da linguagem até o período em que realmente decidem refatorar código. Além da complexidade do projeto influenciar na decisão de migrar o projeto para uma versão mais recente da linguagem, a preocupação com retrocompatibilidade com versões anteriores da

linguagem também podem ser uma barreira para que projetos comecem a evoluir o código juntamente com a evolução da linguagem.

Ainda assim, foi possível observar que a preocupação com a co-evolução do software/-linguagem existe entre os desenvolvedores de projetos, mas a realização da migração ainda acontece lentamente, devido à diversos fatores que precisam ser levados em consideração para a manutenção do projeto.

Para trabalhos e contribuições futuras, seria interessante desenvolver uma ferramenta que pudesse analisar diretamente no código-fonte as oportunidades de aplicação de expressões lambda, dessa forma, poderiam ser analisadas as porcentagens de conversão de código para Java 8 e obter melhores conclusões sobre a importância que os projetos estão dando para a evolução de seus códigos. Outro fator interessante seria aplicar mais *Visitors* à ferramenta *static-analysis* e fazer este mesmo estudo aplicado à outras características da linguagem.

# Referências


[1] Kagdi, Huzefa, Michael L. Collard e Jonathan I. Maletic: *A survey and taxonomy of approaches for mining software repositories in the context of software evolution*, 2007. ix, 3, 8, 9, 10, 11

[2] DBeaver: *Dbeaver - features*. `http://dbeaver.jkiss.org/docs/features/`. ix, 29

[3] Eclipse: *Vert.x commit: Java8-ify code*. `https://github.com/eclipse/vert.x/commit/93f95e9c77419f442a42fe711b6eeaef88192fd3`. ix, 42

[4] Google: *Guava commit: Release java 8 changes to guava*. `https://github.com/google/guava/commit/73e382fa877f80994817a136b0adcc4365ccd904`. ix, 43

[5] Google: *Agera commit: Collapsed testapp with retrolambda*. `https://github.com/google/agera/commit/5e518b7b05f9e7a34a314945f68deff7b2c3e334`. ix, 44

[6] ReactiveX: *Rxjava commit: 2.x: full jdk 6 compatible backport + including bugfixes up to today*. `https://github.com/ReactiveX/RxJava/commit/000a174d87a901135f9665d3b11f3627fd2dc4ed`. ix, 45

[7] Godfrey, M. W. e D. M. German: *The past, present, and future of software evolution*. Em *2008 Frontiers of Maintenance*, páginas 129–138, Sept 2008. 1, 6

[8] Favre, J M: *Languages evolve too! changing the software time scale*. Em *Principles of Software Evolution, Eighth International Workshop on*, páginas 33–42. IEEE, 2005. 1, 5, 7

[9] Overbey, Jeffrey L e Ralph E Johnson: *Regrowing a language: refactoring tools allow programming languages to evolve*. Em *ACM SIGPLAN Notices*, volume 44, páginas 493–502. ACM, 2009. 1, 5, 7, 8, 12

[10] Grechanik, Mark, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie e Carlo Ghezzi: *An empirical investigation into a large-scale java open source code repository*. Em *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, páginas 11:1–11:10, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0039-1. `http://doi.acm.org/10.1145/1852786.1852801`. 1

[11] TIOBE: *Tiobe*. `https://www.tiobe.com/tiobe-index/`. 1, 14

[12] Gyori, Alex, Lyle Franklin, Danny Dig e Jan Lahoda: *Crossing the gap from imperative to functional programming through refactoring.* Em *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, páginas 543–553, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2237-9. `http://doi.acm.org/10.1145/2491411.2491461`. 2, 11, 24, 39

[13] OpenJDK: *State of the lambda.* `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html`. 2

[14] Han, Jiawei, Micheline Kamber e Jian Pei: *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edição, 2011, ISBN 0123814790, 9780123814791. 3

[15] Fowler, Martin e Kent Beck: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. 5, 7

[16] Lehman, M M., J F. Ramil, P D. Wernick, D E. Perry e W M. Turski: *Metrics and laws of software evolution - the nineties view.* Em *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, páginas 20–, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-8093-8. `http://dl.acm.org/citation.cfm?id=823454.823901`. 6

[17] Hassan, Ahmed E. e Tao Xie: *Software intelligence: The future of mining software engineering data.* Em *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, páginas 161–166, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0427-6. `http://doi.acm.org/10.1145/1882362.1882397`. 8, 9

[18] Hassan, Ahmed E: *The road ahead for mining software repositories.* Em *Frontiers of Software Maintenance, 2008. FoSM 2008.*, páginas 48–57. IEEE, 2008. 8

[19] Berry, Michael J. e Gordon Linoff: *Data Mining Techniques: For Marketing, Sales, and Customer Support.* John Wiley & Sons, Inc., New York, NY, USA, 1997, ISBN 0471179809. 10

[20] Parnin, Chris, Christian Bird e Emerson Murphy-Hill: *Java generics adoption: how new features are introduced, championed, or ignored.* Em *Proceedings of the 8th Working Conference on Mining Software Repositories*, páginas 3–12. ACM, 2011. 12

[21] Oracle: *The java language specification.* `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`. 13

[22] Oracle: *The history of java technology.* `http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html`. 13

[23] Murphy, Kieron: *So why did they decide to call it java?* `http://www.javaworld.com/article/2077265/core-java/so-why-did-they-decide-to-call-it-java-.html`. 13

[24] McGraw, Tata: *Object-oriented Programming with Java: Essentials and Applications.* Hill Education. 13

[25] Oracle: *Differences between java ee and java se.* http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html. 13

[26] Lindsey, Clark S.: *The History of Java.* Cambridge. 14

[27] Oracle: *The java language environment.* http://www.oracle.com/technetwork/java/intro-141325.html. 14

[28] Oracle: *Jdk 8.* http://openjdk.java.net/projects/jdk8/. 15, 24

[29] Oracle: *Enhancements in java se 8.* http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8. 15, 16, 17

[30] Oracle: *Lambda expressions.* http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. 16, 18, 21

[31] Oracle: *Method references.* http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html. 16

[32] Oracle: *Default methods.* http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html. 17

[33] Oracle: *Repeating annotations.* http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html. 17

[34] Oracle: *Type annotations.* http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. 17

[35] Oracle: *Anonymous inner classes.* https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html. 19

[36] Kothari, C.R.: *Research Methodology: Methods and Techniques.* New Age International (P) Limited, 2004, ISBN 9788122415223. https://books.google.com.br/books?id=8c6gkbKi-F4C. 22

[37] Nakakoji, Kumiyo, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida e Yunwen Ye: *Evolution patterns of open-source software systems and communities.* Em *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, páginas 76–85, New York, NY, USA, 2002. ACM Press, ISBN 1581135459. http://dx.doi.org/10.1145/512035.512055. 23

[38] Rahman, Foyzur, Christian Bird e Premkumar Devanbu: *Clones: What is that smell?* Empirical Software Engineering, 17(4-5):503–530, 2012. 24

[39] Chacon, Scott: *Pro Git.* Apress, Berkely, CA, USA, 1st edição, 2009, ISBN 1430218339, 9781430218333. 25

[40] Cavalcanti, Thiago Gomes e Vinícius Correa de Almeida: *Caracterização do uso de construções da linguagem java em projetos open-source.* Publicação online, 2016. http://bdm.unb.br/handle/10483/15733. 27

[41] Parr, Terence: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edição, 2009, ISBN 193435645X, 9781934356456. 27

[42] Janssen, Thorben: *5 reasons to use jpa / hibernate*. Publicação online. `https://www.sitepoint.com/5-reasons-to-use-jpa-hibernate/`. 27

[43] Franklin, Lyle, Alex Gyori, Jan Lahoda e Danny Dig: *Lambdaficator: From imperative to functional programming through automated refactoring*. Em *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, páginas 1287–1290, Piscataway, NJ, USA, 2013. IEEE Press, ISBN 978-1-4673-3076-3. `http://dl.acm.org/citation.cfm?id=2486788.2486986`. 29

[44] Google: *Agera*. `https://github.com/google/agera/wiki`. 34

[45] Eclipse: *Eclipse vert.x*. `http://vertx.io/`. 35

[46] Cejas, Fernando: *Android cleanarchitecture*. `https://github.com/android10/Android-CleanArchitecture`. 36

[47] ReactiveX: *Rxjava*. `https://github.com/ReactiveX/RxJava`. 37

[48] Luontola, Esko: *Retrolambda*. `https://github.com/orfjackal/retrolambda`. 43

# Anexo I

# Projetos utilizados

fastjson

platform_frameworks_base

netty

material-dialogs

kotlin

okhttp

tinker

AndroidUtilCode

RxJava

spring-boot

java-design-patterns

elasticsearch

ExoPlayer

kafka

vert.x

realm-java

guava

zipkin

bazel

stetho

Signal-Android

glide

agera

fresco

plaid

presto

libgdx

spark

disruptor
lottie-android
flexbox-layout
PocketHub
zxing
spring-framework
deeplearning4j
dropwizard
interviews
BaseRecyclerViewAdapterHelper
uCrop
DanmakuFlameMaster
lottie-react-native
android-UniversalMusicPlayer
AndroidInterview-Q-A
greenDAO
retrofit
MaterialDrawer
BottomBar
druid
MPAndroidChart
Hystrix
guice
recyclerview-animators
dubbo
androidannotations
RxJava-Android-Samples
PhotoView
clojure
CircleImageView
AndroidSwipeLayout
jedis
MaterialViewPager
butterknife
junit4
RxBinding
leakcanary
SimianArmy
picasso
UltimateRecyclerView

AndroidViewAnimations

AppIntro

FizzBuzzEnterpriseEdition

ion

cheesesquare

physical-web

RxAndroid

Android-CleanArchitecture

Calligraphy

Android-Bootstrap

iosched

Android_Data

MaterialDesignLibrary

ListViewAnimations

EventBus

java8-tutorial

AndroidSlidingUpPanel

dagger

okhttputils

logger

HomeMirror

Material-Animations

android-Ultra-Pull-To-Refresh

android-async-http

Android-ObservableScrollView

Android-Universal-Image-Loader

ActionBarSherlock

SlidingMenu

storm

PagerSlidingTabStrip

Android-PullToRefresh