

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Bernardo Petry Prates

**REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS
JAVA**

Santa Maria, RS
2018

Bernardo Petry Prates

REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

Orientador: Prof. Dr. Eduardo Kessler Piveta

Santa Maria, RS

2018

Petry Prates, Bernardo

Refatorações para a evolução de programas Java / por Bernardo Petry Prates. – 2018.

149 f.: il.; 30 cm.

Orientador: Eduardo Kessler Piveta

Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Pós-Graduação em Ciência da Computação, RS, 2018.

1. Refatoração. 2. Evolução de programas. 3. Java. 4. Funções de detecção. I. Kessler Piveta, Eduardo. II. Refatorações para a evolução de programas Java.

© 2018

Todos os direitos autorais reservados a Bernardo Petry Prates. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.

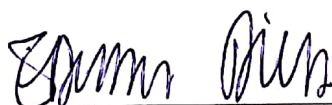
E-mail: bprates@inf.ufsm.br

Bernardo Petry Prates

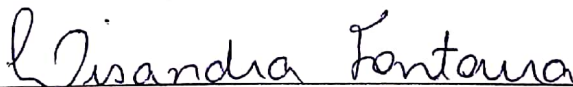
REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS JAVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

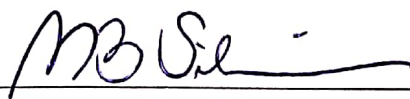
Aprovada em 20 de Dezembro de 2018:



Eduardo Kessler Piveta, Dr. (UFSM)
(Presidente/Orientador)



Lisandra Manzoni Fontoura, Dr. (UFSM)



Maicon Bernardino da Silveira, Dr. (Unipampa)

Santa Maria, RS

2018

DEDICATÓRIA

Dedico este trabalho à minha amada avó paterna Bartira Dalva Lima Prates.

AGRADECIMENTOS

Sendo este trabalho a representação do meu Mestrado em Ciência da Computação, agradeço:

- *aos meus avós Marley Prates e Bartira Prates por me darem todo o suporte para a realização deste trabalho, fornecendo muito mais que bens materiais, mas todas as ferramentas necessárias para me sustentar durante todo esse processo. Aos meus pais Edilene Petry e Darlan Prates por me apoiarem emocionalmente e não medirem esforços em todas as vezes em que precisei;*
- *ao meu orientador e professor Eduardo Piveta, por ser um guia nesse processo, por estar sempre disponível para ajudar e contribuir com o trabalho. Pelas sugestões, ideias, comentários e por toda confiança depositada em mim;*
- *a minha namorada Bruna Balbinot que me apoiou incondicionalmente, foi meu alento e minha principal motivadora. Quando eu pensava em desistir ela estava pronta para me levar nas costas se precisasse. Seu apoio certamente será lembrado por toda minha vida;*
- *aos professores da UFSM, em especial à Deise, à Juliana e ao Alencar pelas ideias e contribuições;*
- *aos meus parceiros de sempre: Douglas Haubert, Kadico Puiati, Tobias Senger e Leonardo Nicorena e*
- *às minhas irmãs Bianca Prates e Brenda Pereira.*

*“Comece fazendo o que é necessário,
depois o que é possível, em breve estarás
fazendo o impossível...”*

(SÃO FRANCISCO DE ASSIS)

RESUMO

REFATORAÇÕES PARA A EVOLUÇÃO DE PROGRAMAS JAVA

AUTOR: BERNARDO PETRY PRATES
ORIENTADOR: EDUARDO KESSLER PIVETA

Com a evolução de uma linguagem, surge a possibilidade de atualizar construções antigas e se beneficiar de novas funcionalidades. Porém, à medida que os sistemas são modificados, adaptados e atualizados, seu código e outros artefatos envolvidos podem se tornar mais complexos, propensos a erros e se afastarem de sua concepção original, podendo diminuir sua qualidade. Então, de forma a guiar desenvolvedores para evolução de programas, este trabalho propõe um catálogo com trinta refatorações para a evolução de programas em Java e a definição de quinze funções de detecção para a busca de oportunidades para aplicar as refatorações propostas. Após isso, com o objetivo de avaliar e validar as refatorações e funções apresentadas, foi desenvolvida uma *API* para a realização de um estudo de caso utilizando cinco projetos de código aberto.

Palavras-chave: Refatoração. Evolução de programas. Java. Funções de detecção.

ABSTRACT

REFACTORINGS FOR THE EVOLUTION OF JAVA PROGRAMS

AUTHOR: BERNARDO PETRY PRATES
ADVISOR: EDUARDO KESSLER PIVETA

With the evolution of a language, the possibility of updating old constructions and benefiting from new features arises. However, as systems are modified, adapted and updated, their code and other artifacts may become more complex, prone to errors and departed from their original design, thus decreasing their quality. We present a catalog of thirty refactorings for the evolution of Java programs and the definition of detection functions for the search of opportunities to apply those refactorings in order to guide developers towards program evolution. Furthermore, we developed an *API* to perform a case study using five open source projects in order to evaluate and to validate the refactorings and functions presented.

Keywords: Refactoring. Programs Evolution. Java. Detection Functions.

LISTA DE FIGURAS

Figura 1 –	Diagrama temporal de execução de tarefas do programa	27
Figura 2 –	Arquitetura do <i>AOPJungle</i> (FAVERI et al., 2013)	32
Figura 3 –	Diagrama de classes adicionadas no <i>AOPJungle</i>	72
Figura 4 –	Diagrama de classes da modelagem das oportunidades de refatoração	73
Figura 5 –	Exemplo de visualização das funções de detecção	75
Figura 6 –	Quantidade de oportunidades de refatoração para <i>Agrupar blocos catch</i>	77
Figura 7 –	Quantidade de oportunidades de refatoração para <i>Mover recurso para o try-with-resources</i>	81
Figura 8 –	Quantidade de oportunidades de refatoração para <i>Adicionar método map na interação com coleções</i>	84
Figura 9 –	Quantidade de oportunidades de refatoração para <i>Converter para uma operação aritmética lambda</i>	89
Figura 10 –	Quantidade de oportunidades para <i>Converter if para ifPresent</i>	92
Figura 11 –	Quantidade de oportunidades de refatoração para <i>Converter iterator.remove para removeIf</i>	95
Figura 12 –	Quantidade de oportunidades de refatoração para <i>Converter iterator.set para replaceAll</i>	97
Figura 13 –	Quantidade de oportunidades de refatoração para <i>Concatenar uma coleção com join</i>	100
Figura 14 –	Quantidade de oportunidades de refatoração para <i>Converter interface Iterator para Spliterator</i>	103
Figura 15 –	Quantidade de oportunidades para <i>Adicionar interface Predicate</i>	106
Figura 16 –	Quantidade de oportunidades de refatoração para <i>Converter para contagem lambda</i>	112
Figura 17 –	Quantidade de oportunidades de refatoração <i>Adicionar método of</i>	114

LISTA DE TABELAS

Tabela 1 –	Refatorações propostas por TEIXEIRA JÚNIOR et al.....	20
Tabela 2 –	Tabela comparativa com trabalhos relacionados.	35
Tabela 3 –	Projetos usados no estudo de caso.	70

LISTA DE ANEXOS

ANEXO A – CATÁLOGO DE REFATORAÇÕES INVERSAS	127
---	-----

LISTA DE ABREVIATURAS E SIGLAS

AJDT	<i>AspectJ Development Tools</i>
API	<i>Application Programming Interface</i>
AQL	<i>Aspect Query Language</i>
AST	<i>Abstract Syntax Tree</i>
JEP	<i>JDK Enhancement Proposal</i>
JVM	<i>Java Virtual Machine</i>
SWT	<i>Standard Widget Toolkit</i>
UFSM	<i>Universidade Federal de Santa Maria</i>

LISTA DE SÍMBOLOS

\forall	Operador <i>Para todo</i>
\in	Operador <i>Pertencente</i>
\exists	Operador <i>Existe pelo menos um</i>
	Operador <i>Tal que</i>
\vee	<i>Ou lógico</i>
\wedge	<i>E lógico</i>
\neg	<i>Negação lógica</i>
\geq	Operador <i>Maior do que ou igual</i>
\leq	Operador <i>Menor do que ou igual</i>
$>$	Operador <i>Maior que</i>
$<$	Operador <i>Menor que</i>

SUMÁRIO

1	INTRODUÇÃO	15
2	REVISÃO BIBLIOGRÁFICA	18
2.1	REFATORAÇÃO	18
2.2	EVOLUÇÃO DA LINGUAGEM JAVA	19
2.2.1	Java 7	20
2.2.2	Java 8	22
2.2.2.1	<i>Coleções e Strings</i>	23
2.2.2.2	<i>A API de Streams</i>	24
2.2.2.3	<i>Classe Optional</i>	28
2.2.2.4	<i>Splititerator</i>	29
2.2.2.5	<i>Predicate</i>	30
2.2.3	Java 9 e 10	30
2.3	FRAMEWORK AOPJUNGLE	31
2.4	TRABALHOS RELACIONADOS	33
2.5	CONSIDERAÇÕES FINAIS	35
3	UM CATÁLOGO DE REFATORAÇÕES PARA EVOLUÇÃO DE PROGRAMAS JAVA	36
3.1	CATÁLOGO	36
3.2	FUNÇÕES DE DETECÇÃO	60
3.3	CONSIDERAÇÕES FINAIS	68
4	ESTUDO DE CASO	69
4.1	PROJETOS SELECIONADOS	69
4.2	UMA API PARA DETECTAR OPORTUNIDADES DE REFATORAÇÃO	70
4.3	IMPLEMENTAÇÃO E EXECUÇÃO DAS FUNÇÕES DE DETECÇÃO	75
4.3.1	Agrupar blocos <i>catch</i>	76
4.3.2	Mover recurso para o <i>try-with-resources</i>	79
4.3.3	Adicionar método <i>map</i> na interação com coleções	83
4.3.4	Agrupar com <i>groupBy</i>	86
4.3.5	Converter para uma operação aritmética <i>lambda</i>	88
4.3.6	Converter <i>if</i> para <i>ifPresent</i>	91
4.3.7	Converter <i>iterator.remove</i> para <i>removeIf</i>	93
4.3.8	Converter <i>iterator.set</i> para <i>replaceAll</i>	96
4.3.9	Concatenar uma coleção com <i>join</i>	98
4.3.10	Converter interface <i>Iterator</i> para <i>Splititerator</i>	102
4.3.11	Adicionar interface <i>Predicate</i>	104
4.3.12	Converter para média <i>lambda</i>	108
4.3.13	Converter para contagem <i>lambda</i>	110
4.3.14	Adicionar método <i>of</i>	113
4.3.15	Converter para <i>underscore</i>	115
4.4	DISCUSSÃO	117
4.5	CONSIDERAÇÕES FINAIS	118
5	CONCLUSÃO	119
5.1	APRIMORAMENTOS DO PROJETO E TRABALHOS FUTUROS	120
	REFERÊNCIAS	123
	ANEXOS	126

1 INTRODUÇÃO

A cada nova versão de uma linguagem de programação podem surgir novos mecanismos de abstração e composição. Com isso, pode ser necessário atualizar os sistemas existentes, pois alguns dos recursos da linguagem são modificados, depreciados, removidos ou mesmo considerados inadequados. A linguagem de programação Java, por exemplo, em sua versão 8, adicionou novos recursos para auxiliar no desenvolvimento de software, incluindo o suporte a expressões lambda, novos métodos advindos do paradigma funcional e uma *API* para tratamento de coleções (*API Stream*).

Com a evolução de uma linguagem, surge a possibilidade de atualizar construções antigas e se beneficiar de novas funcionalidades. Porém, a medida que os sistemas são modificados, adaptados e atualizados, seu código e outros artefatos envolvidos podem se tornar mais complexos, propensos a erros e se afastarem de sua concepção original, podendo diminuir sua qualidade. Então, para usar novas funcionalidades, sem diminuir a qualidade interna do sistema, muitos desenvolvedores utilizam técnicas para facilitar a evolução de programas.

Um dos recursos utilizados para o aprimoramento de sistemas de software é a refatoração, que consiste no processo de modificar um sistema para melhorar sua qualidade interna preservando seu comportamento externamente observável (GRISWOLD, 1991; OPDYKE, 1992; MENS; TOURWÉ, 2004). Nesse sentido, alguns trabalhos identificam oportunidades de refatoração para guiarem o desenvolvedor, sugerindo quais estruturas podem ser modificadas (GAMMA, 1995; FOWLER; BECK, 1999; PIVETA et al., 2006; KERIEVSKY, 2009; PIVETA, 2009; MACIA BERTRAN; GARCIA; STAA, 2011; TEIXEIRA JÚNIOR et al., 2014). Então, de forma a sugerir melhorias, esses trabalhos apresentam um catálogo com problemas, estruturas antigas ou descontinuadas encontradas em códigos de programas, sugerindo refatorações para essas estruturas.

A identificação de refatorações a serem aplicadas, muitas vezes, envolve a inspeção manual do código-fonte, o que rapidamente pode se tornar inviável, à medida que o tamanho de um sistema aumenta. Então, com o tempo, algumas abordagens semi-automáticas foram desenvolvidas. Elas sugerem melhorias em código por meio da utilização de métricas e heurísticas para identificar lugares possíveis de serem refatorados (SIMON; STEINBRUCKNER; LEWERENTZ, 2001; MARINESCU; RATIU, 2004; MUNRO, 2005; PIVETA, 2009). A criação de heurísticas, ou funções de detecção, para buscar oportunidades de refatoração, pode ajudar o de-

envolvedor a identificar partes de código que podem ser atualizadas, obtendo vantagens como paralelismo e outras funcionalidades que são adicionadas em novas versões das linguagens de programação.

Atualmente, alguns trabalhos são propostos para evolução de código Java, principalmente na transformação de iterações externas para internas, fazendo a conversão de laços de repetição para métodos adicionados na *Stream API* em Java 8 (GYORI et al., 2013; DAVID; KESSELI; KROENING, 2017; RAHAD; CAO; CHEON, 2017). Existem também *IDEs* tais como *Eclipse*, *NetBeans* e *IntelliJ IDEA* que fornecem refatorações automatizadas para as novas versões de Java.

Porém, existem alguns problemas. Tanto os trabalhos, quanto as *IDEs* apresentam um número limitado de refatorações e estruturas que podem ser refatoradas. As *IDEs* procuram por oportunidades de refatoração e apresentam uma lista com as oportunidades encontradas, apresentando as modificações para o usuário aceitar. Porém, elas não apresentam uma formalização da mecânica aplicada em suas refatorações e não fornecem oportunidades de se realizar mudanças na hora de aplicar a refatoração.

Desta forma, o objetivo deste trabalho é a criação de um catálogo com refatorações para atualização de programas para as versões mais recentes da linguagem Java, de forma a ampliar o leque de refatorações para atualização de código e guiar desenvolvedores para evolução de programas. Além disso, este trabalho fornece a definição de um conjunto de funções de detecção para a busca de oportunidades para aplicar as refatorações propostas.

As principais contribuições são:

- A definição de um catálogo com 30 refatorações, apresentadas no formato canônico, sendo 15 refatorações para atualização de código Java para as versões 7, 8, 9 e 10 e outras 15 refatorações inversas.
- A definição e implementação de 15 funções de detecção para a busca de oportunidades de refatoração do catálogo, estando de acordo com as refatorações para atualização de código.

Este trabalho complementa o catálogo proposto por Teixeira Júnior (TEIXEIRA JÚNIOR et al., 2014), adicionando novas refatorações e utilizando algumas das refatorações já propostas em seu catálogo nas mecânicas das propostas neste trabalho.

Para a avaliação das refatorações propostas, um estudo de caso foi desenvolvido, sendo implementada uma *API* por meio das informações fornecidas pelo *framework AOPJungle* (FAVERI et al., 2013).

Esta dissertação está organizada como segue:

Para o Capítulo 2, a base conceitual para esta dissertação, incluindo tópicos como refatoração, evolução da linguagem Java, sendo apresentada as principais estruturas que fazem parte do catálogo de refatorações e o *framework AOPJungle*, além de apresentar os trabalhos relacionados a esta dissertação.

Para o Capítulo 3, um conjunto de 15 refatorações em um formato contendo: (i) nome, (ii) motivação, (iii) mecânica e (iv) um exemplo da estrutura antes e depois da aplicação da refatoração. O capítulo apresenta também a definição de 15 funções de detecção para busca das refatorações de evolução de código, em um formato padrão contendo um nome, uma explicação e uma definição formal.

Para o Capítulo 4, a implementação das funções de detecção. O capítulo ainda expõe informações sobre o desenvolvimento da *API* para extração e visualização de informações sobre código, os resultados da execução das funções de detecção em cinco programas de código aberto e exibe exemplos de código sugeridos pela ferramenta desenvolvida.

Para o Capítulo 5, as contribuições desta dissertação e algumas indicações e sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo faz uma revisão da literatura referente a alguns tópicos importantes para o desenvolvimento deste trabalho. Os seguintes assuntos serão tratados: refatoração, evolução da linguagem Java, principais funcionalidades e estruturas atualizadas nas novas versões de Java e o *framework AOPJungle*. Além disso, este capítulo apresenta os trabalhos relevantes que estão relacionados ao tema desta dissertação.

2.1 REFATORAÇÃO

A refatoração é um processo de modificação das estruturas internas de um sistema de software, para torná-lo mais fácil de ser entendido e menos custoso de ser modificado, sem alterar seu comportamento observável (FOWLER; BECK, 1999). O processo de refatoração pode envolver a remoção de duplicações, a simplificação de lógica condicional e a transformação em um código mais claro. Tais aperfeiçoamentos podem envolver algo simples, como trocar o nome de uma variável, ou mais elaborado como unificar duas hierarquias. Existem várias razões e motivações para se refatorar código. Dentre elas, as principais são tornar a adição de código novo mais simples, melhorar o código de projetos existente, obter um melhor entendimento a respeito da codificação, entre outros benefícios (KERIEVSKY, 2009).

A refatoração auxilia nas mudanças em um sistema de software, com objetivo de ajudar a melhorar sua compreensão e evolução com o passar do tempo. GRISWOLD e OPDYKE apresentaram as primeiras formalizações da refatoração de programas orientados a objeto. OPDYKE identificou vinte e três refatorações primitivas e apresentou algumas condições prévias para garantir que as transformações preservam o comportamento dos programas. Ele ainda apresentou três refatorações mais complexas mesclando as refatorações primitivas.

Porém, aplicar refatorações no código de um sistema de forma manual é tediosa e propensa a erros. Por esse motivo, alguns trabalhos apresentam ferramentas para automatizá-las. ROBERTS, por exemplo, automatizou as refatorações básicas propostas por OPDYKE através de uma ferramenta para a linguagem *Smalltalk*. ROBERTS; BRANT; JOHNSON afirmaram que se pequenas modificações (refatorações) propostas estiverem corretas, então às grandes mudanças no código de um sistema utilizando o conjunto dessas modificações, também estarão corretas.

FOWLER; BECK formalizaram um catálogo de refatorações em seu trabalho. Cada refatoração apresentada no catálogo possui basicamente cinco partes: (i) um nome, para ser identificada a refatoração, (ii) um breve resumo, (iii) uma descrição da motivação para refatoração, (iv) um passo-a-passo de como aplicar essa refatoração, (v) um pequeno exemplo, de forma a ilustrar o seu funcionamento. Podem existir diferenças entre usar uma refatoração e reduzi-la aos passos mecânicos como está presente neste trabalho. Eventualmente, podem ocorrer problemas que surgem em circunstâncias muito específicas. Ao utilizar uma refatoração baseada em catálogo, medidas de cautela devem ser tomadas, pois algumas refatorações devem ser adaptadas às particularidades do sistema (FOWLER; BECK, 1999).

Outros trabalhos também identificam oportunidades de refatoração para guiarem o desenvolvedor, sugerindo quais estruturas podem ser modificadas, estruturando um catálogo com problemas encontrados em códigos de programas, ou até mesmo estruturas antigas e sugerindo refatorações (MONTEIRO; FERNANDES, 2006; PIVETA et al., 2006; MACIA BERTRAN; GARCIA; STAA, 2011; RISSETTI et al., 2011; TEIXEIRA JÚNIOR et al., 2014). Por exemplo, MONTEIRO; FERNANDES propuseram um catálogo com refatorações que ajudam a migrar algumas estruturas de um sistema orientado a objetos para um sistema orientado a aspectos. RISSETTI et al. propuseram um catálogo com dez refatorações para atualizar estruturas antigas em programas na linguagem *Fortran*. Outro exemplo pode ser encontrado em TEIXEIRA JÚNIOR et al. que propuseram um catálogo com dez refatorações voltadas para o uso de expressões lambda em programas Java.

A Tabela 1 resume as principais refatorações presentes no trabalho de TEIXEIRA JÚNIOR et al., 2014. O catálogo de refatoração presente neste trabalho (Capítulo 3) complementa esse catálogo.

2.2 EVOLUÇÃO DA LINGUAGEM JAVA

Java é uma linguagem de programação de propósito geral, concorrente, baseada em classes e orientada a objetos. Assim como outras linguagens, Java passou por diversas modificações e melhorias ao longo dos anos. A cada nova versão da linguagem, geralmente são adicionadas novas funcionalidades e ferramentas, realizadas correções de *bugs* e melhorias nas estruturas internas.

Podem ocorrer problemas com as novas estruturas adicionadas, pois novas palavras reservadas podem ser acrescentadas na nova versão, o que pode resultar em conflitos com nomes

Tabela 1 – Refatorações propostas por TEIXEIRA JÚNIOR et al..

Refatoração	Descrição
<i>Convert Functional Interface Instance to Lambda Expression</i>	Substitui uma instancia de interface funcional por uma expressão lambda
<i>Convert Enhanced For to Lambda Enhanced For</i>	Substitui uma construção for each em uma expressão lambda usando o método <code>forEach</code>
<i>Convert Collections.sort to sort</i>	Substitui chamada ao método <code>Collections.sort</code> por chamada ao método <code>sort</code> da própria coleção
<i>Convert Enhanced For with If to Lambda Filter</i>	Substitui um controle de seleção por um filtro
<i>Convert Functional Interface to Default Functional Interface</i>	Substitui as interfaces funcionais <i>ad hoc</i> por interfaces padrões
<i>Convert Abstract Interface Method to Default Method</i>	Implementa método <i>default</i> na interface, quando possível
<i>Convert Inter-Type Method Declaration to Default Interface Method</i>	Substitui a declaração inter-tipos por um método <i>default</i>
<i>Extract Method Reference</i>	Substitui uma expressão lambda para uma referência a método
<i>Convert Interface to Functional Interface</i>	Transforma uma interface não funcional em uma interface funcional
<i>Convert Enhanced For to Parallel For</i>	Substitui um <code>for</code> sequencial por uma construção paralela utilizando os recursos da API <code>Stream</code>

de variáveis, métodos, constantes, entre outros (URMA; FUSCO; MYCROFT, 2014). A evolução de uma linguagem, muitas vezes, requer que os desenvolvedores façam um balanço entre os benefícios de permitir novos recursos de codificação e os custos envolvidos na manutenção de código antigo que não foi corretamente desenvolvido (GOETZ, 2017).

O uso da refatoração pode ajudar desenvolvedores a fazer atualizações em seu código, sugerindo estruturas que podem ser atualizadas, e como realizar essa atualização. As próximas seções apresentam as atualizações das versões 7 e 8 da linguagem Java, bem como outras atualizações presentes nas versões 9 e 10. Todas as funcionalidades, métodos e estruturas que são utilizadas no catálogo de refatorações proposto no Capítulo 3 são expostas neste capítulo.

2.2.1 Java 7

A versão 7 de Java acrescentou algumas novas funcionalidades interessantes de serem refatoradas. Uma das novidades é a possibilidade de tratar mais de uma exceção num único bloco `catch`. Essa nova funcionalidade foi nomeada de *multi-catch*. Esse recurso pode reduzir a duplicação de código e diminuir a possibilidade de os desenvolvedores usarem muitas

exceções diferentes em um único bloco *try-catch*, podendo dificultar o entendimento do código por outros desenvolvedores (ORACLE, 2017a). A Listagem 3 mostra um exemplo do funcionamento do recurso *multi-catch*. O caractere pipe (|) funciona como um operador lógico *OU*, ou seja, se ocorrer alguma das exceções presentes nos argumentos do bloco `catch` nas linhas 5 e 6 do código, este bloco será executado fazendo o tratamento a exceção. Todas as exceções dentro de um mesmo `catch` serão tratadas pelo mesmo bloco de código.

Listagem 3: Funcionamento do *multi-catch*.

```

1 var div = new Scanner(System.in);
2 try {
3     var divisor = div.nextInt();
4     System.out.print(2 / divisor);
5 } catch (ArithmeticException | IllegalArgumentException |
6         NullPointerException e) {
7     System.out.println("Erro: " + e.getMessage());
8 }

```

Outra novidade interessante é a funcionalidade *try-with-resources*, que mudou e simplificou a forma de implementar o bloco `try`. Com essa nova funcionalidade é possível declarar um ou mais recursos nos argumentos do bloco. Recursos são os objetos que devem ser fechados após a execução da estrutura *try-catch*. Geralmente é usado o bloco `finally` para fechar recursos abertos no bloco `try`, porém, com essa nova funcionalidade, o desenvolvedor não necessita mais fazer isso manualmente, o compilador fará o trabalho automaticamente e solucionará os casos especiais que ocorrerem. Porém, para usar essa funcionalidade as classes destes objetos devem implementar a interface `java.lang.AutoCloseable`, o que já inclui os objetos que implementam a interface `java.io.Closeable` (ORACLE, 2017a).

A Listagem 4 mostra um exemplo de uso da construção *try-with-resources*. A declaração do recurso que deve ser fechado aparece entre parênteses imediatamente após a palavra reservada `try` (linha 1). A classe `FileReader` implementa a interface `AutoCloseable`. O objeto está sendo criado e instanciado nos argumentos da instrução `try`, então, essa forma de construção, se beneficiará da funcionalidade *try-with-resources*. Sendo assim, o recurso será fechado, independentemente de o bloco `try` ser executado até o fim ou ser lançada alguma exceção. Não será necessário colocar o bloco `finally` para fechar o recurso manualmente, como geralmente ocorria em versões anteriores.

Listagem 4: Recurso fechado automaticamente, utilizando *try-with-resources*.

```

1 try (var arquivo = new FileReader("texto.txt") ) {
2     //...
3 } catch (IOException e) {
4     System.err.printf("Erro: %s.\n", e.getMessage());
5 }

```

2.2.2 Java 8

A versão 8 de Java realizou algumas mudanças em suas estruturas internas, especialmente para se adaptar ao mercado de hardware. Atualmente a maioria dos computadores pessoais, empresariais e servidores adota processadores *multinúcleo*, pois possuem uma série de vantagens em relação aos processadores com apenas um núcleo, entre elas realizar várias tarefas ao mesmo tempo, aumentando a velocidade de processamento. Porém, escrever código que realmente funciona em paralelo é um processo mais complicado. Em Java 7 foi adicionado o *fork/join framework*, que ajudava os desenvolvedores a trabalharem com múltiplos núcleos. Com essa funcionalidade, é possível dividir o código em pequenas partes. Esse *framework* fez o desenvolvimento paralelo ser mais prático, porém ainda é difícil e trabalhoso de desenvolver ou atualizar programas com ele (URMA; FUSCO; MYCROFT, 2014).

Em Java 8 foram introduzidos novos conceitos, como as expressões lambdas, originárias da programação funcional. Com as expressões lambdas é possível obter uma série de vantagens na programação, como a facilidade em codificar instruções com execução paralela e a obtenção de um código mais enxuto.

O literal para uma expressão lambda consiste em uma lista de argumentos (zero ou mais) seguida do operador " \rightarrow ", que é um *token* utilizado para denotar a aplicação do parâmetro à sua implementação, seguida de uma expressão que deve produzir um valor.

A Listagem 5 apresenta alguns exemplos de expressões lambdas. A linha 1 apresenta uma expressão que não recebe nenhum argumento e retorna sempre o número 27, a linha 2 exhibe uma expressão que recebe um argumento e retorna o seu quadrado e a linha 3 mostra uma expressão que recebe dois argumentos e retorna a soma entre os dois.

Listagem 5: Exemplos de expressões lambdas.

```

1     () -> 27
2     x -> x * x
3     (x, y) -> x + y

```

2.2.2.1 Coleções e Strings

Esta seção apresenta alguns conceitos relacionados às coleções e *strings*, como os novos métodos que foram adicionados e que estão presentes no catálogo de refatorações deste trabalho. A interface *Collection* em Java contém as principais funcionalidades para se trabalhar com coleções. A partir de Java 8, algumas estruturas foram adicionadas e atualizadas nessa interface a fim de dar suporte a expressões lambda, fluxos e operações agregadas. Uma das principais características é a possibilidade de interagir com uma coleção sem a necessidade de percorrer seus elementos explicitamente, com a utilização de *loops* (interação externa), por exemplo. Com as novas estruturas, é possível realizar uma iteração interna, em que o compilador fica encarregado de realizar certas tarefas e o desenvolvedor pode focar no que deve ser feito e não na forma como deve interagir com a coleção.

Um exemplo de um método adicionado na interface *Collection* e que utiliza uma interação interna é o `removeIf`. Com esse método é realizada uma interação interna, sendo possível a remoção de elementos da coleção apenas chamando o método `removeIf` da própria interface e passar uma expressão lambda com a condição da remoção.

A Listagem 6 mostra um exemplo de uso desse método, onde são removidos da lista todos os nomes iguais a "Pedro". Note que é necessário implementar a interface funcional *Predicate* para usar o método `removeIf`. Para que essa interface funcione corretamente é necessário que se atribua uma expressão *lambda* que retorne um valor booleano.

Listagem 6: Remoção de elemento de uma lista utilizando o método `removeIf`.

```
1 lista.removeIf(person -> "Pedro".equals(person.getName()));
```

Outro método funcional adicionado em algumas coleções, como as listas (*List*) e *maps* (*Map*), é o método `replaceAll`. Esse método realiza uma interação interna que modifica, através de um método ou alguma outra operação, cada elemento da coleção e o substitui pelo resultado da operação aplicada. Anteriormente para chamar algum método, ou fazer algum tipo de operação nos elementos de uma coleção, era necessário utilizar um laço de repetição para acessar cada elemento e aplicar a modificação necessária. Atualmente, basta chamar o método `replaceAll` da própria coleção e passar uma expressão lambda com a operação desejada.

A Listagem 7 apresenta um exemplo desse novo método. É possível observar que o método `replaceAll` está sendo executado em uma lista. Ele aplica a cada elemento dessa lista o resultado da expressão *lambda* fornecida. Note que essa expressão contém apenas um

argumento (antes do *token*) e esse argumento sofre a alteração dependendo do corpo da expressão (após o *token*). No exemplo é chamado o método `toUpperCase`. Embora não tenha um retorno explícito, a expressão *lambda* retornará o argumento em letra maiúscula, pois é essa a funcionalidade do método aplicado. Resumindo, para cada elemento da lista será aplicada a expressão *lambda*, entrando o próprio elemento como argumento dessa expressão e tendo como retorno o elemento modificado.

Listagem 7: Transformar elementos de uma lista utilizando o método `replaceAll`.

```
1 lista.replaceAll(s -> s.toUpperCase());
```

As *Strings* não são um tipo primitivo na linguagem Java. O tipo `String` é uma classe que representa um conjunto de caracteres e possui seus próprios métodos e funcionalidades. Esta classe existe desde a primeira versão de Java e agora foi adicionado um novo método (`join`) que utiliza o conceito de interação interna para concatenar várias *strings*, ou um *array* de *strings*, em uma única variável do tipo texto, unindo-as por meio de um delimitador, que pode ser um carácter ou um conjunto de caracteres selecionado pelo desenvolvedor.

A Listagem 8 mostra um exemplo do uso do método `join`. Antes de Java 8, para agrupar uma lista de *strings*, caracteres e outros tipos nativos, era necessário uma estrutura de repetição para acessar cada elemento da lista e então utilizar algum operador de concatenação para unir os elementos. Atualmente, pode ser utilizado o método `join` da classe `String`, junto com o delimitador e o conjunto de *strings*, como é apresentado na linha 1 da Listagem 8, ou com a coleção de `Strings`, como exibido na linha 2. É possível também agrupar valores de outros tipos primitivos, convertendo-os em *strings*. O resultado na linha 1 é uma variável do tipo `String`, com os elementos texto, passados no argumentos do método `join`, concatenados utilizando o caractere *espaço*. Já a linha 2, apresenta a concatenação de elementos de uma coleção utilizando o delimitador *ponto*.

Listagem 8: Concatenar uma lista de strings utilizando o método `join` da classe `String`.

```
1 var text = String.join(" ", "abacate", "uva", "laranja");
2 var text2 = String.join(".", lista);
```

2.2.2.2 A API de Streams

A *Stream API* trouxe novas propostas para a manipulação de coleções em Java, com o objetivo de facilitar o processo de codificação para o desenvolvedor, removendo a preocupação

com a forma que programará o comportamento, deixando as tarefas relacionadas ao controle de fluxo a cargo da *API*. Essa *API*, quando combinada com as expressões lambda, pode proporcionar uma forma diferente de lidar com coleções de elementos, oferecendo ao desenvolvedor uma maneira simples e concisa de escrever código (SILVA, 2017). Conceitualmente, uma *stream* pode ser definida como uma sequência de elementos de uma fonte de dados que possibilita operações de agregação.

As principais características de uma *stream* são: (i) prover uma interface para um conjunto sequencial de valores de um determinado tipo, (ii) processar elementos de uma coleção sobre demanda, (iii) necessitar de uma fonte de dados, que pode ser uma coleção, como arrays, ou listas e (iv) dar suporte a operações comuns a linguagens funcionais, como filtrar, modificar, transformar os elementos, etc.

As operações realizadas por uma *stream* tem algumas características, como, por exemplo, que todas as iterações realizadas pelas operações são classificadas como iterações internas, ou seja, não possuem explicitamente estruturas para interagir com coleções, conforme explicado na Seção 2.2.2.1. Outra característica das operações é que são divididas em dois tipos, as operações intermediárias e as operações terminais.

Operações intermediárias retornam outra *stream* e, conseqüentemente, podem ser realizadas outras operações, ou seja, é possível realizar um encadeamento de múltiplas operações intermediárias. As operações terminais tem por objetivo fechar o processo realizado, retornando um resultado diferente de uma *stream*, como um valor, objeto ou coleção. Dessa forma, a *Stream API* trabalha convertendo uma fonte de dados em uma variável do tipo *stream*, em que é realizado algum processamento nos dados, de forma paralela ou não, e retorna algum valor, coleção, etc. Nesta seção são exploradas algumas das funções utilizadas no catálogo deste trabalho.

O funcionamento da *Stream API* é explicado através de um exemplo. O exemplo apresentado é a realização de um cálculo da soma das idades de pessoas do sexo feminino em uma coleção.

A Listagem 9 mostra um exemplo de um trecho de código utilizando a *Stream API*. Na primeira linha, a coleção chama o método `stream`. Esse método transforma a coleção em uma *stream* e a partir disso é possível chamar as operações intermediárias ou terminais. Na segunda linha aparece o método `filter`, que filtra um elemento de acordo com alguma condição, que no exemplo são os objetos em que o método `getGenero` retorna "Feminino". A terceira

linha mostra o método `mapToInt`, o qual extrai alguma informação daquela *stream*. No exemplo, esta linha transforma a *stream* de objetos em uma *stream* de números inteiros obtendo os valores de acordo com método `getIdade`. Por fim, a quarta linha retorna um único inteiro com a soma dos valores obtidos. Nota-se que os métodos `filter` e `mapToInt` são operações intermediárias, já o método `sum` é uma operação terminal. Ainda é possível chamar o método `parallelStream` na linha 1, que realizaria todas as operações de forma paralela.

Listagem 9: Realizar a soma da idade sobre uma lista de pessoas com gênero igual a "Feminino".

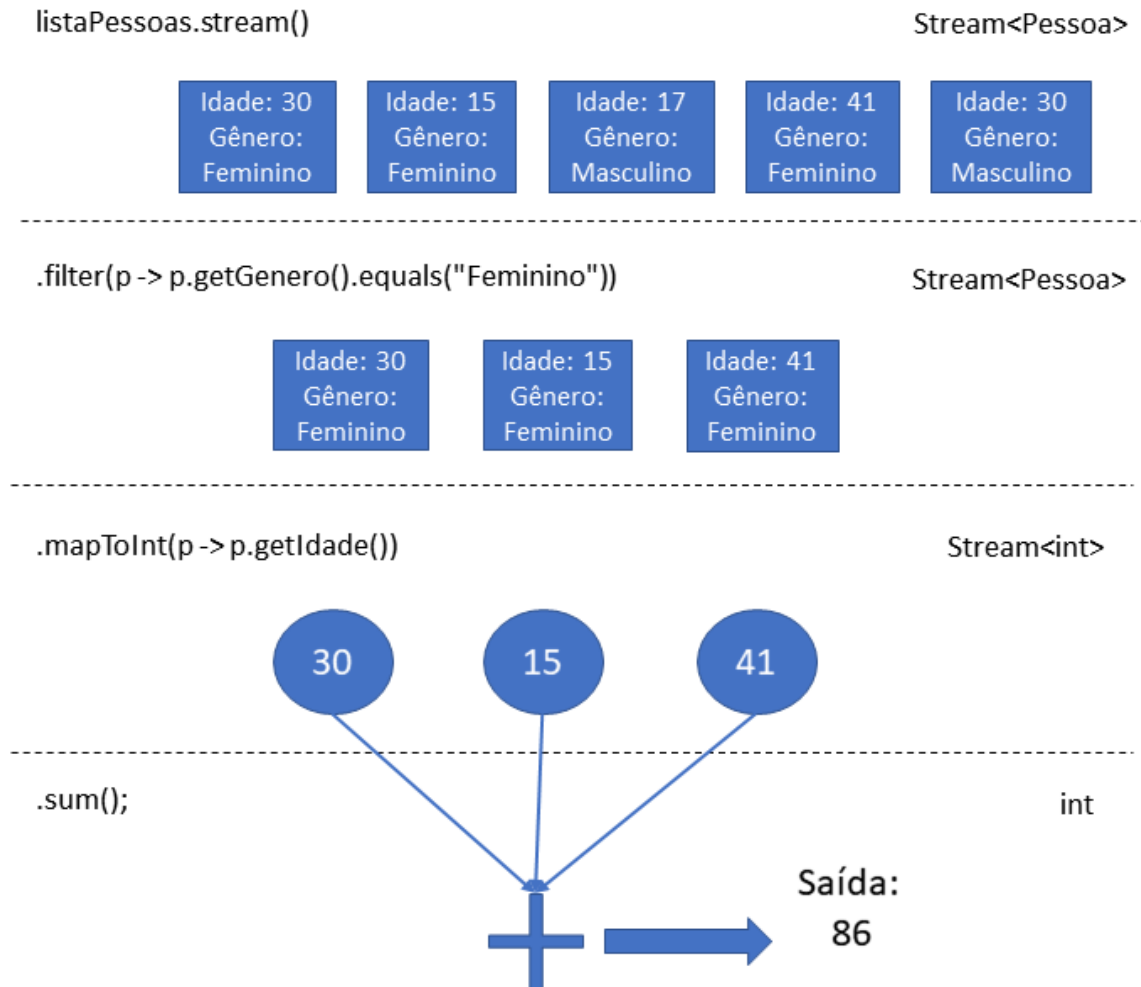
```
1 var soma = listaPessoas.stream()
2     .filter(p -> p.getGenero().equals("Feminino"))
3     .mapToInt(p -> p.getIdade())
4     .sum();
```

A Figura 1 exemplifica o que acontece com os dados da *stream*, os tipos que eles assumem em cada processo e o valores que passam para a próxima etapa até chegar na operação terminal `sum` para mostrar o resultado final.

Alguns métodos da classe `Stream` foram a base para o processo de refatoração das estruturas antigas analisadas neste trabalho. Um exemplo é a operação intermediária `map`. Essa estrutura aplica um método, ou faz uma operação em cada um dos valores de uma *stream*, retornando uma nova *stream* modificada de acordo com o resultado da operação de uma expressão *lambda*. Existem algumas variações dessa estrutura, como a `mapToInt` que pode ser aplicada quando se deseja extrair valores inteiros de uma *stream* de objetos, então essa operação intermediária retorna um `IntStream`. Assim como existem as variações `mapToLong`, que retorna um `LongStream`, e `mapToDouble`, que retorna um `DoubleStream`. A principal diferença entre o método `map` e suas variações é que nestas o desenvolvedor especifica que retornará um valor numérico, sendo possível chamar alguns métodos terminais específicos, como `average`, que retorna a média dos números da *stream*, `sum` onde se obtém a soma, como exemplificado na Listagem 9, dentre outras funções específicas para valores numéricos.

Um exemplo de operação terminal é a operação `reduce`. Essa operação executa uma redução nos elementos da *stream*, ou seja, calcula um resultado usando todos os elementos da *stream*. A Listagem 10 mostra um somatório de valores obtidos através do método `getIdade` de uma lista com objetos do tipo `Pessoa`. É um exemplo da utilização dos métodos `map` e `reduce` em conjunto. Na linha 2 aparece o método `map` (nesse caso poderia ser

Figura 1 – Diagrama temporal de execução de tarefas do programa



utilizada também a operação `mapToInt`, considerando que o método `getIdade` retorne um inteiro).

A linha 2 deveria ter uma expressão *lambda* como acontece na Listagem 9, porém um conceito que surgiu em Java 8 é o da referência de método. Esse conceito pode substituir uma expressão *lambda* que chama algum método em específico, inclusive métodos construtores. Então, se no corpo da expressão *lambda* possui apenas uma chamada a um método específico é possível substituir esta expressão por uma referência de método. O oposto é válido da mesma forma, ou seja, sempre onde há uma referência de método é possível substituir por uma expressão *lambda*. TEIXEIRA JÚNIOR et al. propõem uma refatoração que transforma expressões *lambdas* em referências de método.

Na Listagem 10, como o objetivo é obter a idade, a expressão *lambda* apenas faz a chamada ao método `getIdade`, sendo possível substituir a expressão *lambda* pela referência ao método. Por fim, a linha 3 mostra um exemplo de uso do método `reduce`, onde é realizada

a soma de todos os valores da `stream`. O primeiro argumento desse método é o valor inicial da soma, que no exemplo é 0, e após isso a expressão lambda de acordo com a operação que será feita.

Listagem 10: Realizar a soma da idade de um conjunto utilizando o método `reduce`.

```
1 var soma = lista.stream()
2     .map(Pessoa::getIdade)
3     .reduce(0, (a, b) -> a+b);
```

Outros exemplos de operações deste trabalho pertencem a classe `Collectors`. Essa classe implementa vários métodos para reduções, acumulações e transformações de acordo com vários critérios. Por exemplo, existem métodos na classe `Collectors` para transformar uma `stream` em uma lista (`toList`), em um conjunto (`toSet`) ou em uma coleção (`toCollection`). A Listagem 11 mostra um exemplo do uso da classe `Collectors`. O resultado é uma lista com os nomes de todos os objetos cujo retorno do método `getGenero` é igual a "Feminino". Outro método da classe `Collectors` pertinente para o desenvolvimento deste trabalho é o `groupingBy`, que fornece uma funcionalidade semelhante à cláusula `GROUP BY` da linguagem SQL. Esta classe agrupa objetos de acordo com alguma propriedade e armazena os resultados em uma instância do tipo `Map<K, V>`.

Listagem 11: Lista do tipo `String` com nomes de objetos que o método `getGenero` retorna "Feminino".

```
1 ver nomes = lista.stream()
2     .filter(p -> p.getGenero().equals("Feminino"))
3     .map(Pessoa::getNome)
4     .collect(Collectors.toList());
```

2.2.2.3 Classe *Optional*

`Optional` é uma classe, inserida em Java 8, que encapsula vários métodos úteis para tratar blocos de código crítico que necessitam ser verificados. Essa classe é como um *container* que pode possuir um valor ou não. O método `isPresent` pode ser chamado, retornando um valor booleano, indicando a presença ou não de um valor. Ainda, é possível usar o método `get` que retornará o valor caso presente (ORACLE, 2017b). A classe `Optional` ajuda os desenvolvedores a estruturarem o tratamento de valores nulos, evitando a exceção `NullPointerException`, considerada crítica quando ocorre na execução de um programa.

A Listagem 12 mostra um trecho de código que utiliza a classe `Optional` para verificar um objeto. A linha 1 utiliza o método `ofNullable` para retornar um `Optional` vazio e não uma exceção, caso o valor for nulo. A linha 2 mostra uma chamada ao método `ifPresent`, que só executará bloco de código em seu argumento se o valor estiver presente. No exemplo, será impresso no terminal o resultado do método `getNome` do objeto `Pessoa`, caso o objeto esteja presente.

Listagem 12: Verificação de valores nulos utilizando a classe `Optional`.

```
1 Optional<Pessoa> pessoa = Optional.ofNullable(getPessoa(nome));
2 pessoa.ifPresent(p -> System.out.println(p.getNome()));
```

2.2.2.4 *Spliterator*

A interface `Spliterator` é utilizada para percorrer e particionar coleções. Essa nova interface foi adicionada recentemente e, embora a função seja semelhante, a sua forma de funcionamento é um pouco diferente da interface `Iterator`, pois além de aceitar expressões lambda, possui estruturas que combinam os métodos `hasNext` e `next` da interface `Iterator`. A API do `Spliterator` foi projetada para dar suporte a operações paralelas, além do percurso sequencial, possibilitando a decomposição da iteração em uma coleção (ORACLE, 2017b). Dessa forma, a interface `Spliterator` pode ser usada para dividir uma coleção em vários conjuntos, realizar alguma operação ou cálculo em cada conjunto utilizando diferentes *threads* de forma independente e, possivelmente, aproveitando o paralelismo dos processadores.

A Listagem 13 mostra um exemplo da utilização da interface `Spliterator`. A linha 1 apresenta uma variável que recebe uma lista de *strings*. A linha 2 mostra, em um laço de repetição, a utilização do método `tryAdvance`. Seu funcionamento é parecido com a combinação dos métodos `hasNext` e `next` da interface `Iterator`. Ou seja, ele executa a ação presente na expressão lambda e caso existam mais elementos na lista o método retorna *true*. Caso não existam elementos na lista ele retorna *false*, o que fará com que a execução do programa saia do laço.

Listagem 13: Interação com uma lista utilizando a interface `Spliterator`.

```
1 var is = nomeLista.spliterator();
2 while(is.tryAdvance((n) -> System.out.println(n)));
```

2.2.2.5 Predicate

`Predicate` é uma interface funcional, adicionada em Java 8, que pode ser utilizada para avaliar uma determinada condição, retornando um valor booleano de acordo com o teste realizado. Esta interface pode ser utilizada como destino para uma atribuição de expressão *lambda* ou método de referência (ORACLE, 2017b). Existem algumas vantagens ao utilizar essa interface, tais como (i) mais facilidade para testar e alterar condições, (ii) agrupar as condições em um ponto central, (iii) evitar duplicidade do código e (iv) evitar a manutenção em vários lugares.

A Listagem 14 mostra um exemplo de como poderia ser utilizada a interface `Predicate`. No exemplo, a linha 1 apresenta uma variável do tipo `Integer`, encapsulada pela interface `Predicate`, que recebe uma expressão *lambda*, com uma operação lógica que retorna *true* ou *false*. Agora com a variável criada é possível chamar os métodos da interface para tratar algum valor. O exemplo apresenta uma chamada ao método `test`, que avalia a expressão do predicado de acordo com o argumento dado. Usando essa estrutura, se algum valor é alterado, será modificado apenas a expressão *lambda* do predicado da linha 1. Se não fosse usada a interface, seria necessário alterar todos os *ifs* que foram construídos utilizando a mesma verificação (as linhas 2 e 6 do exemplo).

Listagem 14: Exemplo da utilização da interface `Predicate`.

```

1 Predicate<Integer> testeIdade = e -> e > 18 && e < 36;
2 if (testeIdade.test(idade)) {
3     //...
4 }
5 //...
6 if (testeIdade.test(idade)) {
7     //...
8 }

```

2.2.3 Java 9 e 10

Em Java 9 algumas melhorias foram adicionadas na *API* de coleções. Um método pertinente para este trabalho é o utilizado para a criação de coleções não modificáveis. Por exemplo, para criar uma coleção estática não modificável, atualmente, é necessário instanciar a coleção, inserir os valores através de um método de inserção e depois chamar um método específico da classe `Collections` que transforma aquela coleção em não modificável. Em Java 9 pode ser

criada uma coleção não modificável utilizando o método `of` da coleção e passar os valores que se desejava adicionar. A Listagem 15 mostra exemplos de três coleções imutáveis diferentes sendo criadas. A linha 1 apresenta uma coleção do tipo `Map`, a linha 3 uma coleção do tipo `List` e a linha 5 uma coleção do tipo `Set`.

Listagem 15: Criação de coleções não modificáveis no Java 9.

```
1 Map.of(1, "Turini", 2, "Paulo", 3, "Guilherme");
2
3 List.of(1, 2, 3);
4
5 Set.of("SP", "BSB", "RJ");
```

Java 10 apresenta algumas mudanças significativas no contexto deste trabalho. É possível citar o exemplo da inferência de tipos, um recurso que permite aos desenvolvedores substituírem a declaração do tipo explícito de variáveis com uma palavra-chave `var`. Há melhorias nas funcionalidades de expressões lambdas também, a citar o compilador ser capaz de inferir no tipo de retorno de uma expressão lambda. Anteriormente, mesmo o contexto fornecendo informações suficientes, era necessário o desenvolvedor informar os tipos dos argumentos manualmente.

Uma outra modificação importante para este trabalho é o uso do carácter *underscore* para parâmetros não utilizados em uma expressão *lambda*. Existem cenários em que um *lambda* com vários parâmetros é esperado, embora o corpo não use todos eles, forçando o desenvolvedor a usar nomes indicativos para os parâmetros não utilizados. Com essa alteração, permite-se o uso do sublinhado para tais parâmetros. A Listagem 16 mostra um exemplo onde um parâmetro não utilizado foi substituído pelo carácter *underscore*.

Listagem 16: Forma de declarar parâmetros não utilizados em expressão lambdas em Java 10.

```
1 salarios.replaceAll( (_, salario) -> salario + 10000 );
```

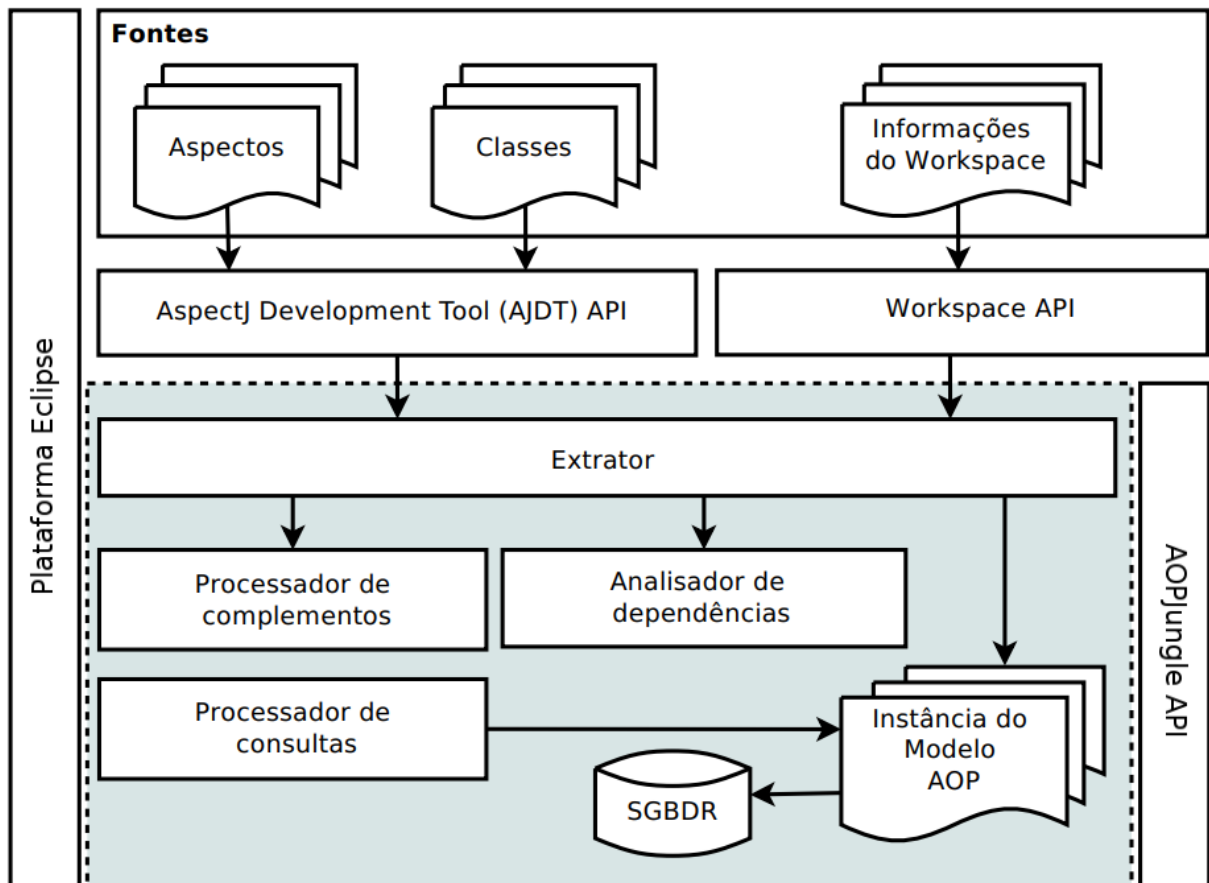
2.3 FRAMEWORK AOPJUNGLE

Uma função de detecção pode ser utilizada para identificar trechos de código onde pode existir uma oportunidade para aplicar uma determinada refatoração. Para localizar oportunidades de refatoração é necessário obter meta-informações a respeito do código fonte de uma

aplicação. Uma forma de obter essas informações é utilizando o *framework AOPJungle* (FAVERI et al., 2013), que foi desenvolvido na UFSM, no grupo de linguagens de programação e bancos de dados.

O *framework* tem como objetivo extrair informações de código fonte e poder acessá-las por meio de uma linguagem específica de domínio ou por estruturas de dados disponíveis no próprio *framework*. A Figura 2 apresenta a arquitetura geral do *AOPJungle*, desenvolvido para a plataforma *Eclipse* como um *plug-in*. Ele possui integrações com a ferramenta *AJDT* (*AspectJ Development Tools*) e com a *API* do *workspace* do *Eclipse* para obter informações de código e para obter dados sobre os recursos da plataforma, tais como projetos, arquivos, caminhos e configurações.

Figura 2 – Arquitetura do *AOPJungle* (FAVERI et al., 2013)



O módulo *Extrator* é responsável pela análise do código fonte de cada projeto do ambiente *Eclipse* e por instanciar o metamodelo, sendo o principal módulo do *AOPJungle*. O módulo *Extrator* utiliza a *API* do *Workspace* para identificar os projetos *AspectJ/Java* disponíveis no *IDE*. A partir da lista de projetos, o *Extrator* utiliza a *API* do *AJDT* para obter os pacotes e as unidades de compilação, como estruturas, métodos e classes. Nesse processo de extração da

informação, o *framework* percorre as *ASTs* (*Abstract Syntax Trees*) gerada pelo *AJDT* obtendo as informações necessárias de cada unidade de compilação. Para fazer isso, o *AOPJungle* implementa o padrão *Visitor*, que permite definir uma nova operação em uma estrutura de objetos sem alterar as classes dos objetos nos quais ela opera (PALSBERG; JAY, 1998). A principal classe responsável por obter as informações é a *AOJungleVisitor*, contida no pacote `br.ufsm.aopjungle.aspectj`. Essa classe estende a classe abstrata *AjASTVisitor* do *AJDT*.

Na classe *AOJungleVisitor* são implementados o método `visit` para iniciar a visita em uma unidade de compilação da *AST* e o método `endVisit` para encerrar a visita nessas estruturas. Um exemplo pode ser visto na Listagem 17, onde o método acessa as estruturas `if` de um código. Esse método será invocado sempre quando houver uma estrutura `if`. Dentro desse método, é possível obter informações a respeito do código presente nessa estrutura. Além disso, pode-se obter outras informações como nome do pacote onde se encontra a estrutura, o nome da classe que ela pertence, o número da linha onde começa a estrutura, etc. No exemplo da Listagem 17 a estrutura é inserida na árvore interna do *AOPJungle*.

Listagem 17: Exemplo de um método `visit` da classe *AOJungleVisitor*.

```

1  @Override
2  public boolean visit(IfStatement node) {
3      AOJIfStatement aojNode = new AOJIfStatement(node,
4          (AOJProgramElement)getLastMemberFromStack());
5      AOJBehaviourKind behaviourKind = getAOJBehaviourKind(
6          getLastMemberFromStack());
7      if (behaviourKind != null)
8          behaviourKind.getStatements().add(aojNode);
9      elementStack.push(aojNode);
10     return super.visit(node);
11 }
12
13 @Override
14 public void endVisit(IfStatement node) {
15     elementStack.pop();
16     super.endVisit(node);
17 }

```

2.4 TRABALHOS RELACIONADOS

Em busca de trabalhos voltados para a prática de refatorações para evolução de programas, foram encontrados alguns projetos que possuem similaridade. RAHAD; CAO; CHEON

e DAVID; KESSELI; KROENING apresentam um estudo sobre refatorações para laços de repetição em Java convertendo-os para estruturas da *Stream API* de Java 8. Em RAHAD; CAO; CHEON, a abordagem é definir um conjunto de refatorações com base em padrões encontrados apenas em estruturas dos laços de repetição. Basicamente, os laços são analisados através de outras ferramentas e propostas. Após, são identificados padrões nesses laços através de um catálogo já existente (BARUA; CHEON, 2014) e, por fim, o código do laço é transformado de acordo com as regras identificadas. DAVID; KESSELI; KROENING propõe uma ferramenta chamada *Kayak*, cuja finalidade é realizar refatorações para conversão de iterações externas para internas. Esta dissertação foi desenvolvida utilizando uma abordagem mais ampla, analisando, além dos laços, uma série de novos recursos apresentado nas novas versões da linguagem Java e propondo refatorações para a evolução das estruturas mais antigas.

Outro estudo (TEIXEIRA JÚNIOR et al., 2014) apresenta um catálogo de refatorações semelhante ao proposto neste trabalho, sendo esta dissertação uma complementação desse catálogo já proposto. Dessa forma, esta dissertação, além de propor refatorações diferentes das propostas no trabalho de TEIXEIRA JÚNIOR et al., é proposto um conjunto de funções de detecção para busca das refatorações elencadas e o desenvolvimento de uma ferramenta automatizada para busca de oportunidades de refatoração.

Grande parte dos ambientes de desenvolvimento (*IDEs*) fornece algum tipo de suporte para refatoração. O *IDE IntelliJ IDEA* em sua última versão (2018.2) possui algumas refatorações similares às presentes nesse catálogo. As refatorações similares identificadas são: (i) *Agrupar blocos catch (Identical 'catch' branches in 'try' statement)*, (ii) *Mover recurso para o try-with-resources ('try finally' replaceable with 'try' with resources)* (iii) *Adicionar método map na interação com coleções (Replace with collect)* e (iv) *Converter if pelo método ifPresent (Refactoring to Optional)*. As refatorações *Agrupar blocos catch* e *Mover recurso para o try-with-resources* também existem em outras *IDEs*, como *NetBeans* e *Eclipse*. Pode-se notar algumas heterogeneidades, como a refatoração *Agrupar blocos catch* sugerida neste trabalho poder definir um grau de similaridade entre as estruturas *catch*. Na maioria das refatorações similares as estruturas *catch* devem ser exatamente iguais. Embora as *IDEs* implementem algumas das refatorações propostas, não há uma descrição da mecânica envolvida no processo, o desenvolvedor apenas aceita ou não determinada modificação.

FRANKLIN et al. apresentaram uma ferramenta chamada *LambdaFicator*, construída com o propósito de realizar refatorações de forma automática envolvendo expressões lambda

em Java. O trabalho apresenta duas ferramentas válidas, a primeira é a *AnonymousToLambda* que visa transformar classes anônimas em uma sintaxe correspondente utilizando expressões lambda. Outra ferramenta apresentada é *ForLoopToFunctional* que propõe converter laços `for` em uma operação funcional utilizando expressões lambda. O trabalho analisado possui uma proposta diferente, sendo direcionado para laços de repetição `for` que interagem com coleções, enquanto que esta dissertação é focada na atualização de código Java.

Tabela 2 – Tabela comparativa com trabalhos relacionados.

Trabalhos Relacionados / IDEs	Quantidade de refatorações	Apresenta as mecânicas das refatorações	Detecta oportunidades de refatoração
Esta dissertação	30	Sim	Sim
RAHAD; CAO; CHEON	Apenas para laços de repetição	Sim	Não
DAVID; KESSELI; KROENING	Apenas para laços de repetição	Sim	Sim
TEIXEIRA JÚNIOR et al.	20	Sim	Não
FRANKLIN et al.	2	Sim	Sim
IDEs	Depende da ferramenta	Não	Sim

2.5 CONSIDERAÇÕES FINAIS

Esse capítulo forneceu o embasamento teórico para entender alguns conceitos que serão apresentados no desenvolvimento deste trabalho. Foi apresentado o conceito acerca de refatoração, analisando alguns trabalhos na área e foram apresentadas algumas refatorações já catalogadas por TEIXEIRA JÚNIOR et al. (2014). Foram apresentados os conceitos e funcionalidades das estruturas que são propostas no catálogo de refatoração, demonstrando o seu funcionamento e pequenos exemplos de trechos de código. Por fim, uma explanação do *framework AOPJungle* que será utilizado para a implementação das funções de detecção apresentadas nesta dissertação.

3 UM CATÁLOGO DE REFATORAÇÕES PARA EVOLUÇÃO DE PROGRAMAS JAVA

Este capítulo apresenta um catálogo de refatorações para evolução de programas Java. São apresentadas 15 refatorações para evolução de programas. Outras 15 são suas respectivas refatorações inversas e são descritas no Anexo A. Todas as refatorações são descritas no formato canônico, ou seja, são descritas por um nome, uma motivação, a mecânica (um conjunto de passos para realizar a refatoração) e um exemplo. O capítulo está estruturado da seguinte forma: a Seção 3.1 apresenta as 15 primeiras refatorações propostas para evolução de programas, a Seção 3.2 apresenta a definição de funções de detecção para a busca de oportunidades de refatoração para a evolução de programas Java e a Seção 3.3 finaliza este capítulo com algumas considerações finais.

3.1 CATÁLOGO

Esta seção descreve 15 refatorações que tem por objetivo a evolução de código em Java. Elas estão separadas de acordo com a versão final que objetiva a refatoração após aplicada. As refatorações para Java 7 possuem suporte em muitos *IDEs*, uma vez que são funcionalidades com certo grau de maturidade na comunidade de desenvolvedores. Já as refatorações para Java 8 são funcionalidades mais recentes, inseridas relativamente há pouco tempo na linguagem. São também propostas algumas refatorações para Java 9 e 10. São elas:

- Java 7

- Agrupar blocos *catch*

- Mover recurso para o *try-with-resources*

- Java 8

- Adicionar método *map* na interação com coleções

- Agrupar com *groupingBy*

- Converter para uma operação aritmética *lambda*

- Converter *if* para *ifPresent*

- Converter *iterator.remove* para *removeIf*

Converter `iterator.set` para `replaceAll`

Concatenar uma coleção com `join`

Converter interface `Iterator` para `Spliterator`

Adicionar interface `Predicate`

Converter para média lambda

Converter para contagem lambda

- Java 9

Adicionar método `of`

- Java 10

Converter para *underscore*

Agrupar blocos `catch`

Você faz tratamento de exceções com código similar em diferentes blocos `catch`.

◇◇◇

Agrupe os blocos `catch` similares em um bloco único.

Essa refatoração pode ser vantajosa para diminuir a duplicação, simplificando o tratamento de exceções e melhorando a legibilidade do código. Esta refatoração faz atualização do bloco `catch` da estrutura *try-catch* para funcionalidade adicionada em Java 7 denominada *multicatch*, que permite o tratamento de múltiplas exceções em um único bloco `catch`. A mecânica é descrita da seguinte forma:

1. Criar um novo bloco `catch` vazio.
2. Verificar para quais classes de exceção nas estruturas `catch` existe código de tratamento similar.
3. Inserir as classes das exceções identificadas no passo 2 no novo bloco `catch`, juntamente com o parâmetro de exceção que receberá o objeto que representa a exceção.
4. Copiar o tratamento do bloco `catch` escolhido como o mais adequado para o novo bloco `catch`, atualizando as referências à variável que representa o objeto de exceção para o novo nome, adaptando se necessário (no caso de não serem idênticas).

5. Remover os blocos `catch` antigos.
6. Compilar o código e testar.

A Listagem 1 apresenta um exemplo de código a ser refatorado. Note que pode ocorrer uma exceção nas linhas 3 e 4, dependendo da entrada de dados fornecida pelo usuário. Então, é necessário escrever um bloco `catch` para cada exceção que se deseja tratar (linhas 5 a 11).

Listagem 1: Estrutura *try-catch* tradicional.

```

1 var div = new Scanner(System.in);
2 try {
3     var divisor = div.nextInt();
4     System.out.print(2 / divisor);
5 } catch (ArithmeticException e) {
6     System.out.println("Erro: " + e.getMessage());
7 } catch (IllegalArgumentException e) {
8     System.out.println("Erro: " + e.getMessage());
9 } catch (NullPointerException e) {
10    System.out.println("Erro: " + e.getMessage());
11 }

```

A Listagem 2 apresenta a estrutura refatorada usando a funcionalidade *multi-catch*, onde terá apenas um bloco `catch` com as exceções que se deseja tratar separadas pelo operador *pipe* (`|`), como apresenta o exemplo nas linhas 5 e 6.

Listagem 2: Estrutura refatorada utilizando a funcionalidade *multi-catch*.

```

1 var div = new Scanner(System.in);
2 try {
3     var divisor = div.nextInt();
4     System.out.print(2 / divisor);
5 } catch (ArithmeticException | IllegalArgumentException |
6         NullPointerException e) {
7     System.out.println("Erro: " + e.getMessage());
8 }

```

Mover recurso para o *try-with-resources*

Você abre e fecha um recurso manualmente utilizando a estrutura try-catch-finally.

◇◇◇

Utilize a funcionalidade try-with-resources.

A funcionalidade *try-with-resources*, reduz a complexidade do gerenciamento de recursos. Ela garante que os recursos declarados no bloco `try` serão fechados após terminar a execução do bloco. A mecânica é descrita da seguinte forma:

1. Escolha o recurso a ser movido.
2. Verifique se o recurso implementa a interface *AutoCloseable*. Caso implemente, siga com os próximos passos.
3. Agrupe e mova a criação e instanciação da variável que recebe o recurso para o argumento do bloco `try`.
4. Apague todas as chamadas ao método `close` dentro dos blocos da estrutura *try-catch-finally*.
5. Remova as estruturas condicionais que ficaram vazias após o passo 4.
6. Caso o bloco `finally` fique vazio, deve ser removido.
7. Compilar o código e testar.

A Listagem 3 mostra uma das formas de abrir e fechar um arquivo texto em Java. No exemplo, na linha 9, alguns problemas podem ocorrer no momento de abrir o arquivo no bloco *try*, um desses problemas é a exceção de referência nula (`NullPointerException`). Por esse motivo, é necessária a verificação realizada na linha 8, pois, se o recurso não foi aberto, a variável `arquivo` é nula.

Listagem 3: Recurso fechado manualmente.

```

1 FileReader arquivo = null;
2 try {
3     arquivo = new FileReader("texto.txt");
4     //...
5 } catch (IOException e) {
6     System.err.printf("Erro: %s.\n", e.getMessage());
7 } finally {
8     if (arquivo != null)
9         arquivo.close();
10 }
```

A Listagem 4 mostra um exemplo no qual o recurso será fechado automaticamente, usando a funcionalidade do *try-with-resources*. Cabe ressaltar que a variável criada nos argumentos do `try` (linha 1) funcionará apenas dentro do bloco.

Listagem 4: Recurso fechado automaticamente, utilizando *try-with-resources*.

```

1 try (FileReader arquivo = new FileReader("texto.txt")) {
2     //...
3 } catch (IOException e) {
4     System.err.printf("Erro: %s.\n", e.getMessage());
5 }

```

Adicionar método `map` na interação com coleções

Você realiza uma modificação em elementos de uma coleção, por meio de uma iteração externa.

◇◇◇

Utilize uma iteração interna do método `map`.

O método `map` da interface `Stream`, adicionado em Java 8, pode ser utilizado para aplicar uma modificação em cada elemento de uma coleção, desde que o método retorne o mesmo tipo dos elementos da coleção. A motivação para essa refatoração é que o método `map` permite realizar mudanças em cada elemento de uma coleção sem a necessidade de variáveis intermediárias ou laços de repetição explícitos. A mecânica é descrita da seguinte forma:

1. Copiar a declaração da variável que recebe a coleção modificada.
2. Atribuir à variável copiada uma referência à coleção que será modificada, encadeando a esta uma chamada ao método `stream`.
3. Encadear a chamada do método `map` ao método `stream`, copiar o argumento do método `add` para o método `map`.
4. Aplicar a refatoração *Convert Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014) no código copiado para a estrutura `map`. Se possível aplicar a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014).
5. Encadear com o método `collect` para converter a `stream` na coleção desejada.
6. Nos argumento do método `collect` chama a classe `Collectors` e encadear com um dos seguintes métodos de acordo com o tipo da coleção:

`toCollection` para o tipo `Collection`;

`toList` para o tipo `List`;

`toSet` para o tipo `Set`.

7. Apagar a estrutura antiga.

8. Compilar o código e testar.

O primeiro exemplo apresenta uma refatoração simples com apenas um método modificando os elementos da lista. Por exemplo, na Listagem 5 é necessário criar uma variável extra (`s`, linha 2) para acessar cada elemento da lista e uma interação explícita (`for`, linhas 2 e 3).

Listagem 5: Método convencional para modificar objetos de uma lista.

```
1 var nomesMaiusculo = new ArrayList<String>();
2 for (var s : nomes)
3     nomesMaiusculo.add(s.toUpperCase());
```

A Listagem 6 não há necessidade de uma interação explícita (apenas implícita com uso do método `map`, linha 2) para acessar os elementos da lista. Neste exemplo foi possível aplicar a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014) (linha 2).

Listagem 6: Aplicar uma modificação em elementos de uma lista utilizando o método `map`.

```
1 var nomesMaiusculo = nomes.stream()
2     .map(String::toUpperCase)
3     .collect(Collectors.toList());
```

O próximo exemplo mostra a refatoração sendo aplicada em um caso onde há outros métodos modificando a lista. Na Listagem 7 são aplicados dois métodos para modificar a lista, o `toUpperCase` (linha 3) e o `trim` (linha 4).

Listagem 7: Método convencional para modificar elementos de uma lista.

```
1 var nomesMaiusculo = new ArrayList<String>();
2 for (var s : nomes) {
3     s = s.toUpperCase();
4     s = s.trim();
5     nomesMaiusculo.add(s);
6 }
```

A Listagem 8 mostra a estrutura refatorada, onde é possível observar a utilização do método `map` (linha 2), para aplicar as modificações nos elementos da coleção e o método `collect` para transformar a `stream` em uma lista (método `toList`, linha 3). Neste exemplo não foi possível aplicar a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014) (linha 2). Porém, aplicou-se a refatoração *Convert Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014) no argumento do método `map` (linha 2)

Listagem 8: Aplicando modificações em elementos de uma lista utilizando o método `map`.

```
1 var nomesMaiusculo = nomes.stream()
2     .map(s -> s.toUpperCase().trim())
3     .collect(Collectors.toList());
```

Agrupar com `groupingBy`

Você faz um agrupamento em uma coleção.

◇◇◇

Você pode utilizar o método `groupingBy`.

O método `groupingBy` da classe `Collectors` fornece uma funcionalidade similar à cláusula `GROUP BY` da linguagem SQL. Ou seja, utilizando o método é possível agrupar objetos de acordo com alguma propriedade e armazenar os resultados em uma instância do tipo `Map`. Essa nova funcionalidade dispensa o uso de laço de repetição e variáveis extras para realizar agrupamentos, tornando um código mais enxuto e de simples compreensão. A mecânica é descrita da seguinte forma:

1. Copiar a declaração da variável do tipo `map`.
2. Fazer a variável receber a coleção que será agrupada, chamar o método `stream` e, em sequência, o método `collect`.
3. Implementar a interface `Collector` requerida pelo método `collect`, chamando o método `groupingBy` da classe `Collectors`.
4. Copiar para o primeiro argumento do método `groupingBy` o método de agrupamento.
5. Extrair a referência do método de agrupamento do passo 4, pode ser aplicada a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014), para guiar no processo da extração da referência.
6. Colocar no segundo parâmetro do método `groupingBy` o método `counting` da classe `Collectors`.
7. Excluir a estrutura antiga.
8. Compilar o código e testar.

A Listagem 9 apresenta uma das formas utilizadas para contar elementos de uma determinada lista, que possui objetos de acordo com algum atributo, antes de Java 8. Portanto, era necessário primeiro criar uma estrutura `Map` (linha 1), onde as contagens eram acumuladas e então percorrer a lista para realizar a avaliação (linha 2 a 11). As contagens são realizadas adicionando à estrutura `Map` o nome de acordo com o atributo que se deseja realizar a contagem (podendo ser obtido através de algum método `get`), junto com uma variável contadora, como observado na estrutura condicional da linha 5 a 10.

Listagem 9: Forma convencional para fazer contagem de objetos.

```

1 var tipo = new HashMap<String, Long>();
2 for (var p : pessoaList) {
3     var tipoPessoa = p.getPessoaTipo();
4     var pessoasPorTipo = tipo.get(tipoPessoa);
5     if (pessoasPorTipo == null) {
6         pessoasPorTipo = Long.valueOf(1);
7         tipo.put(tipoPessoa, pessoasPorTipo);
8     }
9     else
10        tipo.put(tipoPessoa, ++pessoasPorTipo);
11 }

```

A Listagem 10 mostra a estrutura refatorada utilizando método `groupingBy` da classe `Collectors`. A linha 2 apresenta o primeiro parâmetro do método `groupingBy`, que é o atributo utilizado para a contagem dos elementos, obtido através do método `getPessoaTipo`. O segundo parâmetro (linha 3) é a operação realizada, que no caso do exemplo é uma contagem (`counting`).

Listagem 10: Contagem de objetos utilizando o método `groupingBy`.

```

1 var tipo = pessoaList.stream().collect(
2     Collectors.groupingBy(Pessoa::getPessoaTipo,
3         Collectors.counting()));

```

Converter para uma operação aritmética lambda

Você realiza uma operação aritmética em todos os elementos de uma coleção.

◇◇◇

Você pode realizá-la aplicando o método `reduce`.

O método `reduce`, adicionado em Java 8, realiza operações de reduções, onde combina todos os elementos de uma coleção em um único resultado. Essa refatoração pode reduzir o tamanho do código, é compatível com execução paralela e não necessita a criação de variáveis extras para acumular valores, ou laços de repetição para acessar cada elemento. A mecânica é descrita da seguinte forma:

1. Identificar e copiar a declaração da variável que acumula o resultado da operação.
2. Fazer a variável receber a coleção onde é feita a operação aritmética e chamar a interface `stream`.
3. Se for uma coleção de objetos não numéricos, o qual obtém os valores através da chamada a um método `get`, chamar o método:

`mapToInt`, se o método retorna um valor do tipo `int`;

`mapToFloat`, se o método retorna um valor do tipo `float`;

`mapToDouble`, se o método retorna um valor do tipo `double`;

4. Colocar no argumento do método criado no passo 3 a referência ao método utilizado para a obtenção dos elementos. Caso necessário é possível utilizar a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014).
5. Chamar o método `reduce`.
6. Copiar o valor inicial da variável acumuladora e colocar como primeiro argumento do método `reduce`.
7. Criar uma expressão lambda com dois parâmetros e realizar, no corpo da expressão, o cálculo aritmético entre eles, de acordo com a operação aritmética da estrutura antiga.
8. Apagar as estruturas antigas.
9. Compilar o código e testar.

Uma das formas de somar elementos de uma lista é combinar cada elemento de forma iterativa usando o operador de soma para gerar um resultado. A Listagem 11 mostra um exemplo da soma de uma coleção com tipos primitivos e numéricos. Esse tipo de estrutura necessita da criação de duas variáveis, uma para receber a soma (linha 1), onde é inicializada com o valor

0, e outra para receber o valor de cada elemento da lista (variável `x`, linha 2). Após isso o laço de repetição acessa cada elemento da lista para obter o resultado (linhas 2 e 3).

Listagem 11: Somar valores de uma coleção com tipos primitivos e numéricos.

```
1 var soma = 0;
2 for (var x : numeros)
3     soma = soma + x;
```

Utilizando o método `reduce` (linha 2), declara-se apenas a variável para receber o resultado da operação (linha 1), como pode ser visto na Listagem 12.

Listagem 12: Soma de valores de uma coleção com tipos primitivos e numéricos utilizando o método `reduce`.

```
1 var soma = numeros.stream()
2     .reduce(0, (a, b) -> a + b);
```

As Listagens 13 e 14 mostram a aplicação da refatoração em uma lista de objetos, onde a operação aritmética é realizada de acordo com algum atributo do objeto. A Listagem 13 utiliza, além do laço de repetição (linhas 2 e 3), o método `getIdade` (linha 3) para obter os valores utilizados na operação.

Listagem 13: Forma convencional para somar valores de uma coleção de objetos.

```
1 var soma = 0;
2 for (var p : list)
3     soma = soma + p.getIdade();
```

Na Listagem 14 foi aplicada a refatoração. O método `mapToInt` (linha 2) foi utilizado para extrair os valores da operação através da referência ao método `getIdade` (linha 2).

Listagem 14: Soma de valores de uma coleção de objetos utilizando o método `reduce`.

```
1 var soma = list.stream()
2     .mapToInt(Pessoa::getIdade)
3     .reduce(0, (a, b) -> a + b);
```

Converter `if` para `ifPresent`

Você verifica se um objeto existe fazendo uma comparação com o valor `null`.

◇◇◇

Você pode utilizar o método `ifPresent` da classe `Optional`.

`Optional` é uma classe, adicionada em Java 8, que encapsula vários métodos para tratar blocos de código crítico que necessitam ser verificados. A classe `Optional` força o programador a analisar tratamento de valores ausentes. A classe possui o método `ifPresent`, que elimina a necessidade de comparar com o valor `null` explicitamente. A mecânica é descrita da seguinte forma:

1. Copiar a declaração da variável a qual é realizada a comparação com o valor `null`, caso seja Java 10, substituir o tipo da declaração por `var`, caso contrário encapsular o tipo com a classe `Optional`.
2. Fazer a variável receber o método `ofNullable` da classe `Optional`.
3. No argumento do método `ofNullable`, copiar o valor que é recebido pela variável na estrutura anterior.
4. Em substituição a estrutura `if`, chamar o método `ifPresent` da variável criada.
5. Implementar a interface funcional `Consumer`, requerida pelo método `ifPresent`, movendo o conteúdo da estrutura condicional para o método da interface `Consumer`.
6. Aplicar a refatoração *Convert Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014) para a instância da interface `Consumer`.
7. Apagar as estruturas antigas.
8. Compilar o código e testar.

A Listagem 15 apresenta como uma verificação de objetos nulos era estruturada antes de Java 8. Anteriormente, era necessário utilizar uma estrutura de repetição (`if`, linha 2) e fazer a verificação se o objeto existe, comparando-o explicitamente com um valor `null` (linha 2).

Listagem 15: Forma convencional para verificação de valores nulos.

```

1 var pessoa = getPessoa(nome);
2 if (pessoa != null)
3     System.out.println("Nome completo: " + pessoa.getName());

```

A Listagem 16 apresenta os objetos encapsulados com a classe `Optional` (linha 1). Agora é possível utilizar métodos específicos para verificação de valores nulos (como o `ifPresent`, linha 2), diminuindo os códigos duplicados e, como ganho, aumentando a segurança no tratamento desses valores, pois essa classe foi adicionada com esse objetivo.

Listagem 16: Verificação de valores nulos utilizando a classe `Optional`.

```

1 var pessoa = Optional.ofNullable(getPessoa(nome));
2 pessoa.ifPresent(p -> System.out.println("Nome completo: "
3     + p.getName()));

```

Converter `iterator.remove` para `removeIf`

Você remove elementos de uma coleção utilizando a interface `Iterator`.

◇◇◇

Você pode utilizar o método `removeIf` da própria coleção.

Em Java 8 foi adicionado o método `removeIf` na interface `Collection`. A utilização dele pode diminuir a complexidade de remover elementos de uma coleção. Utilizando a abordagem anterior, com o uso da interface `Iterator`, é necessário implementar um laço de repetição e uma estrutura condicional. Atualmente, é possível usar o método `removeIf` passando a condição para remoção do elemento. Para uma coleção com grandes quantidades de dados esse método é mais conciso e eficiente, quando comparado com a abordagem `Iterator` (ORACLE, 2017b). A mecânica é descrita da seguinte forma:

1. Copiar a declaração que representa a coleção que se deseja remover o elemento e chamar o método `removeIf` da própria coleção.
2. Criar uma expressão lambda nos argumentos do método `removeIf`, passando no parâmetro da expressão a variável que recebe o valor corrente.
3. Copiar para o corpo da expressão lambda a condição de remoção, para auxiliar nos passos 2 e 3 é possível utilizar a refatoração *Convert Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014).
4. Se necessário, chamar o método para obtenção dos dados no corpo da expressão lambda.
5. Apagar as estruturas antigas.
6. Compilar o código e testar.

A Listagem 17 apresenta uma forma de remover um elemento de uma coleção, antes de Java 8. Anteriormente era necessário criar uma estrutura de repetição (`for`, linha 1), para poder percorrer todos os elementos dessa coleção, utilizar a interface `Iterator` (linha 1),

para realizar o acesso a cada elemento e uma estrutura condicional para implementar a remoção (através do método `remove`, linha 4).

Listagem 17: Remoção de elementos utilizando a interface `Iterator`.

```
1 for (var i = list.iterator(); i.hasNext();) {
2     var person = i.next();
3     if ("Peter".equals(person.getName()))
4         i.remove();
5 }
```

Com o método `removeIf` (linha 1) basta colocar a condição de remoção, como é possível observar na estrutura refatorada da Listagem 18.

Listagem 18: Remoção de elemento utilizando o método `removeIf`.

```
1 list.removeIf(person -> "Peter".equals(person.getName()));
```

Converter `iterator.set` para `replaceAll`

Você aplica um método em cada elemento de uma coleção.

◇◇◇

Você pode utilizar o método `replaceAll` da própria coleção.

Em Java 8 foi implementado um método que realiza uma ação em todos os elementos de uma coleção, chamado `replaceAll`. Uma consideração a ser feita é que a ação realizada por esse método não pode alterar o tipo dos elementos da coleção. Essa refatoração elimina o uso de `iterator` e laço de repetição para percorrer a lista, o que pode diminuir a complexidade e a quantidade de linhas do código. A mecânica é descrita da seguinte forma:

1. Copiar a declaração que representa a coleção que se deseja alterar e chamar o método `replaceAll` da própria coleção.
2. Criar uma expressão lambda com um parâmetro.
3. Copiar o argumento do método `set` para o método `replaceAll`.
4. Substituir o método `next`, juntamente com a variável que o chama, pelo parâmetro da expressão lambda, se necessário.
5. Apagar as estruturas antigas.

6. Compilar o código e testar.

A Listagem 19 apresenta uma forma de converter elementos de uma lista de *Strings* para letra maiúscula, antes de Java 8. Era necessário percorrer essa lista com um laço de repetição (linha 1), utilizando a interface `Iterator`, por exemplo, para acessar cada um dos elementos e modificar com o método `set` (linha 2).

Listagem 19: Transformar elementos de uma lista utilizando a interface `Iterator`.

```
1 for (var i = list.listIterator(); i.hasNext();)
2     i.set(i.next().toUpperCase());
```

Atualmente é possível aplicar ações em todos os elementos da lista utilizando o método `replaceAll` (linha 1) da própria coleção, como pode ser observado na estrutura refatorada da Listagem 20.

Listagem 20: Transformar elementos de uma lista utilizando o método `replaceAll`.

```
1 list.replaceAll(s -> s.toUpperCase());
```

Concatenar uma coleção com `join`

Você concatena elementos da coleção em uma única variável.

◇◇◇

Você pode utilizar o método `join` da classe `String`.

O método `join` foi adicionado na classe `String` em Java 8. O método concatena um conjunto de valores em uma única `String`, com os elementos separados por algum delimitador. Essa refatoração elimina o uso de laço de repetição para percorrer os elementos de uma coleção, é compatível com os novos métodos e expressões lambdas adicionados recentemente, possibilitando diminuir a duplicidade, o número de linhas e a complexidade do código. A mecânica é descrita da seguinte forma:

1. Copiar a declaração e inicialização da variável.
2. Se a variável é inicializada com algum valor, concatenar o valor ao método `join`, caso contrário chamar a classe `String` juntamente com o método `join`.
3. Copiar o delimitador para o primeiro argumento do método `join`.

4. Se a coleção for do tipo `String`, copiar a coleção para o segundo argumento do método e pular para o passo 6.

5. Se a coleção for de outro tipo básico, no segundo argumento do método `join`:

Copiar a coleção que será concatenada, chamar o método `stream`, encadear com método `map`.

Nos argumentos do método `map` criar uma referência ao método `toString`.

Encadear o método `map` com o `collect`, para converter a `stream` na coleção desejada.

Nos argumento do método `collect` chamar a classe `Collectors` e encadear com um dos seguintes métodos de acordo com o tipo da coleção:

`toCollection` para o tipo `Collection`.

`toList` para o tipo `List`.

`toSet` para o tipo `Set`.

6. Se necessário, encadear a função `concat` e copiar o delimitador em seu argumento para manter o mesmo resultado da estrutura anterior.

7. Apagar as estruturas antigas.

8. Compilar o código e testar.

O primeiro exemplo mostra a utilização da refatoração na concatenação de uma lista de *strings*. Na Listagem 21 é feita uma concatenação utilizando um laço de repetição (linha 2 à 5) e operadores (linhas 3 e 4).

Listagem 21: Forma anterior de concatenar uma lista de *strings*.

```
1 var tex = new String();
2 for (var s : lista) {
3     tex += s;
4     tex += " ";
5 }
```

A Listagem 22 apresenta a estrutura refatorada, utilizando o método `join` (linha 1) é possível obter a mesma funcionalidade da estrutura anterior, sem uso de uma interação externa. A utilização do método `concat` (linha 1) foi necessária para obter o mesmo resultado que a estrutura anterior.

Listagem 22: Concatenar uma lista de strings utilizando o método `join`.

```
1 var tex = String.join(" ", lista).concat(" ")
```

O próximo exemplo apresenta a utilização da refatoração na concatenação de uma lista com elementos não *strings*. Na Listagem 23 a concatenação é feita utilizando um laço de repetição (linha 2) e operadores (linhas 3 e 4).

Listagem 23: Forma convencional de concatenar uma lista de inteiros

```
1 var tex = "[ ";
2 for (var s : lista) {
3     tex += s;
4     tex += " ";
5 }
```

A listagem 24 apresenta a estrutura refatorada. Com a nova estrutura é possível fazer a mesma concatenação utilizando o método `join` (linha 1). Porém, é necessário converter a lista para *strings*, o que pode ser feito utilizando o método `map` e passando a referência do método `toString` para retornar uma lista com os números convertidos de `int` para `String` (linha 2). Neste exemplo, foi necessária a utilização do método `concat`.

Listagem 24: Concatenar uma lista de inteiros utilizando o método `join` da classe `String`

```
1 var tex = "[ ".join(" ", lista.stream()
2     .map(Object::toString)
3     .collect(Collectors.toList())) .concat(" ");
```

Converter interface `Iterator` para `Spliterator`

Você utiliza a interface `Iterator` para percorrer uma coleção.

◇◇◇

Atualize para a interface `Spliterator`.

`Spliterator`, adicionada em Java 8, é um tipo especial de `Iterator` utilizado para percorrer e particionar coleções. Essa nova interface é projetada para trabalhar com execução paralela e aceitar as expressões lambda. Ela possui métodos que combinam os métodos `hasNext` e `next` da interface `Iterator`. A mecânica é descrita da seguinte forma:

1. Copiar a declaração e instanciação da variável alterando a interface `Iterator` pela interface `Spliterator`, quando necessário, e a chamada do método `iterator` pelo método `spliterator` da coleção.

2. Copiar o laço de repetição trocando o método `hasNext` da interface `Iterator` pelo método `tryAdvance` da interface `Spliterator`.
3. Criar uma expressão lambda de um parâmetro para a interface `Consumer` do método `tryAdvance`, pode ser usada a refatoração *Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014) para auxiliar nesse processo.
4. Mover para o corpo da expressão lambda do passo 3 o conteúdo do laço de repetição.
5. Trocar as ocorrências da variável que chama o método `next`, juntamente com o método, pelo argumento da expressão lambda gerada no passo anterior.
6. Apagar as estruturas antigas.
7. Compilar o código e testar.

A Listagem 25 apresenta uma interação com a lista feita através da interface `Iterator`, usando o método `hasNext` (linha 2) para testar se a lista possui mais elementos e o método `next` (linha 3) para receber o elemento da interação, em conjunto com o laço de repetição `while`. Sendo que o código ainda não foi atualizado para Java 10.

Listagem 25: Iteração utilizando a interface `Iterator` e laço de repetição `while`.

```
1 Iterator<String> is = nomeLista.iterator();
2 while (is.hasNext())
3     System.out.println(is.next());
```

A Listagem 26 apresenta a estrutura refatorada utilizando a interface `Spliterator` (linha 1). O método `tryAdvance` (linha 2) realiza os testes e interage com a lista sem necessidade de outro método, obtendo, assim, vantagens com o uso de expressões lambdas.

Listagem 26: Iteração utilizando a interface `Spliterator` e laço de repetição `while`.

```
1 Spliterator<String> is = nomeLista.spliterator();
2 while (is.tryAdvance(p -> System.out.println(p)));
```

A refatoração obterá o mesmo resultado independentemente do laço de repetição utilizado. No próximo exemplo, a Listagem 27 mostra a refatoração sendo aplicada em um código que utiliza o laço de repetição `for` (linha 1). O exemplo apresenta um código já atualizado para Java 10.

Listagem 27: Iteração utilizando a interface `Iterator` e laço de repetição `for`.

```
1 for (var i = nomeLista.iterator(); i.hasNext());
2     System.out.println(i.next());
```

A Listagem 28 mostra como ficará essa estrutura após ser refatorada, sendo utilizado método `tryAdvance` (linha 2) da classe `Spliterator`.

Listagem 28: Iteração utilizando a interface `Spliterator` e laço de repetição `for`.

```
1 for (var i = nomeLista.spliterator();
2     i.tryAdvance(e -> System.out.println(e)));
```

Adicionar interface `Predicate`

Você faz pequenos testes similares no decorrer de um método.

◇◇◇

Você pode atualizar para a interface `Predicate`.

`Predicate` é uma interface funcional, adicionada em Java 8, que pode ser utilizada para avaliar uma determinada condição, retornando um valor booleano de acordo com o teste realizado. Essa refatoração pode ser utilizada quando existem estruturas condicionais que fazem testes similares. Algumas vantagens são obtidas ao aplicar essa refatoração: tratar as condições em um ponto central, evitar duplicidade do código e compatibilidade com expressões lambdas. A mecânica é descrita da seguinte forma:

1. Identificar, no corpo de um método, expressões booleanas semelhantes em estruturas condicionais, se for encontrada uma expressão que se repita continuar no próximo passo.
2. Identificar a variável que mais se repete na expressão booleana identificada no passo 1.
3. Identificar o tipo da variável do passo 2, declarar uma nova variável encapsulando seu tipo com a interface funcional `Predicate`.
4. Inicializar a variável criada no passo 3 com uma expressão lambda, inserindo um parâmetro qualquer.
5. Copiar para o corpo da expressão lambda criada no passo 4 uma das expressões booleanas identificadas no passo 1, trocando a variável identificada pelo parâmetro criado na expressão lambda.

6. Trocar todas as expressões booleanas identificadas no passo 1 pela variável criada no passo 3, encadeando com método `test` e usar como argumento do método a variável identificada no passo 2.
7. Compilar o código e testar.

Muitas vezes os programadores fazem verificações condicionais pequenas diretamente na estrutura `if`, porém se alterar alguma das condições verificadas, provavelmente deverá ser feita a mesma alteração em todas as estruturas que realizam essa verificação. Na Listagem 29, por exemplo, se alguma condição alterar é necessário modificar as linhas 1 e 5.

Listagem 29: Forma convencional para fazer verificação de estruturas *ifs*.

```

1  if (idade > 18 && idade < 36) {
2      //...
3  }
4  //...
5  if (idade > 18 && idade < 36) {
6      //...
7  }
8  //...
```

A Listagem 30 apresenta a estrutura após a aplicação da refatoração. A variável que mais se repete na estrutura anterior é a variável `idade`, que é uma variável do tipo `int`. Por esse motivo a variável é criada e encapsulada com a interface `Predicate`, utilizando como argumento a classe `Integer` (linha 1). Usando a interface `Predicate` podemos centralizar essas verificações diretamente na interface. Por exemplo, se alguma condição for alterada, basta modificar a linha 1 na Listagem 30.

Listagem 30: Verificar condições em estruturas *ifs* utilizando a interface `Predicate`.

```

1  Predicate<Integer> testeIdade = e -> e > 18 && e < 36;
2  if (testeIdade.test(idade)) {
3      //...
4  }
5  //...
6  if (testeIdade.test(idade)) {
7      //...
8  }
```

Converter para média lambda

Você realiza uma média em uma coleção.

◇◇◇

Faça a média utilizando uma expressão lambda.

Esta refatoração é vantajosa por dar suporte a operações paralelas e diminuir o número de variáveis intermediárias na realização de uma média. Além disso, não é necessário o uso de laço de repetição para acessar cada elemento da coleção, o que, em alguns casos, diminui a complexidade do código. A mecânica é descrita da seguinte forma:

1. Copiar a declaração da variável que recebe a média e atribuir à coleção que será realizado o cálculo, chamando o método `stream` da própria coleção.
2. Encadear o método `mapToDouble` ao `stream` do passo 1.
3. No argumento do método `mapToDouble` criar uma expressão lambda com um parâmetro e no corpo da expressão colocar o próprio parâmetro.
4. Caso exista algum método para obter o valor do cálculo, encadeá-lo ao corpo da expressão lambda do passo 3.
5. Encadear o método `average` ao `mapToDouble`, seguido do método `getAsDouble`.
6. Apagar as estruturas antigas.
7. Compilar o código e testar.

No primeiro exemplo, as Listagens 31 e 32, apresentam a aplicação da refatoração em uma coleção com elementos numéricos e primitivos em Java. É possível verificar que a Listagem 31 está realizando uma média, pois há uma variável que recebe o valor inicial de zero (linha 1) e após isso, acumula os valores de uma coleção (linha 3) e divide-os pelo tamanho total de elementos da coleção (linha 4).

Listagem 31: Forma de realizar a média em uma coleção com tipos numéricos e primitivos.

```
1 var soma = 0;
2 for (var i : listaInt)
3     soma += i;
4 var media = soma/listaInt.size();
```


A Listagem 32 apresenta a estrutura refatorada realizando o cálculo da média com métodos adicionados em Java 8. O método `mapToDouble` (linha 2) irá obter os elementos como números do tipo `double`. Após isso, o método `average` (linha 3) irá realizar a média e o método `getAsDouble` (linha 4) irá transformar a média em um elemento do tipo `double`.

Listagem 32: Realizar a média utilizando operação lambda e métodos funcionais em uma coleção com tipos numéricos e primitivos.

```
1 var media = listaInt.stream()
2     .mapToDouble(i -> i)
3     .average()
4     .getAsDouble();
```

As Listagens 33 e 34 apresentam um exemplo de utilização dessa refatoração em uma coleção de objetos. A Listagem 33 apresenta a realização da média em uma coleção de objetos (`list`, linha 2), sendo necessário chamar o método `getIdade` (linha 3) para se obter os valores numéricos para a realização do cálculo.

Listagem 33: Forma convencional realizar a média em uma coleção de objetos.

```
1 var soma = 0;
2 for (var p : list)
3     soma += p.getIdade();
4 var media = soma / list.size();
```

Uma estrutura como na Listagem 33 está prevista na refatoração. A Listagem 34 apresenta a estrutura refatorada, onde o método de obtenção dos valores para realizar a média (`getIdade`) é colocado na função lambda do método `mapToDouble` (linha 2).

Listagem 34: Realizar a média utilizando operação lambda e métodos funcionais em uma coleção de objetos.

```
1 var media = list.stream()
2     .mapToDouble(p -> p.getIdade())
3     .average()
4     .getAsDouble();
```

Converter para contagem lambda

Você realiza uma contagem de elementos de uma coleção de acordo com alguma característica.

◇ ◇ ◇

Você pode realizar essa contagem utilizando expressões lambda.

Algumas vezes, é necessário realizar contagens de objetos de acordo com um determinado atributo. Nesses casos esta refatoração pode ser vantajosa, pois as novas estruturas a partir de Java 8 possuem suporte para operações paralelas e expressões lambda, o que pode melhorar o desempenho final. A mecânica é descrita da seguinte forma:

1. Copiar a variável que recebe a contagem e atribuir à coleção que é realizado o cálculo, chamando o método `stream` da própria coleção.
2. Encadear o método `filter` ao `stream` do passo 1.
3. Criar uma expressão lambda com um parâmetro no argumento do método `filter` do passo 2.
4. Passar para o corpo da expressão lambda a expressão booleana da condição para a contagem, substituído a variável do laço de repetição pelo argumento da expressão lambda.
5. Encadear o método `mapToInt` ao `filter` do passo 2, inserindo no argumento do método uma expressão booleana com um parâmetro e no corpo o valor inteiro 1.
6. Encadear o método `sum` ao `mapToInt` do passo 5.
7. Apagar as estruturas antigas.
8. Compilar o código e testar.

Observando a Listagem 35, nota-se a realização da contagem de uma lista de objetos de acordo com algum atributo. Por esse motivo, é utilizado um laço de repetição (linha 2) em conjunto com uma estrutura condicional (linha 3). Além disso, é necessário o uso de uma variável (linha 1) para acumular quantos objetos com determinada parâmetro existem na lista.

Listagem 35: Forma convencional de realizar a contagem de uma lista de objeto de acordo com algum atributo.

```

1 var count = 0;
2 for (var p : list)
3     if (p.getGenero().equals("Feminino"))
4         count++;

```

A Listagem 36 apresenta a estrutura refatorada utilizando os métodos funcionais: `filter` (linha 2), para selecionar os elementos que serão contados, `mapToInt` (linha 3) para retornar o valor numérico 1 para cada objeto encontrado e `sum` (linha 4), para realizar a contagem.

Listagem 36: Realizar a contagem de uma lista de objetos utilizando métodos funcionais e expressões Lambda.

```
1 var count = list.stream()
2     .filter(p -> p.getGenero().equals("Feminino"))
3     .mapToInt(p -> 1)
4     .sum();
```

Adicionar método `of`

Você cria uma coleção imutável através de um método `unmodifiable`.

◇◇◇

Você pode utilizar o método `of` para isso.

Java 9 apresentou alguns métodos novos, alguns deles permitem a criação de coleções imutáveis de forma rápida e simples. Em versões anteriores, para criar uma coleção pequena e não modificável, era necessária a construção de uma variável local, a adição dos valores à coleção e a chamada ao método específico para transformar essa coleção em não modificável. Esta refatoração pode diminuir o número de linhas e tornar mais simples a criação de coleções não modificáveis. A mecânica é descrita da seguinte forma:

1. Se os elementos foram adicionados manualmente:

Copiar a declaração da variável e atribuir à interface da própria coleção chamando o método `of`.

Identificar os elementos que foram adicionados manualmente e acrescentá-los nos argumentos do método `of`.

Apagar as estruturas anteriores.

2. Se os elementos foram adicionados através de uma estrutura de repetição:

Substituir a estrutura `Collections.unmodifiable + nome da coleção` pela interface da mesma coleção chamando o método `of`.

3. Compilar o código e testar.

A Listagem 37 mostra um exemplo da criação de uma coleção do tipo lista (linha 1), não modificável (`unmodifiableList`, linha 5), em Java. Onde é realizada a adição manual dos valores através do método `add` (linha 2 à 4).

Listagem 37: Forma convencional para criação de listas não modificáveis.

```
1 List<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 list = Collections.unmodifiableList(list);
```

A Listagem 38 mostra como ficaria essa estrutura após refatorada, utilizando o método `of` (linha 1), nativo em Java 9.

Listagem 38: Criação de listas não modificáveis utilizando o método `of`.

```
1 List<String> list = List.of("a", "b", "c");
```

A Listagem 39 apresenta um exemplo da criação de uma coleção do tipo lista (linha 1), não modificável (`unmodifiableList`, linha 5) e obtendo os valores dinamicamente através de um laço de repetição (`for`, linhas 2 e 3).

Listagem 39: Forma convencional para criação de listas não modificáveis.

```
1 var list = new ArrayList<>();
2 for (var i = 0; i < 3; i++)
3     list.add(i);
4 list = Collections.unmodifiableList(list);
```

A Listagem 40 exhibe a estrutura refatorada, utilizando o método `of` (linha 4). Sendo um método presente na `Interface List` (linha 4). Porém, outras interfaces possuem esse método também, por exemplo a interface `Set` e `Map`.

Listagem 40: Criação de listas não modificáveis utilizando o método `of`.

```
1 var list = new ArrayList<>();
2 for (var i = 0; i < 3; i++)
3     list.add(i);
4 list = List.of(list);
```

Converter para *underscore*

Você possui uma expressão lambda em que alguns parâmetros não são utilizados no corpo da expressão.

◇◇◇

*Você pode mudar o nome desses parâmetros para o caractere *underscore*.*

Em algumas expressões lambdas pode existir a necessidade de criar vários parâmetros, mesmo que o corpo não use todos eles. Isso força o desenvolvedor usar nomes indicativos para todos os parâmetros, embora não sejam utilizados. Em Java 10 pode ser usado o caractere sublinhado (*underscore*) para tais parâmetros. A mecânica é descrita da seguinte forma:

1. Identificar no corpo da expressão lambda o(s) parâmetro(s) que não é(são) utilizado(s).
2. Trocar o nome dos parâmetros não utilizados no argumento da expressão lambda pelo caractere sublinhado (*underscore*).
3. Compilar o código e testar.

A Listagem 41 mostra um exemplo de um código com uma estrutura `Map<String, Double>`, a qual possui nomes de pessoas com seus respectivos salários. O desenvolvedor necessita fazer um cálculo para aumentar o salário em 10%, para isso basta multiplicar o valor salário por "1.1". Como é uma estrutura `Map`, tanto o *key* (`String`) quanto o *value* (`Double`) devem ir nos parâmetros da expressão lambda. Caso for apenas um parâmetro, a expressão considerará apenas o primeiro valor, que seria uma `String`, com o nome da pessoa, e certamente causaria um erro ao tentar realizar uma expressão aritmética, por esse motivo, ambos os parâmetros necessitam ir nos argumentos da expressão lambda.

Listagem 41: Forma convencional de declarar parâmetros não utilizados em uma expressão lambda.

```
1 salarios.replaceAll((nome, salario) -> salario * 1.1);
```

A Listagem 42 exhibe um exemplo de como o código ficaria após a refatoração, onde o argumento não utilizado no corpo da expressão foi suprimido pelo caractere *underscore*.

Listagem 42: Forma de declarar parâmetros não utilizados em expressão lambda em Java 10.

```
1 salarios.replaceAll(_, salario) -> salario * 1.1);
```

3.2 FUNÇÕES DE DETECÇÃO

Esta seção apresenta algumas funções de detecção para a busca de oportunidades de refatoração para evolução de códigos apresentadas na Seção 3.1. Cada função de detecção é uma

expressão, baseada ou não em métricas, que indica certas características dos artefatos avaliados. A aplicação das funções de detecção retorna trechos de código que provavelmente podem ser atualizados. Porém, a aplicação da refatoração recomendada fica a cargo do desenvolvedor. O benefício do uso das funções de detecção desenvolvidas é indicar trechos de código que podem ser atualizados para usufruir de novas funcionalidades adicionadas na linguagem. Cada função possui um nome, uma breve descrição textual e uma definição formal. A implementação das funções são apresentadas no Capítulo 4. As funções de detecção apresentadas nesta seção são:

- Detecção de blocos *catch* similares
- Detecção de estruturas que implementam a interface `AutoCloseable`
- Detecção da chamada ao método `add` em laços de repetição
- Detecção de estruturas do tipo `Map<String, Long>`
- Detecção de operações aritméticas
- Detecção de comparações nulas
- Detecção de chamadas ao método `remove`
- Detecção de chamadas ao método `set`
- Detecção de concatenações
- Detecção de declarações do tipo `Iterator`
- Detecção de *ifs* com as mesmas expressões de comparação
- Detecção de médias aritméticas
- Detecção de contagens de objetos
- Detecção de chamadas aos métodos *unmodifiable*
- Detecção de parâmetros não utilizados em expressões lambdas

Detecção de blocos *catch* similares

Esta função de detecção tem como objetivo encontrar lugares para a aplicação da refatoração *Agrupar blocos catch*. Refatoração aplicada em estruturas *try-catch*, especificamente em

blocos *catch* que tenham certo grau de semelhança. Sendo assim, é necessário que haja uma comparação de similaridade entre os trechos de código de tratamento das estruturas *catch*. Esta comparação pode ser feita utilizando uma função de similaridade.

Definição: Considere A como sendo uma construção *try*, B como sendo uma lista dos blocos *catch* da estrutura A e $f(x, y)$ uma função de similaridade. Onde x e y representam blocos *catch* passados para realizar a comparação, que retorna um número real entre 0 e 1, onde quanto mais próximo de 1, mais similares são os blocos comparados. A constante c , definida pelo usuário, de acordo com o nível de similaridade pretendido. Pode-se dizer que existe uma oportunidade de aplicar a refatoração Agrupar blocos *catch* em A , agrupando os blocos b e b' se $\exists b \in B, \exists b' \in B \mid b \neq b' \wedge f(b, b') \geq c$.

Detecção de estruturas que implementam a interface `AutoCloseable`

Esta função de detecção encontra oportunidades para a aplicação da refatoração *Mover recurso para o try-with-resources*. Todo o recurso aberto dentro de um *try* deve implementar a interface `AutoCloseable` para poder utilizar a funcionalidade *try-with-resources* (ORACLE, 2017a). Considerando isso, um bloco *try* só poderá utilizar esta refatoração se o recurso aberto em sua estrutura implementa a interface `AutoCloseable`. Uma das formas de encontrar essa oportunidade de refatoração é fazer a comparação dos recursos abertos no bloco *try* com os recursos que suportam essa funcionalidade.

Definição: Considere $\text{implementsAutoCloseable}(x)$ como uma função que verifica se a classe x implementa a interface `AutoCloseable`, A como um conjunto das classes dos recursos abertos dentro de uma estrutura *try* e a como uma classe qualquer. Pode-se dizer que existe uma oportunidade de aplicar a refatoração *Mover recurso para o try-with-resources* em A , se $\exists a \in A \mid \text{implementsAutoCloseable}(a)$ seja verdadeiro

Detecção da chamada ao método `add` em laços de repetição

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Adicionar método `map` na interação com coleções*. A refatoração é aplicada para trechos de código que modificam uma lista de objetos. Porém, o resultado são duas listas, uma com os objetos modificados e a outra não. Então, uma provável definição para esta função de

detecção é procurar pelo método `add` dentro de laços de repetição, onde a variável de interação do laço esteja contida no método.

Definição: Considere A uma lista com todas as expressões que chamam um método em uma estrutura de repetição, a como uma expressão, x a variável de interação do laço e y outra lista qualquer, é possível que haja uma oportunidade de aplicar a refatoração Adicionar método `map` na interação com coleções em A , se $\exists a \in A \mid a = y.add(x)$ for verdadeiro.

Detecção de estruturas do tipo `Map<String, Long>`

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Agrupar com `groupBy`*. A estrutura `groupBy` proposta pela refatoração sempre retorna uma variável do tipo `Map<String, Long>`. Então, aplica-se essa refatoração se já existe uma variável com esse tipo no código. O objetivo principal dessa funcionalidade é realizar uma contagem de objetos, o que era feito através de um laço de repetição em Java.

Definição: Considere A como uma lista com todas as declarações de variáveis de um código, a uma declaração de variável, x o tipo `String` e y o tipo `Long`, $tipoDe(z)$ uma função que retorna o tipo de uma variável e $subInterface(z)$ se é uma sub-interface de Z , pode existir uma oportunidade de aplicar a refatoração *Agrupar com `groupBy`* em a , se $\exists a \in A \mid tipoDe(a) = Map<x,y> \vee subInterface(Map<x,y>)$.

Detecção de operações aritméticas

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Converter para uma operação aritmética `lambda`*. Existem duas formas principais de realizar operações aritméticas em Java, com os operadores de atribuição (`+=`, `-=`, `*=` ou `/=`) e repetindo o valor que recebe o cálculo (`y = y + x`, `y = y - x`, `y = y * x` ou `y = y / x`). Todas as formas devem estar previstas em uma função de detecção. Dependendo da linguagem de programação, talvez existam outras. Uma maneira de procurar por oportunidades desta refatoração é encontrar essas operações dentro de laços de repetição.

Definição: Considere A como uma lista com todas as operações aritméticas de um laço de repetição, a como uma variável que recebe uma operação aritmética, x como a variável de

interação do laço e y como uma variável numérica qualquer, considere op como o número de operações aritméticas possíveis na linguagem, pode existir uma oportunidade de aplicar a refatoração Converter para uma operação aritmética λ se $\exists a \in A | a = (y \text{ op } x)$ for verdadeira.

Detecção de comparações nulas

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração Converter *if* para *ifPresent*. A condição principal desta função de detecção é encontrar estruturas condicionais que façam comparações com valores nulos. Então uma possível definição é encontrar o valor `null` nas expressões condicionais da estrutura *if*.

Definição: Considere A uma lista com tokens de uma expressão lógica em uma estrutura condicional *if*, a como um token contido em A , pode existir uma oportunidade de aplicar a refatoração Converter *if* para *ifPresent* em A , se $\exists a \in A | a = \text{null}$ for verdadeiro.

Detecção de chamadas ao método `remove`

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração Converter *iterator.remove* para *removeIf*. Uma das formas de realizar essa função é buscar diretamente pelo método `remove`, que em Java é o método utilizado para remover elementos de uma coleção. Então, dentro de um laço de repetição é verificada a existência de uma estrutura condicional. Se a variável de interação do laço chama o método `remove` dentro da estrutura condicional é uma possível oportunidade para refatoração.

Definição: Considere A como uma lista de expressões que chamam um método em uma estrutura condicional, a como uma expressão qualquer e x a variável de interação de um laço de repetição, pode existir uma oportunidade de aplicar a refatoração Converter *iterator.remove* para *removeIf* em A , se $\exists a \in A | a = x.remove()$ seja verdadeiro.

Detecção de chamadas ao método `set`

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração Converter *iterator.set* para *replaceAll*. A principal característica dessa função de detecção é procurar por chamadas ao método `set`, que é utilizado para substituir elementos de uma lista. Então esta função deve procurar por laços de repetição que usem esse

método em seu corpo. Uma das formas de fazer isso é encontrar os laços de repetição em que a sua variável de iteração utilize o método `set`.

Definição: Considere A como uma lista de expressões de declarações de um laço de repetição, a como uma expressão qualquer, x a variável de iteração da lista. Pode ser uma estrutura passível de aplicação da refatoração Converter `iterator.set` para `replaceAll` em A , se $\exists a \in A \mid a = x.set()$ for verdadeira.

Detecção de concatenações

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Concatenar uma coleção com `join`*. A principal característica desta função de detecção é encontrar trechos de código onde uma concatenação de elementos resulte em uma *string*, pois o `join` é um método pertencente à classe `String`. Em Java existem algumas formas para concatenar um elemento em uma *string*, uma delas é utilizando o operador `+=`, outra com a utilização do método `append` da classe `StringBuilder`, sendo esse último o mais recomendado para esse tipo de operação.

Definição: Considere A como uma lista de expressões (`ExpressionStatement`) em um laço de repetição, a como uma expressão qualquer, $operação(x)$ como uma função que retorna a operação de uma expressão e op como as formas de concatenar uma *String* na linguagem, pode existir uma oportunidade de aplicar a refatoração *Concatenar um array com o método `join`* em A , se $\exists a \in A \mid operação(a) = op$ seja verdadeiro.

Detecção de declarações do tipo `Iterator`

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Converter interface `Iterator` para `SplitIterator`*. A interface `SplitIterator` tem um propósito semelhante à `Iterator`, mas foi projetada para executar iterações em paralelo, sendo possível substituir uma pela outra. Sendo assim uma definição possível é procurar trechos de código onde aparece a interface `Iterator`.

Definição: Considere A uma lista com declarações de variáveis em um método, a como uma declaração qualquer e $tipoDeclacao(x)$ como uma função que retorna o tipo da de-

claração. Pode existir uma oportunidade de aplicar a refatoração *Converter Iterator* para *SplitIterator* em A , se $\exists a \in A \mid \text{tipoDeclaracao}(a) = \text{Iterator}$ for verdadeiro.

Detecção de ifs com expressões de comparação similares

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Adicionar interface Predicate*. A aplicação dessa refatoração é viável caso a expressão de comparação nas estruturas *ifs* se repete em uma ou mais estruturas dentro de um mesmo método.

Definição: Considere A uma lista de expressões condicionais de um método, sendo a e a' expressões diferentes pertencentes a A , pode existir uma oportunidade de aplicar a refatoração *Adicionar interface Predicate* em A , se $\forall a \in A, \exists a' \in A \mid a = a'$ seja verdadeiro.

Detecção de médias aritméticas

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Converter para média lambda*. O conceito de média aritmética é a soma total dos termos dividida pelo número total deles. Essa refatoração é aplicada em variáveis do tipo `Double`, pois é um requisito da estrutura final que será refatorada. Então, se o método contém uma variável do tipo `Double`, onde possui uma operação de divisão entre uma variável acumuladora e o total de valores de uma determinada lista, essa estrutura é uma candidata a receber essa refatoração. Lembrando que até o presente momento, uma variável local e acumuladora em Java sempre deve ser inicializada.

Definição: Considere A uma lista com todas as expressões aritmética declaradas em um método, a e a' como diferentes expressões aritméticas pertencentes a A . Considere ainda, *operacaoLadoDireito(x)* uma função que retorna o lado após a igualdade (direito) em uma expressão aritmética, *variavel(x)* uma função que retorna a variável de uma expressão aritmética, z como uma variável qualquer, pode existir uma oportunidade de aplicar a refatoração *Converter para média lambda* em A , se $\forall a \in A, \exists a' \in A \mid \text{operacaoLadoDireito}(a) = 0 \wedge a' = \text{variavel}(a) / z$ for verdadeiro.

Detecção de contagens de objetos

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Converter para contagem lambda*. Essa refatoração é aplicada para contagens de objetos em uma lista. Em Java, isso geralmente é feito percorrendo a lista e incrementado o valor em um. Sendo assim, uma possível definição para essa função de busca é procurar por laços de repetição que implementam uma atribuição de mais um (+1) ou mais-mais (++) em seu código.

Definição: Considere A uma lista com todas as expressões aritméticas de um laço de repetição, a como uma expressão aritmética de A , $operador(x)$ retorna o operador da expressão, $operacaoLadoDireito(x)$ uma função que retorna o lado após a igualdade (direito) em uma expressão aritmética e x uma variável numérica qualquer, pode existir uma oportunidade de aplicar a refatoração *Converter para contagem lambda* em A , se $\exists a \in A \mid operacaoLadoDireito(a) = x + 1 \vee (operador(a) = (+) \wedge operacaoLadoDireito(a) = 1) \vee operador(a) = (++)$ for verdadeiro.

Detecção de chamadas aos métodos *unmodifiable*

Esta função de detecção tem como objetivo encontrar oportunidades para a aplicação da refatoração *Adicionar método $\circ f$* . A forma utilizada em Java para criar coleções imutáveis é com a utilização dos métodos `unmodifiable`, junto com o nome da coleção que se deseja implementar, por exemplo, `unmodifiableList` para estruturas do tipo `List` e `unmodifiableSet` para estruturas do tipo `Set`. Então, uma forma de encontrar refatorações deste tipo é procurar por essas palavras reservadas em atribuições.

Definição: Considere A como uma lista de expressões de atribuições de um método, a como uma expressão de atribuição de A e $f(x)$ uma função que retorna verdadeiro caso existe um método para transformar as coleções em imutáveis, pode existir uma oportunidade de aplicar a refatoração *Adicionar método $\circ f$* em A , se $\exists a \in A \mid f(a)$ for verdadeiro.

Detecção de parâmetros não utilizados em expressões lambdas

Esta função de detecção é utilizada para encontrar refatorações do tipo *Converter para underscore*. Para encontrar refatorações deste tipo, é necessário percorrer o corpo das expressões lambdas e achar ocorrências de seus parâmetros. Caso algum parâmetro não tenha nenhuma ocorrência no corpo da expressão lambda é possível que haja uma oportunidade para aplicar essa refatoração.

Definição: Considerando que *A* recebe uma lista com as variáveis dos parâmetros de uma expressão lambda e que *B* recebe a lista com as expressões do corpo da expressão lambda, *variavel(x)* uma função que retorna a variável de uma expressão, pode existir uma oportunidade de aplicar a refatoração *Converter para underscore* em uma expressão lambda, se $\forall a \in A \neg \exists b \in B \mid a = \text{variavel}(b)$ for verdadeiro.

3.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentou um catálogo com 15 refatorações para atualização de código na linguagem Java. O catálogo possui refatorações que atendem à necessidade de quatro versões de Java e em complemento às 10 refatorações propostas por TEIXEIRA JÚNIOR et al.. O Anexo A descreve 15 refatorações inversas às propostas neste Capítulo.

Por vezes, é complicado aplicar as refatorações na prática, seja pela grande quantidade de linhas de código em sistemas de médio e grande porte, ou pela complexabilidade do código. Por esse motivo faz-se necessárias, muitas vezes, ferramentas automáticas que aplicam algum tipo de função para detectar partes de um sistema em que essas refatorações possam ser aplicadas. Nesse sentido este capítulo apresentou sugestões de 15 funções de detecção para as 15 refatorações de atualização de código propostas.

O Capítulo 4 apresenta a implementação das funções de detecção propostas na Seção 3.2. Além de apresentar o desenvolvimento de uma pequena API, para encontrar trechos em um código onde seja possível aplicar as refatorações propostas.

4 ESTUDO DE CASO

Este capítulo apresenta a implementação das funções de detecção apresentadas no catálogo e um estudo de caso sobre a aplicação das refatorações propostas. Para isso, foram coletados dados de cinco projetos com código aberto disponibilizados na web, sendo o menor com 6.329 linhas de código e o maior com 108.066. Para análise desses programas foi implementada uma *API* que funciona em conjunto com a ferramenta *AOPJungle* (FAVERI et al., 2013), onde são coletadas informações sobre os projetos analisados. O capítulo está organizado da seguinte forma. A Seção 4.1 apresenta algumas informações dos projetos escolhidos para realização do estudo de caso. A Seção 4.2 descreve alguns detalhes da implementação e funcionamento da *API* desenvolvida. A Seção 4.3 mostra uma implementação das funções de detecção de acordo com cada refatoração, os resultados obtidos após a execução das mesmas e exemplos de como ficará a estrutura após a aplicação das refatorações propostas para as oportunidades encontradas. Por fim, a Seção 4.4 apresenta uma discussão acerca dos resultados obtidos.

4.1 PROJETOS SELECIONADOS

Os projetos selecionados levaram em conta o ano do desenvolvimento, assim como a versão inicial de Java utilizada para a implementação. Embora grande parte das refatorações apresentadas no catálogo do Capítulo 3 seja para evolução de código para Java 8, é importante testar códigos já atualizados para a nova versão (Apache-Ant v1.10.3), para testar algumas das refatorações desenvolvidas para as versões 9 e 10. Os demais códigos selecionados foram desenvolvidos inicialmente para diferentes versões Java (conforme a Tabela 3). A coleta de projetos heterogêneos, com diferentes tamanhos e aplicações, proporcionou uma maior variedade nas análises das refatorações com uma maior cobertura na avaliação proposta. A Tabela 3 apresenta um resumo contendo nome, descrição, versão e número de linhas dos projetos selecionados.

As funções de detecção desenvolvidas para a busca das refatorações sugeridas nesta dissertação, foram implementadas como um *plug-in* para o Eclipse e funcionam em conjunto com a ferramenta *AOPJungle*. A próxima seção apresenta informações em relação à ferramenta desenvolvida para a visualização dos resultados.

Tabela 3 – Projetos usados no estudo de caso.

Projeto	Descrição	Versão e LOC
Apache-Ant ant.apache.org	Uma biblioteca Java Java 8	1.10.3 108.066
Apache-log4j logging.apache.org	Uma ferramenta Java para Log de dados Java 4	1.2.17 15.657
JMule jmule.org	Um cliente para redes eDonkey2000 Java 5	0.5.8 68.739
SQLJet sqljet.com	Uma implementação Java do SQLite Java 6	1.1.10 32.574
AOPJungle goo.gl/BEmzfp	Um framework de análise estática Java 7	0.1 6.329

4.2 UMA API PARA DETECTAR OPORTUNIDADES DE REFATORAÇÃO

As funções de detecção foram implementadas como complemento do *AOPJungle*, utilizando recursos da plataforma Eclipse e de *plug-ins* (*JDT*, *AJDT* e do próprio *AOPJungle*). A *API* se beneficia das funcionalidades do *AOPJungle* para percorrer trechos de código e obter as informações necessárias. Então, ela extrai os dados relevantes que são utilizados nas implementações das funções de detecção. Para não afetar o funcionamento da ferramenta *AOPJungle* foi adicionado um novo pacote (*package*) no projeto, em que foram criadas classes para tratar cada função de detecção. Cada função estende a classe abstrata *Heuristic*, pois elas tem características em comum, como um nome para identificá-la, o local onde foi encontrada e alguns métodos em comum. A cada refatoração identificada é salvo o local completo (*packageName*, *className*) e a linha onde se encontra. A Listagem 1 mostra parte do código dessa classe.

Listagem 1: Códigos da classe *Heuristic*.

```

1 public abstract class Heuristic {
2     private String refactoringName;
3     private String packageName, className;
4     private int lineNumber;
5
6     //getters and setters
7
8     public static boolean compareMethodBodies(Block body,
9         String method) {...}
10
11    public static boolean compareMethodExpressions(
12        ExpressionStatement es, String method) {...}
13
14 }

```

A Listagem 2 apresenta o desenvolvimento da primeira função de detecção (*Detecção de blocos catch similares*), que será apresentada como exemplo. O método `visit` para estruturas do tipo `TryStatement` foi implementado no *AOPJungle*, pois no projeto original esses tipos de estruturas não eram relevantes (FAVERI et al., 2013). A classe `AOJTryStatement` também foi adicionada ao projeto *AOPJungle*. Com ela, são extraídas informações sobre a estrutura que é analisada, como o código interno, as variáveis, blocos *catch* (pois é uma estrutura do tipo *try-catch*), entre outras informações.

Listagem 2: Implementação da função de detecção *Detecção de blocos catch similares*.

```

1 public boolean visit(TryStatement node) {
2     var aojNode = new AOJTryStatement(node,
3         (AOJProgramElement) getLastMemberFromStack());
4     var behaviourKind = getAOJBehaviourKind(
5         getLastMemberFromStack());
6     if (behaviourKind != null) {
7         var catchClauses = node.catchClauses();
8         if (aojNode.similarity(0.7, catchClauses)) {
9             var aux = new GroupCatchBlocks(
10                behaviourKind.getPackage().getName(),
11                behaviourKind.getCompilationUnit().getName(),
12                cUnit.getLineNumber(node.getStartPosition()));
13            refactoringList.addGroupCatchBlocks(aux);
14        }
15    }
16    return super.visit(node);
17 }

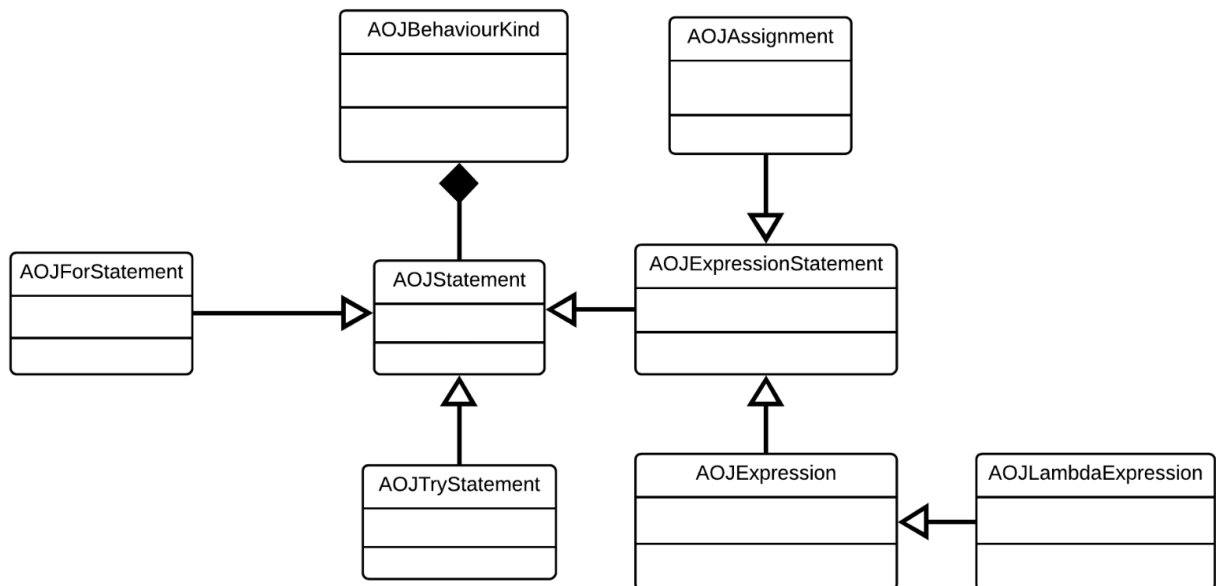
```

A classe `AOJBehaviourKind` (linha 4) já fazia parte do projeto original. Ela é responsável por armazenar algumas informações como: código da classe onde a estrutura está inserida, algumas métricas, parâmetros, exceções, entre outras informações em relação ao sistema em geral (nome de classes, localização do arquivo, número da linha onde inicia a estrutura, entre outras informações). A linha 8 mostra um método que foi implementado na classe `AOJTryStatement` (`similarity`), onde foi implementado o código necessário para o funcionamento da função de detecção. Caso o método apresentado retornar verdadeiro, então o sistema adiciona na lista as informações da estrutura avaliada como uma possível oportunidade de refatoração.

As classes das funções de detecção quando instanciadas salvam os possíveis locais de oportunidades de refatoração. Dessa forma, cria-se um objeto para cada tipo de função de detecção e, após isso, eles são salvos em uma única lista. Foram implementadas novas

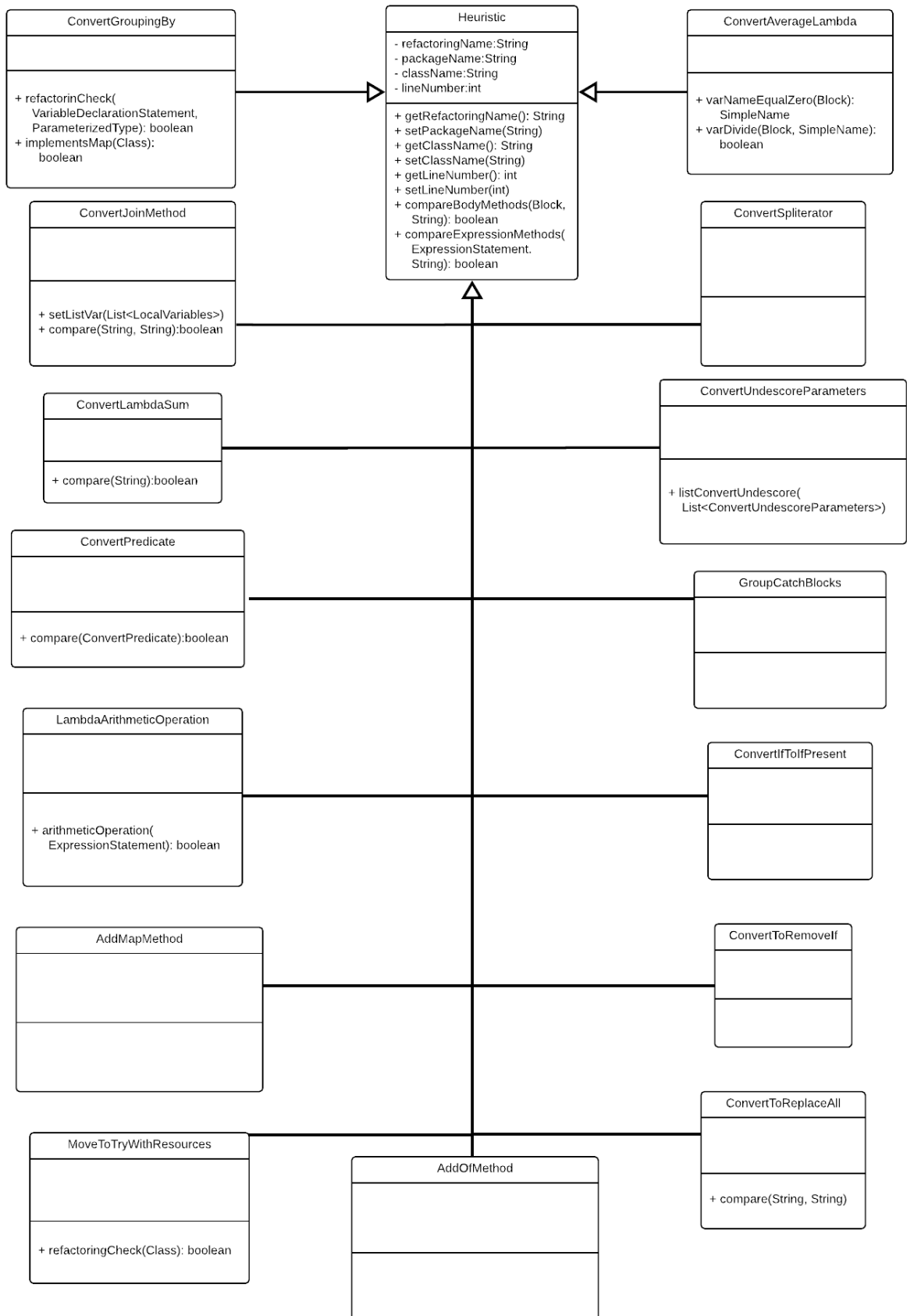
classe no *AOPJungle*. No pacote `br.ufsm.aopjungle.metamodel.java` foram adicionadas as classes (i) `AOJAssignment` para armazenar informações de atribuições feitas no código, (ii) `AOJForStatement` para armazenar informações de estruturas de repetição `for`, (iii) `AOJLambdaExpression` para armazenar informações de expressões lambdas e (iv) `AOJTryStatement` para armazenar informações de estruturas `try-catch`. O Diagrama 3 apresenta as estruturas que foram adicionadas, entre outras já existentes. A classe `AOJBehaviourKind`, já explicada anteriormente, é a base da estrutura para o armazenamento das informações. No decorrer do desenvolvimento das funções de detecção outros detalhes da implementação são descritos.

Figura 3 – Diagrama de classes adicionadas no *AOPJungle*



A Figura 4 apresenta o diagrama de classes da implementação da *API*. Como é possível observar, cada refatoração possui uma classe própria a fim de agrupar as oportunidades encontradas. Quando uma oportunidade de refatoração é encontrada, a classe, de acordo com o tipo de refatoração, é instanciada. Salva-se no objeto informações sobre (i) o caminho onde está localizada a oportunidade de refatoração, (ii) o nome da classe onde essa oportunidade se encontra e (iii) o número da linha onde se inicia a implementação da oportunidade encontrada.

Figura 4 – Diagrama de classes da modelagem das oportunidades de refatoração



As outras funções de detecção seguem o mesmo princípio que a descrita anteriormente. Para a visualização, é gerada uma lista separada pelo tipo de refatoração, como é observado na linha 13 da Listagem 2 (`refactoringList`). A lista é impressa no console do Eclipse utilizando a ferramenta *Eclipse View*, que é baseada em *SWT (Standard Widget Toolkit)* e possui uma visualização utilizando conceitos de hierarquia (SPRINGGAY, 2001). Então, a *API* gera um relatório completo com o nome da refatoração, oportunidades de refatoração encontradas e a localização das possíveis estruturas que podem ser refatoradas.

A Figura 5 apresenta a tela com os resultados da execução da *API* no projeto *AOPJungle*. No primeiro nível da hierarquia é gerada na tela apenas a imagem de um círculo vermelho, quando existem refatorações, ou um círculo verde, quando não existem, junto com o nome da refatoração e a quantidade total de oportunidades encontradas. O segundo nível apresenta a localização das classes que possuem as oportunidades encontradas, de acordo com a refatoração. O terceiro nível apresenta o nome das classes. Por fim, o último nível exibe a(s) linha(s) onde está(ão) essa(s) estrutura(s). Uma mesma estrutura pode aparecer em duas oportunidades de refatoração diferentes, caso ela atenda aos requisitos das duas.

Resumidamente, a *API* gera uma estrutura do tipo árvore, onde é possível obter os locais aos quais existem possíveis oportunidades de refatoração. A próxima seção apresentará informações sobre a implementação das funções de detecção e a execução delas nos projetos listados na Seção 4.1.

Figura 5 – Exemplo de visualização das funções de detecção



4.3 IMPLEMENTAÇÃO E EXECUÇÃO DAS FUNÇÕES DE DETECÇÃO

Esta seção apresenta a implementação das funções de detecção e os resultados obtidos com as execuções delas nos projetos apresentados. Os resultados serão retirados diretamente da execução da *API*. Após isso, pequenos exemplos de estruturas sugeridas para refatoração serão comentados e expostos. As funções de detecção serão apresentadas na mesma ordem das refatores no Capítulo 3, juntamente com um gráfico que mostra as quantidades de oportunidades de refatoração encontradas para cada projeto, três exemplos de trechos de código identificados pela ferramenta e um exemplo de como ficaria o código após a aplicação da refatoração.

4.3.1 Agrupar blocos *catch*

Para a implementação desta função, foi necessária a criação de uma nova classe no *AOPJungle*, que realiza o tratamento das estruturas *try-catch*. Seguiu-se o padrão já adotado para nomear novas classes, começando com *AOJ* seguido pelo nome da estrutura, por exemplo, o nome adotado para a classe criada foi `AOJTryStatement.java`. Essa classe, quando instanciada, salva algumas informações importantes que foram utilizadas na implementação da função de detecção, como, por exemplo, o código das estruturas *catch*.

A Listagem 3 apresenta o método desenvolvido para verificar a similaridade. É utilizado o algoritmo `Jaccard`, que para calcular a similaridade que: (i) conta os números de membros que são compartilhados nos dois conjuntos, (ii) conta o número total de membros em ambos os conjuntos, (iii) divide o número de membros compartilhados pelo número total de membros e (iv) multiplica o número encontrado no passo anterior por 100 (COHEN; RAVIKUMAR; FIENBERG, 2003). Foi o algoritmo que apresentou os melhores resultados em comparações realizadas. Foram testados também os algoritmos *JaroWinkler* (WINKLER, 1990) e *Cosine* (WILKINSON; HINGSTON, 1991). O método retorna `true` caso existam pelo menos dois *catchs* similares.

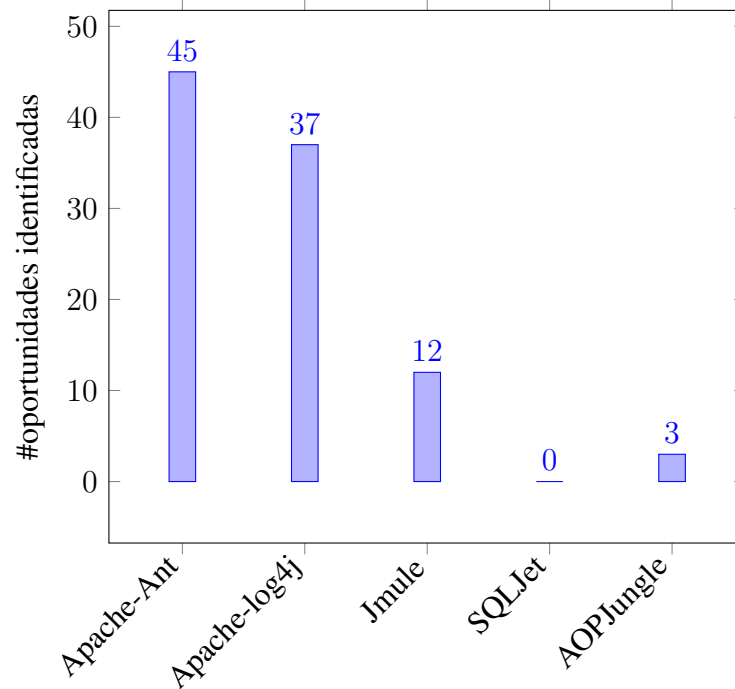
Listagem 3: Método que implementa a similaridade.

```

1 public boolean areSimilar(double threshold,
2     List<CatchClause> catches) {
3     var j2 = new Jaccard(2);
4     for (var i = 0; i < catches.size() - 1; i++)
5         for (var j = i + 1; j < catches.size(); j++)
6             if (j2.similarity(catches.get(i).toString(),
7                 catches.get(j).toString()) > threshold)
8                 return true;
9     return false;
10 }
```

Após a execução desta função de detecção foram encontradas 97 oportunidades de refatoração, sendo que o *Apache-Ant* e *Apache-log4j* foram os projetos que mais retornaram oportunidades, sendo 47% e 38% do montante, respectivamente. Não foram encontradas oportunidades de refatoração no projeto *SQLJet*. A Figura 6 apresenta um gráfico com a quantidade de oportunidades encontradas de acordo com o projeto.

Figura 6 – Quantidade de oportunidades de refatoração para *Agrupar blocos catch*



As listagens 4, 5 e 6 apresentam algumas das oportunidades de refatoração recomendadas pela *API*. São trechos de código retiradas aleatoriamente do projeto *Apache-log4j*. A Listagem 4 apresenta a primeira oportunidade identificada, a estrutura é passível de refatoração, ou, ao menos, os dois últimos blocos `catch` (linhas 5 e 6). Abaixo da oportunidade encontrada é realizada a refatoração seguindo os passos recomendados (da linha 9 à 14).

Listagem 4: Primeira oportunidade encontrada para a refatoração *Agrupar blocos catch*.

```

1 try {
2     istream.close();
3 } catch (InterruptedException ignore) {
4     Thread.currentThread().interrupt();
5 } catch (IOException ignore) {
6 } catch (RuntimeException ignore) {
7 }
8 //Estrutura refatorada
9 try {
10    istream.close();
11 } catch (InterruptedException ignore) {
12    Thread.currentThread().interrupt();
13 } catch (RuntimeException | IOException ignore) {
14 }

```

A Listagem 5 apresenta outra oportunidade encontrada. Ela foi retornada como oportunidade de refatoração devido ao grau de similaridade escolhido para essa função de busca (que

pode ser alterado). Porém, cada bloco *catch* possui pequenas diferenças em relação as mensagens adicionadas em alguma estrutura de acordo com a variável LOG. A cada classe de exceção diferente, uma mensagem diferente é adicionada (linhas 4, 6, 8 e 10). Então, para a aplicação desta refatoração, as mensagens também devem se fundir (linha 18 à 20).

Listagem 5: Segunda oportunidade encontrada para a refatoração *Agrupar blocos catch*.

```

1  try {
2  //...
3  } catch (EOFException e) {
4      LOG.info("Reached EOF, closing connection");
5  } catch (SocketException e) {
6      LOG.info("Caught SocketException, closing connection");
7  } catch (IOException e) {
8      LOG.warn("Got IOException, closing connection", e);
9  } catch (ClassNotFoundException e) {
10     LOG.warn("Got ClassNotFoundException, closing connection",
11             e);
12 }
13 //Estrutura refatorada
14 try {
15 //...
16 } catch (EOFException | SocketException |
17         IOException | ClassNotFoundException e) {
18     LOG.info("Reached EOF or Caught SocketException
19             or Got IOException or Got ClassNotFoundException,
20             Closing connection: ", e);
21 }

```

A Listagem 6 apresenta outra oportunidade encontrada similar à anterior. Ela retornou devido ao grau de similaridade escolhido. Porém essa oportunidade é um falso positivo, pois as mensagens são diferentes, podendo ocasionar erros quando for realizada uma fusão entre os blocos *catch*. Aumentando o grau de similaridade na função de detecção, provavelmente estruturas como a da Listagem 6 não seriam detectadas como oportunidades de refatoração.

Listagem 6: Terceira oportunidade encontrada para a refatoração *Agrupar blocos catch*.

```

1  try {
2      ...
3  } catch (NoClassDefFoundError e) {
4      LOG.info("Missing classes for XML parser", e);
5      JOptionPane.showMessageDialog(
6          this,
7          "XML parser not in classpath -
8          unable to load XML events.",
9          "CHAINSAW",

```



```

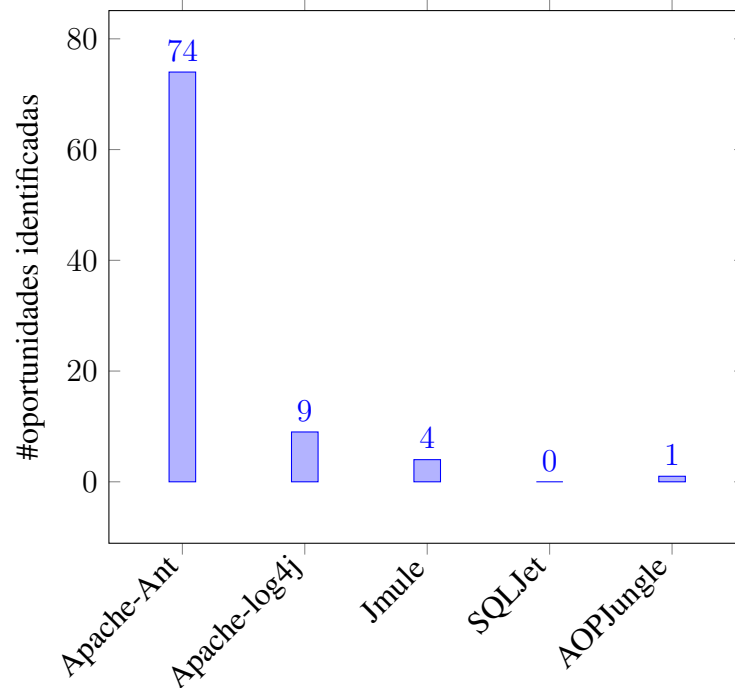
18         .getStartPosition()));
19         refactoringList.addMoveTryWithResouces((aux));
20     }
21     } catch (ClassNotFoundException e) {
22         e.printStackTrace();
23     }
24 }
25 public class AOJTryStatement {
26 private String code;
27 private TryStatement tryCatches;
28
29 public AOJTryStatement(TryStatement node, AOJProgramElement owner) {
30     this.tryCatches = node;
31     setCode(node.toString());
32 }
33
34 public List<String> getNameResources() {
35     var resources = new ArrayList<String>();
36     for (var a: tryCatches.getBody().statements())
37         if (a instanceof ExpressionStatement) {
38             var b = (ExpressionStatement) a;
39             if (b.getExpression() instanceof Assignment) {
40                 var c = (Assignment) b.getExpression();
41                 if (c.getRightHandSide() instanceof
42                     ClassInstanceCreation) {
43                     var d = (ClassInstanceCreation) c
44                         .getRightHandSide();
45                     resources.add(d.getType());
46                 }
47             }
48         }
49     return resources;
50 }
51
52 public class MoveToTryWithResources extends Heuristic {
53 //...
54     public static boolean refactoringCheck(Class clazz) {
55         try {
56             var autoCloseableClass = ClassFinder.findClass("
57                 java.lang.AutoCloseable");
58             return autoCloseableClass.isAssignableFrom(clazz);
59         } catch (ClassNotFoundException e) {
60             e.printStackTrace();
61         }
62         return false;
63     }
64 }

```

Após a execução desta função de detecção foram encontradas 88 oportunidades de refatoração, sendo que 84% delas foram identificadas no projeto *Apache-Ant*. O projeto *SQLJet*

não retornou oportunidades de refatoração. A Figura 7 apresenta um gráfico com a quantidade de oportunidades encontradas de acordo com os projetos analisados.

Figura 7 – Quantidade de oportunidades de refatoração para *Mover recurso para o try-with-resources*



As Listagens 8, 9 e 10 apresentam oportunidades encontradas pela *API*. Os exemplos são retirados do projeto *Apache-Ant*, visto que foi o código que mais apresentou esse tipo de refatoração. A Listagem 8 apresenta uma oportunidade possível de ser refatorada (linha 1 à 6), uma vez que a classe `InputStreamReader` (linha 1 e 3) implementa a interface `AutoCloseable` (ORACLE, 2017a).

Listagem 8: Primeira oportunidade encontrada para a refatoração *Mover recurso para o try-with-resources*.

```

1 InputStreamReader isr = null;
2 try {
3     isr = new InputStreamReader(is, "UTF-8");
4     //...
5 } catch (UnsupportedEncodingException e) {
6     isr = new InputStreamReader(is);
7 }
8 //Estrutura refatorada
9 try (InputStreamReader isr = new InputStreamReader(is, "UTF-8")) {
10     //...
11 } catch (UnsupportedEncodingException e) {

```

```

12     isr = new InputStreamReader(is);
13 }

```

A Listagem 9 apresenta outra oportunidade passível de refatoração, uma vez que a classe `PrintStream` (linhas 1 e 5) implementa a interface `AutoCloseable` (ORACLE, 2017a). Após a oportunidade é apresentada uma possível estrutura após a aplicação da refatoração (linha 10 à 14).

Listagem 9: Segunda oportunidade encontrada para a refatoração *Mover recurso para o try-with-resources*.

```

1 PrintStream logTo = null;
2 //...
3 try {
4     //...
5     logTo = new PrintStream(Files.newOutputStream(
6         logfile.toPath())); //NOSONAR
7     isLogFileUsed = true;
8 }
9 //Estrutura refatorada
10 try (PrintStream logTo = new PrintStream(Files
11     .newOutputStream(logfile.toPath())) {
12     //...
13     isLogFileUsed = true;
14 }

```

A Listagem 10 apresenta uma outra oportunidade encontrada que também é possível de ser refatorada, uma vez que a classe `BufferedReader` (linha 1 e 3) implementa a interface `AutoCloseable` (ORACLE, 2017a). Após a oportunidade encontrada é apresentada como ficaria a estrutura depois da aplicação da refatoração (linha 7 à 10).

Listagem 10: Terceira oportunidade encontrada para a refatoração *Mover recurso para o try-with-resources*.

```

1 BufferedReader r = null;
2 try {
3     r = new BufferedReader(new InputStreamReader(getInputStream()));
4     //...
5 }
6 //Estrutura refatorada
7 try (BufferedReader r = new BufferedReader(
8     new InputStreamReader(getInputStream())) {
9     //...
10 }

```

4.3.3 Adicionar método `map` na interação com coleções

A Listagem 11 apresenta os principais trechos de código para o desenvolvimento desta função de detecção. Então, o código apresentado da linha 2 até a 18 foi implementado no método `visit` para laços de repetição. Os dois últimos métodos apresentados na Listagem 11 (`compareMethodBodies` e `compareMethodExpressions`) obtêm, por reflexão, os nomes dos métodos invocados em expressões dentro do laço e os compara com o nome enviado no segundo argumento dos métodos implementados (`method`). São métodos que foram implementados na classe `Heuristic`, uma vez que eles são utilizados em outras refatorações.

Então, quando faz a chamada a um dos métodos implementados deve passar nos argumentos o corpo do laço e o nome do método procurado, que, no caso da Listagem 7, é o método `add`. Quando o laço de repetição possui apenas uma linha, ou seja, apenas uma expressão, o método `getBody` retorna uma estrutura do tipo `ExpressionStatement`. Se possuir mais de uma linha, ou seja, mais de uma expressão em seu corpo, o método retorna uma estrutura do tipo `Block`. Sendo assim, é necessário verificar de qual tipo é a estrutura retornada, para poder fazer o tratamento adequado.

Listagem 11: Métodos para encontrar oportunidades de refatorações *Adicionar método `map` na interação com coleções.*

```

1  if (node.getBody() instanceof ExpressionStatement) {
2      if (AddMapMethod.compareMethodExpressions(
3          (ExpressionStatement) node.getBody(), "add")) {
4          var aux = new AddMapMethod(
5              behaviourKind.getPackage().getName(),
6              behaviourKind.getCompilationUnit().getName(),
7              cUnit.getLineNumber(node.getStartPosition()));
8          refactoringList.addAddMapMethod(aux);
9      }
10 }
11 if (node.getBody() instanceof Block)
12     if (AddMapMethod.compareMethodBodies(
13         (Block) node.getBody(), "add")) {
14         var aux = new AddMapMethod(
15             behaviourKind.getPackage().getName(),
16             behaviourKind.getCompilationUnit().getName(),
17             cUnit.getLineNumber(node.getStartPosition()));
18         refactoringList.addAddMapMethod(aux);
19     }
20
21 public static boolean compareMethodBodies(Block body,
22     String method) {

```

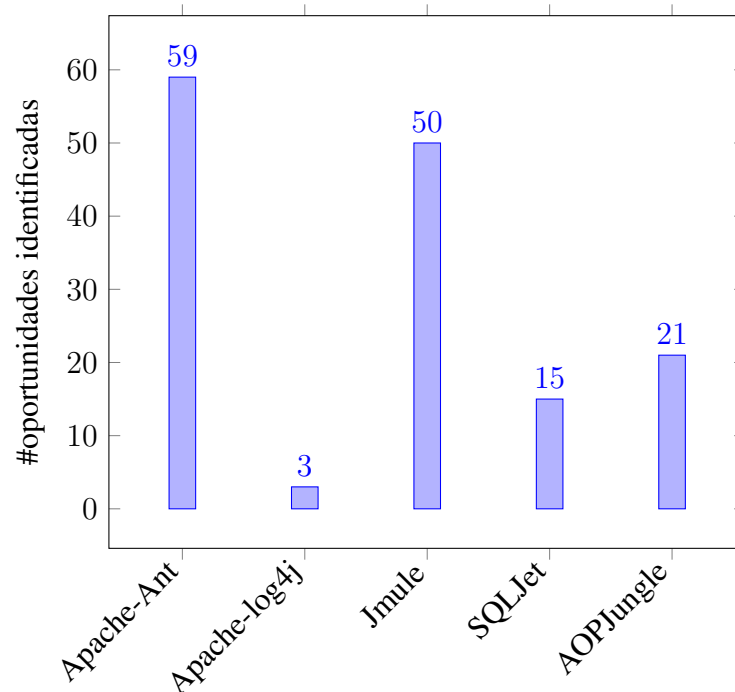
```

23     for (var a: body.statements())
24         if (a instanceof ExpressionStatement)
25             return compareMethodExpressions((ExpressionStatement) a,
26                 method);
27     return false;
28 }
29
30 public static boolean compareMethodExpressions(
31     ExpressionStatement es, String method) {
32     if (es.getExpression() instanceof MethodInvocation) {
33         var met = (MethodInvocation) es.getExpression();
34         return met.getName().getIdentifier().compareTo(method) == 0;
35     }
36     return false;
37 }

```

Após a execução da função de detecção implementada foram encontradas 148 oportunidades. O projeto *Apache-Ant* corresponde a 40% desse total e o *Imule* corresponde a 34% do total. Todos os projetos retornaram oportunidades de refatoração. As quantidades nos demais projetos podem ser observadas na Figura 8.

Figura 8 – Quantidade de oportunidades de refatoração para *Adicionar método map na interação com coleções*



As Listagens 12, 13 e 14 apresentam alguns exemplos das oportunidades encontradas no projeto *AOPJungle*. A Listagem 12 apresenta o primeiro exemplo, onde a coleção para obtenção dos dados é obtida pelo método `getMethods` (linha 4). A coleção que recebe os valores é a

methods. Logo após a oportunidade encontrada é apresentada a mesma estrutura refatorada (linha 8 à 10). A princípio a oportunidade é passível de refatoração.

Listagem 12: Primeira oportunidade encontrada para a refatoração *Adicionar método map na interação com coleções.*

```

1 public AOJInterface(Class<?> c) {
2     this.fullyQualifiedName = c.getName();
3     this.methods = new ArrayList<String>();
4     for (var m : c.getMethods())
5         methods.add(m.getName());
6 }
7 //Estrutura refatorada
8 public AOJInterface(Class<?> c) {
9     this.fullyQualifiedName = c.getName();
10    this.methods = c.getMethods().stream()
11        .map(m -> m.getName())
12        .collect(Collectors.toList());
13 }

```

A Listagem 13 apresenta outro exemplo de oportunidade encontrada. A coleção para obtenção dos dados é obtida através do método `getParameters` (linha 2). A coleção que recebe os resultados é a `listParameters` (linha 1). Após a oportunidade encontrada é apresentada a estrutura refatorada (linha 5 à 7).

Listagem 13: Segunda oportunidade encontrada para a refatoração *Adicionar método map na interação com coleções.*

```

1 Collection<String> listParameters = new ArrayList<>();
2 for (AOJParameter methodParameter : method.getParameters())
3     listParameters.add(methodParameter.getType().getName());
4 //Estrutura refatorada
5 Collection<String> listParameters = method.getParameters().stream()
6     .map(p -> p.getType().getName())
7     .collect(Collectors.toCollection());

```

A Listagem 14 apresenta outra oportunidade obtida pela *API*. As informações são obtidas da coleção `descriptors` (linha 3). A variável `result` (linha 2 e 4) recebe os resultados. Após a oportunidade encontrada é apresentada a estrutura refatorada. Foi possível aplicar a refatoração *Extract Method Reference* (TEIXEIRA JÚNIOR et al., 2014) conforme é apresentado no argumento do método `map` (linha 10).

Listagem 14: Terceiro exemplo de oportunidade encontrada para refatoração *Adicionar método map na interação com coleções.*

```

1 public List<String> getNames() {
2     List<String> result = new ArrayList<String>();
3     for (AOJModifierEnum modifier : descriptors)
4         result.add(modifier.toString());
5     return result;
6 }
7 //Estrutura refatorada
8 public List<String> getNames() {
9     List<String> result = descriptors.stream()
10         .map(Object::toString)
11         .collect(Collectors.toList());
12     return result;
13 }

```

4.3.4 Agrupar com `groupBy`

A Listagem 15 apresenta trechos de código dos principais métodos implementados. O primeiro método é o `visit` que é executado em declarações de variáveis. A implementação procura por declarações de variáveis do tipo `Map`, por reflexão. O segundo método (`implementsMap`) é para verificar se o tipo da variável é `Map` ou se é uma sub-interface de `Map`.

Listagem 15: Parte da implementação da função de detecção *Agrupar com `groupBy`*.

```

1 public boolean visit(VariableDeclarationStatement node) {
2     var behaviourKind = getAOJBehaviourKind(
3         getLastMemberFromStack());
4
5     if (node.getType() instanceof ParameterizedType) {
6         var pt = (ParameterizedType) node.getType();
7         if (ConvertGroupingBy.refactorinCheck(node, pt)) {
8             var vd = (VariableDeclarationFragment) node.fragments().get(0);
9             addRefactoringGroupingBy(node, behaviourKind, vd);
10        }
11    }
12    //...
13 }
14
15 private void addRefactoringGroupingBy(
16     VariableDeclarationStatement node,
17     AOJBehaviourKind behaviourKind,
18     VariableDeclarationFragment vd) {
19     if (vd.getInitializer() instanceof ClassInstanceCreation) {
20         var cic = (ClassInstanceCreation) vd.getInitializer();
21         if (cic.getType() instanceof ParameterizedType) {
22             var pt = (ParameterizedType) cic.getType();
23             for (AOJImportDeclaration importClass: behaviourKind

```

```

24         .getCompilationUnit().getImportsDeclaration())
25     if (importClass.qualifiedName()
26         .contains(pt.getType().toString())) {
27         try {
28             Class clazz = ClassFinder
29                 .findClass(importClass.qualifiedName());
30             if (ConvertGroupingBy.implementsMap(clazz)) {
31                 var aux = new ConvertGroupingBy(
32                     behaviourKind.getPackage().getName(),
33                     behaviourKind.getCompilationUnit().getName(),
34                     cUnit.getLineNumber(node.getStartPosition()));
35                 refactoringList.addConvertGroupingBy(aux);
36             }
37         } catch (ClassNotFoundException e) {
38             e.printStackTrace();
39         }
40     }
41 }
42 }
43 }
44
45 public static boolean refactoringCheck
46     (VariableDeclarationStatement node, ParameterizedType pt) {
47     return pt.typeArguments().size() == 2 &&
48         pt.typeArguments().get(0).toString().contains("String") &&
49         pt.typeArguments().get(1).toString().contains("Long") &&
50         node.fragments().get(0) instanceof VariableDeclarationFragment;
51 }
52
53 public static boolean implementsMap(Class clazz) {
54     try {
55         var mapClass = ClassFinder.findClass("java.util.Map");
56         return mapClass.isAssignableFrom(clazz) ||
57             mapClass.equals(clazz);
58     } catch (ClassNotFoundException e) {
59         e.printStackTrace();
60     }
61     return false;
62 }

```

Após a execução desta função de detecção não foram apresentados resultados. Um dos possíveis motivos é por ser uma refatoração muito específica. Sua funcionalidade, geralmente, é implementada diretamente na linguagem *SQL* e raramente tratada nos programas. Geralmente os desenvolvedores realizam a funcionalidade da estrutura `groupBy` diretamente no banco de dados dos sistemas.

4.3.5 Converter para uma operação aritmética lambda

A Listagem 16 apresenta os principais trechos de código para a função de detecção implementada para esta refatoração. O primeiro trecho (linha 2) mostra a implementação realizada nos métodos `visit` das estruturas de repetição. No corpo de um laço de repetição, procura-se por expressões aritméticas. O primeiro `if` (linha 1) é para laços que contém apenas uma linha, já o `if` da linha 10 procura em laços que possuem mais de uma linha. O método `arithmeticOperation` faz a comparação, por reflexão, se é implementada alguma operação aritmética de Java.

Listagem 16: Principais métodos da função de busca para encontrar oportunidades de refatoração *Converter para uma operação aritmética lambda*.

```

1  if (node.getBody() instanceof ExpressionStatement)
2      if (LambdaArithmeticOperation.arithmeticOperation(
3          (ExpressionStatement) node.getBody())) {
4      var refactor = new LambdaArithmeticOperation(
5          behaviourKind.getPackage().getName(),
6          behaviourKind.getCompilationUnit().getName(),
7          cUnit.getLineNumber(node.getStartPosition()));
8      refactoringList.addLambdaArithmeticOperation(refactor);
9  }
10 else if (node.getBody() instanceof Block) {
11     var block = (Block) node.getBody();
12     for (var a : block.statements())
13         if (a instanceof ExpressionStatement) {
14             if (LambdaArithmeticOperation.arithmeticOperation(
15                 (ExpressionStatement) a)) {
16                 var refactor = new LambdaArithmeticOperation(
17                     behaviourKind.getPackage().getName(),
18                     behaviourKind.getCompilationUnit().getName(),
19                     cUnit.getLineNumber(node.getStartPosition()));
20                 refactoringList.addLambdaArithmeticOperation(refactor);
21             }
22         }
23 }
24
25 public static boolean arithmeticOperation
26     (ExpressionStatement body) {
27     if (body.getExpression() instanceof Assignment) {
28         var assign = (Assignment) body.getExpression();
29         if (assign.getRightHandSide() instanceof InfixExpression) {
30             var ie = (InfixExpression) assign.getRightHandSide();
31             return ie.getOperator().equals(
32                 InfixExpression.Operator.PLUS) ||
33                 ie.getOperator().equals(

```

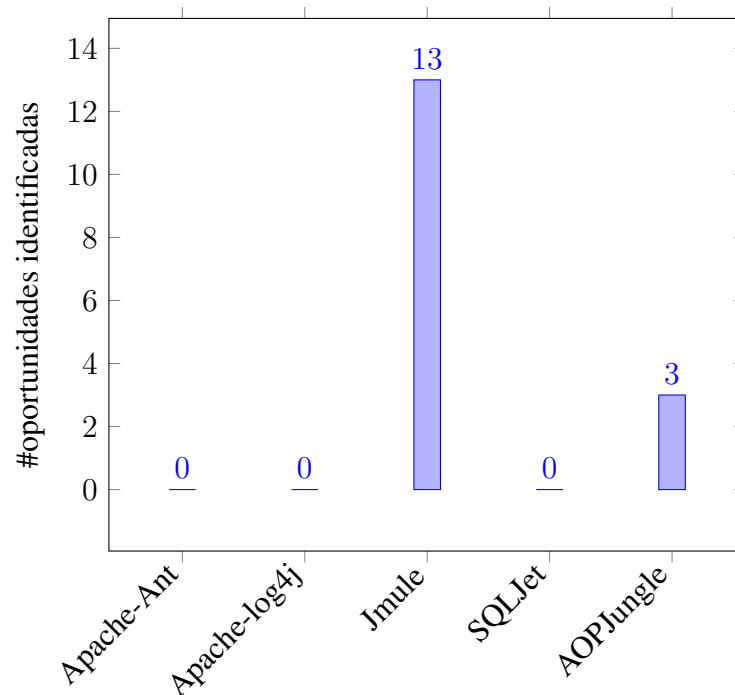
```

34         InfixExpression.Operator.MINUS) ||
35         ie.getOperator().equals(
36             InfixExpression.Operator.TIMES) ||
37         ie.getOperator().equals(
38             InfixExpression.Operator.DIVIDE) ||
39         ie.getOperator().equals(
40             InfixExpression.Operator.REMAINDER);
41     }
42 }
43 return false;
44 }

```

Após a execução da função de detecção foram encontradas 16 oportunidades no total, sendo que dois projetos retornaram refatorações deste tipo. O *Jmule* que retornou treze oportunidades e o projeto *AOPJungle* que retornou três oportunidades de refatoração. O gráfico da Figura 9 apresenta estes resultados.

Figura 9 – Quantidade de oportunidades de refatoração para *Converter para uma operação aritmética lambda*



As Listagens 17, 18 e 19 mostram três exemplos de oportunidades encontradas no projeto *Jmule*. A Listagem 17 apresenta o primeiro exemplo de uma oportunidade identificada pela *API*. A linha 1 mostra a variável `download_peers_count`, em que é uma possível oportunidade para aplicar esta refatoração. Como é uma coleção que obtém os valores por meio da chamada a um método `get` o passo três da refatoração deve ser executado, chamando o método `mapToInt` (linha 7), pois a variável que recebe os dados é do tipo `int`. Adicionou-

se também uma referência ao método `getPeerCount` (linha 7), no argumento do método `mapToInt`, de acordo com o passo quatro da refatoração. No segundo argumento do método `reduce` (linha 8) criou-se uma expressão lambda realizando a soma dos números, uma vez que a operação realizada é uma soma.

Listagem 17: Primeiro exemplo de oportunidade de refatoração *Operação aritmética lambda*.

```

1 int download_peers_count = 0;
2 for (DownloadSession session : session_list.values()) {
3     download_peers_count += session.getPeerCount();
4 }
5 //Estrutura refatorada
6 int download_peers_count = session_list.values().stream()
7     .mapToInt(DownloadSession::getPeerCount)
8     .reduce(0, (a, b) -> a + b);

```

A Listagem 18 apresenta um outro exemplo de oportunidade encontrada. Esta estrutura é possível de ser refatorada, possui uma diferença em relação a estrutura anterior, a variável que recebe a operação aritmética é do tipo `float`. Então, será necessário alterar o método de obtenção dos dados para `mapToFloat` (linha 11), de acordo com o passo três da refatoração. Realiza-se a referência ao método `getDownloadSpeed` (linha 11), no argumento do método `mapToFloat`. Ao final o método `reduce` (linha 12) recebe no primeiro argumento o valor 0 e no segundo argumento uma expressão lambda realizando uma soma.

Listagem 18: Segundo exemplo de oportunidade de refatoração *Operação aritmética lambda*.

```

1 public float getSpeed() {
2     float downloadSpeed = 0;
3     for (Peer peer : session_peers) {
4         downloadSpeed += peer.getDownloadSpeed();
5     }
6     return downloadSpeed;
7 }
8 //Estrutura refatorada
9 public float getSpeed() {
10    float downloadSpeed = session_peers.stream()
11        .mapToFloat(Peer::getDownloadSpeed)
12        .reduce(0, (a, b) -> a + b);
13    return downloadSpeed;
14 }

```

A Listagem 19 apresenta outro exemplo de uma oportunidade de refatoração encontrada pela API. Os dados da operação são obtidos por meio do método `values`, chamado pela variável `session_list` (linha 2). A variável que recebe o resultado é a `total_upload_peers`

(linhas 1 e 3). Este exemplo encontrado é similar ao primeiro, sendo aplicado o método `mapToInt` (linha 7) para obtenção dos resultados e a operação identificada é uma soma.

Listagem 19: Terceiro exemplo de uma oportunidade de refatoração *Operação aritmética lambda*.

```

1 int total_upload_peers = 0;
2     for(UploadSession session : session_list.values()) {
3         total_upload_peers += session.getPeerCount();
4     }
5 //Estrutura refatorada
6 int total_upload_peers = session_list.values().stream()
7     .mapToInt(UploadSession::getPeerCount)
8     .reduce(0, (a, b) -> a + b);

```

4.3.6 Converter `if` para `ifPresent`

A Listagem 20 apresenta uma possível implementação da função de detecção para este tipo de refatoração. A expressão lógica das estruturas `if` são analisadas, verificando a existência do valor `null`. Caso exista é salva como uma possível oportunidade de refatoração. Na classe `AOJIfStatement` foi implementado o método `getExpression` para obter a expressão lógica das estruturas `if`.

Listagem 20: Trecho de código para encontrar oportunidades de refatoração *Converter if para ifPresent*.

```

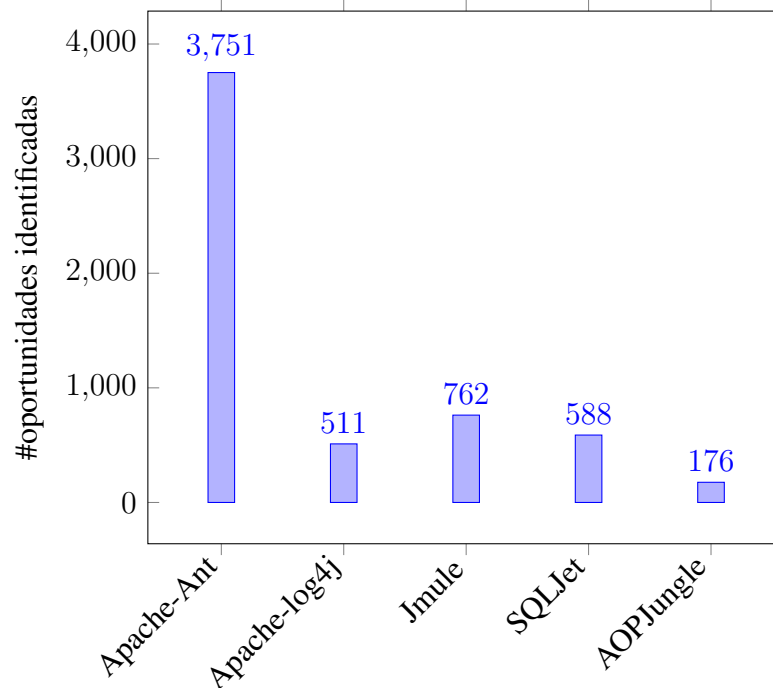
1 public boolean visit(IfStatement node) {
2     var aojNode = new AOJIfStatement(node,
3         (AOJProgramElement) getLastMemberFromStack());
4     if (aojNode.getExpression().contains("null")) {
5         ConvertIfToIfPresent aux = new ConvertIfToIfPresent(
6             behaviourKind.getPackage().getName(),
7             behaviourKind.getCompilationUnit().getName(),
8             cUnit.getLineNumber(node.getStartPosition()));
9         refactoringList.addConvertIfToIfPresent(aux);
10    }
11 }

```

Esta refatoração aborda uma funcionalidade comumente implementada por grande parte dos programadores, fazer comparações com o valor `null`. Assim já eram esperados altos valores como resultado da execução desta função, o que realmente foi confirmado na execução da *API*. Foram retornados 5.788 resultados. Todos os projetos retornaram um valor relativamente

alto, sendo o *Apache-Ant* com o maior número de oportunidades encontradas (3.751 resultados) e o *AOPJungle* o menor (176 resultados). O gráfico da Figura 10 mostra os valores de acordo com os projetos.

Figura 10 – Quantidade de oportunidades para *Converter if para ifPresent*



As Listagens 21, 22 e 23 apresentam exemplos de oportunidades de refatoração sugeridas pela *API*. Todos os códigos foram retirados do projeto *Jmule*. A Listagem 21 apresenta o primeiro exemplo de uma oportunidade de refatoração encontrada. A linha 2 apresenta uma comparação com o valor *null*, então é possível aplicar a refatoração. Após a oportunidade é apresentada a estrutura refatorada, utilizando o método *ifPresent* no lugar da estrutura *if*.

Listagem 21: Primeiro exemplo de oportunidade de refatoração *Converter if para ifPresent*.

```

1 DownloadSession downloadSession = session_list.get(fileHash);
2 if (downloadSession != null)
3     downloadSession.addDownloadPeers(peerList);
4 //Estrutura refatorada
5 var downloadSession = Optional.ofNullable(
6     session_list.get(fileHash));
7 downloadSession.ifPresent(p -> p.addDownloadPeers(peerList));

```

A Listagem 22 apresenta outra oportunidade encontrada, onde também é possível aplicar a refatoração. A refatoração é aplicada à variável *result* (linha 1). O método *ifPresent*

irá substituir a estrutura condicional `if`.

Listagem 22: Oportunidade para aplicação da refatoração *Converter `if` para `ifPresent`*.

```

1 List<NodeValue> result = traverse(node.getRightChild());
2 if (result != null)
3     list.addAll(result);
4 //Estrutura refatorada
5 var result = Optional.ofNullable(traverse(
6     node.getRightChild()));
7 result.ifPresent(p -> list.addAll(p));

```

A Listagem 23 também é possível aplicar a refatoração. Para essa aplicação vamos supor que o ambiente não esteja preparado para Java 10, então não será possível colocar a palavra reservada `var`, o passo um da refatoração prevê este problema. É possível resolver este impasse apenas encapsulando o tipo da variável com a classe `Optional` (linha 8). Em todas as estruturas apresentadas é possível aplicar essa refatoração, pois os laços condicionais fazem uma comparação com valores nulos.

Listagem 23: Oportunidade para aplicação da refatoração *Converter `if` para `ifPresent`*.

```

1 Tag tag = this.tagList.getTag(ED2KConstants.SL_UDPFLAGS);
2 if (tag != null) {
3     int value = ((IntTag)tag).getValue();
4     serverFeatures.addAll(Utils.scanTCPServerFeatures(value));
5     serverFeatures.addAll(Utils.scanUDPServerFeatures(value));
6 }
7 //Estrutura refatorada
8 Optional<Tag> tag = Optional.ofNullable(this.tagList.getTag(
9     ED2KConstants.SL_UDPFLAGS));
10 tag.ifPresent(p -> {
11     int value = ((IntTag)p).getValue();
12     serverFeatures.addAll(Utils.scanTCPServerFeatures(value));
13     serverFeatures.addAll(Utils.scanUDPServerFeatures(value));
14 });

```

4.3.7 Converter `iterator.remove` para `removeIf`

A Listagem 24 apresenta a implementação desta função de detecção. O código foi implementado no método `visit` para estruturas condicionais. São utilizados dois métodos apresentados anteriormente (`compareMethodBodies` e `compareMethodExpressions`) na refatoração *Adicionar método `map` na interação com coleções*. Os métodos retornam verdadeiro caso exista uma chamada ao método `remove` em uma estrutura condicional.

Listagem 24: Implementação da função de detecção para a refatoração *Converter*

iterator.remove para removeIf.

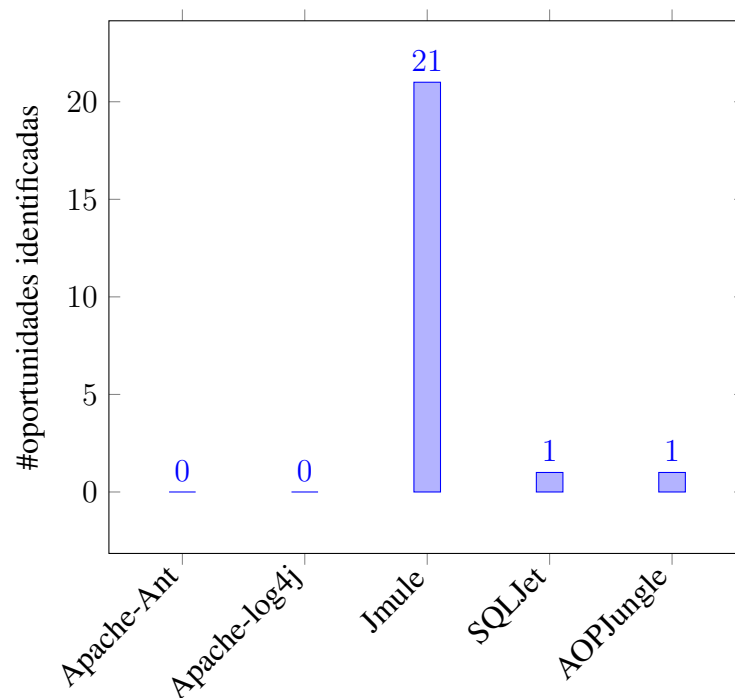
```

1 public boolean visit(IfStatement node) {
2     if (node.getThenStatement() instanceof ExpressionStatement)
3         if (ConvertToRemoveIf.compareMethodExpressions(
4             (ExpressionStatement) node.getThenStatement(),
5             "remove")) {
6         var aux = new ConvertToRemoveIf(
7             behaviourKind.getPackage().getName(),
8             behaviourKind.getCompilationUnit().getName(),
9             cUnit.getLineNumber(node.getStartPosition()));
10        refactoringList.addConvertToRemoveIf(aux);
11    } else if (node.getThenStatement() instanceof Block)
12        if (ConvertToRemoveIf.compareMethodBodies(
13            (Block) node.getThenStatement(), "remove")) {
14            ConvertToRemoveIf aux = new ConvertToRemoveIf(
15                behaviourKind.getPackage().getName(),
16                behaviourKind.getCompilationUnit().getName(),
17                cUnit.getLineNumber(node.getStartPosition()));
18            refactoringList.addConvertToRemoveIf(aux);
19        }
20 }

```

Após a execução da função de detecção, foram encontradas 23 oportunidades de refatoração. Sendo que 91% do total foram encontradas no projeto *Jmule*. A Figura 11 mostra a quantidade de oportunidades encontradas por projeto.

Figura 11 – Quantidade de oportunidades de refatoração para *Converter iterator.remove para removeIf*



As Listagens 25, 26 e 27 apresentam exemplos das oportunidades de refatorações encontradas no projeto *Jmule*. A Listagem 25 apresenta o primeiro exemplo de uma oportunidade encontrada. Os elementos são removidos da coleção `temporary_banned_peers` (linha 4), então é necessário chamar o método `removeIf` desta coleção. No argumento do método `removeIf` é criada uma expressão lambda que recebe a condição de remoção em seu corpo (linhas 6 à 8).

Listagem 25: Primeiro exemplo de Oportunidade de refatoração *Converter*

iterator.remove para removeIf.

```

1 for (TemporaryBannedIP banned_ip: temporary_banned_peers)
2     if((System.currentTimeMillis() - banned_ip.getWhenBanned())
3         >= banned_ip.getHowLong() )
4         temporary_banned_peers.remove( banned_ip );
5 //Estrutura refatorada
6 temporary_banned_peers.removeIf(banned_ip -> System
7     .currentTimeMillis() - banned_ip.getWhenBanned()) >=
8     banned_ip.getHowLong());

```

A Listagem 26 apresenta outro exemplo de oportunidade encontrada. A refatoração é aplicada na coleção `published_source`, é necessário utilizar o passo quatro da refatoração e chamar o método `keySet` no corpo da expressão lambda para obtenção dos dados corretos.

Listagem 26: Segundo exemplo de oportunidade de refatoração *Converter*

iterator.remove para removeIf.

```

1 for(Int128 id: published_source.keySet())
2     if (System.currentTimeMillis() - published_source.get(id)
3         > JKadConstants.TIME_5_HOURS)
4         published_source.remove(id);
5 //Estrutura refatorada
6 published_source.removeIf(id -> System.currentTimeMillis()
7     - published_source.get(id.keySet())
8     > JKadConstants.TIME_5_HOURS);

```

A Listagem 27 apresenta outro exemplo de uma oportunidade encontrada. Essa estrutura é similar a segunda, utilizando a variável `payload_peers.payload_loosed_peers` (linha 8) para chamar o método `removeIf`.

Listagem 27: Terceiro exemplo de Oportunidade de refatoração *Converter*

iterator.remove para removeIf.

```

1 for(UserHash hash: payload_peers.payload_loosed_peers.keySet()) {
2     if (System.currentTimeMillis()
3         - payload_peers.payload_loosed_peers.get(hash)
4         > ED2KConstants.TIME_24_HOURS)
5         payload_peers.payload_loosed_peers.remove(hash);
6     }
7 //Estrutura refatorada
8 payload_peers.payload_loosed_peers.removeIf((p ->
9     System.currentTimeMillis()
10    - payload_peers.payload_loosed_peers.get(p.keySet())
11    > ED2KConstants.TIME_24_HOURS);

```

4.3.8 Converter `iterator.set` para `replaceAll`

A Listagem 28 apresenta o trecho de código que obtém oportunidades de refatoração de acordo com a função de detecção. Os métodos que foram desenvolvidos para encontrar chamadas a outros métodos de acordo com o seu nome foram utilizados na implementação desta função de detecção (`compareMethodExpressions` e `compareMethodBodies`). Então, o código desenvolvido procura, por reflexão, chamadas ao método `set` diretamente nas estruturas `visit` para laços de repetição.

Listagem 28: Função com a expressão regular para encontrar chamadas ao método `set`.

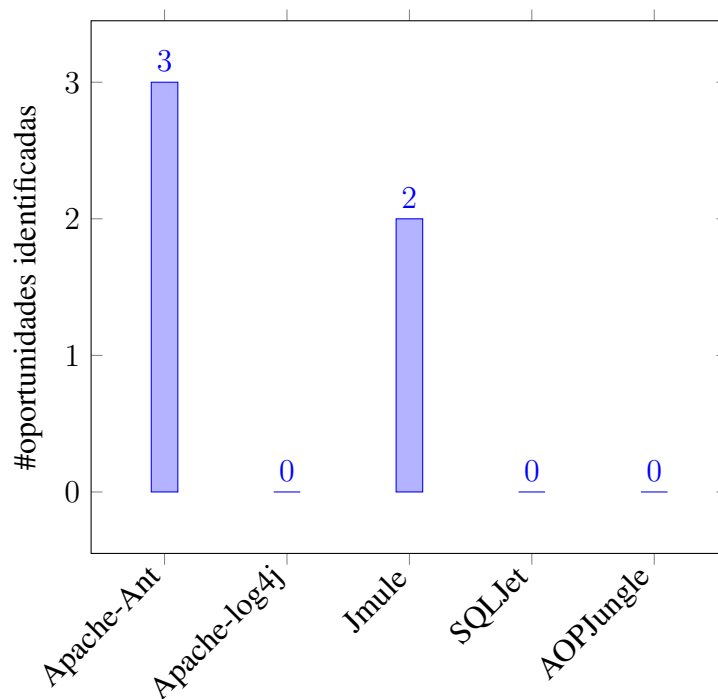
```

1  if (node.getBody() instanceof ExpressionStatement)
2      if (ConvertToReplaceAll.compareMethodExpressions(
3          (ExpressionStatement) node.getBody(), "set")) {
4      var aux = new ConvertToReplaceAll(
5          behaviourKind.getPackage().getName(),
6          behaviourKind.getCompilationUnit().getName(),
7          cUnit.getLineNumber(node.getStartPosition()));
8      refactoringList.addConvertToReplaceAll(aux);
9  }
10 if (node.getBody() instanceof Block)
11     if (ConvertToReplaceAll.compareMethodBodies(
12         (Block) node.getBody(), "set")) {
13     var aux = new ConvertToReplaceAll(
14         behaviourKind.getPackage().getName(),
15         behaviourKind.getCompilationUnit().getName(),
16         cUnit.getLineNumber(node.getStartPosition()));
17     refactoringList.addConvertToReplaceAll(aux);
18 }

```

Após a execução desta função de detecção foram encontradas cinco oportunidades de refatoração. Sendo que três no projeto *Apache-Ant* e outras duas no projeto *Imule*. A Figura 12 apresenta o total de oportunidades encontradas de acordo com os projetos analisados.

Figura 12 – Quantidade de oportunidades de refatoração para *Converter iterator.set para replaceAll*



A Listagens 29, 30 e 31 apresentam as oportunidades de refatoração retornadas do có-

digo *Apache-Ant*. A Listagem 29 apresenta a primeira estrutura, onde é possível aplicação da refatoração, visto que a variável `logicalLines` é uma *lista*.

Listagem 29: Primeiro exemplo de oportunidade encontrada da refatoração *Converter*

iterator.set para replaceAll.

```
1 private List<LogicalLine> logicalLines = new ArrayList<>();
2 for (pos++; pos <= end; pos++) {
3     logicalLines.set(pos, null);
4 }
5 //Estrutura refatorada
6 logicalLines.replaceAll(a -> a, null);
```

A Listagem 30 apresenta o segundo exemplo de oportunidade de refatoração encontrada. Essa estrutura foi um falso positivo para esse tipo de refatoração uma vez que essa refatoração é aplicada em coleções, não sendo possível sua aplicação em uma variável.

Listagem 30: Segundo exemplo de oportunidade encontrada da refatoração *Converter*

iterator.set para replaceAll.

```
1 BitSet bhtab = new BitSet(nBhtab);
2 for (i = 0; i < 256; i++) {
3     bhtab.set(ftab[i]);
4 }
```

A Listagem 31 apresenta o terceiro exemplo de oportunidade encontrada. Esta estrutura também não é possível a aplicação da refatoração, visto que a variável `bhtab` (linha 1) não é uma coleção e a refatoração é aplicada em coleções.

Listagem 31: Terceiro exemplo de oportunidade encontrada da refatoração *Converter*

iterator.set para replaceAll.

```
1 BitSet bhtab = new BitSet(nBhtab);
2 for (i = 0; i < 32; i++) {
3     bhtab.set(nblock + 2 * i);
4 }
```

4.3.9 Concatenar uma coleção com `join`

A Listagem 32 apresenta um trecho do código implementado para procurar por oportunidades de refatoração. Operações de concatenação são procuradas dentro de laços de repetição. O código foi implementado diretamente nas estruturas `visit` dos laços de repetição. Caso uma

atribuição dentro do laço possui o operador de concatenação salva-se a localização da estrutura como uma possível oportunidade de refatoração.

Listagem 32: Implementações para encontrar refatorações *Concatenar uma coleção com join*.

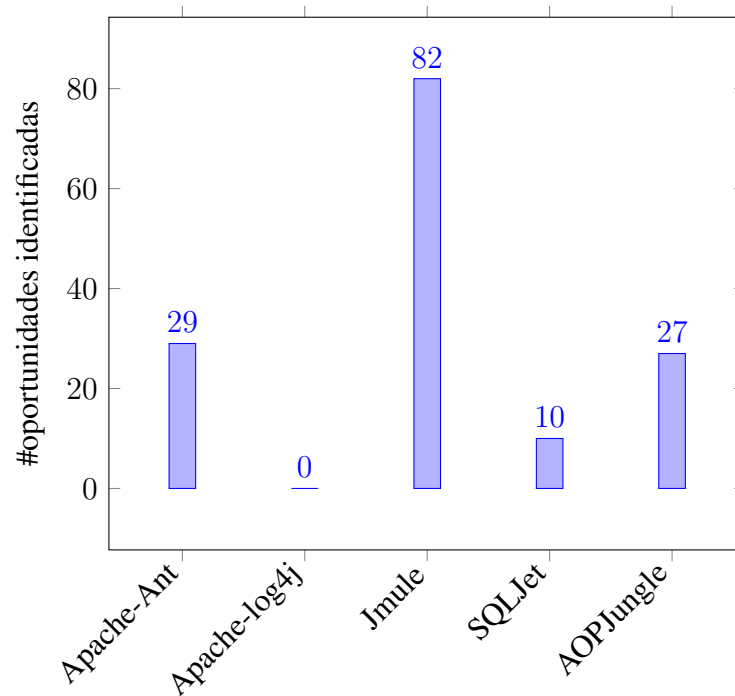
```

1  if (node.getBody() instanceof ExpressionStatement) {
2      var es = (ExpressionStatement) node.getBody();
3      if (ConvertJoinMethod.compareMethodExpressions(es, "append")) {
4          var aux = new ConvertJoinMethod(
5              behaviourKind.getPackage().getName(),
6              behaviourKind.getCompilationUnit().getName(),
7              cUnit.getLineNumber(node.getStartPosition()));
8          refactoringList.addConvertJoinMethod(aux);
9      }
10     else if (es.getExpression() instanceof Assignment) {
11         var assig = (Assignment) es.getExpression();
12         if (assig.getOperator().equals(Assignment.Operator.PLUS_ASSIGN)) {
13             var aux = new ConvertJoinMethod(
14                 behaviourKind.getPackage().getName(),
15                 behaviourKind.getCompilationUnit().getName(),
16                 cUnit.getLineNumber(node.getStartPosition()));
17             refactoringList.addConvertJoinMethod(aux);
18         }
19     }
20 }
21 if (node.getBody() instanceof Block) {
22     var block = (Block) node.getBody();
23     if (ConvertJoinMethod.compareMethodBodies(block, "append")) {
24         var aux = new ConvertJoinMethod(
25             behaviourKind.getPackage().getName(),
26             behaviourKind.getCompilationUnit().getName(),
27             cUnit.getLineNumber(node.getStartPosition()));
28         refactoringList.addConvertJoinMethod(aux);
29     }
30     else
31         for (var a : block.statements())
32             if (a instanceof ExpressionStatement) {
33                 var es = (ExpressionStatement) a;
34                 if (es.getExpression() instanceof Assignment) {
35                     var assig = (Assignment) es.getExpression();
36                     if (assig.getOperator().equals(
37                         Assignment.Operator.PLUS_ASSIGN)) {
38                         var aux = new ConvertJoinMethod(
39                             behaviourKind.getPackage().getName(),
40                             behaviourKind.getCompilationUnit().getName(),
41                             cUnit.getLineNumber(node.getStartPosition()));
42                         refactoringList.addConvertJoinMethod(aux);
43                     }
44                 }
45     }

```

Após a execução desta função de detecção foram encontradas 148 oportunidades de refatoração, sendo que 55% dessas estruturas foram encontradas no projeto *Jmule*. O projeto *Apache-log4j* não retornou locais para esse tipo de refatoração. A Figura 13 apresenta todas as oportunidades de refatoração encontradas de acordo com os projetos analisados.

Figura 13 – Quantidade de oportunidades de refatoração para *Concatenar uma coleção com join*



A Listagens 33, 34 e 35 apresentam oportunidades de refatoração sugeridas pela *API* no projeto *Jmule*. A Listagem 33 apresenta o primeiro exemplo de uma oportunidade de refatoração encontrada. Na primeira estrutura (linha 4) é possível aplicar a refatoração na variável `str` (linha 2), utilizando a lista `session_peers` (linha 3). Como não é uma lista com objetos do tipo `String` é necessário aplicar o passo cinco da refatoração, obtendo os valores da lista `session_peers` em `String` (linha 11 à 13). Ao final ainda é necessário realizar o passo seis da refatoração, aplicando o método `concat` (linha 13) para obter o mesmo valor que a estrutura anterior.

Listagem 33: Primeiro exemplo encontrado de oportunidade para aplicação da refatoração

Concatenar uma coleção com join.

```

1 public String toString() {
2     String str = "[ ";
3     for (Peer peer : session_peers)

```

```

4         str += peer + "\n";
5     str += "]";
6     return str;
7 }
8 //Estrutura refatorada
9 public String toString() {
10     String str = "[ ".join("\n", session_peers.stream()
11         .map(Object::toString)
12         .collect(Collectors.toList())) .concat("\n");
13     str += "]";
14     return str;
15 }

```

A Listagem 34 apresenta outro exemplo de oportunidade encontrada. Nesta estrutura é possível aplicar a refatoração na variável `result` (linha 2), também sendo necessário utilizar o passo cinco da refatoração. Nessa refatoração também foi necessário realizar o passo seis, encadeando o método `concat` (linha 13) para manter o padrão da estrutura anterior.

Listagem 34: Segundo exemplo encontrado de oportunidade para aplicação da refatoração

Concatenar uma coleção com `join`.

```

1 public String toString() {
2     String result = "";
3     for (PeerDownloadInfo download_status : peer_status_list) {
4         result += download_status + "\n";
5     }
6     return result;
7 }
8 //Estrutura refatorada
9 public String toString() {
10     String result = "".join("\n", peer_status_list.stream()
11         .map(Object::toString)
12         .collect(Collectors.toList())) .concat("\n");
13     return result;
14 }

```

A Listagem 35 apresenta outro exemplo de oportunidade encontrada pela *API*. Nesta estrutura também é possível aplicar a refatoração na variável `result` (linha 2), sendo utilizados os passos cinco e seis, conforme descrito nas refatorações anteriores.

Listagem 35: Primeiro exemplo encontrado de oportunidade para aplicação da refatoração

Concatenar uma coleção com `join`.

```

1 public String toString() {
2     String result = "";
3     for (Tag tag : tagList) {

```

```

4         result += tag + "\n";
5     }
6     return result;
7 }
8 //Estrutura refatorada
9 public String toString() {
10     String result = "".join("\n", tagList.stream()
11         .map(Object::toString)
12         .collect(Collectors.toList())).concat("\n");
13     return result;
14 }

```

4.3.10 Converter interface `Iterator` para `Spliterator`

A Listagem 36 mostra uma possível implementação da função de detecção para aplicação desta refatoração. Dentro do método `visit`, executado quando existe uma declaração de variável, é verificada a existência de uma declaração do tipo `Iterator`. Caso exista alguma variável desse tipo, é uma possível oportunidade de refatoração.

Listagem 36: Trechos de código para encontrar refatorações para nova interface `Spliterator`.

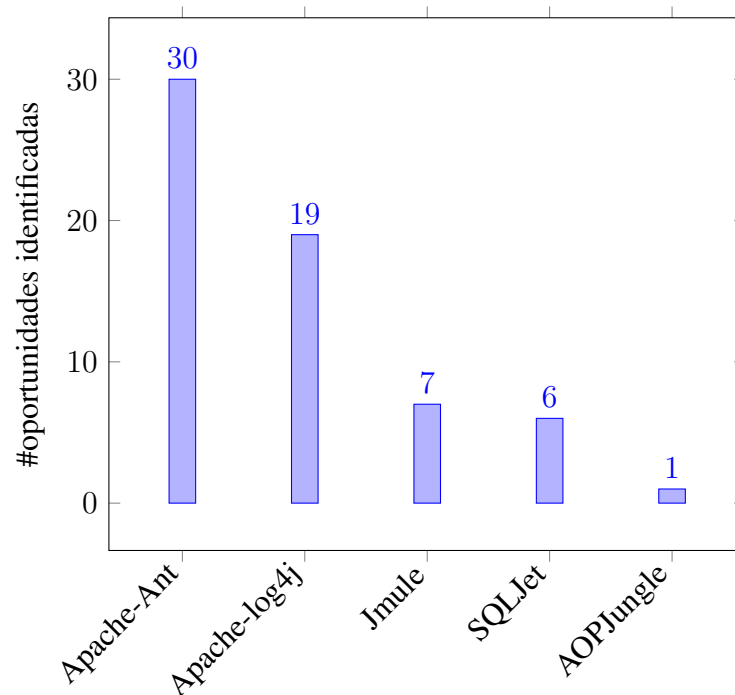
```

1 public boolean visit(VariableDeclarationStatement node) {
2     if (node.getType().toString().contains("Iterator")) {
3         ConvertSpliterator aux = new ConvertSpliterator(
4             behaviourKind.getPackage().getName(),
5             behaviourKind.getCompilationUnit().getName(),
6             cUnit.getLineNumber(node.getStartPosition()));
7         refactoringList.addConvertSpliterator(aux);
8     }
9 }

```

Após a execução desta função de detecção foram encontradas 63 oportunidades de refatoração, sendo que 48% desse total foram detectadas no projeto `Apache-Ant` e 30% no projeto `Apache-log4j`. Todos os projetos possuem ao menos uma oportunidade de refatoração. A Figura 14 apresenta a quantidade de oportunidades encontradas de acordo com os projetos analisados.

Figura 14 – Quantidade de oportunidades de refatoração para *Converter interface Iterator para Spliterator*



As Listagens 37, 38 e 39 mostram alguns exemplos de oportunidades de refatoração encontradas no projeto Apache-log4j. A Listagem 37 apresenta o primeiro exemplo das oportunidades detectadas. A estrutura é passível de refatoração, então o método `hasNext` (linha 2) da interface `Iterator` é substituído pelo método `tryAdvanced` (linha 10) da classe `Spliterator`. A variável de iteração (`it`, linha 3) junto com o método `next` são substituídos pelo parâmetro da expressão lambda (`p`, linha 10).

Listagem 37: Primeiro exemplo de oportunidade encontrada para a refatoração *Converter interface Iterator para Spliterator*.

```

1 final Iterator it = mPendingEvents.iterator();
2 while (it.hasNext()) {
3     final EventDetails event = (EventDetails) it.next();
4     mAllEvents.add(event);
5     toHead = toHead && (event == mAllEvents.first());
6     needUpdate = needUpdate || matchFilter(event);
7 }
8 //Estrutura refatorada
9 final Spliterator it = mPendingEvents.spliterator();
10 while (it.tryAdvance(p -> {
11     final EventDetails event = (EventDetails) p;
12     mAllEvents.add(event);
13     toHead = toHead && (event == mAllEvents.first());

```



```

14     needUpdate = needUpdate || matchFilter(event);
15 });});

```

A Listagem 38 apresenta outro exemplo de oportunidade encontrada pela *API*. A estrutura também é passível de refatoração. Sendo substituídos as funcionalidade da interface *Iterator* pelas da interface *Spliterator*.

Listagem 38: Segundo exemplo de oportunidade encontrada para a refatoração *Converter interface Iterator para Spliterator*.

```

1 Iterator converterIter = converters.iterator();
2 while (converterIter.hasNext()) {
3     Object converter = converterIter.next();
4     patternConverters[i] = (LoggingEventPatternConverter) converter;
5     i++;
6 }
7 //Estrutura refatorada
8 Spliterator converterIter = converters.spliterator();
9 while (converterIter.tryAdvance(p -> {
10     Object converter = p;
11     patternConverters[i] = (LoggingEventPatternConverter) converter;
12     i++;
13 }));

```

A Listagem 39 apresenta o terceiro exemplo de oportunidade de refatoração. Esta estrutura também pode ser refatorada, sendo que por ter apenas uma linha dentro do bloco *while* é possível suprimir o uso das chaves e ; na expressão lambda, passo que é tratado na refatoração *Functional Interface Instance to Lambda Expression* (TEIXEIRA JÚNIOR et al., 2014).

Listagem 39: Oportunidade de refatoração *Converter interface Iterator para Spliterator*.

```

1 Iterator it = logLevels.iterator();
2 while (it.hasNext()) {
3     register((LogLevel) it.next());
4 }
5 //Estrutura refatorada
6 Spliterator it = logLevels.spliterator();
7 while (it.tryAdvance(p -> register((LogLevel) p)));

```

4.3.11 Adicionar interface **Predicate**

A Listagem 40 apresenta a implementação da função de detecção para essa refatoração. Com a lista de expressões das estruturas *ifs* do código, o método verifica se existem estru-

ras condicionais com expressões de comparação iguais em um mesmo local (mesmo pacote e classe). Se existir, salva como uma possível oportunidade de refatoração.

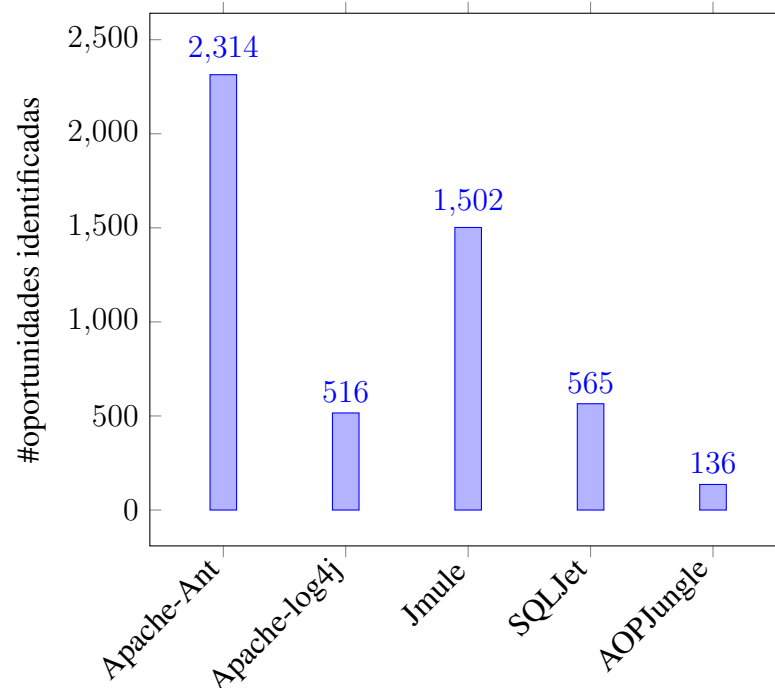
Listagem 40: Trecho de código para encontrar oportunidades de refatoração *Adicionar interface Predicate*.

```

1 public List<ConvertPredicate> getConvertPredicate () {
2     var newConvertPredicate = new ArrayList<ConvertPredicate>();
3     for (var i=0 ; i < this.convertPredicate.size(); i++) {
4         int aux = 0;
5         for (var j = 0; j < this.convertPredicate.size(); j++)
6             if (i == j)
7                 continue;
8             else if (this.convertPredicate.get(i).compare(
9                 this.convertPredicate.get(j)))
10                aux++;
11            if (aux != 0)
12                newConvertPredicate.add(
13                    this.convertPredicate.get(i));
14        }
15    return newConvertPredicate;
16 }
17 //...
18 public boolean compare (ConvertPredicate ob2) {
19     return this.getPackageName().equals(ob2.getPackageName()) &&
20         this.getClassName().equals(ob2.getClassName()) &&
21         this.getExpression().equals(ob2.getExpression());
22 }

```

Após a execução esta função de detecção foram encontradas 5.033 oportunidades, sendo que 46% desse total foram detectadas no projeto Apache-Ant e 30% no projeto Jmule. Todos os projetos retornaram um número relativamente alto de oportunidades de refatoração. O projeto que apresentou o menor número de oportunidades detectadas foi o AOPJungle com 136 oportunidades de refatoração. Os demais resultados podem ser observados no gráfico da Figura 15.

Figura 15 – Quantidade de oportunidades para *Adicionar interface Predicate*

As Listagens 41, 42 e 43 mostram exemplos de oportunidades de refatoração obtidas no projeto *Jmule*. A Listagem 41 apresenta o primeiro exemplo encontrado pela *API*. A estrutura é passível de refatoração, a variável `region_code` é a que mais tem ocorrências na expressão booleana, então ela foi escolhida para ser substituída na expressão lambda, de acordo com o passo dois da refatoração. Também identificou-se que a variável é do tipo `int`, sendo utilizado, então, a classe `Integer` (linha 13) na estrutura `Predicate` da refatoração.

Listagem 41: Primeiro exemplo de oportunidade detectada para a refatoração *Adicionar interface Predicate*.

```

1  if ((region_code.charAt(1) >= 48) & (region_code.charAt(1) <
2      (48 + 10))) {
3      region_code2 = ((region_code.charAt(0)-65)*10)+(region_code
4          .charAt(1)-48)+100;
5  }
6  //...
7  if ((region_code.charAt(1) >= 48) & (region_code.charAt(1) <
8      (48 + 10))) {
9      region_code2 = ((region_code.charAt(0)-48)*10)+(region_code
10         .charAt(1)-48);
11 }
12 //Estrutura refatorada
13 Predicate<Integer> testNum = p -> (p.charAt(1) >= 48) &
14     (p.charAt(1) < (48 + 10));

```

```

15 if (testNum.test(region_code)) {
16     region_code2 = ((region_code.charAt(0)-65)*10)+(region_code
17         .charAt(1)-48)+100;
18 }
19 //...
20 if (testNum.test(region_code)) {
21     region_code2 = ((region_code.charAt(0)-48)*10)+(region_code
22         .charAt(1)-48);
23 }

```

A Listagem 42 apresenta outro exemplo de oportunidade detectada. A variável que mais se repete é a variável `value`. Identificou-se também que a variável é do tipo `int`. As linhas 14 e 18 apresentam um outro exemplo de estruturas que podem ser refatoradas, dois *ifs* iguais, em lugares diferentes e com o mesmo tratamento de erro.

Listagem 42: Segundo exemplo de oportunidade detectada para a refatoração *Adicionar interface Predicate*.

```

1 if ( ( value >= 0 ) && ( value <= 65535 ) )
2     throw new ConfigurationManagerException("The port
3         can be between 0 and 65535, not " + value + " ");
4 //...
5 if ( ( value >= 0 ) && ( value <= 65535 ) )
6     throw new ConfigurationManagerException("The
7         port can be between 0 and 65535, not " + value + " ");
8 //Estrutura refatorada
9 Predicate<Integer> testVal = p -> ( p >= 0 ) && ( p <= 65535 );
10 if (testVal.test(value))
11     throw new ConfigurationManagerException("The port
12         can be between 0 and 65535, not " + value + " ");
13 //...
14 if (testVal.test(value))
15     throw new ConfigurationManagerException("The
16         port can be between 0 and 65535, not " + value + " ");

```

A Listagem 43 apresenta outra oportunidade encontrada. É possível utilizar a refatoração neste estrutura, porém como é utilizado um método para realizar a verificação, pode haver poucos benefícios em utilizar essa refatoração. A única varável da expressão é a `contact` e foi identificada que é do tipo `KadContact`.

Listagem 43: Oportunidades de refatoração *Adicionar interface Predicate*.

```

1 if (contact.supportKad2())
2     packet = PacketFactory.getBootStrapReq2Packet();
3 //...
4 if (contact.supportKad2())

```

```

5     packet = PacketFactory.getHelloReq2Packet (
6         TagList.EMPTY_TAG_LIST);
7 //Estrutura refatorada
8 Predicate<KadContact> testContact = p -> p.supportKad2();
9 if (testContact.test (contact))
10    packet = PacketFactory.getBootStrapReq2Packet ();
11 //...
12 if (testContact.test (contact))
13    packet = PacketFactory.getHelloReq2Packet (
14        TagList.EMPTY_TAG_LIST);

```

4.3.12 Converter para média lambda

As Listagens 44, 45 e 46 vão apresentar os métodos construídos para a implementação desta função de detecção. A Listagem 44 apresenta o trecho de código implementado no método `visit` para declarações de métodos (`MethodDeclaration`). Então, a *API* sempre entra neste método `visit` quando existe uma declaração de algum método. Dentro do método `visit` é possível obter acesso a todos os elementos dos métodos visitados. Sendo assim, a linha 3 faz a verificação se o corpo do método possui um bloco de código (`Block`). Na linha 4 é verificado se, no interior do método, possui alguma variável que seja inicializada por zero (partindo do princípio que poderá ser realizada uma acumulação de valores). Caso exista, é verificado se essa mesma variável é o dividendo de alguma operação (linha 6), caracterizando assim, uma média (soma dos elementos dividido pelo número total deles).

Listagem 44: Método `visit` encontrar oportunidades de refatoração *Converter para média lambda*.

```

1 public boolean visit(MethodDeclaration node) {
2 //...
3 if (node.getBody() instanceof Block) {
4     var varName = ConvertAverageLambda.varNameEqualZero(
5         (Block) node.getBody());
6     if (varName != null)
7         if (ConvertAverageLambda.varDivide((Block) node.getBody(),
8             varName)) {
9         var aux = new ConvertAverageLambda(
10             cUnit.getPackage().getName(),
11             cUnit.getName(),
12             cUnit.getLineNumber(node.getStartPosition()));
13         refactoringList.addConvertAverageLambda(aux);
14     }
15 }
16 //...
17 }

```

A Listagem 45 apresenta o método `varNameEqualZero` que retorna, por reflexão, o nome da variável que é inicializada por 0, dentro de um método, se existir. Primeiramente é quebrado o bloco do método em pedaços, para acessar cada elemento contido no método (linha 2). Após isso, verifica-se dentro do método as declarações de variáveis que existem (linha 3). Após isso, é obtido o a variável do tipo `VariableDeclarationFragment` (linha 7), onde é possível obter a inicialização da variável, por reflexão, então é feita a comparação se a variável é inicializada por 0, se for retorna o nome da variável.

Listagem 45: Método `varNameEqualZero` para encontrar variáveis inicializadas por 0.

```

1 public static SimpleName varNameEqualZero(Block bodyMethod) {
2     for (var a : bodyMethod.statements()) {
3         if (a instanceof VariableDeclarationStatement) {
4             var vds = (VariableDeclarationStatement) a;
5             if (vds.fragments().get(0) instanceof
6                 VariableDeclarationFragment) {
7                 var vdf = (VariableDeclarationFragment) vds.fragments()
8                     .get(0);
9                 if (vdf.getInitializer().toString().equals("0"))
10                    return vdf.getName();
11            }
12        }
13    }
14    return null;
15 }

```

A Listagem 46 apresenta o método `varDivide` que retorna `true` caso a variável passada por parâmetro seja dividendo de uma operação de divisão. Com o bloco do método e o nome da variável que retorna 0, procura-se por operações aritméticas dentro do método. Então, quando é uma declaração de variável (o desenvolvedor cria a variável e já executa o cálculo) o fluxo do programa entra no primeiro `if` (linha 3), caso a variável já foi declarada e só receba o cálculo, então o fluxo do programa pula para o *else-if* da linha 21. Caso entre no primeiro `if` (linha 3), o programa decompõe a variável declarada até chegar na operação matemática, representada pela classe `InfixExpression` (linha 10). Caso chegue até a classe `InfixExpression` é analisada se a operação é uma divisão (`Operator.DIVIDE`, linhas 13 e 24) e se o operador do lado esquerdo da divisão (`getLeftOperand`, linhas 14 e 25) é o mesmo recebido por parâmetro.

Listagem 46: Métodos para encontrar oportunidades de refatoração *Converter para média lambda*.

```

1 public static boolean varDivide(Block bodyMethod,
2     SimpleName varName) {
3     for (var a : bodyMethod.statements())
4         if (a instanceof VariableDeclarationStatement) {
5             var vds = (VariableDeclarationStatement) a;
6             if (vds.fragments().get(0) instanceof
7                 VariableDeclarationFragment) {
8                 var vdf = (VariableDeclarationFragment) vds
9                     .fragments().get(0);
10                if (vdf.getInitializer() instanceof InfixExpression) {
11                    var ie = (InfixExpression) vdf.getInitializer();
12                    if (ie.getOperator().equals(InfixExpression
13                        .Operator.DIVIDE) &&
14                        ie.getLeftOperand() instanceof SimpleName) {
15                        var sn = (SimpleName) ie.getLeftOperand();
16                        return sn.getIdentifier().contains(varName
17                            .toString());
18                    }
19                }
20            }
21        } else if (a instanceof InfixExpression) {
22            var ie = (InfixExpression) a;
23            if (ie.getOperator().equals(InfixExpression
24                .Operator.DIVIDE) &&
25                ie.getLeftOperand() instanceof SimpleName) {
26                var sn = (SimpleName) ie.getLeftOperand();
27                return sn.getIdentifier().contains(varName.toString());
28            }
29        }
30    return false;
31 }

```

Esta função de detecção não retornou oportunidades de refatoração após sua execução. Um dos possíveis motivos é que, por esta refatoração tratar exclusivamente da realização de médias, pode acontecer dos projetos analisados não realizarem esse tipo de cálculo em seus códigos (ou o realizam de uma forma diferente da que foi explorada pela função de detecção).

4.3.13 Converter para contagem lambda

A Listagem 47 apresenta trecho do código da implementação da função de detecção para esta refatoração. No método `visit` para operações de atribuição (`Assignment`) duas estruturas condicionais foram implementadas. A primeira condição verifica se o operador da atribuição é o mais igual (`+=`) e se está incrementando em um (`1`). A segunda condição verifica

se o operador é o sinal de igual (=), se for é realizada uma comparação entre a variável antes e depois da igualdade, caso sejam iguais, a operação seja uma soma (+) e está sendo incrementado em um é adicionado como possível oportunidade de refatoração.

Listagem 47: Trecho de código para encontrar oportunidades de refatoração *Converter para contagem lambda*.

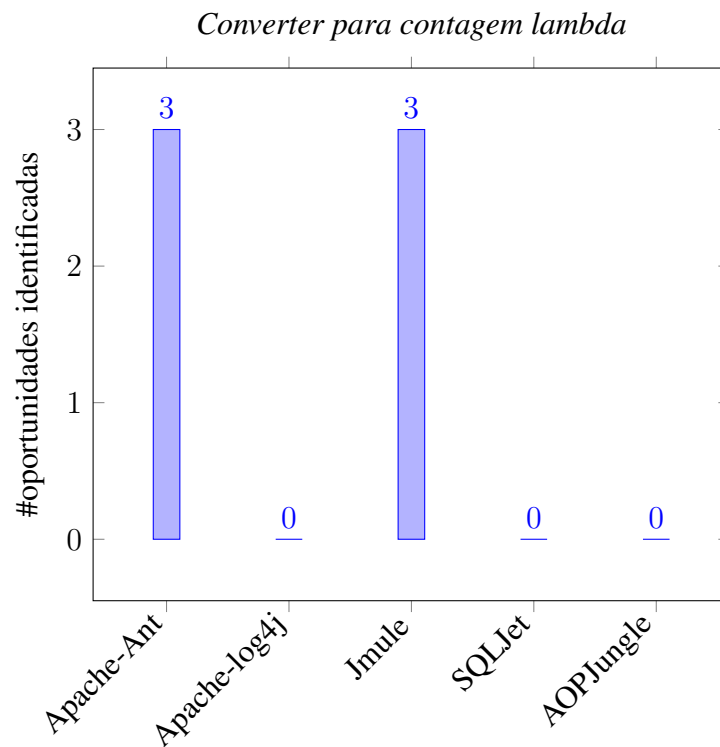
```

1 public boolean visit(Assignment node) {
2 //...
3     if (node.getOperator().equals(
4         Assignment.Operator.PLUS_ASSIGN) &&
5         node.getRightHandSide().toString().contentEquals("1")) {
6         var aux = new ConvertLambdaSum(
7             behaviourKind.getPackage().getName(),
8             behaviourKind.getCompilationUnit().getName(),
9             cUnit.getLineNumber(node.getStartPosition()));
10        refactoringList.addConvertLambdaSum(aux);
11    } else if (node.getOperator().equals(Assignment.Operator.ASSIGN)
12        && node.getRightHandSide() instanceof InfixExpression) {
13        var ie = (InfixExpression) node.getRightHandSide();
14        if (node.getLeftHandSide().toString().contentEquals(
15            ie.getLeftOperand().toString())
16            && ie.getOperator().equals(
17                InfixExpression.Operator.PLUS)
18            && ie.getRightOperand().toString()
19                .contentEquals("1")) {
20            ConvertLambdaSum aux = new ConvertLambdaSum(
21                behaviourKind.getPackage().getName(),
22                behaviourKind.getCompilationUnit().getName(),
23                cUnit.getLineNumber(node.getStartPosition()));
24            refactoringList.addConvertLambdaSum(aux);
25        }
26    }
27    ...
28 }

```

Foram encontradas seis oportunidades de refatoração, sendo três no projeto *Apache-Ant* e outras três no projeto *Imule*. Nos outros projetos não foram encontradas oportunidades, conforme pode ser observado na Figura 16.

Figura 16 – Quantidade de oportunidades de refatoração para *Converter para contagem lambda*



As Listagens 48, 49 e 50 apresentam exemplos de oportunidades encontradas pela *API* no projeto *Apache-Ant*. A Listagem 48 apresenta o primeiro exemplo detectado. A princípio a estrutura é passível de refatoração, pois a variável `tokFile` (linha 2) é uma lista que contém valores.

Listagem 48: Primeiro exemplo de oportunidade detectada para a refatoração *Converter para contagem lambda*.

```

1 depth = 0;
2 while (tokFile.hasMoreTokens()) {
3     depth += 1;
4 }
5 //Estrutura refatorada
6 depth = tokFile.stream()
7     .filter(p -> p.hasMoreTokens())
8     .mapToInt(p -> 1)
9     .sum();

```

A Listagem 50 apresenta a segunda oportunidade detectada pela *API*. É um provável falso positivo, uma vez que, analisando o código, a contagem não é realizada em uma coleção, sendo que essa refatoração é aplicada em coleções.

Listagem 50: Segundo exemplo de oportunidade detectada para a refatoração *Converter para contagem lambda*.

```

1 while (startIndex >= 0
2     && (startIndex + startToken.length()) <= line.length()) {
3     String replace = null;
4     int endIndex = line.indexOf(endToken, startIndex
5         + startToken.length());
6     if (endIndex < 0) {
7         startIndex += 1;
8     }
9 }

```

A Listagem 51 apresenta o terceiro exemplo encontrado, a estrutura também não é possível aplicar a refatoração. Pois, como já comentado, esta refatoração é aplicada em contagem realizadas em coleções e, analisando o código, não foram identificadas coleções.

Listagem 51: Oportunidades de refatoração *Converter para contagem lambda*.

```

1 if (!validToken) {
2     startIndex += 1;
3 }

```

4.3.14 Adicionar método `of`

A Listagem 52 apresenta a implementação da função de detecção para esta refatoração. No método `visit` de atribuições (`Assignment`) é verificada a existência de uma chamada ao método para criar coleções não modificáveis. Então, caso exista uma chamada a um método, e essa chamada possui o conjunto de caractere `unmodifiable`, então a localização desta estrutura é salva como uma possível oportunidade de refatoração.

Listagem 52: Trecho de código para encontrar oportunidades de refatoração *Adicionar método `of`*.

```

1 public boolean visit(Assignment node) {
2     if (node.getRightHandSide() instanceof MethodInvocation) {
3         var mi = (MethodInvocation) node.getRightHandSide();
4         if (mi.getName().toString().contains("unmodifiable")) {
5             var aux = new AddOfMethod(
6                 behaviourKind.getPackage().getName(),
7                 behaviourKind.getCompilationUnit().getName(),
8                 cUnit.getLineNumber(node.getStartPosition()));
9             refactoringList.addAddOfMethod(aux);

```

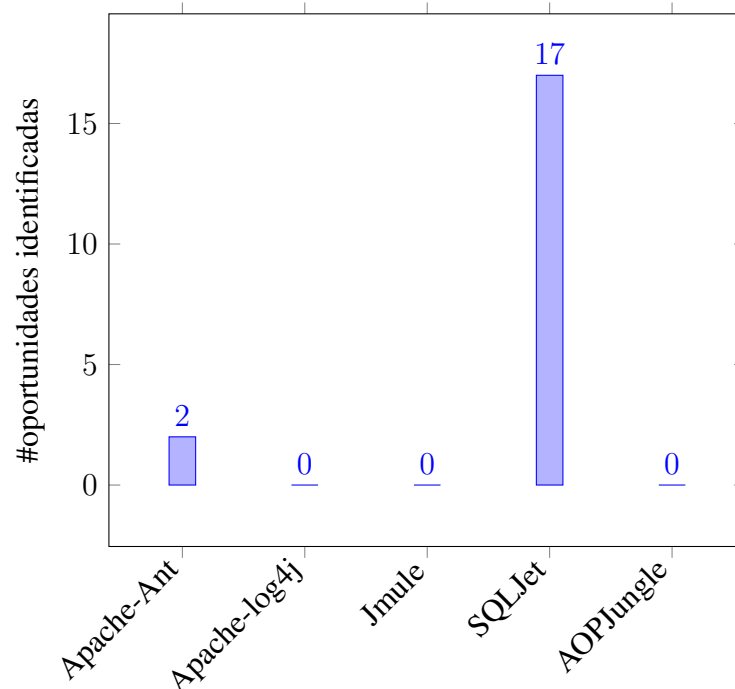
```

10         }
11     }
12 }

```

Após a execução desta função de detecção foram encontradas 19 oportunidades de refatoração, sendo 89% das oportunidades foram encontradas no projeto *SQLJet*, o restante disso no projeto *Apache-Ant*. Os outros projetos não retornaram locais para refatoração. A Figura 17 apresenta a quantidade de oportunidades encontradas pela *API*.

Figura 17 – Quantidade de oportunidades de refatoração *Adicionar método of*



A Listagem 53 apresenta um exemplo retirado do projeto *SQLJet*. Duas listas não modificáveis são criadas (linhas 19 e 20), porém os valores foram inseridos dinamicamente. Como a refatoração trata de casos em que os valores são adicionados tanto manualmente como dinamicamente, as oportunidades descobertas são possíveis de serem refatoradas. A partir da linha 21 é apresentada a estrutura refatorada, sendo utilizado o método `of` (linhas 38 e 39) da interface `List`, pois as variáveis `conditions` (linha 1) e `values` (linha 3) são variáveis do tipo `List`.

Listagem 53: Estruturas recomendadas para refatoração *Adicionar método of*.

```

1 List<ISqlJetExpression> conditions = new
2     ArrayList<ISqlJetExpression> ();
3 List<ISqlJetExpression> values = new

```

```

4         ArrayList<ISqlJetExpression>();
5 while (idx < ast.getChildCount()) {
6     if ("when".equalsIgnoreCase(child.getText())) {
7         ISqlJetExpression condition = create((CommonTree)
8             child.getChild(0));
9         ISqlJetExpression value = create((CommonTree)
10            child.getChild(1));
11        conditions.add(condition);
12        values.add(value);
13        child = (CommonTree) ast.getChild(idx++);
14    } else {
15        break;
16    }
17 }
18 this.conditions = Collections.unmodifiableList(conditions);
19 this.values = Collections.unmodifiableList(values);
20 //Estrutura refatorada
21 List<ISqlJetExpression> conditions = new
22     ArrayList<ISqlJetExpression>();
23 List<ISqlJetExpression> values = new
24     ArrayList<ISqlJetExpression>();
25 while (idx < ast.getChildCount()) {
26     if ("when".equalsIgnoreCase(child.getText())) {
27         ISqlJetExpression condition = create((CommonTree)
28            child.getChild(0));
29         ISqlJetExpression value = create((CommonTree)
30            child.getChild(1));
31        conditions.add(condition);
32        values.add(value);
33        child = (CommonTree) ast.getChild(idx++);
34    } else {
35        break;
36    }
37 }
38 this.conditions = List.of(conditions);
39 this.values = List.of(values);

```

4.3.15 Converter para *underscore*

A Listagem 41 apresenta trechos de código da implementação da função de detecção. Foi criado o método `visit` para expressões lambda (`LambdaExpression`), que ainda não havia sido implementado no *AOPJungle*. A classe `AOJLambdaExpression` foi também implementada no *AOPJungle* para armazenar informações sobre as expressões lambda. Por último, foi criado um método que recebe uma lista de expressões lambda do código (linha 19). A cada expressão lambda, os seus parâmetros são percorridos, procurando ocorrências deles em seu corpo. Se alguma variável dos argumentos não existir no corpo da expressão, é

incrementada a variável `countRefactoring` na linha 28 e essa estrutura permanece na lista de refatorações. Caso todos os parâmetros existirem no corpo da expressão, a variável não é incrementada e a expressão lambda é removida da lista de refatorações.

Listagem 41: Trechos de código para encontrar oportunidades de refatoração *Converter para underscore*.

```

1 public boolean visit(LambdaExpression node) {
2     AOJLambdaExpression aojNode = new AOJLambdaExpression(node,
3         (AOJProgramElement) getLastMemberFromStack());
4     AOJBehaviourKind behaviourKind =
5         getAOJBehaviourKind(getLastMemberFromStack());
6     if (behaviourKind != null) {
7         ConvertUnderscoreParameters refactorUnderscore =
8             new ConvertUnderscoreParameters(
9                 behaviourKind.getPackage().getName(),
10                behaviourKind.getCompilationUnit().getName(),
11                cUnit.getLineNumber(node.getStartPosition()),
12                aojNode);
13        refactoringList.addConvertUnderscoreParameters(
14            refactorUnderscore);
15    }
16    return super.visit(node);
17 }
18
19 public static void listConvertUnderscore(
20     List<ConvertUnderscoreParameters> listRefactor) {
21     for (ConvertUnderscoreParameters lambdaEx : listRefactor) {
22         int countRefact = 0;
23         for (VariableDeclaration var : lambdaEx.getAojNode()
24             .getParameters()) {
25             int varInBody = 0;
26             if (lambdaEx.getAojNode().getBody().contains(var
27                 .toString()))
28                 varInBody++;
29             if (varInBody == 0)
30                 countRefact++;
31         }
32         if (countRefact == 0)
33             listRefactor.remove(lambdaEx);
34     }
35 }

```

Não foram encontrados locais para esse tipo de refatoração. Um dos possíveis motivos pode ser o fato das expressões lambda serem um conceito relativamente novo para os desenvolvedores Java. Então, grande parte dos projetos, embora sejam atualizados, ainda não usam as novas estruturas lambdas, e se a usam, provavelmente são em um número relativamente pe-

queno, não se encaixando nesse tipo de refatoração. Porém, futuramente, esta refatoração pode ser importante.

4.4 DISCUSSÃO

Para o desenvolvimento das funções de detecção propostas neste trabalho, o projeto *AOPJungle* foi aprimorado, sendo adicionadas novas classes e funcionalidades para obtenção de informações. Porém, suas estruturas internas não foram modificadas, optando-se por um desenvolvimento paralelo às funcionalidades da ferramenta.

Este estudo de caso forneceu subsídios suficientes para avaliar as refatorações e funções de detecção criadas. Foram desenvolvidas 15 funções de detecção e 12 retornaram oportunidades de refatoração. Na maioria das estruturas retornadas foi possível aplicar a refatoração pretendida. Porém, em algumas oportunidades, não foi possível a aplicação da refatoração. Um exemplo é a refatoração *Converter para contagem lambda* em que apenas uma oportunidade foi possível aplicar a refatoração. Outro exemplo é a refatoração *Converter iterator.set para replaceAll*. Geralmente essa refatoração não pode ser realizada quando as operações no código são realizadas sem o uso de uma interface de coleção, ou pelo menos quando a coleção não foi identificada. Porém, com a análise do desenvolvedor do código, ou uma análise mais substancial, podem existir formas de refatorar as oportunidades encontradas.

Após o término deste estudo, três funções de detecção não retornaram resultados de acordo com os projetos analisados. Um dos motivos é por serem refatorações de estruturas bem específicas, como funcionalidades implementadas diretamente no banco de dados, a exemplo da refatoração *Agrupar com groupingBy*, ou realização de médias aritméticas, como a refatoração *Converter para média lambda*, ou uma refatoração aplicada em Java 10, como a refatoração *Converter para underscore*. Futuramente essas funções de detecção podem ser aprimoradas ampliando a sua cobertura, adicionando outras formas de funcionamento, já que em alguns casos podem existir outros meios e maneiras de fazer o que a refatoração propõe.

A descrição do catálogo é informal, como geralmente ocorre em catálogos de refatorações ou padrões de design. Isso pode levar a imprecisões ao aplicar as refatorações (manualmente ou usando uma ferramenta). Não é possível precisar se todas as oportunidades encontradas possam ser refatoradas. Nos catálogos descritos usando o formato canônico, geralmente os desenvolvedores avaliam a segurança das refatorações usando casos de teste. Abordagens mais formais para descrever essas refatorações podem ser aplicadas no futuro para aliviar isso.

Geralmente, descrições informais de refatorações são retratadas antes de qualquer declaração rigorosa estar em vigor. Este é o caso, por exemplo, do catálogo de refatoração de FOWLER; BECK, cujas descrições formais completas só apareceram cinco anos depois. Além disso, os critérios usados para procurar oportunidades de refatoração consideraram uma estratégia ampla (todos os locais onde as refatorações podem ser aplicadas), o que pode levar a um grande número de oportunidades encontradas. A avaliação de precisão e recuperação de um subconjunto das oportunidades pode ajudar na definição de heurísticas para classificar as oportunidades de refatoração encontradas.

4.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou as informações necessárias acerca da construção de uma *API*, a qual faz uso de recursos e funcionalidades de outras ferramentas, como *Eclipse* e *AOPJungle*. Com o uso da *API*, foi possível realizar a implementação das funções de detecção apresentadas ao final do catálogo e obter as informações necessárias para a realização de um estudo de caso, com a finalidade de testar e validar as funções de detecção desenvolvidas.

Neste capítulo, também foram apresentadas informações a respeito de cinco projetos aberto, utilizados para realizar uma análise de suas estruturas internas e mostrar exemplos reais de trechos de código que poderiam ser refatorados. Então, para cada refatoração, foi apresentada a quantidade total de oportunidades encontradas, exemplos dos trechos de código de algumas estruturas apontadas para refatoração e essas mesmas estruturas após a aplicação da refatoração. Para finalizar, foi feita uma discussão em relação aos resultados obtidos. O próximo capítulo apresenta as considerações finais e os trabalhos futuros.

5 CONCLUSÃO

Esta dissertação apresentou procedimentos e ferramentas a fim de guiar desenvolvedores para atualização de código Java. A reestruturação de um sistema para utilizar novas ferramentas e funcionalidades recentes de uma linguagem pode aproveitar vantagens como a utilização do paralelismo dos processadores, estruturas mais rápidas com um melhor desempenho, obtendo-se, assim, um ganho competitivo para um mercado com sistemas cada vez mais potentes, com mais linhas de código e muitas informações sendo processadas. Este trabalho oferece algumas ferramentas que podem ser úteis para a detecção de oportunidades de atualização, a fim de oferecer um melhor desempenho, redução de custos e esforços em manutenção.

O escopo principal deste trabalho pode ser usado e adaptado para outras linguagens de programação, uma vez que os procedimentos apresentados foram descritos da forma mais geral possível, embora, neste trabalho, as propostas foram pensadas e implementadas para a linguagem Java. Primeiramente um total de 15 refatorações para evolução de código Java foram apresentadas, em um formato canônico e como complemento das refatorações exposta por TEIXEIRA JÚNIOR et al., 2014. Outras 15 refatorações inversas são apresentadas e estão nos anexos dessa dissertação. O catálogo total, incluindo as refatorações apresentadas por TEIXEIRA JÚNIOR et al., 2014, corresponde a 50 refatorações que podem ser usadas para evolução de código Java.

Um dos problemas enfrentados por muitos desenvolvedores é localizar lugares para aplicar refatorações em sistemas com muitas linhas de código. Pesando nisso, ao final do catálogo foram propostas 15 funções de detecção para as refatorações de evolução de código, fornecendo subsídios suficientes para o desenvolvimento de uma ferramenta com a finalidade de localizar automaticamente possíveis trechos de código que podem ser aplicadas as refatorações apresentadas.

No desenvolvimento deste trabalho muitas decisões foram tomadas, a mais importante foi em relação a forma que as funções de detecção seriam desenvolvidas. Em um primeiro momento pensou-se em desenvolver uma ferramenta completa para extrair informações de código fonte. Porém, já havia sido desenvolvida no Grupo de Linguagens de Programação e Bancos de Dados da UFSM uma ferramenta para extração de informações e dados de código fonte, chamado *AOPJungle*. Optou-se, então, de forma paralela ao funcionamento da ferramenta, obter as informações necessárias para o desenvolvimento das funções de detecção, o que demonstrou-se

um desempenho satisfatório e suficiente para implementação e execução das funções propostas.

Por fim, para a implementação das funções de detecção e obtenção dos resultados, foi desenvolvida uma *API* que utilizou, principalmente, de funcionalidades do *framework AOP-Jungle*. Algumas estruturas foram adicionadas no *framework*, deixando-o mais robusto e com novas funcionalidades que, inclusive, podem ser utilizadas para outros propósitos futuramente. Com isso, as funções de detecção desenvolvidas foram colocadas em teste. Um estudo de caso de uso foi realizado em um total de cinco projetos com código aberto. Foram encontrados 11.437 oportunidades para refatoração, um número relativamente expressivo.

Em conjunto com a realização do estudo de caso, algumas refatorações e funções de detecção foram modificadas, pois não estavam de acordo com os resultados obtidos. Ao final, as estruturas exemplificadas no estudo de caso mostraram que tanto as refatorações, como as funções de detecção, podem ser comumente utilizadas pelos desenvolvedores para aplicação manual de melhoria no código fonte. Algumas das funções de detecção implementadas não retornaram resultados. Acredita-se que os motivos podem ser os projetos analisados realmente não apresentarem as funcionalidades esperadas. Porém, algumas melhorias podem ser implementadas no projeto e serão sugeridas na próxima seção.

5.1 APRIMORAMENTOS DO PROJETO E TRABALHOS FUTUROS

Priorização das oportunidades de refatoração: Pode ser importante desenvolver uma classificação entre as oportunidades de refatoração. Uma forma de realizar esse processo pode ser através do desenvolvimento de critérios pré-definidos para cada tipo de refatoração, estabelecendo níveis de importâncias entre as oportunidades encontradas ou de outros mecanismos de classificação (PIVETA, 2009).

Novas refatorações: Embora um número expressivo de refatorações já tenha sido sugeridas, um complemento poderia ser implementado, já que muitas refatorações limitaram-se ao escopo da versão 8 de Java. Nas versões 9 e 10 de Java novas refatorações podem ser propostas, é possível citar algumas como: (i) a evolução da estrutura *try-with-resources*, que em novas versões da linguagem não é necessário usar variáveis intermediárias para abrir recursos dentro do bloco, (ii) foi adicionada uma nova classe para tratar *strings*, chamada de `CompactString`, que dependendo do caso pode reduzir o consumo de memória pela metade, (iii) foram adicionados novos métodos na interface *Stream* em Java 9, como `takeWhile`, `dropWhile`, `ofNullable`, `iterate`, entre outros que merecem uma atenção e, dependendo do caso, até

a definição de uma refatoração.

Construção e melhorias das funções de detecção: Esse trabalho propôs funções de detecção para todas as refatorações de evolução de código, porém algumas das refatorações propostas por TEIXEIRA JÚNIOR et al., 2014 ainda não possuem função de detecção. Seria interessante implementá-las, além de propor melhorias para as funções presente neste trabalho, principalmente as que não retornaram resultados.

Estender a Linguagem AQL: A AQL (Aspect Query Language) é uma linguagem declarativa, projetada para pesquisas em dados estruturados, de acordo com um metamodelo (FAVERI et al., 2013). O metamodelo é extraído da ferramenta *AOPJungle*, o que significa que algumas atualizações já foram realizadas no metamodelo principal, porém falta organizar a compatibilidade da linguagem (palavras reservadas, novas classes, etc) para as estruturas adicionadas.

REFERÊNCIAS

- BARUA, A.; CHEON, Y. **A catalog of while loop specification patterns**. [S.l.]: Technical Report 14-65, Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, 2014.
- COHEN, W.; RAVIKUMAR, P.; FIENBERG, S. A comparison of string metrics for matching names and records. In: KDD WORKSHOP ON DATA CLEANING AND OBJECT CONSOLIDATION. **Anais...** [S.l.: s.n.], 2003. v.3, p.73–78.
- DAVID, C.; KESSELI, P.; KROENING, D. Kayak: safe semantic refactoring to Java streams. **arXiv preprint arXiv:1712.07388**, [S.l.], 2017.
- FAVERI, C. d. et al. **Uma linguagem específica de domínio para consulta em código orientado a aspectos**. 2013. Dissertação — Universidade Federal de Santa Maria-RS. Brasil.
- FOWLER, M.; BECK, K. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 1999.
- FRANKLIN, L. et al. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.1287–1290.
- GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education India, 1995.
- GOETZ, B. Language designer's notebook: quantitative language design. **<http://www.ibm.com/developerworks/java/library/j-ldn1/>**, [S.l.], 2017.
- GRISWOLD, W. Program Restructuring as an Aid to Software Maintenance. **Ph. D. Thesis, University of Washington, Dept. of Computer Science & Engineering, Seattle, WA**, [S.l.], 1991.
- GYORI, A. et al. Crossing the gap from imperative to functional programming through refactoring. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.543–553.
- KERIEVSKY, J. **Refatoração para padrões**. [S.l.]: Bookman Editora, 2009.

- MACIA BERTRAN, I.; GARCIA, A.; STAA, A. von. An exploratory study of code smells in evolving aspect-oriented systems. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT. **Proceedings...** [S.l.: s.n.], 2011. p.203–214.
- MARINESCU, R.; RATIU, D. Quantifying the quality of object-oriented design: the factor-strategy model. In: REVERSE ENGINEERING, 2004. PROCEEDINGS. 11TH WORKING CONFERENCE ON. **Anais...** [S.l.: s.n.], 2004. p.192–201.
- MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Transactions on software engineering**, [S.l.], v.30, n.2, p.126–139, 2004.
- MONTEIRO, M. P.; FERNANDES, J. M. Towards a catalogue of refactorings and code smells for AspectJ. **Lecture notes in computer science**, [S.l.], v.3880, p.214, 2006.
- MUNRO, M. J. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In: SOFTWARE METRICS, 2005. 11TH IEEE INTERNATIONAL SYMPOSIUM. **Anais...** [S.l.: s.n.], 2005. p.15–15.
- OPDYKE, W. Refactoring object-oriented frameworks. **PhD thesis, University of Illinois at Urbana-Champaign**, [S.l.], 1992.
- ORACLE. **Java™ Platform, Standard Edition 7 API Specification**. Acessado em 23/10/2017, <https://docs.oracle.com/javase/7/docs/>.
- ORACLE. **Java Platform Standard Edition 8 Documentation**. Acessado em 23/10/2017, <https://docs.oracle.com/javase/8/docs/>.
- PALSBERG, J.; JAY, C. B. The essence of the visitor pattern. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 1998. COMPSAC'98. PROCEEDINGS. THE TWENTY-SECOND ANNUAL INTERNATIONAL. **Anais...** [S.l.: s.n.], 1998. p.9–15.
- PIVETA, E. K. **Improving the search for refactoring opportunities on object-oriented and aspect-oriented software**. 2009. PHD thesis — UFRGS.
- PIVETA, E. K. et al. **Detecting Bad Smells in AspectJ**. 2006. 811–827p. v.12, n.7.
- RAHAD, K.; CAO, Z.; CHEON, Y. A Thought on Refactoring Java Loops Using Java 8 Streams. **Departmental Technical Reports (CS). 1153.**, [S.l.], 2017.

- RISSETTI, G. et al. **Catálogo de refatorações para a evolução de programas em linguagem fortran**. 2011. Dissertação — Universidade Federal de Santa Maria-RS. Brasil.
- ROBERTS, D. B. **Practical Analysis for Refactoring**. 1999. Ph. D. Thesis — University of Illinois at Urbana-Champaign.
- ROBERTS, D.; BRANT, J.; JOHNSON, R. A refactoring tool for Smalltalk. **Theory and Practice of Object systems**, [S.l.], v.3, n.4, p.253–263, 1997.
- SILVA, C. A. **Iniciando o desenvolvimento com a Streams**. Acessado em 27/11/2017, <https://www.infoq.com/br/articles/java8-iniciando-desenvolvimento-com-a-streams-api>.
- SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics based refactoring. In: SOFTWARE MAINTENANCE AND REENGINEERING, 2001. FIFTH EUROPEAN CONFERENCE ON. **Anais...** [S.l.: s.n.], 2001. p.30–38.
- SPRINGGAY, D. **Creating an Eclipse View**. Acessado em 20/07/2018, <https://www.eclipse.org/articles/viewArticle/ViewArticle2.html>.
- TEIXEIRA JÚNIOR, J. E. et al. **Um catálogo de refatorações envolvendo expressões lambda em Java**. 2014. Dissertação — Universidade Federal de Santa Maria-RS. Brasil.
- URMA, R.-G.; FUSCO, M.; MYCROFT, A. **Java 8 in Action: lambdas, streams, and functional-style programming**. [S.l.]: Manning Publications Co., 2014.
- WILKINSON, R.; HINGSTON, P. Using the cosine measure in a neural network for document retrieval. In: ANNUAL INTERNATIONAL ACM SIGIR CONFERENCE ON RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL, 14. **Proceedings...** [S.l.: s.n.], 1991. p.202–210.
- WINKLER, W. E. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. **ERIC**, [S.l.], 1990.

ANEXOS

ANEXO A – Catálogo de Refatorações inversas

Neste anexo serão apresentadas 15 refatorações inversas às propostas para a evolução de código do catálogo presente no Capítulo 3. Grande parte das refatorações catalogadas possuem uma refatoração inversa, no trabalho de FOWLER; BECK, 1999 é encontrado a refatoração *Extrair Método*, que transforma um fragmento de código em um método cujo nome explique o seu propósito. Essa refatoração possui a sua correspondente inversa *Internalizar Método*, que remove o método quando seu corpo é tão claro quanto ao nome. Há diversos motivos para se pensar em refatorações inversas, dentre eles pode ser citado: (i) certas estruturas novas serem incompatíveis com a versão utilizada pelos sistemas; (ii) os desenvolvedores não estarem habituados com a programação funcional e com as novas estruturas propostas; (iii) o uso das novas estruturas deixou a intenção do desenvolvedor menos clara; (iv) a refatoração causou algum erro inesperado pelo sistema. O desenvolvedor também pode usar a refatoração inversa para entender melhor o código e depois aplicar a refatoração que converte novamente. A seguir são apresentadas as refatorações inversas do catálogo.

- Desagrupar blocos `catch`
- Retirar a estrutura `try-with-resources`
- Retirar método `map`
- Remover `groupBy`
- Remover operação aritmética `lambda`
- Trocar o método `ifPresent` pela estrutura `if`
- Converter `removeIf` para `iterator.remove`
- Converter `replaceAll` para `iterator.set`
- Retirar método `join`
- Converter interface `Splitterator` para `Iterator`
- Retirar a interface `Predicate`
- Retirar a média `lambda`

- Retirar contagem lambda
- Retirar método `of`
- Retirar o *underscore* de parâmetros não utilizados em expressões lambda

Desagrupar blocos `catch`

Você possui as classes de exceções agrupadas em apenas um bloco `catch`, porém necessita fazer tratamentos diferentes para cada classe de exceção agrupada.

◇◇◇

Desagrupe as classes de exceções em blocos `catch` diferentes.

Utilizar a estrutura `multi-catch` pode ser vantajosa em vários aspectos, como para diminuir a duplicação de código, simplificar o tratamento das exceções e melhorar a legibilidade do código. Porém, por vezes, é necessário um tratamento diferenciado para cada classe de exceção agrupada, ou a refatoração de agrupamento resultou em alguns erros não esperados, dessa forma é possível utilizar a refatoração inversa para solucionar tais adversidades. A mecânica é descrita da seguinte maneira:

1. Criar um bloco `catch` para cada classe de exceção que a estrutura `multi-catch` possui.
2. Inserir cada classe de exceção em um bloco `catch` diferente junto com a variável da estrutura `multi-catch`.
3. Copiar o tratamento do bloco `multi-catch` para cada um dos blocos `catch` criados.
4. Remover o bloco `multi-catch`.
5. Compilar o código e testar

A Listagem 1 faz uso da estrutura `multi-catch`, agrupando todas as classes de exceção em um mesmo bloco `catch`. A Listagem 2 mostra a estrutura após aplicada esta refatoração inversa, onde passa a possuir um bloco `catch` para cada classe de exceção.

Listagem 1: Estrutura `multi-catch`.

```

1 var div = new Scanner(System.in);
2 try {
3     var divisor = div.nextInt();
4     System.out.print(2 / divisor);
5 } catch (ArithmeticException | IllegalArgumentException |
6     NullPointerException e) {
7     System.out.println("Erro: " + e.getMessage());
8 }

```

Listagem 2: Estrutura try-catch tradicional.

```

1 var div = new Scanner(System.in);
2 try {
3     var divisor = div.nextInt();
4     System.out.print(2 / divisor);
5 } catch (ArithmeticException e) {
6     System.out.println("Erro: " + e.getMessage());
7 } catch (IllegalArgumentException e) {
8     System.out.println("Erro: " + e.getMessage());
9 } catch (NullPointerException e) {
10    System.out.println("Erro: " + e.getMessage());
11 }

```

Retirar a estrutura try-with-resources

Você utiliza a funcionalidade try-with-resources, porém necessita tratar essa variável no bloco catch ou finally.

◇◇◇

Crie a variável que recebe o recurso fora do bloco, instancie ela dentro do bloco try e faça fechamento manual.

O uso de try-with-resources pode reduzir a complexidade do gerenciamento de recursos. Porém, o desenvolvedor pode optar por não fazer uso dessa funcionalidade por problemas de compatibilidade, ou por ser necessário tratar o recurso aberto nos bloco catch ou finally, já que o recurso só existe dentro do bloco try. Então, para esses casos pode ser necessário a aplicação desta refatoração inversa. A mecânica é descrita da seguinte forma:

1. Mover para antes da estrutura try a criação da variável que recebe o recurso e fazer receber o valor null
2. Instanciar a variável criada dentro do bloco try, movendo a instanciação do recurso para essa variável.

3. Criar uma estrutura condicional no bloco `finally` para verificar se a variável criada não é nula, se não for nula chamar o método responsável para fechar o recurso aberto.
4. Compilar o código e testar

A Listagem 3 mostra uma estrutura utilizando o recurso `try-with-resources` para fechamento automático de um arquivo. A Listagem 4 mostra a estrutura final após aplicar a refatoração, sendo removido a funcionalidade `try-with-resources`.

Listagem 3: Recurso fechado automaticamente.

```

1 try (var arquivo = new FileReader("texto.txt") ) {
2     //...
3 } catch (IOException e) {
4     System.err.printf("Erro: %s.\n", e.getMessage());
5 } finally {
6     //...
7 }
```

Listagem 4: Recurso fechado manualmente.

```

1 FileReader arquivo = null;
2 try {
3     arquivo = new FileReader("texto.txt");
4     //...
5 } catch (IOException e) {
6     System.err.printf("Erro: %s.\n", e.getMessage());
7 } finally {
8     if (arquivo != null)
9         arquivo.close();
10    //...
11 }
```

Retirar método `map`

Você utiliza o método `map` para aplicar métodos que modificam objetos pertencentes a uma coleção, porém o código ficou confuso.

◇◇◇

Você pode retirar o método `map`.

O método `map` aplica alguma modificação em cada elemento da coleção sem necessidade de variáveis intermediárias e laços de repetição. Porém, por questão de compatibilidade, ou pelo

fato do desenvolvedor não estar acostumado com alguns dos conceitos de Java 8, como as referências de método, pode querer retirar o método `map` para compreender mais facilmente o código. A mecânica é descrita da seguinte forma:

1. Copiar a criação da variável que recebe a coleção modificada e instanciá-la de acordo com a coleção recebida no método `collect` da estrutura.
2. Criar um laço de repetição para acessar cada elemento da coleção que será modificada.
3. Fazer com que a variável que acessa os elementos do laço receber ela mesma e, em sequência, o método da referência na estrutura `map`;
4. Se existirem mais métodos `map` volte para o item 3, senão vá para o próximo item.
5. Chamar o método `add` da coleção nova e adicionar a variável que contém o elemento modificado.
6. Apagar a estrutura anterior.
7. Compilar o código e testar

A Listagem 5 mostra uma estrutura que possui apenas um método `map` aplicando uma modificação nos elementos de uma coleção. A Listagem 6 apresenta o exemplo da estrutura refatorada, sem a utilização do método `map`, utilizando um laço de repetição para percorrer a lista e adicionar os elementos modificados na nova lista através do método `add`.

Listagem 5: Aplicar uma modificação em elementos de uma lista utilizando o método `map`.

```
1 var nomesMaiusculo = nomes.stream()
2   .map(String::toUpperCase)
3   .collect(Collectors.toList());
```

Listagem 6: Estrutura refatorada para modificar objetos de uma lista.

```
1 var nomesMaiusculo = new ArrayList<String>();
2 for (String s : nomes) {
3     s = s.toUpperCase()
4     nomesMaiusculo.add(s);
5 }
```

As próximas listagens apresentam a refatoração sendo aplicada em um caso onde há outros métodos `map` modificando a lista.

Listagem 7: Aplicando modificações em elementos de uma lista utilizando o método `map`.

```
1 var nomesMaiusculo = nomes.stream()
2   .map(String::toUpperCase)
3   .map(String::trim)
4   .collect(Collectors.toList());
```

Listagem 8: Estrutura refatorada para modificar elementos de uma lista sem utilizar o método `map`.

```
1 var nomesMaiusculo = new ArrayList<>();
2 for (var s : nomes) {
3     s = s.toUpperCase();
4     s = s.trim();
5     nomesMaiusculo.add(s);
6 }
```

Remover `groupingBy`

Você utiliza o método `groupingBy` para fazer agrupamentos em uma coleção, porém você não entendeu o funcionamento deste método.

◇◇◇

Então retire o método `groupingBy`.

Com método `groupingBy` pode-se agrupar objetos de acordo com alguma propriedade e guardar os resultados em uma instância do tipo `Map`. Porém, a estrutura pode apresentar alguns problemas, por exemplo, o método `groupingBy` retorna um número do tipo `Long` na contagem de elementos e o desenvolvedor pode precisar de um outro tipo numérico. Pode ocorrer também que o desenvolvedor não tenha o conhecimento básico sobre o funcionamento dessa nova estrutura, ou que haja uma incompatibilidade com versões anteriores a Java 8, então essa refatoração deve ser aplicada. A mecânica é descrita da seguinte forma:

1. Copiar a criação da variável do tipo `Map` e instanciá-la com a estrutura `HashMap`, ou equivalente.
2. Implementar um laço de repetição que percorra a lista da contagem de elementos.
3. Inserir no laço de repetição uma variável com mesmo tipo que a primeira posição da estrutura `Map`, e fazer ela receber a variável que percorre a lista chamando o método referência da primeira posição do método `groupingBy`.

4. Inserir no laço de repetição uma variável do mesmo tipo da segunda posição da estrutura Map, que receba a variável Map junto com o método `get` e, na passagem de argumentos, a variável criada anteriormente.
5. Verificar se a segunda variável criada no laço de repetição é `null`, se for chamar o método para adicionar um novo valor da variável Map e inicializá-la com 1. Se não chamar o método para adicionar/editar um novo valor da variável Map e adicionar a variável contadora incrementada.
6. Excluir a estrutura anterior.
7. Compilar o código e testar.

A Listagem 9 mostra a contagem de uma coleção de objetos do tipo pessoa de acordo o atributo `pessoaTipo` e armazena o resultado em uma variável do tipo Map utilizando as funcionalidades do método `groupingBy`. A Listagem 10 mostra o resultado após a aplicação da refatoração inversa, fazendo a contagem de uma lista de objetos de acordo com determinado atributo utilizando um laço de repetição e variáveis intermediárias, sendo possível alterar o tipo numérico da estrutura Map.

Listagem 9: Contagem de objetos utilizando o método `groupingBy`.

```
1 var tipo = pessoaList.stream().collect(
2     Collectors.groupingBy(Pessoa::getPessoaTipo,
3     Collectors.counting()));
```

Listagem 10: Forma convencional para fazer contagem de objetos.

```
1 var tipos = new HashMap<String, Long>();
2 for( var p : pessoaList ) {
3     var tipoPessoa = p.getPessoaTipo();
4     var pessoasPorTipo = tipos.get(tipoPessoa);
5     if (pessoasPorTipo == null) {
6         tipos.put(tipoPessoa, Long.getLong("1"));
7     } else {
8         tipos.put(tipoPessoa, ++pessoasPorTipo);
9     }
10 }
```

Remover operação aritmética lambda

Você realiza uma operação aritmética utilizando o método `reduce`, porém o código apresentou uma compreensão complexa.

◇◇◇

Retire esse método e faça a operação com um acumulador.

O método `reduce` realiza operações de reduções, onde combina todos os elementos de uma coleção em um único resultado. Porém, pode surgir a necessidade de eliminar o uso deste método, pois o desenvolvedor pode não possuir o conhecimento necessário para utilizar operações lambda e considerar a codificação mais complexa. Pode ocorrer também problemas de compatibilidade, já que esse método foi adicionado em Java 8. A mecânica é descrita a seguir:

1. Copiar a criação da variável que recebe o valor da operação lambda e fazer ela receber o primeiro argumento do método `reduce`.
2. Criar um laço de repetição para acessar todos os elementos da coleção que será feita a operação aritmética.
3. Fazer a variável acumular o resultado dos elementos de acordo com a operação feita no método `reduce`.
4. Se é uma coleção de objetos fazer a variável de interação chamar o método na referência na estrutura `mapToInt`.
5. Apagar as estruturas anteriores.
6. Compilar o código e testar.

A Listagem 11 expõe um exemplo da soma de uma coleção com valores numéricos utilizando o método `reduce`. A Listagem 12 apresenta a estrutura depois de refatorada sem a utilização do método `reduce`.

Listagem 11: Soma de valores de uma coleção com tipos primitivos e numéricos utilizando o método `reduce`.

```
1 var soma = numeros.stream().reduce(0, (a, b) -> a + b);
```

Listagem 12: Estrutura refatorada sem a utilização do método `reduce`.

```

1 var soma = 0;
2 for (var x : numeros)
3     soma = soma + x;

```

As listagens 13 e 14 mostram a aplicação da mesma refatoração em uma lista de objetos, onde a operação aritmética é realizada de acordo com algum atributo do objeto.

Listagem 13: Soma de valores de uma coleção de objetos utilizando o método `reduce`.

```

1 var soma = list.stream()
2     .mapToInt(Pessoa::getIdade)
3     .reduce(0, (a, b) -> a + b);

```

Listagem 14: Estrutura refatorada sem a utilização do método `reduce`.

```

1 var soma = 0;
2 for (var p : list)
3     soma = soma + p.getIdade();

```

Trocar o método `ifPresent` pela estrutura `if`

Você faz verificação de valores nulos usando o método `ifPresent` da classe `Optional`, porém o código apresentou uma difícil compreensão.

◇◇◇

Então volte a utilizar a estrutura condicional `if`.

`Optional` é uma classe, adicionada em Java 8, que encapsula vários métodos para tratar blocos de código crítico que necessitam ser verificados. Porém, a condição de verificação dessa nova interface é criada através de expressões lambda e utiliza um método para a verificação dessa expressão, o que pode tornar o código complexo. Deve-se ter também muito cuidado ao utilizar o método `ifPresent`, uma vez que o tipo da variável muda para o tipo `Optional`, o que modifica a forma de obter valores, podendo ocasionar erros em outras partes do código. A mecânica é descrita da seguinte forma:

1. Retirar na criação da variável a classe `Optional<T>`, deixando apenas o tipo do argumento (`T`).
2. Retirar o método `ofNullable` e a classe `Optional`.
3. Substituir o método `ifPresent` pela estrutura `if` e no argumento da estrutura verificar se a variável não é nula.

4. Mover o conteúdo presente no método `ifPresent` para dentro da estrutura `if`.
5. Aplicar a refatoração *Convert Lambda Expression to Functional Interface Instance* (TEIXEIRA JÚNIOR et al., 2014) para a instância da interface `Consumer` copiada do método `ifPresent`.
6. Compilar o código e testar.

A Listagem 15 utiliza a classe `Optional` para averiguar valores nulos, onde o método `ifPresent` só executará o código presente no seu argumento caso o valor não for nulo. A Listagem 16 mostra a estrutura resultante da utilização da estrutura condicional `if` para fazer a verificação se o valor não é nulo.

Listagem 15: Verificação de valores nulos utilizando a classe `Optional`.

```
1 var pessoa = Optional.ofNullable(getPessoa(nome));
2 pessoa.ifPresent(p -> System.out.println("Nome completo: "
3     + p.getName()));
```

Listagem 16: Estrutura refatorada para verificação de valores nulos.

```
1 var pessoa = getPessoa(nome);
2 if (pessoa != null)
3     System.out.println("Nome completo: " + pessoa.getName());
```

Converter `removeIf` para `iterator.remove`

Você remove elementos de uma coleção utilizando o método `removeIf`, porém o código ficou complexo e de difícil compreensão.

◇◇◇

Então volte para a interface `iterator`.

O método `removeIf` geralmente é mais conciso e eficiente quando comparando com a abordagem `iterator`. Porém, o desenvolvedor pode não ter conhecimento necessário sobre expressões lambda e o código acabar se tornando de difícil compreensão, ou pode acontecer alguma incompatibilidade com versões anteriores do Java, uma vez que esse método foi adicionado em Java 8. Então, nesses casos pode ser aplicada essa refatoração de inversão. A mecânica é descrita da seguinte forma:

1. Copiar a variável que utiliza o método `removeIf` e implementar um laço de repetição, utilizando a interface `Iterator` e os métodos próprios da interface para a interação com a coleção.
2. Criar uma variável dentro do laço de repetição com o mesmo nome do argumento da expressão lambda (antes do *token*) do método `removeIf` e do mesmo tipo dos elementos da coleção, fazer essa variável receber o elemento corrente.
3. Criar uma estrutura condicional `if` e copiar o corpo da expressão lambda (após o *token*) do método `removeIf` no teste condicional da estrutura.
4. Adicionar na estrutura `if` a variável de interação do laço chamando o método `remove` da interface `Iterator`.
5. Apagar as estruturas anteriores.
6. Compilar o código e testar.

A Listagem 17 mostra um código utilizando o método `removeIf` para remover o elemento de uma coleção. A Listagem 18 mostra o código após a aplicação da refatoração inversa, utilizando a abordagem `Iterator` para acessar os elementos da coleção.

Listagem 17: Remoção de elemento de uma lista utilizando o método `removeIf`.

```
1 list.removeIf(person -> "Peter".equals(person.getName()));
```

Listagem 18: Forma convencional para remoção de elementos de uma lista.

```
1 for (var i = list.iterator(); i.hasNext(); ) {
2     var person = i.next();
3     if ("Peter".equals(person.getName()))
4         i.remove();
5 }
```

Converter `replaceAll` para `iterator.set`

Você utiliza o método `replaceAll` para aplicar alguma ação nos elementos da coleção, porém é necessário fazer outras verificações.

◇◇◇

Então volte para a implementação com `iterator.set`

O método `replaceAll` realiza uma ação em todos os elementos de uma coleção, não sendo possível aplicar a ação em elementos específicos através de alguma condição. Ainda, o desenvolvedor pode encontrar dificuldades ao utilizar esse método, uma vez que as expressões lambda foram adicionadas recentemente. A questão da compatibilidade entre versões do Java também pode ser um fator importante para o uso dessa refatoração, visto que o método `replaceAll` foi adicionado recentemente. A mecânica é descrita da seguinte forma:

1. Copiar a variável que utiliza o método `replaceAll` e implementar um laço de repetição, interagindo com os elementos através da interface `Iterator`.
2. Chamar o método `set` da variável de interação do laço e passar no argumento o elemento corrente da interação junto com o método de alteração utilizado no corpo da expressão lambda na estrutura `replaceAll`.
3. Apagar as estruturas anteriores.
4. Compilar o código e testar.

Na Listagem 19 é possível observar a estrutura `replaceAll` tornando todos os elementos da lista maiúsculos. Já na Listagem 20 é possível observar a estrutura refatorada onde faz uso de um laço de repetição e a interface `Iterator` para modificar os elementos da coleção.

Listagem 19: Transformar elementos de uma lista para letras maiúsculas utilizando o método `replaceAll`.

```
1 list.replaceAll(s -> s.toUpperCase());
```

Listagem 20: Estrutura refatorada que não faz o uso do método `replaceAll`.

```
1 for (var i = list.listIterator(); i.hasNext();)
2     i.set(i.next().toUpperCase());
```

Retirar método `join`

Você concatena elementos utilizando o método `join` da classe `String`, porém ocorreu algum erro por questão de compatibilidade.

◇◇◇

Então volte a utilizar um laço de repetição junto com um operador de concatenação.

O método `join` retorna uma única `String` com todos os elementos de uma coleção separados por algum delimitador. Porém esse método foi adicionado a partir do Java 8, então, por questões de compatibilidade, pode ser necessária a utilização dessa refatoração. A mecânica é descrita da seguinte forma:

1. Copiar a criação da variável e instanciar uma nova `String` que será utilizada na concatenação.
2. Implementar um laço de repetição que passará por cada elemento da coleção, a variável de interação do laço deve ser do mesmo tipo que os elementos da coleção.
3. Fazer a variável de concatenação receber o elemento corrente do laço através do operador de concatenação da linguagem.
4. Fazer a variável de concatenação receber o delimitador do primeiro argumento do método `join`.
5. Apagar as estruturas anteriores.
6. Compilar o código e testar.

Independente do tipo dos elementos que a coleção possuir (texto, numérico, ...) a refatoração irá funcionar da mesma maneira, pois quando utilizamos o operador de concatenação o compilador transforma automaticamente o elemento em `String`. Na Listagem 21 é apresentado um código que concatena uma lista de *strings* e a Listagem 22 mostra a estrutura refatorada.

Listagem 21: Concatenar uma lista de strings utilizando o método `join` da classe `String`.

```
1 var tex = String.join(" ", lista).concat(" ");
```

Listagem 22: Forma convencional de concatenar uma lista de *strings*.

```
1 var tex = new String();
2 for (var s : lista) {
3     tex += s;
4     tex += " ";
5 }
```

O segundo exemplo irá mostrar a utilização da refatoração na concatenação de uma lista de inteiros, que utiliza a mesma mecânica do exemplo anterior. A Listagem 23 apresenta um exemplo da concatenação de elementos que não são textuais, então é necessário converter esses elementos para `String`, o que é feito pelo método `map` (linha 2). A Listagem 24 mostra a estrutura refatorada utilizando um laço de repetição com uma variável do tipo `int`, que necessariamente deve ter o mesmo tipo dos elementos da coleção.

Listagem 23: Concatenar uma lista de inteiros utilizando o método `join` da classe `String`

```
1 var tex = String.join(" ", lista.stream()
2     .map(Object::toString)
3     .collect(Collectors.toList()));
```

Listagem 24: Forma convencional de concatenar uma lista de inteiros.

```
1 var tex = new String();
2 for (var s : lista) {
3     tex += s;
4     tex += " ";
5 }
```

Converter interface `spliterator` para `iterator`

Você utiliza a interface `spliterator` para percorrer uma coleção, porém o código ficou complexo e de difícil compreensão.

◇◇◇

Então volte a utilizar a interface `iterator`.

A interface `Spliterator` é um tipo especial de `Iterator` utilizado para percorrer e particionar coleções. Um desenvolvedor que não está familiarizado com as novas estruturas de Java 8 pode ter dificuldade de entender o código, uma vez que a forma de funcionamento da interface `Spliterator` difere um pouco da interface `Iterator`. Pode ocorrer também algum erro de compatibilidade de acordo com a versão Java utilizada pelo sistema. Sendo assim é possível utilizar essa refatoração. A mecânica é descrita da seguinte forma:

1. Trocar a interface `Spliterator` pela interface `Iterator` e a chamada do método `spliterator` pelo `iterator` da coleção, mantendo o mesmo formato.

2. Trocar o método `tryAdvance` da interface `Spliterator` pelo método `hasNext` da interface `Iterator` no laço de repetição.
3. Mover a interface `consumer` do método `tryAdvance` e colocar dentro da estrutura do laço de repetição utilizado.
4. Remover o `Token` e o(s) parâmetro(s) na expressão lambda e substituir o argumento utilizado no corpo da expressão lambda pela variável da interação junto com o método `next`.
5. Compilar o código e testar.

A Listagem 25 mostra uma interação em uma lista utilizando a interface `Spliterator` e o laço de repetição `while`. Já a Listagem 26 mostra a estrutura refatorada para a interface `Iterator`, utilizando os métodos padrões da interface com o mesmo laço de repetição da estrutura anterior.

Listagem 25: Interação com uma lista utilizando a interface `Spliterator` com laço de repetição `while`.

```
1 var is = nomeLista.spliterator();
2 while(is.tryAdvance((n) -> System.out.println(n)));
```

Listagem 26: Estrutura refatorada utilizando a interface `Iterator` com laço de repetição `while`.

```
1 var is = nomeLista.iterator();
2 while (is.hasNext())
3     System.out.println(is.next());
```

A refatoração irá obter o mesmo resultado independente do laço de repetição utilizado, no próximo exemplo a Listagem 27 mostra a refatoração sendo aplicada em um código que utiliza o laço de repetição `for`. A Listagem 28 mostra como ficará essa estrutura após refatorada.

Listagem 27: Interação com uma lista utilizando a interface `Spliterator` e laço de repetição `for`.

```
1 for (var i = nomeLista.spliterator();
2     i.tryAdvance(e -> System.out.println(e)));
```

Listagem 28: Interação com uma lista utilizando a interface `Iterator` e laço de repetição `for`.

```
1 for (var i = nomeLista.iterator(); i.hasNext();)
2     System.out.println(i.next());
```

Retirar a interface `Predicate`

Você utiliza a interface `Predicate` para realizar testes no decorrer de um método, porém o código ficou de difícil compreensão.

◇◇◇

Retire a interface `Predicate` e faça os testes diretamente nas estruturas `ifs`.

`Predicate` é uma interface funcional que pode ser utilizada para avaliar uma determinada condição, retornando um valor booleano, de acordo com o teste realizado. Essa nova interface foi adicionada recentemente e para implementá-la é necessária a utilização de expressões lambda, assim como as anteriores, pode ser incompreensível para alguns desenvolvedores não habituados com o assunto, também pode ocorrer erros de compatibilidade de acordo com a versão Java utilizado pelo sistema. A mecânica é descrita da seguinte forma:

1. Identificar no corpo de um método estruturas condicionais que fazem a utilização da variável `Predicate<T>` criada nos testes lógicos.
2. Copiar para o argumentos das estruturas condicionais identificadas o corpo da expressão lambda recebida pela variável `Predicate<T>`.
3. Substituir as variáveis no corpo da expressão lambda copiada pela sua correspondente no argumento do método `test`.
4. Apagar todas as variáveis `Predicate<T>` junto com o método `test` das estruturas condicionais identificadas.
5. Apagar a criação da variável `Predicate<T>` e testar, se der erro, por estar sendo utilizada em outra estrutura de alguma outra forma, desfazer a exclusão e manter a estrutura.
6. Compilar o código e testar.

A Listagem 29 mostra um exemplo da utilização da interface `Predicate` para realizar os testes condicionais nas estruturas *ifs*. Já a Listagem 30 mostra um exemplo refatorado, sendo retirada essa nova interface do código.

Listagem 29: Verificar condições em estruturas *ifs* utilizando a interface `Predicate`.

```

1 var testeIdade = e -> e > 18 && e < 36;
2 if (testeIdade.test(idade)) {
3     //...
4 }
5 //...
6 if (testeIdade.test(idade)) {
7     //...
8 }
9 //...
```

Listagem 30: Estrutura refatora fazendo a verificação diretamente nas estruturas *ifs*.

```

1 if (idade > 18 && idade < 36) {
2     //...
3 }
4 //...
5 if (idade > 18 && idade < 36) {
6     //...
7 }
8 //...
```

Retirar a média lambda

Você realiza uma média utilizando expressões lambdas, porém o código se tornou complexo.

◇ ◇ ◇

Retire a expressão lambda e faça a média utilizando um acumulador.

Fazendo uso dos métodos adicionado em Java 8 e combinando-os com expressões lambdas é possível obter algumas vantagens como suporte para operações paralelas, diminuição do número de variáveis e estruturas intermediárias na realização de operações aritméticas e lógicas, entre outros benefícios. Porém, problemas podem ocorrer relacionados com a utilização dessas estruturas, a citar difícil compreensão do código, incompatibilidade com versões anteriores da linguagem, entre outros, sendo necessária a aplicação dessa refatoração. A mecânica é descrita da seguinte forma:

1. Criar uma variável acumuladora para receber a soma dos valores dos elementos da coleção, inicializar essa variável com o valor zero.
2. Copiar a coleção onde se realiza a operação e implementar uma laço de repetição para acessar os elementos da coleção.
3. Fazer a variável criada acumular os valores acessados, se for uma lista de objetos copiar o método para acessar os elementos do método `mapToDouble`.
4. Criar uma variável do tipo real para receber a variável acumuladora dividido pelo número total de elementos da coleção.
5. Apagar as estruturas anteriores.
6. Compilar o código e testar.

O exemplo da Listagem 31 e 32 apresenta a refatoração sendo aplicada em uma coleção com elementos numéricos e primitivos.

Listagem 31: Realizar a média utilizando operação lambda em uma coleção com tipos numéricos e primitivos.

```
1 var media = listaInt.stream()
2   .mapToDouble(i -> i)
3   .average()
4   .getAsDouble();
```

Listagem 32: Estrutura refatorada realizando a média em uma coleção com tipos numéricos e primitivos.

```
1 var soma = 0;
2 for (var i : listaInt)
3   soma = soma + i;
4 var media = soma / listaInt.size();
```

O exemplo da Listagem 33 e 34 expõe um exemplo de utilização desta refatoração em uma coleção de objetos.

Listagem 33: Realizar a média utilizando operação lambda e métodos funcionais em uma coleção de objetos.

```

1 var media = list.stream()
2   .mapToDouble(p -> p.getIdade())
3   .average()
4   .getAsDouble();

```

Listagem 34: Estrutura refatorada realizando a média em uma coleção de objetos.

```

1 var acu = 0;
2 for (var p : list)
3   acu = acu + p.getIdade();
4 double media = acu / list.size();

```

Retirar contagem lambda

Você realiza a contagem de elementos em coleção com os métodos funcionais e expressões lambda, porém o código ficou complexo e de difícil compreensão.

◇◇◇

Então volte para contagem tradicional utilizando acumulador e laço de repetição.

A utilização dos novos métodos advindos do paradigma funcional em Java 8 dão suporte para operações paralelas e utilização expressões lambda, o que pode melhorar o desempenho final. Porém, problemas relacionados a complexidade do código, compatibilidade com versões anteriores da linguagem podem ocorrer. Então, nesses casos pode ser aplicada essa refatoração. A mecânica é descrita da seguinte forma:

1. Criar uma variável acumuladora e fazer ela receber o valor zero.
2. Copiar a coleção que os valores serão acumulados e implementar um laço de repetição para acessar os objetos da coleção.
3. Se a estrutura possuir uma método `filter`, então implementar a estrutura condicional `if` passando a variável de interação concatenada com a mesma condição do corpo da expressão lambda do método. Se não passar para o próximo item.
4. Fazer a variável criada acumular o valor presente no corpo da expressão lambda da estrutura `mapToInt`.
5. Apagar a estrutura anterior.
6. Compilar o código e testar.

Na Listagem 35 pode ser visto uma estrutura utilizando métodos funcionais e expressões lambda para fazer contagem de elementos de uma coleção. A Listagem 36 mostra a estrutura refatorada fazendo a contagem de elementos de uma coleção, utilizando um acumulador e um laço de repetição.

Listagem 35: Realizar a contagem de uma lista de objetos utilizando expressões lambdas.

```
1 var sum = list.stream()
2   .mapToInt(p -> 1)
3   .sum();
```

Listagem 36: Estrutura refatorada que realiza a contagem de uma lista de objeto sem utilização de expressões lambdas.

```
1 var acumula = 0;
2 for (var p : list)
3   acumula = acumula + 1;
```

As Listagens 37 e 38 apresentam a mesma refatoração, porém agora com a presença do método `filter` que adicionará uma estrutura condicional na estrutura final refatorada.

Listagem 37: Realizar a contagem de uma lista de objetos utilizando expressões lambdas.

```
1 var sum = list.stream()
2   .filter(p -> p.getGenero().equals("Feminino"))
3   .mapToInt(p -> 1).sum();
```

Listagem 38: Estrutura refatorada que realiza a contagem de uma lista de objeto sem utilização de expressões lambdas.

```
1 var acumula = 0;
2 for (var p : list)
3   if (p.getGenero().equals("Feminino"))
4     acumula = acumula + 1;
```

Retirar método `of`

Você cria uma coleção imutável utilizando o método `of`, porém por algum motivo ocorreu um erro.

◇◇◇

Então volte para a construção convencional.

O método `of`, previsto em Java 9, permite criação de coleções imutáveis de forma rápida e simples. Porém, essa estrutura ainda não foi implementada e podem acontecer alguns erros não previsto na implementação dessa estrutura, então deve-se utilizar esta refatoração para voltar ao método tradicional para criar coleções imutáveis. A mecânica é descrita da seguinte forma:

1. Se os elementos foram adicionados manualmente:

Copiar a criação da variável e instanciar de acordo com o tipo da coleção.

Chamar um método `add` da variável criada para cada elemento do método `of`.

Quando todos os elementos da lista forem adicionados a variável da coleção deve receber o método `unmodifiableList` da interface `Collections` passando a própria lista nos argumentos do método.

Apagar as estruturas anteriores.

2. Se os elementos foram adicionados através de uma estrutura de repetição:

Substituir a chamada da interface da coleção junto com o método `of` pela a estrutura `Collections.unmodifiableList`.

3. Compilar o código e testar.

A Listagem 37 mostra um exemplo da criação de uma lista não modificável em Java utilizando o método `of`. A Listagem 38 apresenta essas mesmas estruturas após refatorada.

Listagem 39: Criação de listas não modificáveis utilizando o método `of`.

```
1 var list = List.of("a", "b", "c");
```

Listagem 40: Estrutura refatorada para criação de listas não modificáveis.

```
1 var list = new ArrayList<String>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 list = Collections.unmodifiableList(list);
```

As Listagens 41 e 42 mostram como seria a aplicação da refatoração caso os elementos fossem adicionados através de um laço de repetição.

Listagem 41: Criação de listas não modificáveis utilizando o método `of`.

```

1 var list = new ArrayList<String>();
2 for (var i = 0; i < 3; i++)
3     list.add(i);
4 list = List.of(list);

```

Listagem 42: Forma convencional para criação de listas não modificáveis.

```

1 var list = new ArrayList<String>();
2 for (int i = 0; i < 3; i++)
3     list.add(i);
4 list = Collections.unmodifiableList(list);

```

Retirar o *underscore* de parâmetros não utilizados em expressões lambda

Você utiliza o carácter underscore em parâmetros não são utilizados no corpo de uma expressão Lambda, porém ficou difícil de entender a expressão.

◇ ◇ ◇

Insira nomes nos argumentos das expressões lambdas para entender melhor a operação realizada.

Em Java 10 poderá ser usado o caractere sublinhado (*underscore*) para os parâmetros não utilizado em expressões lambda, o que pode tornar o código mais limpo e enxuto. Porém, por vezes, ao atualizar um código pode ser necessário conhecer os argumentos recebidos, para adicionar uma nova modificação em alguma variável até então não utilizada no corpo da expressão lambda. Pode ocorrer também de ficar difícil de entender a expressão lambda sem conhecer os outros parâmetros, nesse sentido essa refatoração pode ser vantajosa. A mecânica é descrita da seguinte forma:

1. Identificar os parâmetros que estão com caractere sublinhado (*underscore*).
2. Encontrar a estrutura que a expressão lambda modifica, ou um dos métodos que faz uso dessa expressão e trocar o caractere sublinhado (*underscore*) por um nome que identifique o campo não utilizado.
3. Compilar o código e testar.

A Listagem 39 mostra um exemplo de um código com uma estrutura `Map<String, Double>`, onde possui nome de pessoas com seus respectivos salários. Porém o primeiro

argumento era desconhecido, poderia ser um nome, um CPF, uma matrícula, ou qualquer outro atributo que possa identificar alguém que recebe um salário. Fazendo a análise do código constatou-se que essa variável representava nome de pessoas. Então a Listagem 40 mostra o código refatorado, onde o carácter sublinhado foi trocado por um nome que identifique o parâmetro.

Listagem 43: Forma de declarar parâmetros não utilizados no corpo de expressões lambdas em Java 10.

```
1 var salarios = new HashMap<String, Long>();
2 salarios.put("Pedro", 400.00);
3 salarios.put("Joquim", 300.00);
4 salarios.put("Manuel", 500.00);
5 ...
6 salarios.replaceAll( (_, salario) -> salario + 10000 );
```

Listagem 44: Forma convencional de declarar parâmetros não utilizados em uma expressão lambda.

```
1 var salarios = new HashMap<String, Double>();
2 salarios.put("Pedro", 400.00);
3 salarios.put("Joquim", 300.00);
4 salarios.put("Manuel", 500.00);
5 ...
6 salarios.replaceAll( (nome, salario) -> salario * 1.1 );
```