UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UM CATÁLOGO DE REFATORAÇÕES ENVOLVENDO EXPRESSÕES LAMBDA EM JAVA

DISSERTAÇÃO DE MESTRADO

Jânio Elias Teixeira Júnior

Santa Maria, RS, Brasil

2014

UM CATÁLOGO DE REFATORAÇÕES ENVOLVENDO EXPRESSÕES LAMBDA EM JAVA

Jânio Elias Teixeira Júnior

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Informática, Área de Concentração em Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Mestre em Ciência da Computação

Orientador: Profª. Dr. Eduardo Kessler Piveta

Santa Maria, RS, Brasil

2014

Teixeira Júnior, Jânio Elias

Um Catálogo de Refatorações Envolvendo Expressões Lambda em Java / por Jânio Elias Teixeira Júnior. – 2014.
92 f.: il.; 30 cm.

Orientador: Eduardo Kessler Piveta
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2014.

1. Refatoração. 2. Expressões Lambda. 3. Closures. I. Piveta, Eduardo Kessler. II. Título.

E-mail: janiojunior@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

UM CATÁLOGO DE REFATORAÇÕES ENVOLVENDO EXPRESSÕES LAMBDA EM JAVA

elaborada por
Jânio Elias Teixeira Júnior

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Eduardo Kessler Piveta, Dr.
(Presidente/Orientador)

Juliana Kaizer Vizzotto, Dr.ª (UFSM)

André Rauber Du Bois, Dr. (UFPEL)

Santa Maria, 28 de Agosto de 2014.

Aos meus pais Jânio e Tina.

AGRADECIMENTOS

Quero agradecer primeiramente ao meu orientador, professor Eduardo Kessler Piveta, pela grande oportunidade e confiança a mim concedidas e aos inúmeros conselhos e ensinamentos que foram fundamentais para o desenvolvimento deste trabalho.

Aos professores, amigos e colegas da UFSM, que me incentivaram e apoiaram de alguma forma na construção deste trabalho. Em especial às professoras Deise e Juliana.

Ao meu amigo Cristiano e sua família, pelo acolhimento, conselhos, sugestões e aprendizado. Vocês se tornaram minha família em Santa Maria.

Ao meu amigo Heres, pela grande amizade e parceria de todos esses anos.

À minha querida Muriel, com seu apoio, carinho e amor, tudo ficou mais fácil.

Por fim, quero agradecer aos meus pais e irmãos pela força, ensinamentos, conselhos e apoio em tudo que idealizo realizar. O amor e a dedicação de vocês serão sempre fundamentais.

“Descobrir consiste em olhar para o que todo mundo está vendo e pensar uma coisa diferente”

— ROGER VON OECH

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

UM CATÁLOGO DE REFATORAÇÕES ENVOLVENDO EXPRESSÕES LAMBDA EM JAVA

AUTOR: JÂNIO ELIAS TEIXEIRA JÚNIOR
ORIENTADOR: EDUARDO KESSLER PIVETA
Local da Defesa e Data: Santa Maria, 28 de Agosto de 2014.

A evolução de uma linguagem de programação fornece espaços para melhorias de programas existentes. Dessa forma, desenvolvedores podem atualizar projetos de sistemas de software, aplicando os novos recursos disponíveis na linguagem. No entanto, ao adaptar, melhorar e modificar um sistema de software, seu código pode se afastar de sua concepção original. Nesse contexto, o uso de técnicas e processos de transformação pode ser interessante, pois reduz a possibilidade de erros ao realizar uma melhoria em uma estrutura de código, por exemplo. A refatoração é um processo de melhoria do projeto de um sistema de software, que altera sua estrutura interna, sem modificar seu comportamento externo observável. A partir desse cenário, este trabalho tem como principal objetivo apresentar um catálogo de refatorações direcionadas às novas funcionalidades da linguagem Java. Tais refatorações estão relacionadas às expressões lambda e visam permitir a transformação de construções implementadas em Java 7 para sua atual versão 8. Para avaliar a aplicabilidade das refatorações propostas, um conjunto de projetos de código aberto foi submetido a uma ferramenta de análise estática, desenvolvida para realizar buscas por oportunidades de refatoração.

Palavras-chave: Refatoração. Expressões Lambda. Closures.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

A CATALOG OF REFACTORINGS INVOLVING LAMBDA EXPRESSIONS IN JAVA

AUTHOR: JÂNIO ELIAS TEIXEIRA JÚNIOR
ADVISOR: EDUARDO KESSLER PIVETA

Defense Place and Date: Santa Maria, August 28st, 2014.

Programming language evolution provides room for improving existing programs. Developers can upgrade their projects, applying new features available in the latest language versions. However, during maintenance activities, the code artefacts can become distant from their original conception. In this context, the use of transformation techniques and processes can be interesting, as it reduces the error-proneness when improving source code structure. Refactoring is a process of improving the design of a software system, modifying its internal structure without changing its external observable behavior. From this scenario, this work presents a refactoring catalog focused on the new features of the Java language. Such refactorings are related to lambda expressions and seek to allow the transformation of features implemented in Java 7 for the current version 8. To evaluate the proposed refactorings applicability, we developed a static analysis and used it in a set of open source projects aiming to search for opportunities to apply those refactorings.

Keywords: Refactoring. Lambda Expressions. Closures.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AJDT	*AspectJ Development Tools*
API	*Application Programming Interface*
AQL	*Aspect Query Language*
AST	*Abstract Syntax Tree*
DSL	*Domain Specific Language*
JDT	*Java Development Tools*
OA	*Orientação a Aspectos*
OO	*Orientação a Objetos*
OR	*Oportunidade de Refatoração*
IDE	*Integrated Development Environment*
LOC	*Linhas de Código*
UC	*Unidade de Compilação*
UML	*Unified Modelling Language*

SUMÁRIO

1 INTRODUÇÃO

Com a evolução das linguagens de programação, novos padrões e novas funcionalidades surgem e, consequentemente, a atualização do projeto de sistemas de software existentes pode tornar-se necessária. Contudo, à medida que um sistema de software é melhorado, modificado e adaptado às novas funcionalidades, seu código pode se tornar mais complexo e se afastar de sua concepção original.

Essas mudanças originam uma necessidade de reestruturação de tais sistemas de software, podendo acarretar em uma diminuição na qualidade do sistema (MENS; TOURWE, 2004). Refatoração é um processo de melhoria do projeto de sistemas de software, que altera sua estrutura interna, sem modificar seu comportamento externo observável (OPDYKE, 1992; FOWLER et al., 1999). Dessa forma, modificações podem ser realizadas através de refatorações, muitas vezes disponíveis em catálogos de refatoração (FOWLER et al., 1999; VAN EMDEN; MOONEN, 2002; OLBRICH et al., 2009; SCHUMACHER et al., 2010).

A linguagem Java, em sua versão 8, adicionou novos recursos para auxiliar no desenvolvimento de software, incluindo o suporte às expressões lambda. Historicamente, algumas linguagens orientadas a objetos, tais como *Scala*, *Xtend*, *JavaScript*, e *Ruby*, dão suporte a expressões lambda desde sua primeira versão. Outras linguagens, como *C#* e *C++*, evoluíram para dar suporte a expressões lambda em versões posteriores. Portanto, com a evolução de uma linguagem, surge a possibilidade de atualizar funcionalidades de programas que utilizam tal linguagem, quando vantajosas, atualizando as construções antigas de forma a se beneficiar das novas construções, evoluindo o projeto de software juntamente com a linguagem.

Ao evoluir o código em resposta às novas construções, como as expressões lambda em Java 8, os desenvolvedores são confrontados com o desafio de reestruturar o código das aplicações, de forma a obter os benefícios de tais construções. Primeiro, há uma necessidade de identificar oportunidades de refatoração em partes do código em que um cenário de refatoração poderia ser aplicado. Em seguida, o desenvolvedor deve avaliar os efeitos da refatoração na qualidade. Além disso, a aplicação de um padrão de refatoração deve manter a consistência das partes do sistema de software que está sendo reestruturado. Finalmente, o comportamento de um sistema de software deve ser preservado ao refatorarmos. Essa sequência de atividades é descrita como um processo de refatoração (MENS; TOURWE, 2004; PIVETA, 2009), porém sua execução é propensa a erros quando realizada manualmente, sem mecanismos, ferramentas

e recursos adequados.

Este trabalho propõe um catálogo de refatorações voltadas para o uso de expressões lambda em programas Java e AspectJ, através das novas construções adicionadas à linguagem Java, disponíveis na especificação JSR-335 (ORACLE, 2014a). As principais contribuições são:

- a definição de um catálogo de 20 refatorações, apresentadas no formato padrão canônico, no qual a metade das refatorações representam conversões de estruturas de código disponíveis em programas da versão 7 da linguagem Java para sua representação usando expressões lambda (Java 8). As demais refatorações correspondem às refatorações inversas; e

- uma extensão do *framework* AOPJungle (FAVERI, 2013), que fornece métricas de software e informações a respeito de programas orientados a objetos (OO) e orientados a aspectos (OA), tendo como finalidade o acesso a meta-informações por meio de consultas a código fonte. Essa extensão é uma melhoria do *framework*, permitindo a inclusão de informações sobre expressões lambda em seu modelo de dados.

Para a avaliação das refatorações propostas, um estudo de caso foi desenvolvido utilizando uma ferramenta de análise estática (λRefactoring), construída no âmbito deste trabalho para a identificação de oportunidades de refatoração em códigos OO e OA, através das informações fornecidas pelo *framework* AOPJungle.
Esta dissertação está organizada da seguinte forma:

- Capítulo 2, *Revisão de literatura*. Apresenta a base conceitual que envolve o escopo desta dissertação. São descritos os seguintes assuntos: refatorações, expressões lambda, expressões lambda em Java, *framework* AOPJungle e trabalhos relacionados.

- Capítulo 3, *Um catálogo de refatorações envolvendo expressões lambda em java*. É apresentado um conjunto de refatorações, em que cada refatoração é descrita no formato de um padrão que inclui, além de outras informações, uma motivação em se utilizar a refatoração e um exemplo de seu uso. Nesse capítulo também estão contidas as refatorações inversas.

- Capítulo 4, *Implementação*. Expõe detalhes sobre a extensão do *framework* AOPJungle. São descritos os seguintes assuntos: definição e adequação do metamodelo, descrevendo

o que foi adicionado e modificado; e a adaptação do *framework* para o suporte a expressões lambda, mostrando o que foi adicionado em função do catálogo de refatorações.

- Capítulo 5, *Estudo de Caso*. Exibe um estudo de caso realizado com a ferramenta *λRefactoring*, criada neste trabalho para identificar oportunidades de refatorações a partir de informações geradas pelo *framework* AOPJungle.

- Capítulo 6, *Conclusão*. Apresenta as contribuições desta dissertação e indica algumas sugestões para trabalhos futuros.

2 REVISÃO DE LITERATURA

Este capítulo apresenta uma revisão de literatura voltada aos seguintes assuntos: refatoração, expressões lambda, expressões lambda em Java, *framework* AOPJungle e trabalhos relacionados.

2.1 Refatoração

Uma característica que se pode observar em um sistema de software é sua necessidade de evoluir. Ao adaptar, melhorar e modificar um sistema, seu projeto pode-se afastar da sua concepção original, diminuindo a qualidade do sistema de software (MENS; TOURWE, 2004). Essa degradação pode levar à redução da legibilidade, da reusabilidade e da manutenibilidade do sistema.

Refatoração (OPDYKE, 1992; FOWLER et al., 1999) é uma técnica que visa minimizar o impacto dessa degradação, através da reestruturação do comportamento interno de projetos de software, sem modificar seu comportamento externo observável. Sua aplicação pode ajudar a melhorar incrementalmente atributos de qualidade (ISO, 2001; BOEHM; IN, 1996) de um sistema, por meio de transformações que preservam o comportamento da aplicação, chamados padrões de refatoração (KATAOKA et al., 2001).

Padrões de refatoração são organizados em catálogos de refatorações, sendo que cada padrão é descrito por um nome, um contexto que poderia ser aplicado, um conjunto de passos para sua aplicação, e um ou mais exemplos, mostrando como a transformação poderia ser executada (FOWLER et al., 1999). Abaixo é apresentada a refatoração *Extract Method*:

Extract Method

Há código fragmentado que pode ser agrupado em um método reutilizável.

◇ ◇ ◇

Portanto, transforme tal fragmento em um novo método cujo nome expresse seu propósito.

Quando o código é muito longo ou necessita de comentário para que se entenda seu propósito, pode-se converter esse trecho em um novo método, com um nome que represente seu objetivo. A sua mecânica é descrita da seguinte forma:

1. Crie um novo método e dê um nome que expresse semanticamente seu objetivo.
2. Copie o trecho de código selecionado para seu novo método criado.
3. Procure no código extraído por quaisquer variáveis locais no escopo do método original.
4. Verifique se algumas das variáveis temporárias são usadas somente dentro do código extraído e declare-as dentro do novo método.
5. Verifique se existem variáveis que eram modificadas pelo código do método extraído. Se uma variável foi modificada, verificar se pode ser tratada como uma consulta e então atribuir o resultado a uma variável em questão. Se existir mais de uma variável nessa situação, antes da extração do método, a variável pode precisar ser tratada, provavelmente usando a refatoração *Split Temporary Variable*[1] e somente depois realizar a extração.
6. Passe as variáveis locais como parâmetros para o novo método.
7. Compile quando você tratou todas as variáveis de escopo local.
8. Substitua o trecho de código extraído pela chamada do novo método.
9. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois da aplicação da refatoração *Extract Method*. Antes da refatoração, duas chamadas ao método *println* eram feitas para imprimir as variáveis *_name* e *amount*. Após a refatoração, um novo método (*printDetails*) foi criado para representar tais chamadas de impressão.

```
1  private String _name;
2
3  // Antes
4  void printOwing(double amount) {
5     printBanner();
6     //imprimir detalhes
7     System.out.println("name:" + _name);
8     System.out.println("amount" + amount);
9  }
10 // Depois
11 void printOwing(double amount) {
12    printBanner();
13    printDetails(amount);
14 }
15 void printDetails(double amount) {
16    System.out.println("name:" + _name);
17    System.out.println("amount" + amount);
18 }
```

[1] Faça uma variável temporária para cada atribuição.

É importante considerar que existem processos de refatoração (MENS; TOURWE, 2004; PIVETA, 2009), os quais consideram, dentre outras, as seguintes atividades: (i) identificar onde aplicar uma refatoração, (ii) avaliar os efeitos da refatoração na qualidade de software, (iii) manter a consistência dos artefatos reestruturados pela refatoração, e (iv) garantir que a aplicação da refatoração preserve o comportamento externamente observável do sistema de software.

Identificar onde refatorar consiste em buscar por locais nos artefatos do sistema de software em que refatorações possam ser aplicadas, considerando deficiências, inadequações ou incompletudes identificadas nesses artefatos. Sendo assim, uma oportunidade de refatoração é definida como uma associação entre um artefato de software, uma limitação (deficiência, inadequação ou incompletude) e um padrão de refatoração (PIVETA, 2009).

A avaliação dos efeitos da refatoração na qualidade de software geralmente é realizada por avaliações quantitativas, aplicando funções de impacto. Funções de impacto são funções matemáticas que calculam o valor esperado de uma métrica ao se aplicar uma refatoração, sem de fato aplicá-la (DU BOIS; MENS, 2003; MENS; TAENTZER; RUNGE, 2005; DU BOIS, 2006). Du Bois (2006) e Piveta (2009) apresentam abordagens para a criação de funções de impacto em código orientado a objetos e código orientado a aspectos, respectivamente.

Outra etapa do processo de refatoração é manter a consistência dos artefatos reestruturados pela refatoração. Por exemplo, se o método de uma classe tem o nome alterado, existe a necessidade de alterar todas a chamadas a esse método. Isso implica em atualizar todos os seus relacionamentos.

A última etapa relacionada ao processo de refatoração, visa garantir que a aplicação da refatoração preserve o comportamento externo do software. Existem diversos formalismos para ajudar nesta atividade, bem como o uso de asserções (pré condições, pós condições e invariantes) e transformações de grafos. Cornélio (2004) proporciona uma discussão interessante sobre a preservação de comportamento externo de software.

2.2 Expressões Lambda

Uma expressão lambda é uma construção que permite a chamada anônima de uma função, isto é, sem a necessidade de um nome como identificador de tal função. Seu conceito é baseado no cálculo lambda (CHURCH, 1936), que é um sistema formal no qual toda computação é reduzida a operações básicas de definição e aplicação de funções (PIERCE, 2002). Por exemplo, $(int\ a, int\ b) \rightarrow a * b$ é uma expressão lambda com dois argumentos do tipo inteiro e

que retorna o resultado da multiplicação de tais argumentos.

Uma das motivações para o uso das expressões lambda tem sido a facilidade em se codificar instruções com execução paralela através de API's específicas. Como a indústria de hardware moveu para o uso de processamento multi-core, a indústria de software se inclina para esconder a complexidade de escrever código paralelo, através de bibliotecas paralelas. Por exemplo, as bibliotecas C#TPL[2] e PLINQ[3] contam com expressões lambda para encapsular funções que são passadas para APIs de biblioteca e logo serem executadas em paralelo.

Várias linguagens de programação baseadas em Java, tais como Scala (EPFL, 2013) e Xtend (ECLIPSE, 2013), utilizam expressões lambda em seus projetos. Recentemente, membros do *Java Community Process* terminaram a especificação JSR-335 (ORACLE, 2014a), que incluiu o suporte para expressões lambda em Java a partir da versão 8. Em Java, as expressões lambda são conhecidas também como *closures* ou *métodos anônimos* (ORACLE, 2014a).

2.3 Expressões Lambda em Java

Em Java, as expressões lambda aplicam-se exclusivamente às interfaces funcionais que, por definição, são as interfaces que contém apenas um método abstrato, como por exemplo, a interface *Runnable* (Listagem 2.1).

Listagem 2.1 – Interface Funcional *Runnable*

```
1 public interface Runnable {
2    public void run();
3 }
```

De um modo geral, a sintaxe padrão de uma expressão lambda segue o seguinte formato: *ParameterSpace* −> *BodySpace*. O *ParameterSpace* é o espaço reservado para os parâmetros definidos no método da interface funcional. O símbolo −> é um token que, além de identificar uma expressão, é utilizado para denotar a aplicação do parâmetro à implementação que vem em seguida. Por fim, o *BodySpace* é o espaço reservado para a implementação do método da interface funcional. A Listagem 2.2 apresenta um trecho de código da implementação de uma interface funcional *Runnable* na forma convencional e, em seguida, a sua implementação utilizando expressões lambda.

[2] http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx.
[3] http://msdn.microsoft.com/en-us/library/dd460688(v=vs.110).aspx.

Listagem 2.2 – Convencional vs. Expressão Lambda

```
1 //Forma Convencional
2 Runnable r = new Runnable() {
3    public void run() {
4       System.out.println("hello!");
5    }
6 };
7
8 // Lambda
9 Runnable r = () -> System.out.println("hello!");
```

Como pode ser visto na Listagem 2.2, com a utilização de expressões lambda, é perceptível a redução do código e também a sua facilidade de leitura. A forma convencional é mais verbosa e utiliza uma maior quantidade de chaves e parenteses, além de apresentar uma maior quantidade de linhas de código (Problema Vertical) (ORACLE, 2013).

2.3.1 Interfaces Funcionais

Em Java, uma interface funcional é uma interface que possui apenas um método abstrato, representando um único contrato de função. Em alguns casos, esse único método pode assumir a forma de vários métodos abstratos com assinaturas *override* equivalentes, herdadas das *super* interfaces. Neste caso, os métodos herdados logicamente representam um único método (ORACLE, 2014a).

Uma expressão lambda pode ser vista como uma instância de uma interface funcional, com uma sintaxe mais simples e compacta. Geralmente uma expressão lambda é solicitada em parâmetros de métodos, portanto sua sintaxe compacta se torna uma vantagem. Utilizando a interface *ActionListener*, a Listagem 2.3 apresenta um trecho de código com uma implementação convencional e logo após com expressões lambda.

Listagem 2.3 – Expressão Lambda de uma ActionListener

```
1 //Forma Convencional
2 buttonClear.addActionListener(new ActionListener() {
3    public void actionPerformed(ActionEvent e) {
4       clear();
5    }
6 });
7
8 // Lambda
9 buttonClear.addActionListener(e -> clear());
```

Como observado na Listagem 2.3, a sintaxe lambda é mais simples que a implementação convencional, reduzindo estrutura e a quantidade de linhas. A próxima subseção detalha as

diferentes formas de se implementar uma expressão lambda.

2.3.2 A Sintaxe de uma Expressão Lambda

Como visto na seção anterior, uma expressão lambda é aplicada de forma anônima, sem referenciar a qual interface e a qual método sua aplicação está sendo feita. Contudo, sua sintaxe pode ser aplicada com algumas variações. Para exemplificar as diferentes formas de sintaxe, serão utilizadas as interfaces funcionais *Comparator* e *ActionListener*. A Listagem 2.4 apresenta instâncias de classes anônimas que implementam essas interfaces.

Listagem 2.4 – Implementação convencional das Interfaces *Comparator* e *ActionListener*

```
1  new Comparator<String>(){
2      public int compare(String o1, String o2) {
3          return o1.compareTo(o2);
4      }
5  };
6
7  new ActionListener(){
8      public void actionPerformed(ActionEvent e) {
9          System.out.println("Closures...");
10     }
11 };
```

Logo abaixo, são apresentadas as variações de sintaxe de uma expressão lambda, são elas:

- **Sintaxe padrão**. Geralmente uma expressão lambda possui a seguinte estrutura: $(\mathcal{P}) \rightarrow \{\mathcal{C}\}$, onde $(\mathcal{P})$ se refere aos parâmetros da assinatura do método de uma interface, $\rightarrow$ é o token que identifica o uso da expressão lambda e $\{\mathcal{C}\}$ representa o código do método. Convertendo o código apresentado na Listagem 2.4, as instâncias das interfaces utilizando expressões lambda seriam:

```
1  // Comparator
2  (String o1, String o2) -> {return o1.compareTo(o2);};
3
4  // ActionListener
5  (ActionEvent e) -> {System.out.println("Closures...");};
```

- **Inferência de tipos**. A linguagem Java permite que os parâmetros utilizados em uma expressão não precisem da especificação dos tipos dos objetos, deixando apenas a variável que representa uma classe ou uma interface. Inferindo os tipos o código ficaria:

```
1  // Comparator
2  (o1, o2) -> {return o1.compareTo(o2);};
3
4  // ActionListener
5  (e) -> {System.out.println("Closures...");};
```

- **Ocultação de parênteses**. Outra simplificação é a possibilidade de não utilizar parênteses na declaração dos parâmetros. Para este caso, o uso dos parênteses é obrigatório apenas quando houver mais de um objeto na assinatura do método. Segue o exemplo abaixo:

```
1 // Comparator
2 (o1, o2) -> {return o1.compareTo(o2);};
3
4 // ActionListener
5 e -> {System.out.println("Closures...");};
```

- **Omissão de chaves e *return***. O código da implementação do método de uma interface funcional pode ser simplificado se possuir apenas uma instrução. Entretanto, quando sua construção computar mais de uma instrução, o uso de chaves se torna obrigatório e a separação de cada instrução é realizada por ponto e vírgula. O modo simplificado abaixo, apresenta apenas a chamada de uma instrução em cada exemplo, portanto não é necessário o uso de chaves e ponto e vírgula:

```
1 // Comparator
2 (o1, o2) -> o1.compareTo(o2);
3
4 // ActionListener
5 e -> System.out.println("Closures...");
```

  Quando implementamos um método e este possui um tipo de retorno, o uso do comando *return* torna-se obrigatório. Porém, quando uma expressão lambda é utilizada em sua forma simplificada, o comando *return* deixa de ser obrigatório. No exemplo anterior, pode-se visualizar a omissão do comando *return* na implementação da Interface *Comparator*.

2.3.3 Coleções

Com a introdução das expressões lambda em Java, uma série de funcionalidades relacionadas foi adicionada e várias outras evoluíram. Dentre elas está a API de coleções. Sua evolução implicou em uma forma diferente de se usar coleções, definindo duas maneiras de iterações sobre uma coleção, a interna e a externa:

- **Iteração externa**. É a forma geralmente utilizada em uma coleção de dados. No exemplo abaixo, uma construção *for each* executa a chamada do método *iterator* da coleção *people*, resultando em um laço de repetição dos objetos da coleção. Dentro do seu escopo, pode-se acessar informações dos objetos utilizados e, especificamente, o exemplo imprime o primeiro nome de uma pessoa.

```
1 for (Person p: people)
2    System.out.println(p.getFirstName());
```

Uma iteração externa é bem simples, mas apresenta alguns problemas (ORACLE, 2014b):

  - é inerentemente serial, portanto seu processamento segue a ordem especificada pela coleção; e
  - impossibilita que métodos da coleção gerenciem o fluxo de controle, impedindo, por exemplo, a reordenação dos dados, paralelismo, *short-circuiting*, ou *laziness* para melhorar o desempenho.

Uma iteração sequencial e em ordem, que é o propósito de uma construção *for each*, pode ser utilizada em várias rotinas, sendo desejável o seu uso nesta forma. Porém, outros recursos também são desejáveis, o que se pode tornar uma complicação ao utilizar uma iteração externa.

- **Iteração interna**. Ao contrário da iteração externa, a interna é a responsável pela iteração de uma coleção. Sua utilização é feita através de operações definidas pela coleção. O exemplo abaixo apresenta um laço de repetição (*for each*) em uma coleção de pessoas (*people*).

```
1 people.forEach(p -> System.out.println(p.getFirstName()));
```

Uma iteração interna permite o acesso a métodos diretamente no objeto da coleção, sem a necessidade de uma estrutura de repetição para sua execução. No exemplo acima, a coleção *people* faz a chamada ao método *forEach*, o qual exige como argumento uma implementação da interface funcional *Consumer*.

2.3.4 Métodos *Default*

Uma interface descreve um conjunto de métodos que podem ser chamados por quaisquer objetos de classes que implementem tal interface, mas não permite implementações concretas em seus métodos. Até a versão 7 da linguagem Java, todos os métodos de uma interface deveriam ser declarados e implementados em todas as classes concretas que a implementam (DEITEL; DEITEL, 2011). No entanto, na versão 8, é possível implementar métodos concretos na própria interface, desde que, estes métodos sejam definidos como *default*. No caso das expressões lambda, em que uma interface funcional só poderia conter apenas um método em sua

estrutura, podem existir outros métodos que possam complementar seu objetivo. O exemplo abaixo apresenta uma interface com um método implementado. Um método *default* de uma interface não precisa ser declarado em classes que a implementam, porém podem ser sobrescritos caso necessário.

```
1 public interface Service<T> {
2   public void save(T t);
3   public void update(T t);
4   public void delete(T t);
5   public default Collection<T> find(T t) {
6      return new ArrayList<T>();
7   }
8 }
```

2.3.5 API *Stream*

API *Stream* disponibiliza um conjunto de classes (*java.util.stream*) para apoiar operações, no estilo funcional, sobre fluxo (*stream*) de elementos, como por exemplo, transformações *map-reduce*[4] em coleções. No exemplo abaixo, é feita uma operação *filter-map-reduce* sobre uma fonte de *stream* (*people.stream()*), obtendo a soma da idade de todas as pessoas do gênero masculino da coleção *people*. O somatório é um exemplo de uma operação de redução (*reduce*).

```
1    List<Person> people = ...
2    people.stream().filter(p -> p.getGender().equals(Gender.MALE))
3       .mapToInt(p -> p.getAge()).sum();
```

A API *Stream* permite realizar operações e obter informações sem alterar a estrutura ou dados originais. Por exemplo, ao realizar uma ordenação em uma coleção (linha 2 do exemplo abaixo), sem utilizar o recurso de *stream*, os dados da própria coleção são ordenados. Porém, ao se aplicar da ordenação utilizando *stream* (linha 5 do exemplo abaixo), apenas o resultado recebe a ordenação, mantendo as informações da coleção em sua forma original.

```
1 // sem stream
2 people.sort((p1, p2) ->  p1.getName().compareTo(p2.getName()));
3
4 // com stream
5 people.stream().sorted((p1, p2) ->  p1.getName().compareTo(p2.getName())).
     toArray();
```

Adicionada na versão 8 do Java, a API *Stream* é dividida em dois conjuntos de operações: intermediárias e finais (ORACLE, 2014a). As operações intermediárias permitem o encadeamento de operações, como o método *filter* no exemplo abaixo.

[4] *Map-reduce* é um modelo computacional distribuído, inspirado nas funções *map* e *reduce*, utilizadas comumente em programação funcional.

```
1 Collection<Person> people = ...
2 people.stream().filter(person -> person.getGender().equals(Gender.MALE))
3    .forEach(person -> System.out.println(person.getName()));
```

Neste exemplo, o método *filter* filtra todas as pessoas do gênero masculino e retorna um novo *stream*. A partir desse retorno, o método *forEach* é invocado, imprimindo apenas as pessoas do gênero masculino. A diferença entre uma operação intermediária e uma final está no tipo do retorno do método. Toda operação intermediária retorna um objeto do tipo *Stream*, permitindo a chamada de outros métodos de forma encadeada, enquanto as operações finais encerram o encadeamento dos métodos.

A API *Stream* disponibiliza vários recursos interessantes, como por exemplo: *filter*, *distinct*, *sorted*, *limit*, *parallel*, os quais pertencem ao grupo de operações intermediárias. E outros recursos como: *forEach*, *toArray*, *min*, *max*, *count*, *findFirst*, pertencentes ao grupo de operações finais.

Streams diferem das coleções de várias maneiras (ORACLE, 2014a):

- **Sem armazenamento**. Um *stream* não armazena valores, apenas os transfere a partir de uma fonte de dados;
- **Funcionais por natureza**. Operações sobre um *stream* produzem um resultado e não modificam sua fonte de dados original;
- ***Laziness-seeking***. Operações como filtragem ou mapeamento podem ser implementadas preguiçosamente, assim, uma operação só é aplicada a elementos da fonte de dados quando necessário.
- **Possivelmente ilimitados**. Enquanto as coleções têm um tamanho finito, os *streams* possuem operações como *limit* ou *findFirst*, que podem permitir cálculos sobre *streams* infinitos, os quais possam ser completados em tempo finito.
- **Consumíveis**. Os elementos de um *Stream* só são visitados apenas uma vez. Como um *Iterator*, um novo *Stream* deve ser gerado para revisitar os mesmos elementos da fonte de dados.

2.3.6 Programação Paralela

A execução da API *Stream* pode ser realizada de maneira sequencial ou paralela. Como visto no exemplo anterior, ao utilizar o método *stream()* de uma coleção, a sua execução é aplicada de forma sequencial. Porém, ao utilizar o método *parallelStream()*, as operações encadeadas seguintes serão executadas de forma paralela, sem uma ordem pré-definida, conforme exemplo abaixo.

```
1 Collection<Person> people = ...
2 people.parallelStream().filter(person -> person.getGender().equals(Gender.
    MALE)).forEach(person -> System.out.println(person.getName()));
```

Todas as operações *streams* podem ser executadas de forma paralela ou sequencial (ORACLE, 2014a). A maioria das operações de *stream* aceita parâmetros que descrevem o comportamento especificado pelo usuário, os quais são sempre instâncias de interfaces funcionais, podendo utilizar expressões lambdas e referências a métodos.

2.3.7 Referências a Métodos

As expressões lambda permitem a definição de métodos anônimos e seu uso como instâncias das interfaces funcionais (ORACLE, 2013). Pode-se dizer que uma referência a método é uma expressão lambda, pois todos os locais que tenham uma expressão podem utilizar uma referência a método. Uma referência a método é uma chamada a um método que possui as mesmas características exigidas por uma interface funcional, ou seja, um método que for usado como referência deve possuir o mesmo tipo de retorno e argumentos (tipo e quantidade) da assinatura do método de uma interface funcional. O exemplo abaixo apresenta um trecho de código utilizando referência a métodos.

```
1 List<Person> list = (...);
2 list.sort(Person::compareName);
```

Na linha 2 do exemplo acima, o método *sort* da coleção de pessoas (*list*) faz uma referência ao método estático *compareName* da classe *Person* (linha 3 do exemplo abaixo). O método *sort* espera uma implementação da interface funcional *Comparator*. Portanto o método *compareName* possui as mesmas características do método *compare* da interface *Comparator* (linha 9 do exemplo abaixo).

```
1  public class Person {
2    ...
3    public static int compareName(Person p1, Person p2) {
4      return p1.getName().compareTo(p2.getName());
5    }
6  }
7
8  public interface Comparator<T> {
9    int compare(T o1, T o2);
10 }
```

O uso de referências a métodos melhora a capacidade de reutilização quando a mesma expressão está sendo utilizada em vários locais de código.

2.4 Framework AOPJungle

O framework AOPJungle (FAVERI, 2013) foi desenvolvido com o objetivo de fornecer meta-informações sobre código orientado a objetos e aspectos. Sua função é executar uma instanciação de seu metamodelo, o qual armazena meta-informações de programas escritos na linguagem Java e AspectJ, admitindo receber e executar consultas sobre esse metamodelo.

Em primeiro momento, o AOPJungle foi concebido como parte de um processo de consulta de informações em código de projetos através de uma linguagem específica de domínio (DSL), chamada AQL (*Aspect Query Language*). Ao utilizar a AQL, o framework AOPJungle retorna informações que podem ser acessadas por meio de uma coleção de classes que representa toda a estrutura de informações coletadas dos projetos analisados. Entretanto, o AOPJungle é uma ferramenta independente e não necessita obrigatoriamente da AQL para sua execução. A Figura 2.1 apresenta a arquitetura do AOPJungle.

O AOPJungle é um plug-in desenvolvido para a plataforma Eclipse e possui dois principais módulos em sua arquitetura: o *extrator*, responsável por coletar informações referentes aos projetos contidos no workspace do eclipse; e o *metamodelo*, estrutura responsável por armazenar as informações coletadas pelo extrator em um modelo orientado a objetos, permitindo o acesso a essas informações.

O módulo *extrator* é responsável por analisar o código fonte dos projetos existentes no *workspace* do eclipse e instanciar o metamodelo. Para a análise dos projetos, duas fontes de informações são utilizadas: a API da plataforma Eclipse (*Eclipse Platform API*), que fornece informações sobre toda a estrutura dos projetos, como por exemplo, a lista dos projetos e seus arquivos; e as ASTs (*Abstract Syntax Tree*), estruturas de dados em árvore que disponibilizam

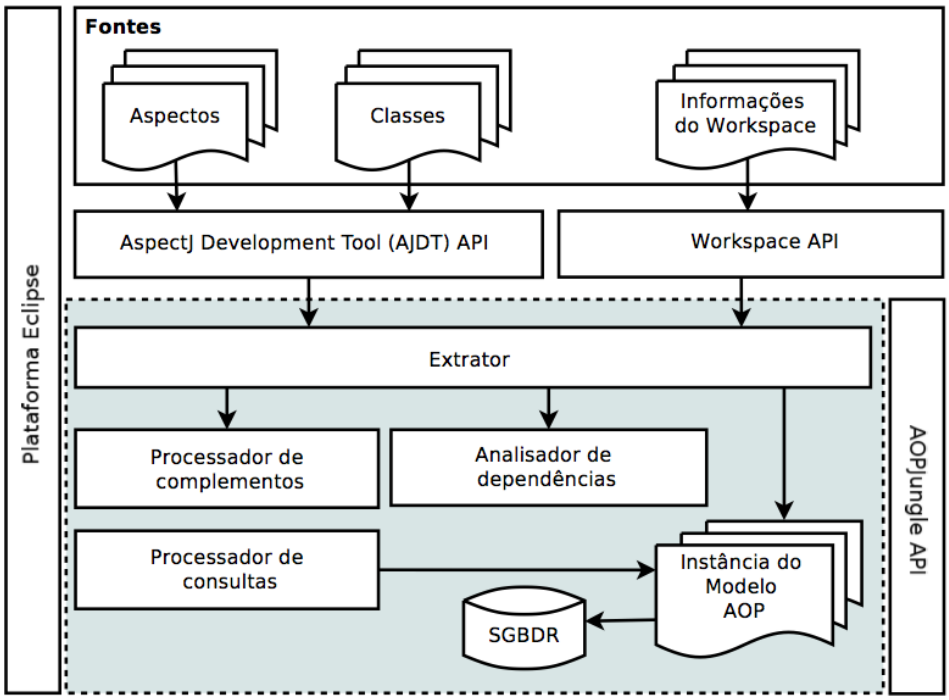

Figura 2.1 – Arquitetura do AOPJungle (FAVERI, 2013)

informações sobre as unidades de compilação[5] (UC).

O processo de extração inicia-se com a identificação dos projetos, Java e AspectJ, pertencentes ao *workspace* da IDE Eclipse. Utilizando uma instância de tipo compatível com a interface (*IWorkSpaceRoot*)[6], é possível obter informações sobre projetos, pastas e arquivos.

A partir da lista de projetos, o Extrator utiliza as APIs JDT (*Java Development Tools*) e AJDT (*AspectJ Development Tools*) para identificar os pacotes e as unidades de compilação. Após a identificação, o Extrator percorre todas as ASTs em busca das informações requisitadas e as adiciona na instância do metamodelo. Ao finalizar a extração das informações, dois módulos são acionados: (i) o *analisador de dependências*, responsável por construir as associações e as dependências entre os elementos do modelo; e (ii) o *processador de complementos*, responsável pela execução de outros módulos após a instanciação do metamodelo, tais como: o cálculo de métricas e estatísticas para os projetos analisados.

2.5 Trabalhos Relacionados

Em busca de trabalhos voltados para a prática de refatorações em projetos orientados a objetos e aspectos, bem como a estudos ligados às expressões lambda em Java, dois trabalhos

[5] Unidade de Compilação (*Compilation Unit*): representa o código fonte de um arquivo Java ou AspectJ.
[6] *IWorkSpaceRoot*: interface da API do Eclipse disponível no pacote *org.eclipse.core.resources*.

se destacaram por serem fortemente relacionados a esse cenário.

O primeiro trabalho apresenta uma ferramenta chamada *LambdaFicator* (FRANKLIN et al., 2013), construída com o propósito de realizar refatorações de forma automática envolvendo expressões lambda em Java. Dois cenários de refatoração foram apresentados, são eles:

- ***AnonymousToLambda***. Visa transformar instâncias de classes anônimas em uma sintaxe correspondente utilizando expressões lambda. A Listagem 2.5 apresenta o código antes e depois de se realizar a refatoração.

Listagem 2.5 – *AnonymousToLambda*

```
1 //Antes
2 String sep = doAction(new PrivilegedAction() {
3    public String run() {
4        return System.getProperty("file.separator");
5    }
6 }) ;
7 //Depois
8 String sep = doAction(() -> System.getProperty("file.separator"));
```

- ***ForLoopToFunctional***. Propõe converter laços *for* com iteração sobre uma coleção, em uma operação funcional utilizando expressões lambda. A Listagem 2.6 apresenta o código antes e depois de se realizar a refatoração.

Listagem 2.6 – *ForLoopToFunctional*

```
1  //Antes
2  private boolean isEngineExisting(String grammarName) {
3     for (GrammarEngine e : importedEngines) {
4        if (e.getGrammarName() == null)
5           continue;
6        if (e.getGrammarName().equals(grammarName))
7           return true;
8     }
9     return false;
10 }
11 //Depois
12 private boolean isEngineExisting(String grammarName) {
13    return importedEngines.stream()
14       .filter(e -> e.getGrammarName() != null)
15       .anyMatch(e -> e.getGrammarName().equals(grammarName));
16 }
```

O estudo de caso apresentado utilizou o *LambdaFicator* sobre nove projetos de código aberto e em sua execução foram detectadas 1.263 oportunidades de refatoração para a *AnonymousToLambda*, sendo que 55% delas foram convertidas para expressões lambda. Para a refatoração *ForLoopToFunctional*, 1.595 oportunidades foram encontradas, sendo 63% delas convertidas pela ferramenta. Outro dado interessante foi o número de linhas modificadas com a

aplicação das refatorações pela ferramenta. Os resultados mostraram que 8.538 linhas foram alteradas, indicando que o uso dessa ferramenta pode auxiliar em trabalhos manuais.

O segundo trabalho faz a análise, o projeto, a implementação e a avaliação das mesmas refatorações do trabalho anterior, e utiliza também a ferramenta *Lambdaficator* para o estudo de caso. Gyori et al. (2013) utilizaram nove projetos de código aberto, encontrando um total de 1.263 oportunidades de refatoração para *AnonymousToLambda* e 1.709 para *ForLoopToFunctional*. Tal estudo mostrou uma nova motivação em relação ao anterior, apontando vantagens de utilizar expressões lambda em situações que necessitem o uso de operações com execução em paralelo. A redução do código e a simplicidade em utilizar paralelismo é visivelmente observada em exemplos e comentários.

Para o estudo de caso desse trabalho foram analisados cinco quesitos: (i) a aplicabilidade das refatorações; (ii) se a refatoração realmente melhora a qualidade do código; (iii) o quanto de esforço o programador é salvo ao utilizar o *LambdaFicator*; (iv) a precisão ao se utilizar o *LambdaFicator*; e (v) a segurança em utilizar o *LambdaFicator*. Em análise das questões enumeradas, a ferramenta *LambdaFicator* mostrou auxiliar na conversão automática de refatorações.

No momento da construção deste trabalho, foi lançada a versão 13.1 da IDE IntelliJ IDEA com suporte para cinco das refatorações propostas neste trabalho, as quais serão apresentadas no próximo capítulo, são elas: *Convert Functional Interface Instance to Lambda Expression* (*Replace With Lambda* no IDEA), *Convert Enhanced For to Lambda Enhanced For* e *Convert Enhanced For with If to Lambda Filter* (ambas referem-se a *Replace With Foreach* no IDEA), *Extract Method Reference* (*Replace Lambda with Method References* no IDEA) e *Convert Abstract Interface Method to Default Method* (*Make Method Default* no IDEA). Este suporte adicional está previsto em outras IDEs, como Eclipse e NetBeans.

2.6 Considerações Finais

Este capítulo discutiu os principais conceitos necessários a esta dissertação. O conceito de refatoração foi apresentado, bem como a estrutura para definir uma refatoração e as etapas que um processo de refatoração deve realizar. A etapa *identificar onde aplicar uma refatoração* será abordada em forma de um estudo de caso no Capítulo 5. Foram apresentados detalhes sobre as expressões lambda adicionadas à linguagem Java, exibindo conceitos e exemplos sobre suas funcionalidades, as quais foram utilizadas na definição do catálogo de refatoração no próximo

capítulo. Este capítulo apresentou também detalhes da arquitetura do *framework* AOPJungle, que foi utilizado para a construção de uma ferramenta para a busca de oportunidades de refatoração. O Capítulo 4 fornece mais informações sobre as modificações e as adições realizadas no *framework*. Por fim, dois trabalhos fortemente relacionados foram analisados, mostrando algumas refatorações ligadas a este trabalho, bem como os resultados da aplicação de tais refatorações em código existente.

3 UM CATÁLOGO DE REFATORAÇÕES ENVOLVENDO EXPRESSÕES LAMBDA EM JAVA

Refatorações em sistemas de software têm sido estudadas há anos e muitos desses estudos estão voltados às linguagens de programação. Em sistemas orientados a objetos, diversos trabalhos exploram a identificação de cenários de refatoração (FOWLER et al., 1999; VAN EMDEN; MOONEN, 2002; OLBRICH et al., 2009; SCHUMACHER et al., 2010), bem como em sistemas orientados a aspectos (PIVETA et al., 2006), (MACIA; GARCIA; STAA, 2010), (HUANG et al., 2011), os quais resultaram em catálogos de refatorações.

Este capítulo apresenta um catálogo de refatorações para o uso de expressões lambdas voltadas para sistemas orientados a objetos e aspectos, cujas refatorações são descritas no formato usual. Geralmente uma refatoração é descrita por um nome, uma motivação, uma mecânica (um conjunto de passos para aplicá-la), e um exemplo. Dessa forma, um conjunto de 20 refatorações é proposto, inclusive as inversas, as quais apresentam detalhes sobre a motivação, a mecânica e os exemplos de utilização.

A seguir são descritas as dez primeiras refatorações, que possuem como objetivo a adaptação para alguns dos novos recursos da versão 8 da linguagem Java:

- *Convert Functional Interface Instance to Lambda Expression*.
- *Convert Enhanced For to Lambda Enhanced For*.
- *Convert Collections.sort to sort*.
- *Convert Enhanced For with If to Lambda Filter*.
- *Convert Functional Interface to Default Functional Interface*.
- *Convert Abstract Interface Method to Default Method*.
- *Convert Inter-Type Method Declaration to Default Interface Method*.
- *Extract Method Reference*.
- *Convert Interface to Functional Interface*.
- *Convert Enhanced For to Parallel For*.

Convert Functional Interface Instance to Lambda Expression

Você tem uma instância de interface funcional que é usada ao longo do código fonte de um projeto.

◇ ◇ ◇

Portanto, substitua o uso dessa instância por uma expressão lambda.

O uso de expressões lambda na implementação de interfaces funcionais apoia o uso das novas funcionalidades da linguagem Java, ajudando a melhorar a sua legibilidade e a facilidade de manutenção, além de reduzir seu código padrão de pouca expressividade. A sua mecânica é descrita da seguinte forma:

1. Implemente uma instância de uma interface funcional utilizando expressões lambda (*ParameterSpace −> BodySpace*).
2. Copie os argumentos do método da instância convencional da interface funcional e posicione-os antes do *token* (−>).
3. Copie o código da implementação do método e coloque-o após o *token*.
4. Substitua a implementação convencional da interface funcional pela expressão lambda.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois da aplicação da refatoração. O código antes da refatoração implementa uma classe anônima *ActionListener*. Após a refatoração, o código apresentado possui a mesma funcionalidade, porém utiliza expressões lambda.

```
1  // Antes
2  ActionListener al = new ActionListener() {
3     public void actionPerformed(ActionEvent e) {
4        System.out.println("Hello!");
5     }
6  };
7  // Depois
8  ActionListener al = e -> System.out.println("Hello!");
```

Convert Enhanced For to Lambda Enhanced For

Você está usando uma construção for each para percorrer uma coleção.

◇ ◇ ◇

Converta esta construção em uma expressão lambda usando o método forEach.

Esta refatoração é mais vantajosa quando o desenvolvedor precisa adicionar outros comportamentos ao percorrer uma coleção, como um filtro (*filter*) ou uma ordenação (*sort*), por exemplo. Neste contexto, o uso de interfaces fluentes (FOWLER, 2010) proporciona um mecanismo para enfileirar as operações para serem aplicadas em conjunto. A mecânica é descrita na seguinte forma:

1. Substitua a construção for each pela chamada do método *forEach* da coleção.
2. Implemente a interface funcional *Consumer* requerida pelo método *forEach*, movendo o conteúdo da estrutura de repetição para o método da interface *Consumer*.
3. Aplique a refatoração *Convert Functional Interface Instance to Lambda Expression* para a instância da interface *Consumer*.
4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois da aplicação da refatoração. Antes da refatoração existe uma coleção de pessoas chamada *people* que é percorrida por uma construção *for each*. O código somente imprime os nomes das pessoas. Após a refatoração, a construção *for each* foi substituída pelo método *forEach* da coleção *people* utilizando uma expressão lambda.

```
1  // Antes
2  Collection<Person> people =  ...
3  for (Person person : people)
4     System.out.println(person.getName());
5
6  // Depois
7  Collection<Person> people =  ...
8  people.forEach(person -> System.out.println(person.getName()));
```

Convert Collections.sort to sort

Você tem uma chamada para o método Collections.sort.

◇ ◇ ◇

Portanto, substitua esta chamada pelo método sort da própria coleção.

A interface *Collection* fornece uma nova funcionalidade que permite a ordenação da própria coleção. Diferentemente do método *sort* da classe *Collectoins*, essa nova funcionalidade utiliza a própria instância da coleção para a ordenação. O processo de ordenação é feito utilizando a implementação da interface funcional *Comparator*. Aplicar uma ordenação diretamente através de uma instância de uma coleção aumenta a capacidade de escrita, exigindo menos argumentos e permitindo o uso de expressões lambda para aplicar os mecanismos de comparação dos itens da coleção. A mecânica é descrita da seguinte forma:

1. Utilize o método *sort* da própria coleção, ao invés de o método *sort* da classe *Collections*.
2. Mova a implementação da classe anônima que estende a interface funcional *Comparator* para a chamada ao método *sort* da instância da coleção.
3. Aplique a refatoração *Convert Functional Interface Instance to Lambda Expression*.
4. Exclua a implementação antiga que utiliza a classe *Collections*.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de se aplicar a refatoração. Note que o código antes da refatoração é mais longo e mais complicado. O método *sort* pode ser combinado com outros métodos da coleção, utilizando-os de forma encadeada.

```
1   // Antes
2   Collections.sort((List<Person>)people,
3      new Comparator<Person>(){
4         public int compare(model.Person p1, model.Person p2) {
5            return p1.getName().compareTo(p2.getName());
6         }
7      });
8
9   // Depois
10  ((List<Person>) people).sort((p1, p2) ->
11     p1.getName().compareTo(p2.getName()));
```

Convert Enhanced For with If to Lambda Filter

Você tem um controle de seleção dentro de uma iteração sobre uma coleção.

◇◇◇

Portanto, substitua o controle de seleção por um filtro.

Uma maneira de implementar um filtro sobre uma coleção em Java é através de um controle de seleção (*if*) dentro de uma iteração. Com expressões lambda, o método *filter* pode ser utilizado para executar a mesma tarefa. Esta refatoração reduz o tamanho do código, permite o encadeamento de filtros através de interfaces fluentes e, se necessário, permite utilizar outros recursos disponíveis na coleção. A mecânica é descrita da seguinte forma:

1. Utilizando o idioma de interfaces fluentes, use o método *stream* e, em sequência, o método *filter* sobre uma coleção.
2. Implemente a interface *Predicate* requerida pelo método *filter*, fazendo a mesma comparação do controle de seleção.
3. Adicione a chamada do método *forEach* de forma encadeada.
4. Implemente a interface *Consumer* no método *forEach* e defina o resto da implementação.
5. Aplique a refatoração *Convert Functional Interface Instance to Lambda Expression*.
6. Apague a estrutura *for each* antiga.
7. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. O código antes da refatoração apresenta uma coleção de pessoas (*people*) que é percorrida por uma construção *for each*. Esse código imprime apenas os nomes de pessoas do gênero masculino. O código depois da refatoração substituiu a construção *for each* pelo método *forEach* e o controle de seleção (*if*) pelo método *filter*, ambos utilizando expressões lambda.

```
1  // Antes
2  for (Person person : people)
3     if (person.getGender().equals(Gender.MALE))
4        System.out.println(person.getName());
5
6  // Depois
7  people.stream()
8     .filter(person -> person.getGender().equals(Gender.MALE))
9     .forEach(person -> System.out.println(person.getName()));
```

Convert Functional Interface to Default Functional Interface

Você criou uma interface funcional própria, mas existe uma interface padrão de mesmo propósito.

◇ ◇ ◇

Portanto, substitua os usos da interface funcional para uma interface funcional padrão adequada.

Em Java 7 há várias interfaces funcionais, tais como *Runnable* e *ActionListener*. Em Java 8, um conjunto de novas interfaces (*Predicate*, *Consumer*, *Function*, *Supplier*, etc) está disponível para ser utilizado com expressões lambdas. Esta refatoração substitui as interfaces funcionais *ad hoc* por interfaces padrões. A mecânica dessa refatoração é descrita em seguida:

1. Identifique a interface funcional a ser substituída, comparando com as interfaces disponíveis (no pacote *java.util.function*, por exemplo), verificando o tipo de retorno e parâmetros (quantidade e tipo).

2. Substitua todas as referências da interface funcional antiga para a correspondente interface padrão.

3. Apague a interface antiga.

4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração, implementações que usavam a interface *Listener* agora usam a interface funcional *Consumer*. Note que ambas interfaces possuem as mesmas características para uma expressão lambda.

```
1  // Antes
2  public interface Listener {
3     public void select(Object object);
4  }
5
6  // Depois
7  public interface Consumer<T> {
8     void accept(T t);
9  }
```

Convert Abstract Interface Method to Default Method

Você tem classes que implementam métodos vazios só porque uma interface os exige.

◇ ◇ ◇

Portanto, proporcione uma implementação de método default na interface, se possível.

Em versões de Java anteriores à 8, todos os métodos de uma interface eram abstratos. Consequentemente, toda interface funcional poderia conter apenas a assinatura de um método em sua estrutura. Agora, as interfaces permitem implementações concretas de seus métodos, desde que eles sejam definidos como um método *default*. Usando esse recurso, é possível a existência de interfaces funcionais com mais de um método, além de permitir que os métodos sejam implementados diretamente em interfaces, quando necessário. Esta refatoração é destinada a ser aplicada a métodos de interface, quando eles possuírem implementações vazias em classes somente para satisfazer o protocolo de uma interface. A mecânica é descrita da seguinte forma:

1. Adicione o modificador *default* no método da interface.
2. Forneça um código ao método *default*, se desejado. Ou deixe o corpo do método vazio, caso contrário.
3. Remova os métodos vazios, referente ao método refatorado, de classes que implementem a interface.
4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de se aplicar a refatoração. Note que, antes o método *find* da interface *Service* era do tipo abstrato e precisava ter uma implementação em cada classe que implementasse a interface. Depois da refatoração, passou a ser um método *default* (sendo desnecessária a implementação vazia nas classes, exceto quando desejado).

```
1   // Antes
2   public interface Service<T> {
3      public void save(T t);
4      public void delete(T t);
5      public Collection<T> find(T t);
6   } // Depois
7   public interface Service<T> {
8      public void save(T t);
9      public void delete(T t);
10     public default Collection<T> find(T t) {
11        return null;
12     }
13  }
```

Convert Inter-Type Method Declaration to Default Interface Method

---

Você tem uma declaração inter-tipos que afeta uma interface somente para emular um método default.

◇◇◇

Portanto, substitua a declaração inter-tipos por um método default.

A programação orientada a aspectos fornece mecanismos de abstração e de composição para a implementação de interesses transversais. Um desses mecanismos é a declaração inter-tipos, que permite a adição de estado e de comportamento em classes existentes. Declarações inter-tipos são frequentemente utilizadas para adicionar métodos concretos em interfaces. Esta refatoração visa converter uma declaração de método inter-tipos que afeta uma interface em um método *default* de interface. A mecânica é descrita da seguinte forma:

1. Adicione na interface um método *default* com o mesmo nome utilizado pela declaração de método inter-tipos.
2. Copie o corpo do método da declaração inter-tipos para o novo método *default*.
3. Remova a declaração de método inter-tipos do aspecto.
4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Note que antes, o aspecto *ModelAspect* adicionava o método *print* na interface *Model*. Após a aplicação refatoração, a implementação do método *print* é feita diretamente na interface *Model*, utilizando um método *default*.

```
1  // Antes
2  public aspect ModelAspect {
3     public String Model.print() {
4        String s = "";
5        for (Field f : this.getClass().getDeclaredFields())
6           s += f.getName() + ": ";
7        return s;
8     }
9     ...
10 }
11 public interface Model {...}
12 // Depois
13 public interface Model {
14    default public String print(){
15       // o mesmo codigo apresentado no metodo Model.print().
16    }
17 }
18 public aspect ModelAspect {...}
```

Extract Method Reference

Você tem uma expressão lambda e seria mais adequado utilizar uma referência a um método.

◇ ◇ ◇

.

As expressões lambda permitem a definição de métodos anônimos e seu uso como instâncias das interfaces funcionais (ORACLE, 2013). Às vezes, é desejável fazer o mesmo com um método. Esta refatoração visa converter uma expressão lambda para uma referência a método. O uso de uma referência a método melhora a capacidade de reutilização quando a expressão está sendo utilizada em vários locais de código. A mecânica é descrita da seguinte forma:

1. Crie um método para ser utilizado através de uma referência, com as mesmas características do método da interface funcional utilizada pela expressão lambda.
2. Implemente o método com o código localizado depois do *token* da expressão lambda.
3. Substitua a expressão lambda pela referência ao método criado.
4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. A expressão lambda é substituída por uma referência ao método *Person::compareName*.

```
1   // Antes
2   List<Person> list = (...);
3   list.sort((p1, p2) -> p2.getName().compareTo(p1.getName()));
4
5   public class Person {...}
6
7   // Depois
8   List<Person> list = (...);
9   list.sort(Person::compareName);
10
11  public class Person {
12     ...
13     public static int compareName(Person p1, Person p2) {
14        return p1.getName().compareTo(p2.getName());
15     }
16  }
```

Convert Interface to Functional Interface

Você tem uma interface e deseja utilizá-la com expressões lambda.

◇ ◇ ◇

Portanto, mantenha/defina um de seus métodos abstratos e o restante transforme em métodos default, caso existam.

Uma expressão lambda é aplicada exclusivamente a uma interface funcional. Dessa forma, interfaces não funcionais que, são utilizadas como classes anônimas por exemplo, possuem a necessidade de serem implementadas com expressões lambda, podem transformar seus métodos (exceto um) em métodos *default*. Portanto, esta refatoração se destina à transformação de uma interface não funcional em uma interface funcional, convertendo seus métodos em métodos *default*. A mecânica é descrita da seguinte forma:

1. Identifique o método da interface que não será transformado em método *default*.
2. Aplique a refatoração *Convert Abstract Interface Method to Default Method* nos demais métodos.
3. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração a interface *Listener* possuía dois métodos abstratos. Após a refatoração, o método *isSelected* foi definido como método *default*, inserindo uma implementação padrão em seu corpo.

```
1   // Antes
2   public interface Listener {
3      public void select(Object object);
4      public Boolean isSelected();
5   }
6
7   // Depois
8   public interface Listener {
9      public void select(Object object);
10     public default Boolean isSelected(){
11        return null;
12     }
13  }
```

Convert Enhanced For to Parallel For

Você está usando uma construção for each de forma sequencial para percorrer um coleção.

◇ ◇ ◇

Portanto, substitua essa construção por uma implementação paralela utilizando os recursos da API Stream.

Embora a linguagem Java possua bibliotecas para a execução de instruções de forma paralela, uma das motivações para o uso das expressões lambda tem sido a facilidade em se codificar esse tipo de instrução. Esta refatoração visa aplicar um paralelismo em uma estrutura de repetição *for each*, utilizando expressões lambda e recursos da API *Stream*. A mecânica é descrita da seguinte forma:

1. Substitua a construção *for each* pela chamada dos métodos *parallelStream()* e *forEach()* da coleção, de forma encadeada.
2. Implemente a interface funcional *Consumer* requerida pelo método *forEach*, movendo o conteúdo da estrutura de repetição para o método da interface *Consumer*.
3. Elimine implementações da classe *Thread*, caso necessário.
4. Aplique a refatoração *Convert Functional Interface Instance to Lambda Expression* para a instância da interface *Consumer*.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração a estrutura de repetição *for each* percorre uma coleção de pessoas e executa o método *save*. Após a refatoração a estrutura de repetição é executada de forma paralela, utilizando os recursos da API *Stream*.

```
1  // Antes
2  Collection<Person> people = new ArrayList<Person>();
3  for (Person person : people)
4     save(person);
5
6  // Depois
7  Collection<Person> people = new ArrayList<Person>();
8  people.parallelStream().forEach(person -> save(person));
```

3.1 Refatorações Inversas

Geralmente, as refatorações possuem suas transformações inversas, pois o contexto da aplicação é diferente para cada uma delas. Por exemplo, a refatoração *Extract Method* é usada quando um método é longo e existe a necessidade de extrair parte de sua funcionalidade para um novo método. A sua refatoração inversa é a *Inline Method*, que é aplicada quando um método curto é utilizado por um ou por outros poucos métodos. *Inline Method* elimina o método, substituindo suas chamadas pelo corpo do método. O mesmo ocorre com outras refatorações, tais como: *Pull Up Method* $\overset{inversa}{\leftrightarrow}$ *Push Down Method*, *Rename (oldName, newName)* $\overset{inversa}{\leftrightarrow}$ *Rename (newName, oldName)*, etc. O uso de refatorações inversas pode ser incomum, porém pode ser necessário. São exemplos de situações que podem requerer o uso de refatorações inversas: (i) o projeto requer uma versão anterior da linguagem que não utiliza expressões lambda; (ii) desenvolvedores que não estão habituados com o uso de expressões lambda podem efetuar transformações inversas temporariamente, para melhor entender o que está acontecendo com um trecho de código que usa expressões lambda; (iii) o uso de uma expressão lambda deixou a intenção do desenvolvedor menos clara que o uso de recursos convencionais; ou (iv) parte da expressão lambda será extraída para outros métodos ou movida para outras classes. Pode ser mais simples usar uma refatoração inversa, modificar o que precisa ser modificado, e depois aplicar uma refatoração que converte novamente para o uso de expressões lambda. A seguir são apresentadas as refatorações inversas propostas:

- *Convert Lambda Expression to Functional Interface Instance*.
- *Convert Lambda Enhanced For to Enhanced For*.
- *Convert sort to Collections.sort*.
- *Convert Lambda Filter to Enhanced For with If*.
- *Convert Default Functional Interface to Functional Interface*.
- *Convert Default Method to Abstract Interface Method*.
- *Convert Default Interface Method to Inter-Type Method Declaration*.
- *Inline Method Reference*.
- *Convert Functional Interface to Interface*.
- *Convert Parallel For to Enhanced For*.

Convert Lambda Expression to Functional Interface Instance

Você utiliza uma expressão lambda para instanciar uma interface funcional, porém o código está extenso e confuso.

◇ ◇ ◇

Portanto, substitua o uso dessa expressão por uma classe anônima correspondente.

O uso de expressões lambda na implementação de interfaces funcionais ajuda a melhorar a legibilidade, a facilidade de manutenção e a redução de código. Porém, se o código possuir muitas instruções e for muito extenso, utilizar uma expressão lambda pode não ser tão vantajoso. Esta refatoração converte uma implementação de uma interface funcional utilizando expressão lambda, por uma implementação de classe anônima. A mecânica é descrita da seguinte forma:

1. No local da expressão lambda, instancie de forma convencional a interface funcional requerida.
2. Dê os mesmos nomes dos argumentos da expressão lambda aos argumentos na assinatura do método da interface funcional, caso existam.
3. Mova o corpo da expressão lambda (localizado após o *token* ->) para o corpo do método da interface.
4. Apague a expressão lambda.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois da aplicação da refatoração. O código antes da refatoração apresenta uma expressão lambda instanciando a interface *ActionListener*. Após a refatoração, a mesma instância é feita de forma convencional.

```
1  // Antes
2  ActionListener al = e -> System.out.println("Hello!");
3
4  // Depois
5  ActionListener al = new ActionListener() {
6     public void actionPerformed(ActionEvent e) {
7        System.out.println("Hello!");
8     }
9  };
```

Convert Lambda Enhanced For to Enhanced For

O uso do método forEach tornou a implementação muito complicada.

◇ ◇ ◇

Portanto, converta este método em uma construção for each tradicional.

Utilizar expressões lambda para percorrer uma lista é mais vantajoso quando o desenvolvedor precisa adicionar outros comportamentos ao percorrer uma coleção, como um filtro ou uma ordenação. Se sua iteração não precisa de comportamentos adicionais, talvez seja interessante utilizar uma construção *for each* tradicional. A mecânica é descrita na seguinte forma:

1. Implemente uma construção *for each* para a coleção utilizada pelo método *forEach*.
2. Mantenha o mesmo nome do argumento na estrutura *for each* utilizada na formatação da expressão lambda (argumento antes do *token*).
3. Mova o corpo da expressão lambda (localizado depois do *token*) para dentro do escopo da construção *for each*.
4. Apague a estrutura de repetição antiga.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois da aplicação da refatoração. Antes da refatoração, o método *forEach* da coleção *people* precorre toda a coleção utilizando expressões lambda e imprime o nome do objeto *person*. Após a refatoração, uma construção *for each* convencional executa a mesma tarefa.

```
1  // Antes
2  Collection<Person> people =  ...
3  people.forEach(person -> System.out.println(person.getName()));
4
5  // Depois
6  Collection<Person> people =  ...
7  for (Person person : people)
8     System.out.println(person.getName());
```

Convert sort to Collections.sort

Você faz uma ordenação utilizando o método sort da própria coleção, mas necessita de uma ordenação por meio de uma classe externa.

◇◇◇

Portanto, substitua esta ordenação pelo método estático Collections.sort.

A interface *Collection* fornece uma funcionalidade que permite a ordenação da própria coleção. Porém, pode existir a necessidade da ordenação por meio de uma classe externa. A classe *Collections* disponibiliza um método estático (*sort*) para realizar uma ordenação. Entretanto, para utilizar este recurso, devem ser passados dois argumentos: a coleção e o critério de ordenação (interface *Comparator*). Esta refatoração, além de utilizar o método estático *sort* da classe *Collections*, converte uma expressão lambda em uma implementação convencional, utilizando a interface funcional *Comparator*. A mecânica é descrita da seguinte forma:

1. Implemente uma nova ordenação utilizando o método *sort* da classe *Collections*.
2. Adicione a instância da interface *Collection* como primeiro argumento do método *sort*.
3. Mova a expressão lambda para local do segundo argumento do método *sort*.
4. Aplique a refatoração *Convert Lambda Expression to Functional Interface Instance*.
5. Apague a ordenação antiga.
6. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de se aplicar a refatoração. Antes da refatoração, a ordenação é feita diretamente na coleção *people*, através do método *sort*. Após a refatoração, a mesma ordenação é feita pelo método estático *Collections.sort* e a expressão lambda foi substituída pela implementação convencional da interface funcional *Comparator*.

```
1   // Antes
2   ((List<Person>) people).sort((p1, p2) ->
3      p1.getName().compareTo(p2.getName()));
4
5   // Depois
6   Collections.sort((List<Person>)people,
7      new Comparator<Person>(){
8         public int compare(model.Person p1, model.Person p2) {
9            return p1.getName().compareTo(p2.getName());
10        }
11     });
```

Convert Lambda Filter to Enhanced For with If

Você tem uma iteração e um filtro sobre uma coleção, utilizando expressões lambda. No entanto, seu filtro tornou-se demasiadamente complexo.

◇◇◇

Portanto, substitua essa instrução por uma construção for each e um controle de seleção (if).

Com expressões lambda há certa praticidade ao se aplicar filtros em uma coleção. Porém, dependendo da complexidade do filtro, é interessante utilizar um controle de seleção (*if*) dentro da iteração. A mecânica desta refatoração é descrita da seguinte forma:

1. Implemente uma construção *for each* para a coleção utilizada pelo método *forEach*;
2. Mantenha o mesmo nome do argumento na construção *for each* utilizada na formatação da expressão lambda (argumento antes do *token*).
3. Implemente um controle de seleção (*if*) e aplique a mesma verificação realizada pelo método *filter*.
4. Mova o código localizado após o *token*, que se encontra no método *forEach*, para dentro do escopo do controle seleção (*if*).
5. Apague a instrução antiga.
6. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. O código antes utiliza expressões lambda e aplica uma iteração e um filtro sobre a coleção *people*, imprimindo apenas o nome das pessoas do gênero masculino. Após a refatoração, os métodos *filter* e *forEach* são substituídos, respectivamente, pelas estruturas *if* e *for each*.

```
1   // Antes
2   people.stream().filter(person -> person.getGender()
3     .equals(Gender.MALE))
4     .forEach(person -> System.out.println(person.getName()));
5
6   // Depois
7   Collection<Person> people = ...
8   for (Person person : people)
9      if (person.getGender().equals(Gender.MALE))
10        System.out.println(person.getName());
```

Convert Default Functional Interface to Functional Interface

Você utiliza uma expressão lambda sobre uma interface funcional disponível na linguagem, mas precisa substituir essa interface funcional por uma interface ad hoc.

◇◇◇

Portanto, substitua por uma nova implementação de uma interface funcional que possua uma assinatura de método e com as mesmas características da interface anterior.

Em Java 7 existem várias interfaces funcionais, tais como *Runnable* e *ActionListener*. Em Java 8, um conjunto de novas interfaces (*Predicate*, *Consumer*, *Function*, *Supplier*, etc) está disponível para ser utilizado com expressões lambdas. Todavia, pode existir a necessidade de se utilizar uma interface funcional específica, ou seja, criada exclusivamente para um determinado propósito, disponibilizando outros recursos como métodos *default*. Esta refatoração substitui o uso de interfaces funcionais padrões (disponíveis na linguagem) por interfaces funcionais *ad hoc*. A mecânica é descrita em seguida:

1. Implemente uma interface funcional que tenha um método abstrato com as mesmas características da interface a ser substituída, verificando o tipo de retorno e parâmetros (quantidade e tipo).

2. Substitua todas as referências da interface funcional antiga para a interface correspondente.

3. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração, implementações que usavam a interface *Consumer* agora usam a interface funcional *Listener*. Note que ambas interfaces possuem as mesmas características exigidas por uma expressão lambda.

```
1  // Antes
2  public interface Consumer<T> {
3     void accept(T t);
4  }
5
6  // Depois
7  public interface Listener {
8     public void select(Object object);
9  }
```

Convert Default Method to Abstract Interface Method

Você possui um método default em uma interface e precisa que este método seja implementado obrigatoriamente nas classes.

◇◇◇

Portanto, transforme este método em abstrato e implemente-o em todas as classes relacionadas.

Na versão 8 da linguagem Java, as interfaces permitem implementações de métodos (*default*). Dessa forma, as classes que implementam essas interfaces não são obrigadas a implementar esses métodos. Podem existir casos nos quais um método *default* não seja a melhor opção para o objetivo da aplicação. Esta refatoração é destinada a converter um método *default* de interface em um método abstrato. A mecânica é descrita da seguinte forma:

1. Adicione um método em todas as classes que implementam a interface com a mesma assinatura do método *default* que será removido.
2. Mova o código do método *default* para todos os métodos adicionados nas classes.
3. Remova o modificador *default* do método da interface.
4. Remova a implementação do método da interface, deixando somente sua assinatura.
5. Compile o código e teste.

O exemplo a seguir descreve o código antes e depois de se aplicar a refatoração. Note que antes o método *find* da interface *Service* era do tipo *default* e depois passou a ser um método abstrato.

```
1   // Antes
2   public interface Service<T> {
3      public void save(T t);
4      public void update(T t);
5      public void delete(T t);
6      public default Collection<T> find(T t) {
7         return null;
8      }
9   }
10  // Depois
11  public interface Service<T> {
12     public void save(T t);
13     public void update(T t);
14     public void delete(T t);
15     public Collection<T> find(T t);
16  }
```

Convert Default Interface Method to Inter-Type Method Declaration

---

Você tem um método default em uma interface e precisa de uma implementação com mais recursos.

◇ ◇ ◇

Portanto, substitua este método por uma declaração inter-tipos em um aspecto.

Declarações inter-tipos são frequentemente usadas para adicionar métodos concretos em interfaces, emulando o novo recurso fornecido pela linguagem Java (métodos *default*). Uma grande vantagem em aplicar uma declaração inter-tipos é a possibilidade de utilizar recursos não disponíveis em interfaces, como atributos e métodos não estáticos. Esta refatoração visa converter um método *default*, em uma declaração de método inter-tipos. A mecânica é descrita em seguida:

1. Adicione uma declaração de método inter-tipos em um aspecto existente ou em um novo aspecto. Esse método deve afetar a interface que possui o método *default* a ser removido.
2. A assinatura do método inter-tipos deve possuir as mesmas características do método *default*.
3. Copie o código do corpo do método *default* e adicione no método inter-tipos do aspecto.
4. Remova o método *default* da interface.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Note que antes a interface *Model* implementava um método *default*. Após a aplicação da refatoração, a mesma implementação é feita pelo método inter-tipos do aspecto *ModelAspect*.

```
1  // Antes
2  public interface Model {
3     default public String print(){
4        String s = "";
5        for (Field f : this.getClass().getDeclaredFields())
6           s += f.getName() + ": ";
7        return s;
8     }
9  }
10 public aspect ModelAspect { ... }
11 // Depois
12 public aspect ModelAspect {
13    public String Model.print() {
14    // o mesmo codigo apresentado no metodo print() da classe Model
15    }
16 }
17 public interface Model { ... }
```

Inline Method Reference

Você utiliza uma referência a método e seria adequado utilizar uma expressão lambda.

◇ ◇ ◇

Portanto, remova a referência e utilize uma expressão equivalente.

As referências a métodos podem ser utilizadas em locais que esperam uma expressão lambda, ou seja, que esperam uma implementação de classe anônima referente a uma interface funcional. Às vezes não é interessante a utilização de referências a métodos, mas sim a aplicação de uma expressão lambda em si. Referencias a métodos geralmente são utilizadas quando uma mesma expressão aparece em vários locais do código, melhorando seu reuso. Esta refatoração visa transformar uma referência a método em uma expressão lambda. A mecânica é descrita da seguinte forma:

1. No local em que se encontra a referência a método, implemente uma expressão lambda de acordo com a interface funcional necessária.
2. Copie os argumentos do método referenciado e substitua na expressão (antes do *token*).
3. Copie o corpo do método e substitua na expressão (depois do *token*).
4. Apague o método referenciado, caso desejado.
5. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. A referência a método, *Person::compareName* (antes da refatoração), foi substituída por uma expressão lambda equivalente (depois da refatoração).

```
1   // Antes
2   List<Person> list = (...);
3   list.sort(Person::compareName);
4
5   public class Person {
6     ...
7     public static int compareName(Person p1, Person p2) {
8       return p1.getName().compareTo(p2.getName());
9     }
10  }
11
12  // Depois
13  List<Person> list = (...);
14  list.sort((p1, p2) -> p2.getName().compareTo(p1.getName()));
15
16  public class Person {...}
```

Convert Functional Interface to Interface

Você tem uma interface com métodos default e deseja transformá-los em abstratos.

◇ ◇ ◇

Portanto, remova o modificador default e a implementação padrão de cada método.

Em Java, interfaces funcionais são aquelas que possuem apenas um método abstrato, mas podem possuir outros métodos, por exemplo, métodos *default*. Tais métodos podem não atender o objetivo de uma determinada interface, pois suas implementações são genéricas e não são obrigadas a serem codificadas em classes que implementam sua interface. Portanto, esta refatoração visa converter uma interface funcional em uma interface sem métodos *default*, transformando-os em métodos abstratos. A mecânica é descrita da seguinte forma:

1. Aplique a refatoração *Convert Default Method to Abstract Interface Method* em cada um dos métodos *default*.

2. Aplique a refatoracão *Convert Lambda Expression to Functional Interface Instance* nas instâncias que utilizam expressões lambda.

3. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração, a interface *Listener* possuía apenas um método abstrato. Após a refatoração, o método *isSelected* foi transformado em abstrato.

```
1   // Antes
2   public interface Listener {
3      public void select(Object object);
4      public default Boolean isSelected(){
5         return null;
6      }
7   }
8
9   // Depois
10  public interface Listener {
11     public void select(Object object);
12     public Boolean isSelected();
13  }
```

Convert Parallel For to Enhanced For

Você esta usando um método de uma coleção para percorrer a própria coleção de forma paralela e precisa realizar essa iteração de forma externa e sequencial.

◇ ◇ ◇

Portanto, converta esse método em uma construção for each tradicional.

Uma das motivações para o uso das expressões lambda é a facilidade em se implementar instruções de forma paralela através de recursos de bibliotecas. Porém, pode haver a necessidade em se executar um conjunto de instruções de forma sequencial. Esta refatoração transforma uma estrutura de repetição com execução paralela, a qual utiliza recursos da API *Stream*, em uma construção *for each* sequencial tradicional. A mecânica é descrita da seguinte forma:

1. Substitua a construção lambda que utiliza recursos da API Stream (*parallelStream*), por uma construção *for each* tradicional.

2. Mova o corpo da expressão lambda do método *forEach* (conteúdo que se encontra após o *token*) para dentro do escopo da estrutura de repetição.

3. Elimine a implementação da estrutura de repetição antiga.

4. Compile o código e teste.

O exemplo a seguir apresenta o código antes e depois de aplicar a refatoração. Antes da refatoração, uma estrutura de repetição é executada de forma paralela, percorrendo a coleção *people* e invocando o método *save*. Após a refatoração, a execução passa a ser de forma sequencial.

```
1  // Antes
2  Collection<Person> people = new ArrayList<Person>();
3  people.parallelStream().forEach(person -> save(person));
4
5  // Depois
6  Collection<Person> people = new ArrayList<Person>();
7  for (Person person : people)
8     save(person);
```

3.2 Considerações Finais

Este capítulo apresentou um catálogo de refatorações em virtude das novas funcionalidades adicionadas à linguagem Java. Foram descritas refatorações que representassem situações de código, que pudessem ser transformadas em implementações com a aplicação de expressões lambda. Para esse objetivo, um conjunto de vinte refatorações foi proposto, sendo que as dez primeiras retratam as transformações diretas, voltadas para o uso de expressões lambda. As demais refatorações foram suas respectivas transformações inversas. O próximo capítulo descreve a extensão do *framework* AOPJungle, utilizado para a aplicação do estudo de caso comentado no Capítulo 5.

4 IMPLEMENTAÇÃO

Faveri (2013) desenvolveu um projeto chamado *framework* AOPJungle com o objetivo de extrair informações sobre código fonte e permitir seu acesso por meio de uma linguagem específica de domínio ou por estruturas de dados disponíveis no *framework*. Visando a continuidade deste trabalho, desenvolvido no Departamento de Linguagens de Programação e Bancos de Dados da UFSM[7], suas funcionalidades foram estendidas para atender às necessidades desta dissertação.

Este capítulo descreve detalhes sobre a implementação e a extensão do *framework* AOPJungle, desenvolvido para a plataforma Eclipse, e possui o objetivo de fornecer dados sobre o código fonte de sistemas orientados a objetos e aspectos. Esta extensão visa adicionar ao *framework* recursos para a coleta de informações sobre expressões lambda e outras funcionalidades da linguagem utilizadas na criação do catálogo de refatorações do Capítulo 3, as quais não faziam parte do seu modelo de dados original. Para realizar essa implementação, foram utilizadas as ferramentas JDT - *Java Development Tools* e AJDT - *AspectJ Development Tools*, responsáveis por fornecer os mecanismos necessários para o acesso às informações de unidades de compilação escritas nas linguagens Java e AspectJ. As seções seguintes apresentam detalhes sobre a extração de informação, a definição e a adequação do metamodelo, e a adaptação do *framework* AOPJungle para o suporte do catálogo de refatorações.

4.1 Extração

O processo de extração de informação do *framework* tem como foco principal os dados disponíveis na AST (*Abstract Syntax Tree*) de cada unidade de compilação. Para isso, o AOPJungle implementa o padrão de projeto *Visitor* (HELM et al., 2002)[8], adotado pelo plug-in AJDT. Sua implementação foi realizada através da extensão da classe *AjASTVisitor* e da sobrescrita dos métodos *visit*, conforme mostra a Listagem 4.1. Cada método *visit* permite o acesso a um nó específico da AST.

Para cada método *visit* implementado, o AOPJungle armazena as informações coletadas em um metamodelo orientado a objetos, visando uma melhor organização dos dados e, através

[7] UFSM - Universidade Federal de Santa Maria.

[8] O *Visitor* é um padrão de projeto comportamental, que permite criar uma operação a ser realizada em elementos de uma estrutura de objetos, sem mudar os elementos das classes os quais ele opera (HELM et al., 2002).

do metamodelo, permitir o acesso por meio de consultas de forma direta e simplificada.

Listagem 4.1 – Sobrescrita do método *visit*

```
1 @Override
2 public boolean visit(AnonymousClassDeclaration node) {
3     AOJAnonymousClassDeclaration aojNode = new AOJAnonymousClassDeclaration
          (node,cUnit);
4     (...)
5     return super.visit(node);
6 }
```

4.2 Definição e adequação do Metamodelo

O metamodelo do AOPJungle é uma estrutura orientada a objetos responsável por armazenar e organizar metadados extraídos de ASTs e também informações obtidas dos projetos do usuário contidos no espaço de trabalho do Eclipse, através dos plug-ins JDT e AJDT.

Todas as classes do modelo herdam, diretamente ou indiretamente, da classe abstrata *AOJMember*, que também possui uma herança com a classe *ASTElement* (Figura 4.1), responsável por armazenar informações específicas das ASTs e que, por sua vez, também possui uma herança com a classe *AOJProgramElement*, a qual é responsável pela navegabilidade de cada elemento do modelo através de sua associação recursiva.

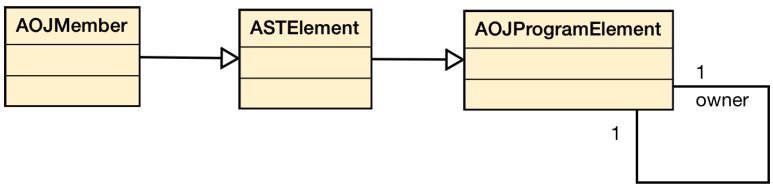

Figura 4.1 – Estrutura Básica do Metamodelo

A Figura 4.2 apresenta uma versão simplificada do metamodelo do AOPJungle, com a finalidade de identificar o que foi modificado e melhorado no modelo. As classes em laranja não sofreram nenhuma modificação, enquanto as demais foram adicionadas ou apenas adaptadas para armazenar mais informações sobre instruções de código de projetos OO e OA.

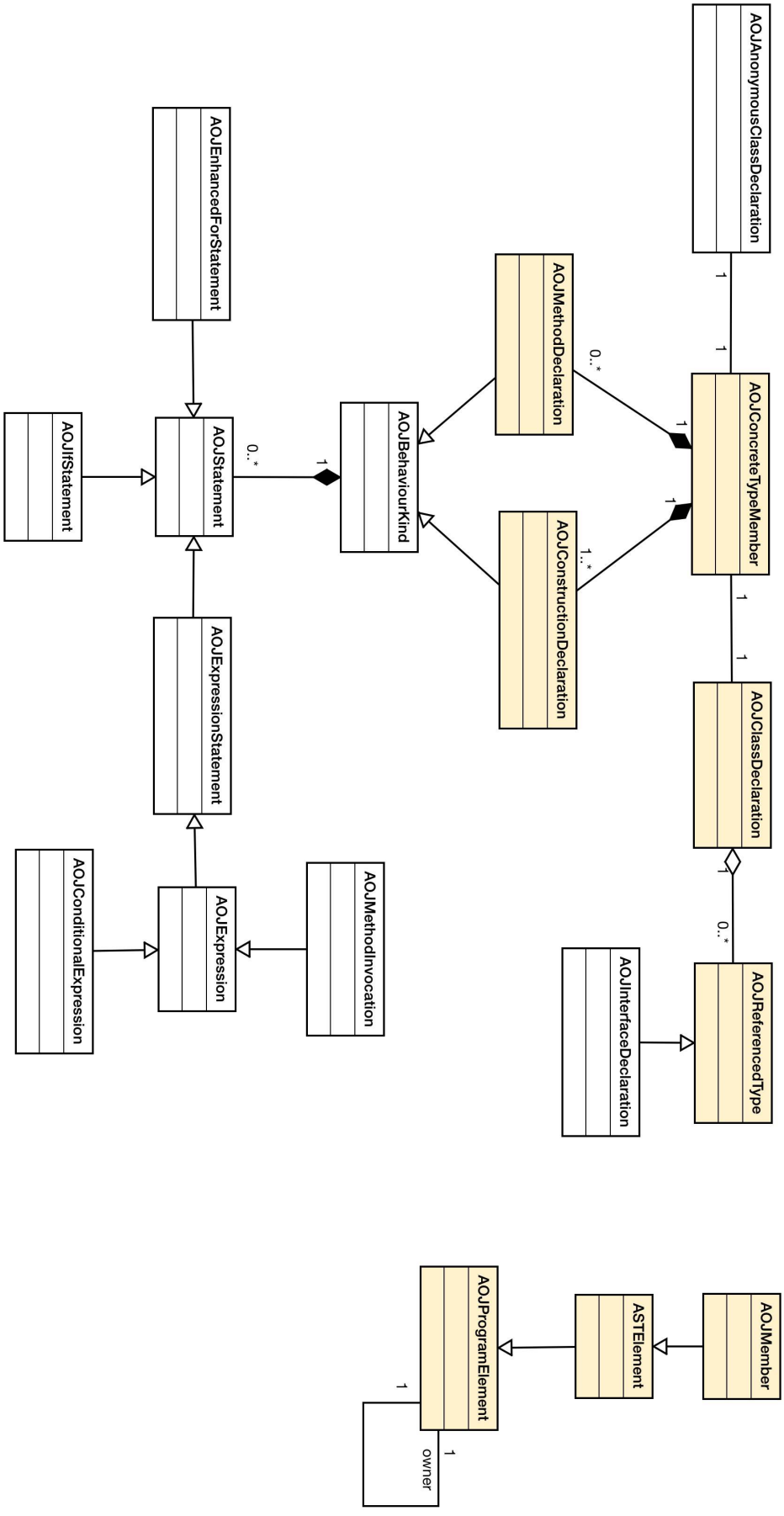

Figura 4.2 – Metamodelo do AOPJungle

A Figura 4.3 apresenta uma parte da estrutura de classes do metamodelo. Tais classes são a base para o armazenamento de uma unidade de compilação, sendo que as classes em cor branca foram modificadas para armazenar informações sobre expressões lambda.

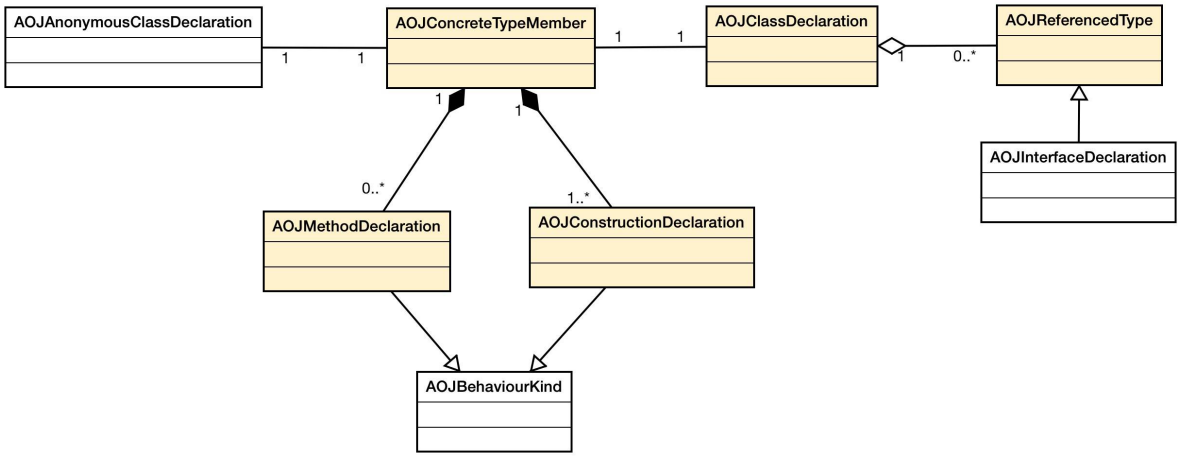

Figura 4.3 – Estrutura Base do Metamodelo

Inicialmente o metamodelo do AOPJungle não possuía classes que representavam instrução de código contido em métodos e construtores, pois não eram relevantes para a pesquisa de Faveri (2013). No entanto, para a identificação das refatorações descritas no Capítulo 3, modificações estruturais no metamodelo foram necessárias, acrescentando algumas classes para a representação das seguintes instruções de código: laço de repetição *ForEach* (*AOJEnhancedForStatement*), estrutura condicional (*AOJIfStatement*), e alguns tipos de expressões, como a invocação de um método (*AOJMethodInvocation*). Estas novas classes podem ser visualizadas na segunda parte do modelo (Figura 4.4).

A classe *AOJBehaviourKind* representa o comportamento em comum dos métodos e construtores, sendo a responsável por armazenar informações sobre instruções de código contidas neles. Na Figura 4.4, pode-se observar a composição entre as classes *AOJBehaviourKind* e *AOJStatement*.

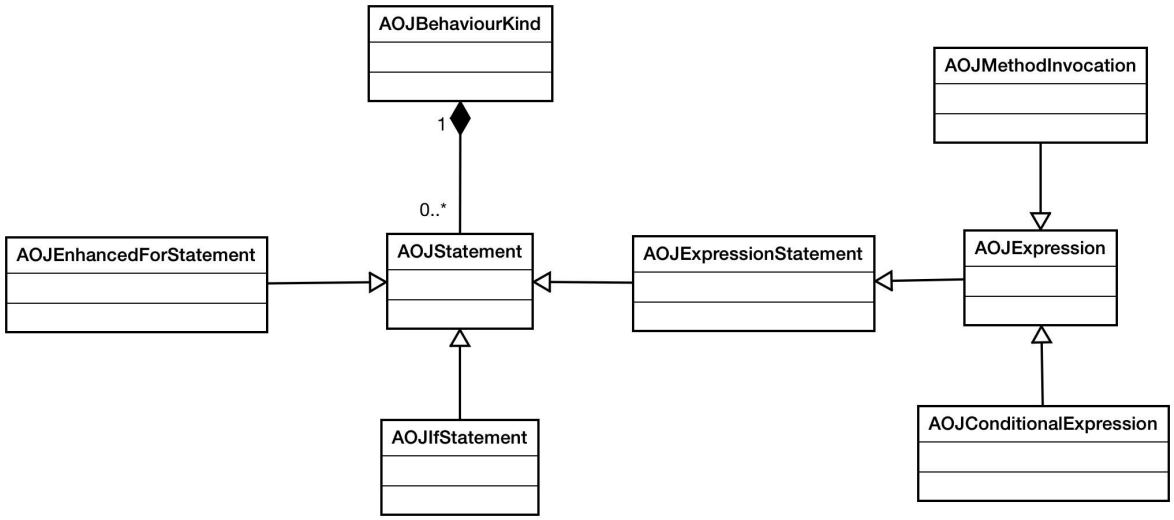

Figura 4.4 – Estrutura do Metamodelo para o Armazenamento de Instruções de Código

A próxima seção apresenta mais detalhes sobre a estrutura de classes do metamodelo e implementações relacionadas às refatorações descritas no Capítulo 3.

4.3 Adaptação do *Framework* AOPJungle para o suporte do Catálogo de Refatorações

O modelo de dados elaborado para o AOPJungle descreve uma estrutura sobre projetos, pacotes, classes, métodos, aspectos, etc., como visto na seção anterior. No entanto, uma adaptação do modelo de dados original foi necessária para dar o suporte ao catálogo de refatorações proposto, pois o mesmo não possuía a estrutura para representar as instruções de código contidas em métodos, por exemplo. As seções abaixo relacionam algumas refatorações com as modificações e adições feitas no AOPJungle, visando a busca de oportunidades de refatoração do catálogo apresentado no Capítulo 3.

4.3.1 *Convert Functional Interface Instance to Lambda Expression*

Identificar se uma instância de interface é do tipo funcional ou não, é a primeira e a principal etapa para a identificação desta refatoração, pois toda expressão lambda está relacionada a uma interface funcional. A classe que pode representar uma instância de uma interface funcional no AOPJungle é a *AOJAnonymousClassDeclaration*. Em sua estrutura foi adicionado um atributo, *functionalInterface* do tipo *Boolean*, o qual identifica se é uma interface funcional.

Para obter esta informação, é preciso verificar se as informações sobre a classe anônima, extraída da AST, são provenientes de uma instância de uma interface e, também, se possui ape-

nas um método abstrato em sua estrutura, requisitos estes que categorizam uma interface funcional. Esta informação é obtida através do construtor da classe *AOJAnonymousClassDeclaration*, conforme mostrado na Listagem 4.2.

Listagem 4.2 – Construtor da Classe AOJAnonymousClassDeclaration

```
1  public AOJAnonymousClassDeclaration(ASTNode node, AOJProgramElement owner){
2      super(node, owner);
3      AnonymousClassDeclaration n = (AnonymousClassDeclaration) node;
4      String simpleName = getSimpleName(node.getParent().toString());
5      (...)
6      members = new AOJConcreteTypeMember();
7      Class<?> clazz = getObjectClass();
8      if (clazz != null){
9          setInterface(clazz.isInterface());
10         setFunctionalInterface(verifyingFunctionalInterface(clazz));
11     }
12 }
```

No código descrito na Linha 7, Listagem 4.2, o AOPJungle faz a instância, por reflexão, da classe anônima. Nas linhas 9 e 10, é verificado se o objeto *clazz* é uma interface e, depois, se é uma interface funcional. Para verificar se uma classe anônima (*AnonymousClassDeclaration*) é realmente uma instância de uma interface, foi necessário utilizar reflexão computacional para obter esta informação.

O processo de identificação de uma interface, Listagem 4.3, inicia-se com a busca do seu nome qualificado (pacote + nome da classe) através de seu nome, fornecido pelo método *getInstanceTypeName* do AOPJungle, linha 2. A identificação do nome qualificado pode ser realizada em até três etapas: (i) primeiramente, é verificado se existe um *import* na própria unidade de compilação; (ii) caso não exista, é feita uma análise por reflexão, verificando se o nome simples pertence a algum pacote do Java que não precisa ser importado. Ex: Classes do pacote *java.lang*; (iii) assume-se que esta classe pertence ao mesmo pacote da unidade de compilação.

Listagem 4.3 – Método *isInterface*

```
1  public Boolean isInterface(){
2    String fullName = getFullyQualifiedNameFromImport(getInstanceTypeName());
3    try {
4      return Class.forName(fullName).isInterface() ? true : false;
5    } catch (ClassNotFoundException e) {
6      Class<?> c = AOJClassLoader.forName(fullName);
7      if (c != null)
8        return true;
9    }
10   return false;
11 }
```

Após obter o nome completo da classe anônima, novamente por reflexão, é feita uma instância desta classe anônima através do método *forName*, linha 4, e depois é verificado se é uma interface através do método *isInterface*. Uma exceção (*ClassNotFoundException*) significa que esta classe não existe, sendo necessário realizar uma nova verificação através da classe *AOJClassLoader*.

Listagem 4.4 – Classe AOJClassLoader

```
1  public class AOJClassLoader {
2      private static List<URLClassLoader> loaders = null;
3
4      public static Class<?> forName(String fullyQualifiedName){
5          getLoaders();
6          for (URLClassLoader loader : getLoaders()) {
7              try {
8                  Class<?> c = loader.loadClass(fullyQualifiedName);
9                  return c;
10             } catch (ClassNotFoundException e) {}
11         }
12         return null;
13     }
14
15     private static List<URLClassLoader> getLoaders() { (...) }
16     (...)
17 }
```

A classe *AOJClassLoader* (Listagem 4.4) foi criada para permitir a instância por reflexão de classes ou interfaces que não estão em execução. Seu método *forName*, linha 4, possui a mesma finalidade do mesmo método da classe *Class*, entretanto, se seu retorno for nulo, significa que não foi possível realizar a instância.

Após descobrir se uma classe anônima é uma interface, é iniciado o processo de validação de uma interface funcional. Basicamente, é verificado se uma interface possui apenas um método abstrato em sua estrutura, ou seja, um método que não possui implementação. Entretanto, ao se usar reflexão, a lista de métodos obtida não identifica se um determinado método é abstrato ou não. Então, uma análise é feita em todos os métodos, eliminando os não abstratos, conforme mostra a linha 7 da Listagem 4.5.

Todo objeto Java possui uma herança da classe *Object*. Dessa forma, os métodos: *equals* e *toString*, por exemplo, podem ser sobrescritos em uma implementação de uma interface. Portanto, não foram contabilizados como métodos válidos. Também não foram contabilizados os métodos estáticos, pois na versão 8 do Java podem existir implementações estáticas em interfaces.

Listagem 4.5 – Verificando se uma interface é funcional

```
1  private boolean verifyingFunctionalInterface(Class clazz) {
2     int  numberOfMethods = 0;
3     if (! isInterface() )
4        return false;
5     if (clazz.getMethods().length == 0)
6        return false;
7     for (Method m : clazz.getMethods())
8        if (m.getName().equalsIgnoreCase("equals") && m.getReturnType().
            getName().equalsIgnoreCase("boolean"))
9        else if (m.getName().equalsIgnoreCase("toString") && m.getReturnType
            ().getName().equalsIgnoreCase("String"))
10       else if (...) // outros metodos da classe Object
11       else if (Modifier.isStatic(m.getModifiers()))
12       else
13          numberOfMethods++;
14    return numberOfMethods == 1;
15 }
```

4.3.2 *Convert Enhanced For to Lambda Enhanced For*

Uma construção *for each* é uma das diversas estruturas de repetição em Java, sendo identificada pela AST como um *for* aprimorado (*Enhanced For*). O foco dessa implementação é descobrir locais do código que possuem uma estrutura de repetição do tipo *for each* e adicionar essa instrução de código ao metamodelo. A Listagem 4.6 apresenta uma sobrescrita do método *visit* que, através da instância da classe *AOJEnhancedForStatement*, obtém as informações desta estrutura de repetição e, por fim, o objeto que a representa é adicionado a uma lista de *Statements* do objeto *behaviourKind*, linha 6.

Listagem 4.6 – Visit Enhanced For

```
1  public boolean visit(EnhancedForStatement node) {
2     AOJEnhancedForStatement aojNode = new AOJEnhancedForStatement(node, (
         AOJProgramElement)getLastMemberFromStack());
3     AOJBehaviourKind behaviourKind = getAOJBehaviourKind(
         getLastMemberFromStack());
4     if (behaviourKind != null)
5        behaviourKind.getStatements().add(aojNode);
6     elementStack.push(aojNode);
7     return super.visit(node);
8  }
```

A classe *AOJBehaviourKind* (Listagem 4.7) é responsável por armazenar algumas informações como: código, métricas, parâmetros, exceções e agora alguns tipos de instruções de código (*AOJStatment*), linha 11. Três classes estendem *AOJBehaviourKind*. São elas: *AOJMethodDeclaration*, *AOJConstructorDeclaration* e *AOJAdviceDeclaration*.

Listagem 4.7 – Classe AOJBehaviourKind

```
1 public abstract class AOJBehaviourKind extends AOJMember implements ... {
2    private List<AOJParameter> parameters;
3    private List<AOJException> thrownExceptions;
4    private AOJBehaviorMetrics metrics;
5    private String code;
6    private List<AOJStatement> statements;
7    (...)
8 }
```

Uma estrutura *for each* pode ser encontrada em locais como: métodos, construtores presentes em classes, classes anônimas e aspectos, e também em *advices* de um aspecto. Esta implementação foi construída para identificar um tipo de laço de repetição (*for each*), contudo sua estrutura pode ser estendida para todos os outros tipos de laços existentes na linguagem Java.

4.3.3 *Convert Collections.sort to sort*

A classe *ExpressionStatement* é a estrutura responsável por representar as expressões da AST, algumas dessas expressões podem ser: *ConditionalExpression*, *Name*, *BooleanLiteral*, *MethodInvocation*, entre outras. Para esta refatoração, foi adicionada ao metamodelo a classe *AOJExpressionStatement*, responsável por armazenar todo tipo de expressão, inclusive *MethodInvocation*, que é a expressão utilizada para identificar esta refatoração.

Listagem 4.8 – Visit ExpressionStatement

```
1 public boolean visit(ExpressionStatement node) {
2    AOJExpressionStatement expression = new AOJExpressionStatement(node, (
        AOJProgramElement)getLastMemberFromStack());
3    AOJBehaviourKind behaviourKind = getAOJBehaviourKind(
        getLastMemberFromStack());
4    if (behaviourKind != null)
5       behaviourKind.getStatements().add(expression);
6    elementStack.push(expression);
7    return super.visit(node);
8 }
```

O método *visit*, Listagem 4.8, é responsável por obter as informações referentes às expressões encontradas no código. Na linha 2, pode-se observar a instância da classe *AOJExpressionStatement*, tento como um de seus parâmetros o objeto *node*, referente à classe *ExpressionStatement* da AST. Em seu construtor, mostrado na Listagem 4.9, é verificado qual o tipo da expressão, linha 5, e é adicionada a expressão encontrada no atributo *aojExpression* .

Listagem 4.9 – Construtor AOJExpressionStatement

```
1  public AOJExpressionStatement(ASTNode node, AOJProgramElement owner) {
2     super(node, owner);
3     ExpressionStatement e = (ExpressionStatement)node;
4     if (e.getExpression() instanceof MethodInvocation)
5       this.aojExpression = new AOJMethodInvocation(owner,e.getExpression());
6     else if (e.getExpression() instanceof ConditionalExpression)
7       this.aojExpression = new AOJConditionalExpression(owner, e.
            getExpression());
8     else //  expressao default
9       this.aojExpression = new AOJExpression(owner) {    };
10 }
```

A classe *AOJMethodInvocation* é a responsável por armazenar informações sobre as chamadas de método (*MethodInvocation*), necessárias para identificar a refatoração *Convert Collections.sort to sort*. As demais expressões foram adicionadas ao modelo como uma expressão padrão (*AOJExpression*), linha 10, com a exceção da expressão *ConditionalExpression*.

4.3.4 *Convert Enhanced For with If to Lambda Filter*

As adaptações realizadas no AOPJungle para esta refatoração são complementares às implementadas na refatoração *Enclose Enhanced For*. Como especificado no Capítulo 3, um *Filter* representa um filtro em uma coleção, ou seja, um filtro é uma estrutura condicional (*if*) dentro de um laço de repetição. Esta implementação adiciona uma estrutura condicional ao metamodelo, a fim de utilizá-la para identificar esta refatoração. A Listagem 4.10 descreve o método *visit* responsável pela seleção dos dados de um *IfStatement*.

Listagem 4.10 – Visit AOJIfStatement

```
1  public boolean visit(IfStatement node) {
2      AOJIfStatement aojNode = new AOJIfStatement(node, (AOJProgramElement)
           getLastMemberFromStack());
3      AOJBehaviourKind behaviourKind = getAOJBehaviourKind(
           getLastMemberFromStack());
4      if (behaviourKind != null)
5          behaviourKind.getStatements().add(aojNode);
6      elementStack.push(aojNode);
7      return super.visit(node);
8  }
```

4.3.5 *Convert Functional Interface to Default Functional Interface*

Uma expressão lambda é uma sintaxe alternativa para instanciar uma interface funcional. Esse tipo de sintaxe não leva em consideração nem o nome da interface e nem o nome do método. No entanto, suas informações relevantes são o tipo de retorno e os argumentos do

método (quantidade e tipo).

A linguagem Java, a partir da versão 8, disponibiliza um conjunto de interfaces funcionais padrões a serem utilizadas, disponível no pacote *java.util.funciton*. A Listagem 4.11 apresenta uma dessas interfaces funcionais.

Listagem 4.11 – Interface Funcional Consumer

```
1 public interface Consumer<T> {
2     void accept(T t);
3 }
```

Essa implementação tem como objetivo adicionar um atributo, *functionalInterface*, na classe *AOJInterfaceDeclaration* (Listagem 4.12), identificando se esta interface é do tipo funcional. Essa informação possibilita encontrar todas as interfaces funcionais existentes no código, permitindo verificar se alguma interface pode ser substituída por uma das novas interfaces funcionais disponíveis na linguagem. Para essa análise, serão comparadas informações como o tipo de retorno do método e seus parâmetros.

Listagem 4.12 – Interface AOJInterfaceDeclaration

```
1  public class AOJInterfaceDeclaration extends AOJTypeDeclaration {
2      (...)
3      private Boolean functionalInterface = null;
4      public Boolean isFunctionalInterface() {
5          if (functionalInterface == null) {
6              Class c = AOJClassLoader.forName(this.getFullQualifiedName());
7              functionalInterface = (c != null) ? c.getMethods().length == 1 :
                   false;
8          }
9          return  functionalInterface;
10     }
11     (...)
12 }
```

A diferença da implementação do método *isFunctionalInterface*, linha 5, com a realizada na identificação de uma classe anônima, é que esta verificação é feita apenas para as interfaces disponíveis no código do projeto.

4.4 Considerações Finais

Este capítulo apresentou a extensão realizada no *framework* AOPJungle, tendo como principais objetivos a adição e a modificação de funcionalidades voltadas para a identificação das dez primeiras refatorações descritas no Capítulo 3. A seção 4.3 relacionou algumas das refatorações propostas com a implementação realizada no *framework*, mostrando a importância de cada alteração do metamodelo e as adições das novas funcionalidades.

Para as refatorações *Convert Abstract Interface Method to Default Method* e *Convert Inter-Type Method Declaration to Default Interface Method*, não foram preciso realizar modificações no *framework* AOPJungle, sendo suficientes as informações disponíveis no seu metamodelo para a busca de oportunidades de refatoração. As demais refatorações (*Extract Method Reference*, *Convert Interface to Functional Interface* e *Convert Enhanced For to Parallel For*) estão diretamente relacionadas às implementações construídas para as refatorações *Convert Functional Interface Instance to Lambda Expression*, *Convert Abstract Interface Method to Default Method* e *Convert Enhanced For to Lambda Enhanced For*. Logo, não houve a necessidade da descrição de suas implementações. Um estudo de caso utilizando essas refatorações e os recursos do AOPJungle são descritos no próximo capítulo.

5 ESTUDO DE CASO

Um processo de refatoração é composto pelas seguintes etapas: (i) identificar onde aplicar refatorações; (ii) avaliar os efeitos da refatoração na qualidade do software; (iii) manter a consistência dos trechos de código estruturados pela refatoração; e (iv) garantir que a aplicação da refatoração preserve o comportamento externo observável do sistema de software. Em particular, a busca por oportunidades de refatoração consiste em procurar locais no código nos quais uma potencial refatoração pode ser aplicada.

Este estudo de caso envolveu a busca por oportunidades de refatoração onde as expressões lambda pudessem ser aplicadas, de acordo com o catálogo de refatorações definido no Capítulo 3. Para tanto, seis projetos de código aberto, listados na Tabela 5.1, foram utilizados, sendo o menor com 4.262 linhas de código e o maior com 451.674.

Tabela 5.1 – Projetos Usados no Estudo de Caso

Projeto	**Descrição**	**Versão e LOC**
AOPJungle	AOPJungle framework	1.0
www.ufsm.br/ppgi/		10.694
Apache-Ant	Java application builder	1.9.3
ant.apache.org		106.246
Apache-log4j	Log services	1.2.17
logging.apache.org		21.050
Aspect Design Patterns	GoF Pattern Implementation	1.11
www.cs.ubc.ca/labs/spl/projects/aodps.html		4.262
EclipseLink	Java Persistence API	2.5.1
www.eclipse.org/eclipselink/downloads/		451.674
MySQL Connector/J	Java MySQL DB driver	5.1.30
dev.mysql.com/downloads/connector/j/		96.695

Para realizar a busca por oportunidades, foram utilizados dois plug-ins construídos na plataforma Eclipse: (i) o plug-in AOPJungle, responsável por selecionar meta-informações sobre projetos orientado a objetos e aspectos, apresentado no capítulo anterior; e (ii) o plug-in λRefactoring, que foi construído para localizar e quantificar cada refatoração do catálogo lambda, através das informações disponibilizadas pelo AOPJungle.

5.1 Plug-in λRefactoring

O plug-in λRefactoring utiliza recursos da plataforma Eclipse e de plug-ins (JDT, AJDT e AOPJungle) para sua implementação. Seu funcionamento é baseado na aplicação de filtros específicos para cada refatoração. De forma geral, o λRefactoring percorre e faz a análise em construtores e métodos de classes, de classes anônimas e de aspectos, incluindo, neste último caso, os *advices*.

Para cada refatoração analisada pelo λRefactoring, foi definido um processo de execução contendo três principais etapas:

1. **Seleção de Classes, Aspectos e Interfaces**. Essa etapa refere-se a seleção de estruturas (classes, aspectos e interfaces) em que o λRefactoring realizará a busca por oportunidades. Para cada tipo de oportunidade de refatoração, é feita uma seleção de estruturas em que tal oportunidade pode ser encontrada. O código da Listagem 5.1 mostra como é feita a seleção das estruturas através da execução do método *run*. Primeiramente são selecionados todos os projetos contidos no *workspace* do Eclipse (linha 2). Em seguida, para todas as classes (linha 3) e todos os aspectos (linha 5) são feitas chamadas para o método *findRefactorings* (linhas 4 e 6).

Listagem 5.1 – Método *run()*

```
1 public void run() {
2   for (AOJProject project : AOJWorkspace.getInstance().getProjects()){
3     for (AOJTypeDeclaration clazz : project.getClasses())
4       findRefactorings(clazz);
5     for (AOJTypeDeclaration aspect : project.getAspects())
6       findRefactorings(aspect);
7   }
8 }
```

2. **Busca por Oportunidades de Refatoração**. A segunda etapa refere-se à identificação da oportunidade. A Listagem 5.2 apresenta o método *findRefactorings*, responsável por encontrar uma oportunidade específica em um trecho de código. Este exemplo executa uma busca por classes anônimas em métodos e construtores de classes. Para cada coleção de classe anônima disponível em um método (linha 2) ou em um *statement* (linha 4), é invocado o método *print* (linhas 3 e 6) para a análise final e registro das informações.

Listagem 5.2 – Método *findRefactorings()*

```
1 private void findRefactorings(AOJTypeDeclaration clazz) {
2    for (AOJMethodDeclaration method : ((AOJClassDeclaration)clazz).
        getMembers().getMethods()) {
3       print(filterAnonymous(method.getAnonymousClasses()), clazz, "
           Method");
4       for(AOJStatement statement : method.getStatements())
5          if (statement instanceof AOJExpressionStatement)
6             print(filterAnonymous( ((AOJExpressionStatement) statement
                 ).getAnonymousClasses()), clazz, "Method");
7    }
8    (...)
9 }
```

3. **Quantificação e Impressão do Resultado**. A terceira etapa é destinada, principalmente, à exibição das informações. O código do método abaixo, Listagem 5.3, registra cada oportunidade encontrada (linha 5), neste caso, as interfaces funcionais, as quais serão exibidas ao final da execução do plug-in.

Listagem 5.3 – Método *print()*

```
1 private void print(List<AOJContainer> listContainer,
     AOJTypeDeclaration typeDeclaration, String type){
2    for (AOJContainer container : listContainer)
3       if (((AOJAnonymousClassDeclaration)container).
           isFunctionalInterface()){
4          String message = String.format( "%s: %s - %s: %s",type, ((
              AOJAnonymousClassDeclaration)container).getName(), ((
              AOJAnonymousClassDeclaration)container).
              getInstanceTypeName(), ((AOJAnonymousClassDeclaration)
              container).getCode() );
5          registerReport(message, typeDeclaration);
6       }
7 }
```

A Figura 5.1 apresenta o resultado da execução do plug-in sobre o projeto *Apache-Ant*. Nos resultados encontrados, as oportunidades de refatorações identificadas são sinalizadas (em vermelho), juntamente com a quantidade de ocorrências. Cada refatoração do catálogo é apresentada em forma de árvore, permitindo a navegação até os nós folha, selecionados na imagem, os quais identificam os locais específicos onde a oportunidade de refatoração se encontra.

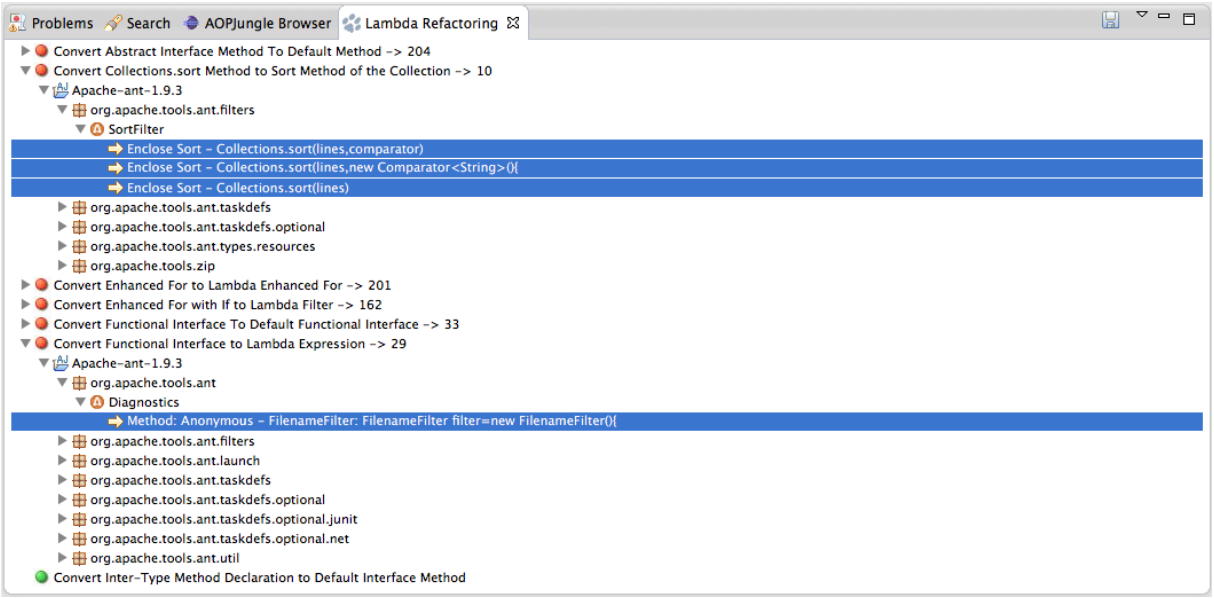

Figura 5.1 – Resultado da Execução do Plug-in λRefactoring

As seções seguintes apresentam detalhes específicos de implementação e os resultados da execução do λRefactoring usando as refatorações descritas no catálogo do Capitulo 3, em exceção das refatorações inversas. Dessa forma, as refatorações avaliadas são voltadas para a aplicação de expressões lambda.

5.2 Convert Functional Interface Instance to Lambda Expression (R1)

Esta refatoração consiste em converter classes anônimas em uma expressão lambda. Logo, buscou-se localizar todas as instâncias de classes anônimas implementadas a partir de interfaces funcionais, as quais podem ser encontradas em locais como: métodos, construtores e *advices*.

5.2.1 Implementação

Na primeira etapa do processo de execução, foram selecionadas todas as classes e aspectos para a análise. Em seguida, buscou-se encontrar as classes anônimas disponíveis em métodos, construtores e *advices*, como apresentado na Listagem 5.2. Para tanto, apenas as classes anônimas implementadas a partir de uma interface foram avaliadas, como mostrado na Listagem 5.4.

Listagem 5.4 – Método *filterAnonymous()*

```
1  private List<AOJContainer> filterAnonymous(List<AOJContainer> list) {
2     List<AOJContainer> result = new ArrayList<AOJContainer>();
3     for (AOJContainer container : list)
4        if (container instanceof AOJAnonymousClassDeclaration)
5           result.add(container);
6     return result;
7  }
```

A última etapa do processo é destinada à quantificação e à exibição dos resultados. No entanto, além do registro desta oportunidade (linhas 3 e 4 da Listagem 5.5), esta etapa verificou os métodos das próprias classes anônimas (linhas 7 e 8) de forma recursiva.

Listagem 5.5 – Método *print()*

```
1  private void print(List<AOJContainer> listContainer, AOJTypeDeclaration
       typeDeclaration, String type){
2     for (AOJContainer container : listContainer){
3        if (((AOJAnonymousClassDeclaration)container).isFunctionalInterface()
            ){
4           String message = String.format( "%s: %s - %s: %s",type, ((
               AOJAnonymousClassDeclaration)container).getName(), ((
               AOJAnonymousClassDeclaration)container).getInstanceTypeName(),
               ((AOJAnonymousClassDeclaration)container).getCode() );
5           registerReport(message, typeDeclaration);
6        }
7        for ( AOJMethodDeclaration method : ((AOJAnonymousClassDeclaration)
            container).getMembers().getMethods())
8           print(filterAnonymous(method.getAnonymousClasses()),
               typeDeclaration, type);
9     }
10 }
```

5.2.2 Resultados

Todos os projetos tiveram essa refatoração encontrada, embora as ocorrências em determinados projetos como AOPJungle e MySQL Connector/J tenham sido baixas. Um total de 120 oportunidades de refatoração foi encontrado em todos os seis projetos. Este número representa pouca utilização de instâncias de interfaces funcionais, porém com o possível aumento do uso de expressões lambda em novos projetos de software, tal número pode aumentar no futuro.

Na Figura 5.2, pode-se observar que o projeto *Apache-Log4J* obteve o maior número de oportunidades encontradas pelo plug-in, com um total de 46 (38%) oportunidades de refatoração. A maioria das oportunidades encontradas é composta de instâncias das interfaces funcionais *ActionListener* (pacote *java.awt.event*) e *Runnable* (pacote *java.lang*).

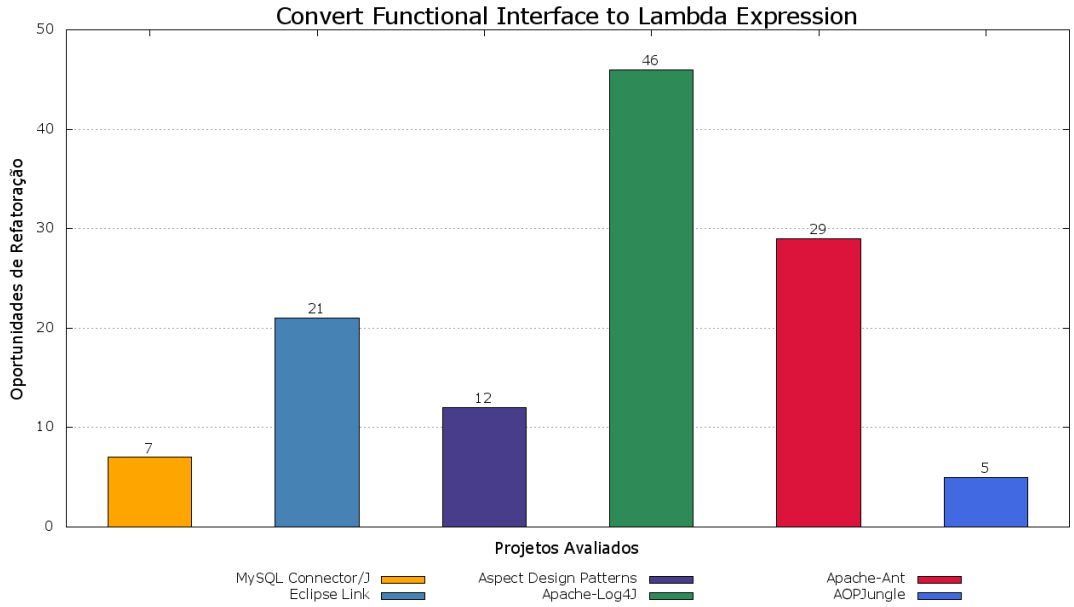

Figura 5.2 – R1 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

5.3 Convert Enhanced For to Lambda Enhanced For (R2)

Para esta refatoração, buscou-se localizar todos os laços de repetição do tipo *for Each*. Os locais de código analisados pelo plug-in foram métodos, construtores e *advices*.

5.3.1 Implementação

Como na primeira etapa do processo de busca da oportunidade de refatoração anterior, foram selecionadas todas as classes e aspectos para a análise. Em seguida, buscou-se encontrar *statements* disponíveis em métodos, construtores e *advices*. A Listagem 5.6 mostra parte do código do método *findRefactorings* para a seleção de *statements* em métodos de classes anônimas (linhas 5 e 6) existentes em construtores e *statements* dos próprios construtores (linhas 7 e 8).

Listagem 5.6 – Método *findRefactorings()*

```
1  private void findRefactorings(AOJTypeDeclaration clazz) {
2     for (AOJConstructorDeclaration constructor : ((AOJClassDeclaration)clazz
          ).getMembers().getConstructors() )  {
3        for (AOJContainer container : constructor.getAnonymousClasses())
4           for(AOJMethodDeclaration methodAnonymous : container.getMembers().
                getMethods())
5              for(AOJStatement statement : methodAnonymous.getStatements())
6                 print(statement, clazz);
7        for(AOJStatement statement : constructor.getStatements())
8           print(statement, clazz);
9     } (...)
10 }
```

A terceira etapa do processo faz a análise de cada *statement*, verificando se o mesmo é uma construção *for each* (*AOJEnhancedForStatement*), como mostrado na Listagem 5.7.

Listagem 5.7 – Método *findRefactorings()*

```
1 private void print(AOJStatement statement,  AOJTypeDeclaration
      typeDeclaration) {
2    if (statement instanceof AOJEnhancedForStatement) {
3       String message = String.format("Enhanced For - %s",    ((
           AOJEnhancedForStatement) statement).getCode());
4       registerReport(message, typeDeclaration);
5    }
6 }
```

5.3.2 Resultados

O total de oportunidades de refatorações foi bem expressivo, 2.302, comparado a outras refatorações. Lembrando que esta refatoração não verificou outros tipos de laços de repetição, o que levaria a um número ainda maior.

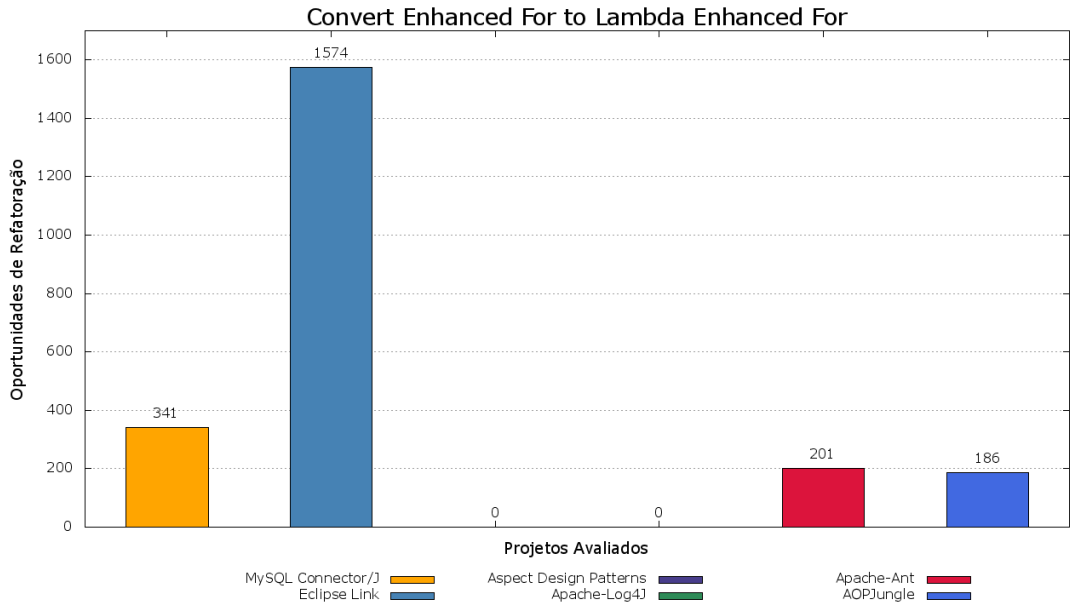

Figura 5.3 – R2 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

Como pode ser visto na Figura 5.3, dois projetos não tiveram oportunidades encontradas. Analisando esta situação, foi observado que se tratava de implementações mais antigas e provavelmente a construção *forEach* não estava disponível na versão utilizada. O projeto que teve o maior índice foi o *Eclipse Link*, com 68% do total de oportunidades de refatorações encontradas, nos seis projetos. Lembrando que o *Eclipse Link* é o maior projeto analisado por este estudo de caso.

5.4 Convert Collections.sort to sort (R3)

Para esta refatoração, o plug-in *λRefactoring* procurou por chamadas do método *sort* da classe *Collections*. Este tipo de instrução de código foi analisado pelo plug-in em locais como métodos, construtores e *advices*.

5.4.1 Implementação

Como nas buscas de oportunidades anteriores, a primeira etapa do processo de execução para a busca dessa refatoração, o *λRefactoring* selecionou todas as classes e aspectos existentes nos projetos. A segunda etapa buscou-se por *statements* em construtores, métodos e *advices*, como mostrado na Listagem 5.6. A terceira etapa verificou se os *statements* selecionados são expressões (AOJExpressionStatement) e se cada expressão é uma invocação de método (linhas 2 e 3). Por fim, foi verificado se o método invocado possui o nome *sort* e se pertence a classe *Collections*.

Listagem 5.8 – Método *print()*

```
1  private void print(AOJStatement statement,  AOJTypeDeclaration
       typeDeclaration) {
2     if (statement instanceof AOJExpressionStatement)
3        if (((AOJExpressionStatement)statement).getAojExpression() instanceof
              AOJMethodInvocation){
4           AOJMethodInvocation m = (AOJMethodInvocation) ((
                AOJExpressionStatement)statement).getAojExpression();
5           if (m.getIdentifier().equals("Collections") && m.getMethodName().
                equals("sort")){
6              String message = String.format( "Collections.sort - %s",((
                   AOJExpressionStatement)statement).getAojExpression().
                   getExpressionCode());
7              registerReport(message, typeDeclaration);
8           }
9        }
10 }
```

5.4.2 Resultados

Esta refatoração teve um baixo índice de oportunidades encontradas, apenas 37, sendo que 65% (Figura 5.4) das oportunidades foram encontradas no projeto *Eclipse Link*. Os projetos *AOPJungle*, *Apache-Ant* e *Aspect Design Patterns* não tiveram nenhuma oportunidade identificada.

Com a facilidade em se aplicar uma ordenação em coleções utilizando Java 8, este cenário poderá melhorar, ainda mais com a utilização de expressões lambda, que atribuiu uma maior

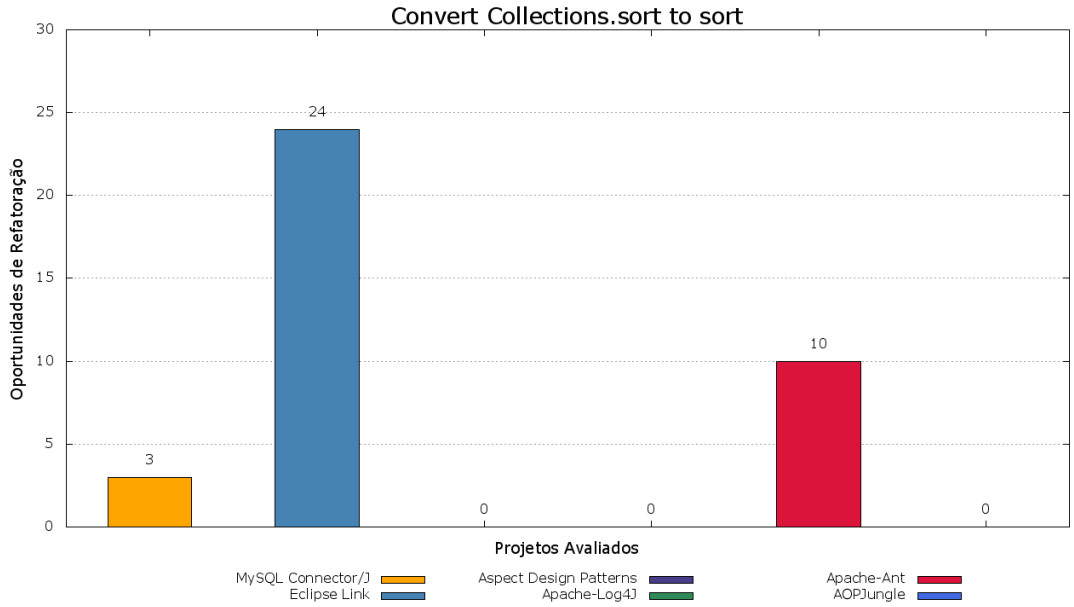

Figura 5.4 – R3 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

praticidade na implementação deste recurso.

5.5 Convert Enhanced For with If to Lambda Filter (R4)

Esta refatoração está diretamente ligada à *Convert Enhanced For to Lambda Enhanced For*, ou seja, além de localizar uma estrutura de repetição *forEach*, o plug-in λRefactoring buscou por estruturas condicionais dentro do escopo do *forEach*.

5.5.1 Implementação

As duas primeiras etapas do processo de execução, para a busca desta refatoração, são idênticas às apresentadas para a refatoração *Convert Enhanced For to Lambda Enhanced For*. A terceira etapa verificou se uma estrutura condicional (*if*) está contida em um laço de repetição *for each*, conforme apresentado nas linhas 2 e 3 da Listagem 5.9.

Listagem 5.9 – Método *print()*

```
1 private void print(AOJStatement statement,  AOJTypeDeclaration
    typeDeclaration) {
2    if (statement instanceof AOJIfStatement){
3       if (((AOJIfStatement)statement).getOwner() instanceof
           AOJEnhancedForStatement){
4          String message = String.format( "Filter - %s",((AOJIfStatement)
              statement).getCode());
5          registerReport(message, typeDeclaration);
6       }
7    }
8 }
```

5.5.2 Resultados

Para esta refatoração, o plug-in analisou construtores, métodos e *advices* e localizou um total de 1.267 oportunidades de refatoração, as quais possuíam pelo menos uma estrutura condicional dentro de um laço de repetição.

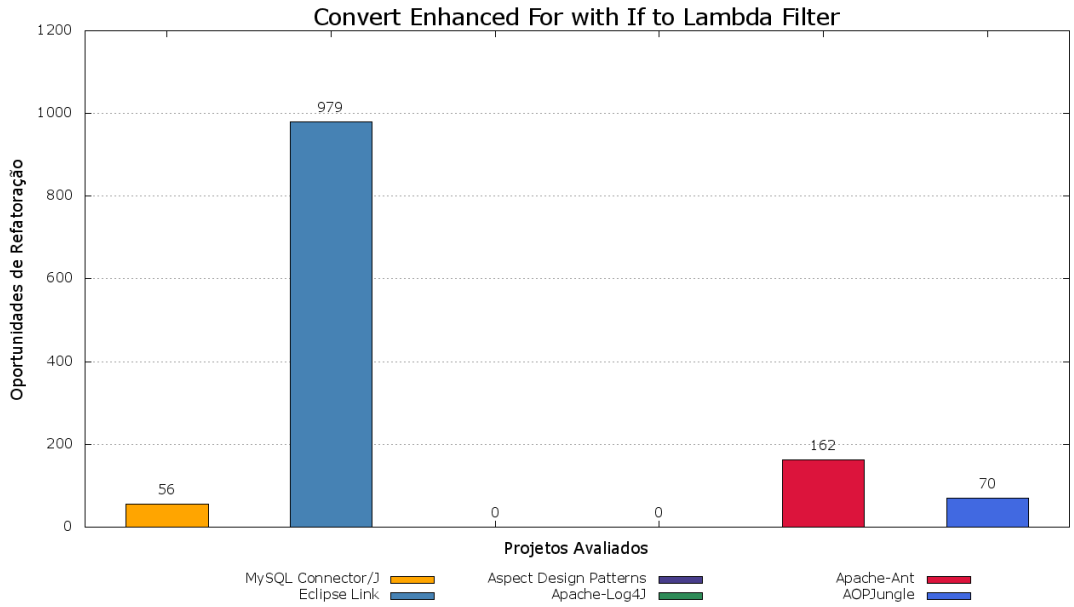

Figura 5.5 – R4 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

Na Figura 5.5 pode-se observar que a maioria das oportunidades localizadas encontram-se no maior projeto analisado (*Eclipse Link*), com um total de 77% das ocorrências. Em dois projetos, *Apache-Log4J* e *Aspect Design Patterns*, não foram encontradas estruturas de repetição, o que ocorreu igualmente para a refatoração *Convert Enhanced For to Lambda Enhanced For*.

Foi observado também que o número de oportunidades encontradas foi um pouco maior que 50%, comparado com a refatoração *Convert Enhanced For to Lambda Enhanced For*. Con-

cluindo que, para cada dois laços de repetição, um possui pelo menos uma estrutura condicional em seu escopo, ou seja, laços de repetição são fortes candidatos à aplicação de filtros lambda.

5.6 Convert Functional Interface to Default Functional Interface (R5)

Uma expressão lambda possui uma sintaxe compacta e sua aplicação é de forma anônima. Isso significa que ao se implementar uma expressão lambda, pouco importa qual a interface funcional ela representa, mas deve-se observar as seguintes características: (i) o tipo de retorno do método; (ii) as quantidades de argumentos do método; e (iii) os tipos desses argumentos. A partir dessas características, este estudo de caso buscou identificar interfaces funcionais existentes nos projetos e compará-las às interfaces funcionais disponíveis no pacote *java.util.function*.

5.6.1 Implementação

Na primeira etapa foram selecionadas todas as interfaces disponíveis nos projetos, como mostrado na Listagem 5.10.

Listagem 5.10 – Método *run()*

```
1 public void run() {
2    for (AOJProject project : AOJWorkspace.getInstance().getProjects())
3       for (AOJInterfaceDeclaration inter : project.getInterfaces())
4          findRefactorings(inter);
5 }
```

Em posse das interfaces, na próxima etapa foram selecionadas apenas as interfaces funcionais (linha 3 da Listagem 5.11). Em seguida foram comparadas suas características com as interfaces funcionais do pacote *java.util.function* (linha 3). Caso a comparação retorne mais de uma interface, a etapa de registro da informação é iniciada (linhas 4 e 5).

Listagem 5.11 – Método *findRefactorings()*

```
1 private void findRefactorings(AOJInterfaceDeclaration inter) {
2    if (inter.isFunctionalInterface()){
3       List<String> listInterfaces = getSimilarDefaultFunctionalInterface(
           inter);
4       if (listInterfaces.size() > 0)
5          print(inter, listInterfaces);
6    }
7 }
```

5.6.2 Resultados

Foram encontradas 117 interfaces funcionais compatíveis com as interfaces do pacote *java.util.function*. No gráfico apresentado na Figura 5.6, pode-se observar que todos projetos possuíam interfaces funcionais. O projeto *Eclipse Link* juntamente com o projeto *Apache Ant* apresentaram o maior número de oportunidades identificadas, sendo 45% e 28% do total de oportunidades, respectivamente.

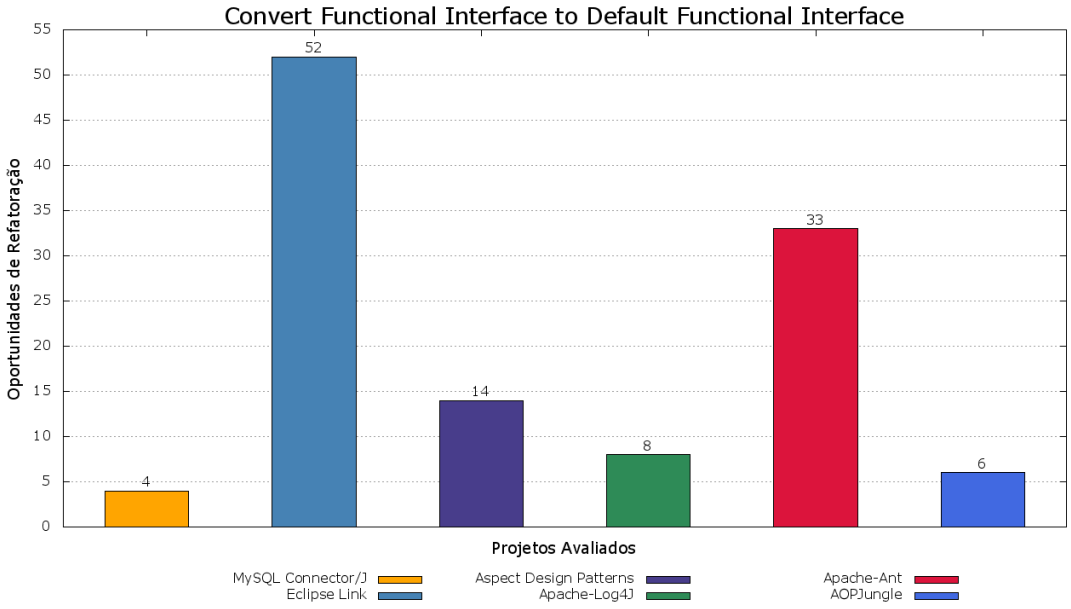

Figura 5.6 – R5 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

Vale observar que, das interfaces localizadas neste estudo de caso, podem existir situações nas quais uma interface funcional faça parte da implementação de uma classe. Portanto, a conversão para uma interface padrão fica sem sentido, pois os métodos implementados pela classe não poderão ser utilizados de forma anônima.

5.7 Convert Abstract Interface Method to Default Method (R6)

Ao se implementar uma classe com um contrato de uma interface, todos os métodos abstratos da interface devem ser implementados, mas nem sempre há a necessidade desta implementação. Sendo assim, para esta refatoração, o *λRefactoring* buscou por métodos concretos de interfaces, implementados em classes, com o seu corpo vazio.

5.7.1 Implementação

Primeiramente, foram selecionadas apenas as classes disponíveis nos projetos. Em sequência, foram filtradas somente as classes que implementavam uma interface. A partir dessas classes, foram analisados apenas os métodos concretos exigidos pela interface, o qual foi verificado se possuíam alguma instrução de código, como pode ser visto na Listagem 5.12.

Listagem 5.12 – Método *findRefactorings()*

```
1 private void findRefactorings(AOJClassDeclaration clazz) { (...)
2    if (method.getName().equalsIgnoreCase(methodInterface)){
3       int size = method.getStatementsBody();
4       if (size == 0 || (size == 1 && !method.getReturnType().getName().
            equals("void"))){
5          String message = String.format( "Class "+ clazz.getName() +": "+
               method.getName() + " ("+ i.getFullyQualifiedName() + ")");
6          list.add(message);
7       }
8    }
9 }
```

5.7.2 Resultados

A falta de necessidade de se implementar métodos pode ser observada pela grande quantidade de oportunidades encontradas para esta refatoração. Foram identificadas 5.650 classes que possuíam métodos vazios, os quais foram implementados somente pela obrigatoriedade de um contrato de interface.

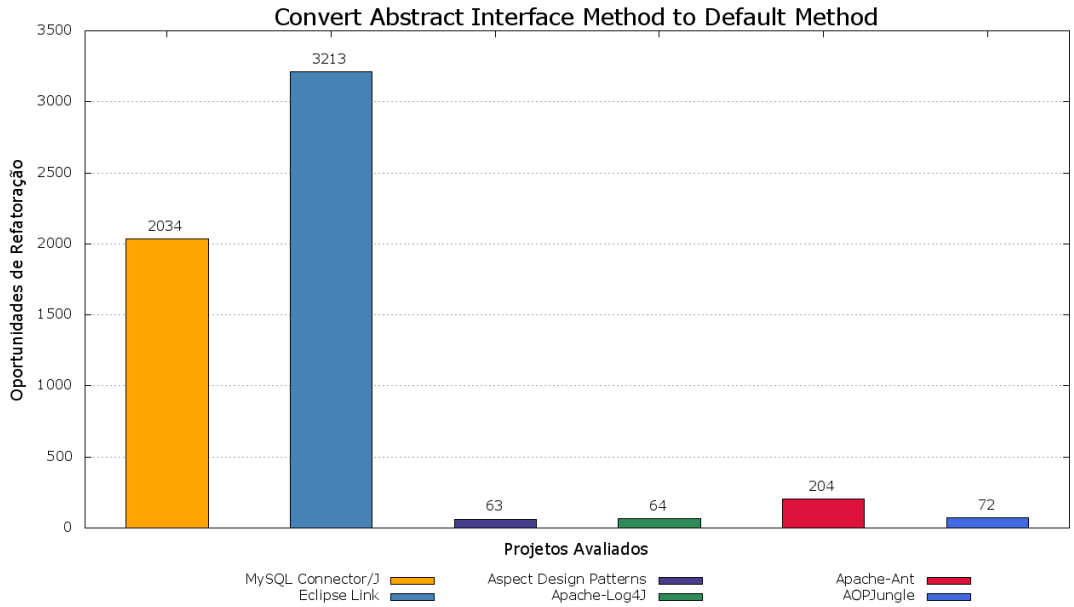

Figura 5.7 – R6 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

O gráfico apresentado na Figura 5.7 mostra que em todos os projetos foram encontradas oportunidades para esta refatoração, e que os projetos *Eclipse Link* e *MySQL Connector/J* tiveram uma maior representatividade, com 57% e 36% de oportunidades encontradas, respectivamente.

5.8 Convert Inter-Type Method Declaration to Default Interface Method (R7)

Esta refatoração é exclusiva para projetos que utilizam uma linguagem orientada a aspectos e sua função é substituir declarações de métodos inter-tipos que adicionam métodos em interfaces, pela mesma implementação diretamente nas interfaces. A linguagem orientada a aspectos permite a adição de métodos implementados em interfaces, porém com a chegada dos métodos *default* à linguagem Java, esse tipo de recurso pode ser aplicado diretamente nas próprias interfaces. O plug-in *λRefactoring* realizou buscas em todos os aspectos que implementavam uma ITMD em uma interface.

5.8.1 Implementação

Na primeira etapa foram selecionados apenas os aspectos existentes nos projetos. Em seguida, para cada aspecto, foram obtidas as suas respectivas declarações inter-tipos, conforme apresentado na Listagem 5.13. Para cada declaração o *λRefactoring* verificou se a estrutura que ela se aplica era uma interface (linhas 3 e 4). Ao identificar uma interface a terceira etapa é realizada.

Listagem 5.13 – Método *findRefactorings()*

```
1 private void findRefactorings(AOJAspectDeclaration aspect) {
2    for (AOJDeclareMethod interTypeMethod : aspect.getMembers().
        getDeclareMethods()) {
3       AOJInterface inter = loadInterface(interTypeMethod.getOnType(),
           aspect);
4       if (inter != null)
5          print(interTypeMethod, aspect);
6    }
7 }
```

5.8.2 Resultados

No gráfico que representa os resultados, Figura 5.8, pode-se observar que 100% das oportunidades foram encontradas no projeto *Aspect Design Patterns*, que é um projeto que implementa padrões de projetos utilizando aspectos. Esta oportunidade foi a de menor ocorrência

nos projetos analisados, com um total de 32 oportunidades encontradas em um único projeto.

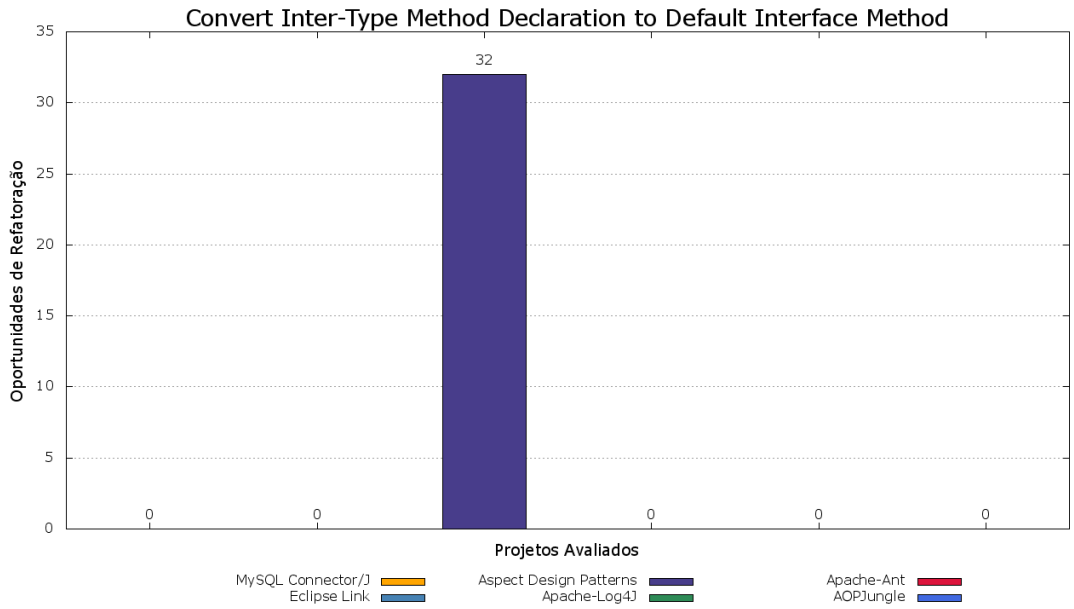

Figura 5.8 – R7 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

5.9 Extract Method Reference (R8)

Métodos concretos que possuem as mesmas características de um método de uma interface funcional, podem substituir um expressão lambda através de uma referência a tal método. Dessa forma, o plug-in *λRefactoring* buscou por locais em que uma expressão pudesse ser aplicada.

5.9.1 Implementação

Para a busca desta refatoração, o plug-in *λRefactoring* usou as pesquisas realizadas para as refatorações *Convert Functional Interface Instance to Lambda Expression*, *Convert Enhanced For to Lambda Enhanced For* e *Convert Enhanced For with If to Lambda Filter*, as quais aplicam expressões lambda em sua conversão.

5.9.2 Resultados

Um total de 3.689 oportunidades de refatoração foi encontrado em todos os seis projetos, sendo que o maior número de oportunidades, 70%, foi encontrado no projeto *Eclipse Link* (Figura 5.9). Vale ressaltar que nem todas as oportunidades encontradas para esta refatoração

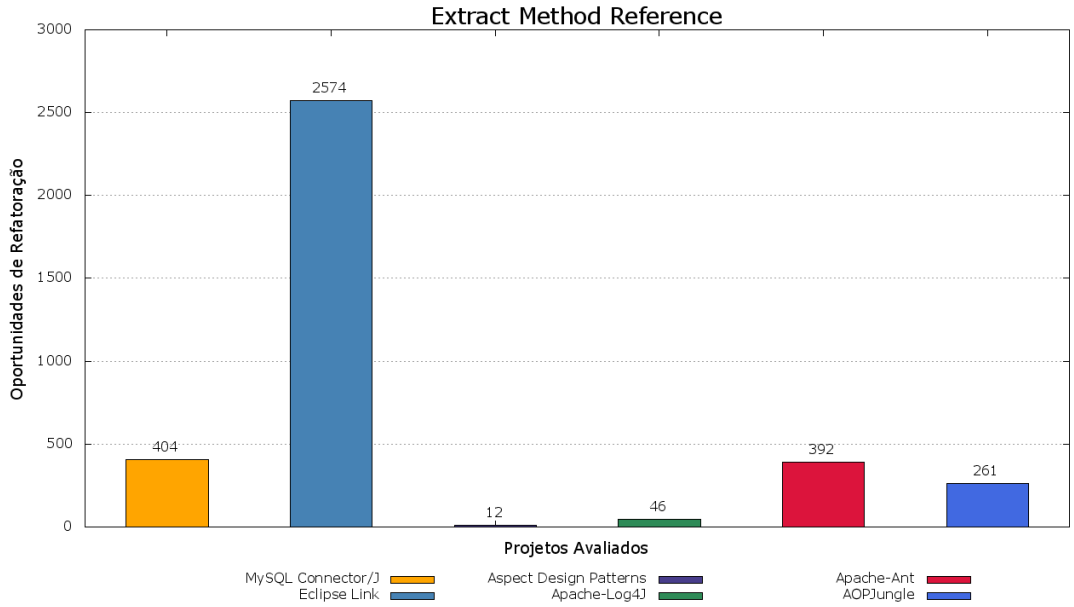

Figura 5.9 – R8 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

são interessantes de serem aplicadas, podendo ser analisadas caso a caso.

5.10 Convert Interface to Functional Interface (R9)

Sabe-se que as expressões lambda se aplicam somente às interfaces funcionais. Dessa forma, para localizar as oportunidades desta refatoração, o *λRefactoring* buscou por todas as interfaces existentes nos projetos.

5.10.1 Implementação

Após a seleção das interfaces, realizada na primeira etapa, o *λRefactoring* verificou se tais interfaces não eram do tipo funcional, conforme mostrado na linha 2 da Listagem 5.14.

Listagem 5.14 – Método *findRefactorings()*

```
1 private void findRefactorings(AOJInterfaceDeclaration inter) {
2    if (!inter.isFunctionalInterface())
3       print(inter);
4 }
```

5.10.2 Resultados

Interfaces são bastante utilizadas em projetos Java, isso se confirma com um total 540 oportunidades encontradas para esta refatoração. O maior número de ocorrências (393) foi detectado no projeto *Eclipse Link*, conforme exibido no gráfico representado pela Figura 5.10.

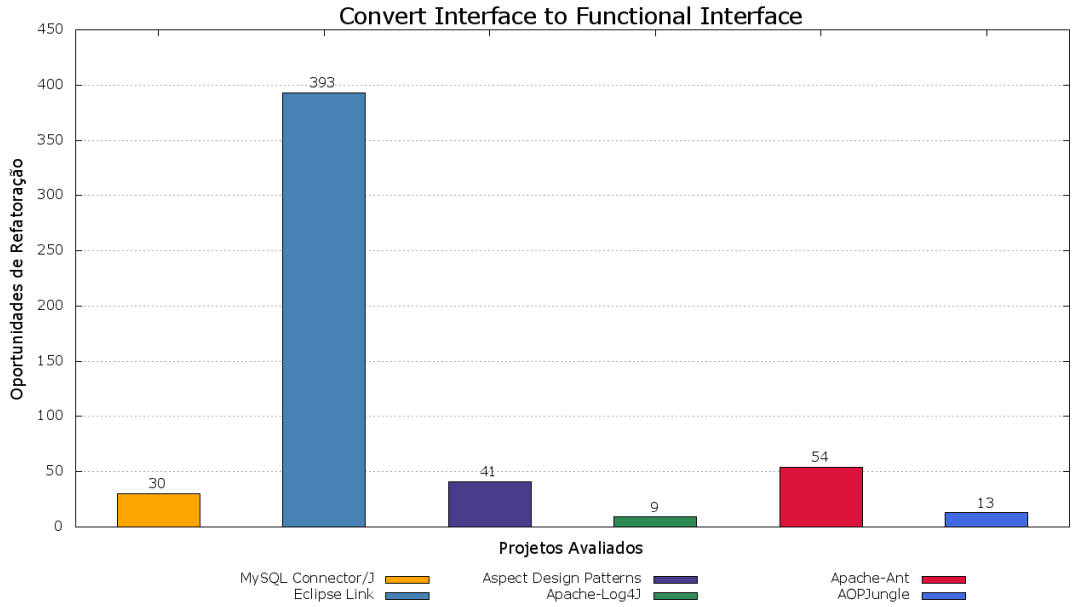

Figura 5.10 – R9 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

5.11 Convert Enhanced For to Parallel For (R10)

Utilizando expressões lambda, uma operação em paralelo pode ser facilmente aplicada em uma estrutura de coleções, através da API *Stream* adicionada ao Java 8. Para localizar as oportunidades desta refatoração, o *λRefactoring* buscou construções *for each* em classes e aspectos.

5.11.1 Implementação

Para a busca de oportunidades desta refatoração, o plug-in *λRefactoring* usou a implementação realizada para a refatoração *Convert Enhanced For to Lambda Enhanced For*, pois tal implementação realiza buscas por laços de repetição (*for each*) em construtores, métodos e *advices*.

5.11.2 Resultados

Nem todos os laços de repetição encontrados poderão ser utilizados de forma paralela. Mas buscando verificar a usabilidade dessa refatoração, um total de 2.302 oportunidades de refatoração foi encontrado em todos os seis projetos, conforme mostrado na Figura 5.11.

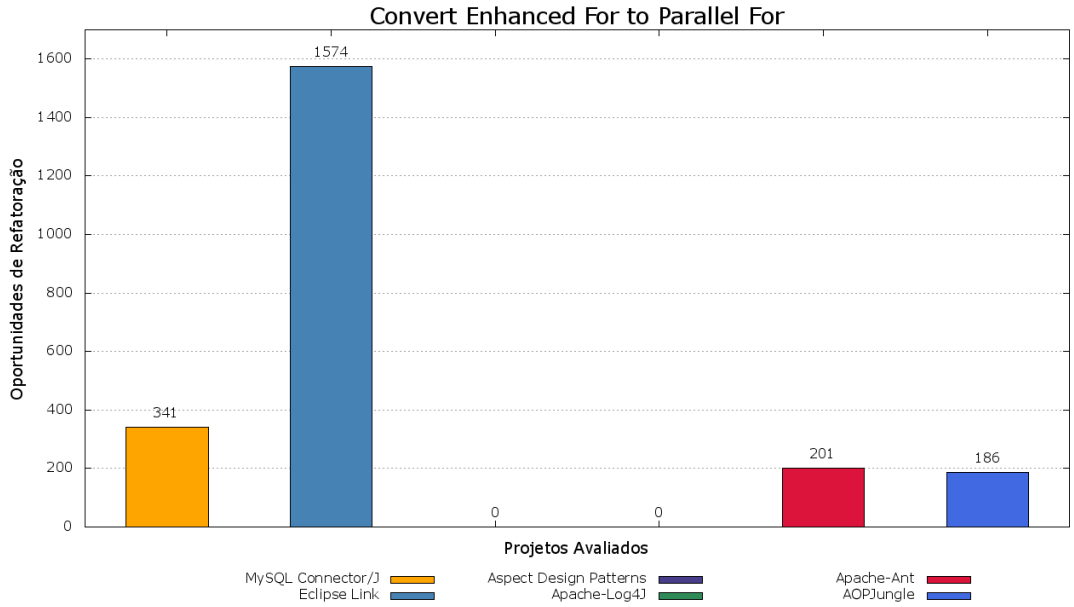

Figura 5.11 – R10 - Gráfico de Oportunidades de Refatoração (OR) Encontradas

5.12 Considerações Finais

Este estudo de caso analisou dez refatorações descritas no catálogo do Capítulo 3, sendo a maioria transformações de estruturas "convencionais" para uma representação equivalente que utilize expressões lambda. O objetivo principal desse estudo foi investigar se as refatorações propostas poderiam ser comumente utilizadas.

Como se pode observar na Tabela 5.2 o plug-in *λRefactoring* encontrou um número alto de oportunidades de refatoração, com um total de 16.056 oportunidades nos seis projetos analisados. O maior número de oportunidades identificadas foi para o maior projeto analisado, o *Eclipse Link*, com um total de 10.404 oportunidades. Algumas refatorações não foram identificadas em alguns projetos, mas todas elas tiveram ocorrências em pelo menos um dos projetos analisados.

Tabela 5.2 – Tabela de OP de Refatorações X Sistemas de Software

Software	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**Total**
AOPJungle	05	186	0	70	6	72	0	261	13	186	**799**
Apache-Ant	29	201	10	162	33	204	0	392	54	201	**1.286**
Apache-Log4J	46	0	0	0	8	64	0	46	09	0	**173**
Aspect Design Patterns	12	0	0	0	14	63	32	12	41	0	**174**
EclipseLink	21	1.574	24	979	52	3.213	0	2.574	393	1.574	**10.404**
MySQL Connector/J	07	341	3	56	4	2.034	0	404	30	341	**3.220**
Total	**120**	**2.302**	**37**	**1.267**	**117**	**5.650**	**32**	**3.689**	**540**	**2.302**	**16.056**

Neste estudo de caso, não foi levada em consideração a qualidade das oportunidades identificadas. Portanto, um trabalho futuro poderia classificar essas oportunidades, através de critérios definidos para cada uma das refatorações, apontando qual oportunidade teria uma maior relevância em relação às outras.

6 CONCLUSÃO

Este trabalho teve como principal objetivo apresentar uma série de refatorações relacionadas aos novos recursos da linguagem Java, descritos na especificação JSR-335 (ORACLE, 2014a), a qual fornece detalhes sobre expressões lambda e funcionalidades diretamente ligadas. Dessa forma, um conjunto de refatorações foi proposto, formando um catálogo com um total de vinte refatorações. É importante ressaltar que o catálogo apresentado incluiu as refatorações inversas, que podem ser utilizadas quando uma construção se tornar mais complexa, por exemplo, e assim permitir retornar a sua forma original, na medida do possível.

Para a validação das refatorações propostas, foi desenvolvida uma ferramenta para a identificação de oportunidades de refatoração, para códigos escritos em Java e AspecJ, com a finalidade em determinar o quão utilizável pode ser cada refatoração. Assim, a implementação da ferramenta se dividiu em duas etapas:

- **AOPJungle**. Foi realizada uma extensão no *framework* AOPJungle, utilizado para abstrair informações sobre códigos OO e OA. Este *framework* foi melhorado para dar suporte às funcionalidades relacionadas às expressões lambda e utilizado para fornecer informações para o plug-in λRefactoring; e

- λ**Refactoring**. Foi construído um plug-in com o objetivo específico de localizar as oportunidades das refatorações propostas, descritas no Capítulo 3, e assim determinar sua usabilidade.

Para tanto, seis projetos de código aberto foram submetidos para a análise do λRefactoring e, após sua execução, um total de 16.056 oportunidades de refatoração foi localizado. Com este número, entende-se que as refatorações propostas podem ser comumente utilizadas, mesmo as que não obtiveram um número tão alto, pois a busca foi realizada apenas em projetos anteriores ao Java 8, em que a ideia de se utilizar uma interface funcional não era usual.

6.1 Trabalhos Futuros

A seguir são apresentadas alguma sugestões para trabalhos futuros.

- **Classificação das Oportunidades**. Um estudo que pode ser realizado sobre as oportunidades de refatoração encontradas neste trabalho é a classificação das oportunidades. Uma forma de realizar esta análise seria através da definição de critérios pré-definidos para cada tipo de refatoração, estabelecendo níveis de importância entre as oportunidades encontradas.

- **Qualidade das Refatorações**. Um importante ponto a ser pesquisado é a qualidade de uma refatoração. Por exemplo, ao utilizar paralelismo em uma coleção, a primeira vista pode-se parecer vantajoso. No entanto, toda uma estrutura é montada para realizar um paralelismo e, dependendo do tipo de instrução contida em um laço de repetição, o *overhead* necessário para prover esse paralelismo pode deixar a execução da estrutura mais lenta.

- **Novas Refatorações Relacionadas às Expressões Lambda**. O catálogo proposto neste trabalho apresentou vinte refatorações relacionadas às expressões lambda. No entanto, existem algumas outras funcionalidades adicionadas à linguagem que não foram abordadas. A API *Stream*, por exemplo, apresenta inúmeros recursos que podem ser passíveis à criação de novas refatorações. Um bom exemplo seriam as refatorações relacionadas às funções do tipo *map* e *reduce*.

- **Aplicação da Refatoração**. O trabalho apresentado buscou definir meios para a localização de oportunidades de refatoração. Entretanto, seria interessante realizar o processo de refatoração por completo. Assim, modificações mais significativas teriam de ser realizadas no *framework* AOPJungle como, por exemplo, a localização exata de uma construção e seus possíveis relacionamentos. No entanto, as principais IDEs já implementam algumas dessas refatorações.

- **Linguagem AQL**. A linguagem *Aspect Query Language*, criada por Faveri (2013), pode ser estendida para analisar os novos recursos da linguagem Java, facilitando, por exemplo, a busca por oportunidades de refatoração. Como a linguagem está relacionada ao *framework* AOPJungle, pode-se dizer que parte da extensão ja foi inicialmente realizada, faltando relacionar os novos recursos adicionados ao AOPJungle à estrutura da linguagem em si.

REFERÊNCIAS

BOEHM, B.; IN, H. Identifying Quality-Requirement Conflicts. **IEEE Software**, [S.l.], v.13, n.2, p.25–35, 1996.

CHURCH, A. An Unsolvable Problem of Elementary Number Theory. **American Journal of Mathematics**, [S.l.], v.58, n.2, p.345–363, 1936.

CORNÉLIO, M. L. **Refactorings as Formal Refinements**. 2004. Tese (Doutorado em Ciência da Computação) — Centro de Informática, Universidade Federal de Pernambuco, Brazil.

DEITEL, P. J.; DEITEL, D. H. M. **Java How to Program**. [S.l.]: Prentice Hall; 9th edition (March 7, 2011), 2011.

DU BOIS, B. **A Study of Quality Improvements by Refactoring**. [S.l.]: Citeseer, 2006. v.68, n.02.

DU BOIS, B.; MENS, T. Describing the Impact of Refactoring on Internal Program Quality. In: INTERNATIONAL WORKSHOP ON EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE APPLICATIONS. **Anais...** [S.l.: s.n.], 2003. p.37–48.

ECLIPSE. **Xtend User Guide**. Acesso em: agosto de 2013, disponível em: http://www.eclipse.org/xtend/documentation/2.4.0/Documentation.pdf.

EPFL. **Scala Language**. Acesso em: agosto de 2013, disponível em: http://www.scala-lang.org/.

FAVERI, C. de. **Uma Linguagem Específica de Domínio para Consultar em Código Orientado a Aspectos**. 2013. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Santa Maria-RS, Brasil.

FOWLER, M. **Domain Specific Languages**. [S.l.]: Addison-Wesley Professional, 2010.

FOWLER, M. et al. **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: INTL. CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2013. p.1287–1290.

GYORI, A. et al. Crossing the Gap From Imperative to Functional Programming Through Refactoring. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 9. **Proceedings...** [S.l.: s.n.], 2013. p.543–553.

HELM, R. et al. **Design Patterns**: elements of reusable object-oriented software. [S.l.]: Addison-Wesley, 2002.

HUANG, J. et al. EXTRACTOR: an extensible framework for identifying aspect-oriented refactoring opportunities. In: SYSTEM SCIENCE, ENGINEERING DESIGN AND MANUFACTURING INFORMATIZATION (ICSEM), 2011 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. v.2, p.222–226.

ISO, I. IEC 9126-1: software engineering-product quality-part 1: quality model. **Geneva, Switzerland: International Organization for Standardization**, [S.l.], 2001.

KATAOKA, Y. et al. Automated Support for Program Refactoring Using Invariants. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'01). **Proceedings...** [S.l.: s.n.], 2001. p.736.

MACIA, I.; GARCIA, A.; STAA, A. von. Defining and Applying Detection Strategies for Aspect-oriented Code Smells. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.60–69.

MENS, T.; TAENTZER, G.; RUNGE, O. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.127, n.3, p.113–128, 2005.

MENS, T.; TOURWE, T. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, [S.l.], v.30, n.2, p.126–139, Feb. 2004.

OLBRICH, S. et al. The Evolution and Impact of Code Smells: a case study of two open source systems. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2009. **Proceedings...** [S.l.: s.n.], 2009. p.390–400.

OPDYKE, W. **Refactoring Object-oriented Frameworks**. 1992. Tese (Doutorado em Ciência da Computação) — University of Illinois.

ORACLE. **State of Lambda**. Acesso em: dezembro de 2013, disponível em: http://cr.openjdk.java.net/ briangoetz/lambda/lambda-state-4.html.

ORACLE. **Java Specification Requests 335**. Acesso em: janeiro de 2014, disponível em: https://www.jcp.org/en/jsr/detail?id=335.

ORACLE. **Project Lambda**. Acesso em: janeiro de 2014, disponível em: http://openjdk.java.net/projects/lambda/.

PIERCE, B. C. **Types and Programming Languages**. [S.l.]: The MIT Press, 2002.

PIVETA, E. K. **Improving the Search for Refactoring Opportunities on Object-oriented and Aspect-oriented Software**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

PIVETA, E. K. et al. Detecting Bad Smells in AspectJ. **Journal of Universal Computer Science**, [S.l.], v.12, n.7, p.811–827, 2006.

SCHUMACHER, J. et al. Building Empirical Support for Automated Code Smell Detection. In: ACM-IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.8.

VAN EMDEN, E.; MOONEN, L. Java Quality Assurance by Detecting Code Smells. In: REVERSE ENGINEERING, 2002. PROCEEDINGS. NINTH WORKING CONFERENCE ON. **Anais...** [S.l.: s.n.], 2002. p.97–106.