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

Transformações de Programa para Suportar a Evolução da Linguagem Java

Reno Medeiros Dantas

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2017

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

Transformações de Programa para Suportar a Evolução da Linguagem Java

Reno Medeiros Dantas

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

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

Prof. Dr. Ricardo Terra
DCC / UFLA

Prof. Dr. Genaina Nunes
CIC / UnB

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

Brasília, 26 de Setembro de 2017

Dedicatória

Dedico este trabalho à minha família e, em especial, à minha filha pelo amor e carinho nas horas mais difíceis.

Agradecimentos

Ao meu orientador, Prof. Dr. Rodrigo Bonifácio, por todo o trabalho, esforço, paciência e por me apresentar o mundo da refatoração automatizada por meio de meta linguagens.

Resumo

Considerando a literatura de Engenharia de Software, tem sido reportado que os custos mais altos de desenvolvimento estão relacionados com as atividades de manutenção de software. De forma frequente, os sistemas precisam se adaptar às inovações negociais e tecnológicas. Similarmente, as linguagens de programação passam por um processo de evolução contínua – conforme os recursos (e limitações) de uma linguagem são melhor compreendidas, novas construções de linguagem são introduzidas. Todavia, quando um novo recurso estende ou modifica uma construção de linguagem, com o intuito de melhorar a facilidade de entendimento, manutenabilidade ou eficiência, recursos linguísticos anteriores passam a ser considerados obsoletos. Nesse sentido, a coexistência de construções novas e legadas em uma linguagem tende a aumentar a complexidade de manutenção dos sistemas existentes, bem como dificultar o aprendizado por parte dos desenvolvedores, que precisam conhecer estratégias diferentes de implementação para uma mesma situação. Esta dissertação tem como objetivo explorar o uso de suporte ferramental para apoiar técnicas de transformação de programas, na intenção de adaptar sistemas legados para usar construções mais recentes da linguagem de programação Java, que vem passando por um processo relativamente contínuo de melhoria. No total, foram implementadas seis transformações, que permitem, por exemplo, introduzir a construção *multi-catch* e expressões *lambda* em sistemas Java legados. Para compreender a aceitação dessas transformações por parte da comunidade de de desenvolvimento de software, foram aplicadas 2.371 transformações de programas em 40 projetos de código aberto. Foi identificado que transformações mais simples (como a introdução do operador diamante) possuem uma aceitação maior do que as transformações que apresentam um maior impacto no código fonte (como as transformações que convertem *enhanced for loops* em expressões *lambda*).

Palavras-chave: Evolução de Linguagem, Transformação de Programas.

Abstract

Considering the Software Engineering literature, it has been reported that most resources are spent during the activites of software maintenance. Often, software systems have to be adapted according either to the changes of the business needs or technology innovation. In a similar way, programming languages also have to evolve continually—as soon as the language resources (and limitations) are better understood, new programming languages constructs and features should be introduced. However, whenever a new feature is introduced in a programming language, some of the old constructs and idioms become obsolete. Accordingly, the coexistence of new and old constructs and idioms makes the programs hard to evolve and understand. In this master dissertation, we explore the use of program transformations that evolve Java legacy systems towards the use of new programming language constructs. We have implemented six transformations that allow, for instance, the automatic introduction of *mult-catch* and *lambda expressions* constructs in Java legacy systems. In order to understand the acceptance of this kind of transformation, we performed 2,371 transformation in 40 open-source projects. We realized that simple transformations (such as the introduction of the diamond operator) are more likely to be accepted than transformations that substantially change the code (such as the transformations that replace *enhanced for loops* by *lambda expressions*).

Keywords: Language Evolution, Program Transformations.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Desenvolvimento de sistemas é um processo de evolução contínua. Conforme os requisitos de um sistema se adequam para refletir novas necessidades negociais, os sistemas necessitam passar por um processo natural de evolução. Processo análogo ocorre com as linguagens de programação. À medida que as demandas e os recursos da linguagem vão sendo melhor compreendidos, construções e abstrações vão sendo aprimoradas para acompanhar essa evolução. Por outro lado, o processo de evolução de linguagens de programação apresenta um desafio: novos recursos são adicionados e/ou aperfeiçoados à linguagem, mas os que se tornam obsoletos em detrimento a eles raramente são removidos [1], pois causaria uma perda de retrocompatibilidade com código legado. Além disso, mesmo que um determinado código esteja atualizado, no momento que a linguagem disponibiliza uma nova maneira de resolver um problema, tudo que foi escrito usando as antigas construções passa a ser considerado legado.

Desse modo, código escrito em versões obsoletas da linguagem tem que coexistir com as novas construções. Isso aumenta a complexidade das linguagens, em termos das diferentes construções sendo usadas por um programa, o que dificulta o seu aprendizado, a manutenção de sistemas legados e a evolução da própria linguagem. Consequentemente, existe um dilema na evolução de linguagens de programação. Se por um lado a decisão em não evoluir uma linguagem faz com que ela se torne estagnada, por outro a decisão de eliminar construções obsoletas acarreta perda de retrocompatibilidade, em detrimento do aumento de complexidade e custo da evolução.

Para lidar com esse problema, Overbey e Johnson sugerem uma estratégia para evoluir linguagens de programação ao mesmo tempo em que mitiga as desvantagens do processo: refatoraçã de código para suportar evolução de linguagens [1]. Refatoraçã corresponde a um tipo particular de transformação de código fonte que preserva comportamento [2, 3]. Até 2009, muitos IDEs[1] conhecidos já suportavam refatoraçãs comuns, como extrair um

[1] *Integrated Development Environment*

bloco de código para um novo método (favorecendo reuso) ou mudar o nome de um método (favorecendo a facilidade de compreensão), por exemplo. Essas características, facilidade de compreensão e reuso, são os benefícios tipicamente esperados como resultado da aplicação de um refatoraçã. Entretanto, essa noção de transformação que preserva comportamento pode ser estendida para outros cenários, como bibliotecas e componentes. Por exemplo, Dig e Johnson observaram que 80% das mudanças em uma quantidade significativa de API's Java poderiam ser obtidas por utilização de *scripts* de refatoraçã automática [4]. Mais ainda, refatorações poderiam ser aplicados para transformar o código escrito em uma versão anterior de uma linguagem para um código que usa construções mais recentes da mesma linguagem [1, 5].

Cenários de evolução de linguagens de programação podem ser facilmente exemplificados. A linguagem C++ passa por constante evolução, com novas versões lançadas em 2003, 2011 e 2014. A linguagem Python possui duas *major releases* (2.7 e 3.x) que coexistem desde 2010. Migrar da versão 2.7 para as versões 3.x tem se mostrado relativamente custoso – particularmente pela incompatibilidade de bibliotecas existentes. Similarmente, a linguagem Java vem passando por uma constante evolução, mas sempre com a preocupação de retrocompatibilidade. Duas *releases* trouxeram um maior impacto sintático e semântico. A versão Java 5 introduziu recursos de polimorfismo parametrizado (Java *Generics*), suporte a anotações em código fonte e um novo comando de iteração sobre coleções de objetos (*enhanced for loops*). Mais recentemente, a versão Java 8 introduziu expressões *Lambda*, um recurso esperado por vários desenvolvedores que usavam a linguagem. Esses são dois marcos relevantes na evolução da linguagem Java, mas outras versões "intermediárias" também incluíram mudanças sintáticas e semânticas na linguagem.

Refatoraçã de código fonte e evolução de linguagens de programação caracterizam o escopo desta pesquisa, que tem como principal interesse investigar a aceitção de transformações de código para apoiar a adoção de construções recentes de uma linguagem de programação. O contexto da pesquisa é inserido na comunidade de desenvolvimento de projetos *open-source* escritos na linguagem de programação Java.

1.1 Questões de Pesquisa

O cenário de evolução de linguagens de programação leva a algumas questões gerais de pesquisa (QGP) ainda não respondidas de forma sistematizada na literatura.

Neste contexto, falsos positivos são situações em que uma transformação poderia ser aplicada mas que, por decisão de projeto, a implementação não aplica a transformação. Ou seja, o design das transformações atuais levaram em consideração aspectos gerenciais

(como custos e prazos de desenvolvimento) e aspectos técnicos envolvendo garantias que tinham como objetivo levar a um resultado que "não quebra o código fonte".

Considerando um exemplo mais concreto, para transformar *Anonymous Inner Classes* (AIC) em expressões *Lambda*, algumas restrições precisam ser observadas [5]. Isso garante uma maior "segurança" (evitando falsos negativos) em uma quantidade maior de situações em que a transformação pode ser aplicada (evitando falsos positivos). Essas restrições devem levar em consideração não apenas a análise do código fonte, mas também os tipos referenciados no contexto da transformação.

Por outro lado, realizando algumas simplificações (com a verificação que a maior parte das classes anônimas implementam um conjunto restrito de interfaces), é possível manter uma abrangência aceitável das transformações e diminuir os custos de implementação (diminuindo, por exemplo, a preocupação com os tipos envolvidos). Durante o primeiro ano de pesquisa, foi possível perceber que essas simplificações levam a um código de transformação mais simples de ser compreendido e a transformação mais fácil de ser usada pelos desenvolvedores.

(QGP1) Qual a percepção dos desenvolvedores de software com a necessidade de se reestruturar um código para torná-lo aderente aos novos recursos de uma linguagem de programação?

(QGP2) Qual o impacto em se conciliar requisitos conflitantes no desenvolvimento de transformações (como custos e prazos de desenvolvimento e expressividade da implementação)? Conforme discutido no Capítulo 4.4, a implementação das transformações discutidas nesta dissertação são bem conservativas, levando a um menor custo de desenvolvimento e a um maior número de falsos positivos.

(QGP3) Qual a aplicabilidade das transformações em projetos reais? A quantidade de oportunidade de aplicação de transformações e muito maior que número de falsos positivos ocorridos? Tal relação fornece um indicativo de melhoria para uma transformação.

1.2 Objetivos

O principal objetivo deste trabalho é explorar o uso de técnicas avançadas de transformação de programas com o intuito de transformar um código fonte legado em um código fonte que suporte novas construções da linguagem. Mais especificamente, este trabalho tem os seguintes objetivos:

OBJ1 Estudar a linguagem de meta-programação Rascal – Uma linguagem de programação que permite realizar análise e refatoração de um código fonte além de possuir um suporte ferramental robusto.

OBJ2 Implementar um conjunto de transformações de programa, voltadas para suportar a evolução da linguagem Java, usando Rascal.

OBJ3 Aplicar as transformações em repositórios de projetos *open-source* e verificar a aceitação dos pedidos de alteração de código pelas respectivas comunidades de desenvolvimento.

OBJ4 Analisar e reportar as decisões de design das transformações, que causam impacto tanto na qualidade das transformações quanto nos custos de desenvolvimento.

1.3 Metodologia

A metodologia envolve uma revisão da literatura para entender a evolução da linguagem Java, e as técnicas e ferramentas que a comunidade científica tem explorado para transformação de programas. A pesquisa envolve ainda a revisão das estratégias existentes para transformação de programas específicas para suportar a evolução da linguagem Java, bem como a implementação de transformações para esse contexto específico.

A primeira questão de pesquisa pode ser respondida analisando a taxa de aceitação das transformações feitas via o mecanismo de *pull requests*—que devem ser submetidos em projetos *open-source*. A segunda questão de pesquisa pode ser respondida comparando, via uma análise mais técnica, o grau de expressividade da implementação da solução proposta nesta dissertação com uma ferramenta desenvolvida anteriormente (Lambdaficator) [5] – que faz uma busca em projetos Java no IDE *NetBeans* por código legado que possa ser refatorado para expressões *Lambda*.

1.4 Organização do Trabalho

O restante deste documento está organizado em mais quatro capítulos. No Capítulo 2, é apresentado um referencial teórico para apoiar o leitor a compreender alguns temas-chave da dissertação, como a evolução da linguagem Java e duas taxonomias relacionadas à transformações de programas e métodos empíricos em Engenharia de Software. No Capítulo 3, são discutidas algumas questões de implementação das transformações voltadas para suportar a adaptação de sistemas legados para o uso de construções mais recentes da linguagem Java. No Capítulo 4, é discutido um estudo empírico conduzido para avaliar

tanto a aceitação dos resultados da aplicação das transformações de código quanto a expressividade de um subconjunto das transformações. Finalmente, o Capítulo 5 apresenta algumas considerações finais sobre a pesquisa realizada e descreve algumas possibilidades de trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo contextualiza o leitor em termos dos principais temas relacionados à essa dissertação, incluindo *Evolução da Linguagem de Programação Java*, *Transformação de Programas* e *Métodos Empíricos em Engenharia de Software*. Com base na leitura deste capítulo, o leitor deve compreender algumas das decisões tomadas ao longo do desenvolvimento da pesquisa, como a linguagem de transformação escolhida para implementar os refatoramentos de programas e o método empírico em engenharia de software selecionado para avaliar os resultados do trabalho.

2.1 Evolução da Linguagem Java

Inicialmente chamada `OAK`, a linguagem Java foi concebida com o intuito de utilização em dispositivos embarcados. Uma das ideias centrais envolvia a abstração da camada de acesso direto ao *hardware*, para facilitar a disponibilização de programas em dispositivos ou aplicações Web[1]. Além disso, a linguagem deveria ser independente de plataforma e de fácil escrita, simples de testar, e possuir características consolidadas de linguagens anteriores. A mudança de nome foi necessária porque a "marca" `OAK` já tinha sido registrado pela *Oak Technology*, então a equipe da *Sun Microsystems*, empresa que era responsável pela concepção e implementação da linguagem, optou por *rebatizar* a linguagem para Java.

Em sua versão inicial, a implementação da linguagem já considerava o uso de uma máquina virtual—a *Java Virtual Machine* (JVM), atualmente considerada um importante legado da linguagem, inclusive permitindo que outras linguagens de programação passaram a *compilar* código fonte para a JVM. A JVM possibilita a independência de plataforma, ou seja, um programa escrito uma vez na linguagem Java (em uma plataforma qualquer) podia ser executado em qualquer outra plataforma (desde que possua

[1] *World Wide Web*

uma implementação da JVM). Essa versão foi bastante difundida por facilitar a criação aplicativos para Web. A grande vantagem era que os aplicativos eram auto-contidos, não sendo necessário procurar por dependências externas a eles. O formato portável e resultante do processo de compilação da linguagem Java (conhecido como *bytecode*) podia ser transferido para a máquina onde estava sendo invocado e executado pela *JVM* local. Posteriormente, o desenvolvimento da linguagem Java ocorreu em torno de diferentes especificações, incluindo a *Java Standard Edition* (JSE) e *Java Enterprise Edition* (JEE). Essa dissertação se concentra na especificação JSE.

Da mesma forma que outras linguagens de programação, a linguagem Java passou, ao longo dos últimos vinte anos, por um processo contínuo de evolução. Algumas *releases* da linguagem apresentaram um enfoque maior em termos de melhorias de implementação da linguagem (envolvendo compiladores e JVM, por exemplo) e introdução de novas classes / bibliotecas à especificação da linguagem (JSE 1.1, JSE 1.2, JSE 1.3 e JSE 1.6). Diferentemente, outras *releases* introduziram novas construções à linguagem de programação Java, sempre com a preocupação em manter compatibilidade retroativa (JSE 1.5, JSE 1.7 e JSE 1.8). Tal preocupação também era algo inicialmente previsto, de tal forma que, na primeira especificação da linguagem, Gosling et al. escreveu [6]:

> *Nós acreditamos que a linguagem Java é uma linguagem madura e pronta pra uso de maneira disseminada. Por outro lado, nós esperamos alguma evolução na linguagem nos próximos anos e temos a intenção de gerenciar essa evolução de uma forma completamente compatível com aplicações existentes.*

As transformações de código desenvolvidas nessa dissertação têm como objetivo evoluir um sistema existente para suportar o uso de construções da linguagem Java introduzidas nas suas duas últimas releases públicas: JSE 1.7 e JSE 1.8. A versão JSE 1.7, intitulada *Dolphin*, introduziu um conjunto significativo de melhorias. Dentre eles, o aprimoramento de bibliotecas existentes de entrada/saída, redes, segurança, renderização 2D, tratamento XML, internacionalização e acesso a bancos de dados. Em relação a novas construções, a versão *Dolphin* introduziu o suporte a representação de literais binários (indicados por `0b` ou `0B` no literal), decimais literais (adição de quantidade arbitrária do caractere "_" entre o segundo e penúltimo algarismo de um literal), inclusão do tipo `String` em definições `switch`, adição do operador diamante (<>) para inferência de tipos para instâncias de classes genéricas, possibilidade de agrupar cláusulas `catch` que possuem corpo similar e definição de `try` com alocação de recursos.

Com o lançamento de sua oitava versão, mais uma vez diversas características foram introduzidas à linguagem, como o suporte a expressões Lambda, que são *funções anônimas* de alta ordem com escopo local (visíveis apenas na expressão que as invoca) o que melhora a legibilidade de algumas construções na linguagem e viabiliza uma nova classe

de construções. Também foi incluído os uso do mecanismo de *Streams*, que permite, a partir de uma coleção, aplicar zero ou mais operações intermediárias envolvendo filtros, transformação sob os elementos da coleção, e redução dos elementos de uma coleção em um valor. É Importante destacar que *expressões Lambda* possuem uma completa sinergia com o mecanismo de *Streams*. Ainda nessa versão foram adicionadas os *default methods*, que permitem escrever métodos concretos em interfaces Java, cuja comportamento se propaga a todas as classes que implementam a interface. Por fim, foram adicionadas diversas bibliotecas para fornecer um maior suporte a concorrência, com algumas delas podendo ser usada com *streams*.

2.2 Transformação de Programas

De acordo com Eelco Visser, transformações de programas são usadas em um espectro amplo de aplicações na engenharia de software, incluindo construção de compiladores, síntese de programas, refatoramento, rejuvenescimento de software e engenharia reversa [7]. Esta seção apresenta um resumo de uma taxonomia, também de autoria de Eelco Visser, relacionada a transformações de programas, útil para contextualizar parte dos objetivos desta dissertação.

2.2.1 Taxonomia para Transformações de Programas

Eelco Visser classifica transformações de programas como *tradução*, quando a linguagem alvo é diferente da linguagem fonte, e *reformulação* quando as linguagens envolvidas (origem e destino da transformação) forem as mesmas. Como tipos de tradução é possível citar a **migração**, quando ocorre uma transformação de código entre linguagens que possuem nível similar de abstração, como C# para Java. **Síntese**, que busca reduzir o nível de abstração para alcançar um maior desempenho; **engenharia reversa**, que, ao contrário da síntese, tem por finalidade aumentar as abstrações disponíveis e, dessa forma, melhorar a compreensão de um programa; e **análise**, que recupera métricas ou representações de um programa a partir de uma análise estática ou dinâmica do software.

Quanto a reformulação, existe a **normalização**, que é comumente usada para eliminar açúcar sintático de um código fonte, substituindo construções complexas por suas formas padrão e, dessa maneira, reduzindo a complexidade de um programa. Há também a **otimização** que busca uma redução de tempo de processamento ou espaço em memória necessário para execução de uma programa; o **refatoramento**, que objetiva modificar o design / trechos de código de modo a melhorar atributos como reuso e facilidade de entendimento, mas preservando o seu comportamento; e a **renovação**, que ao contrário do refatoramento, tem o intuito de modificar o comportamento de um trecho de código,

seja para correção de um erro ou expansão de alguma característica antes inexplorada. No contexto desta dissertação, as transformações propostas são classificadas como refatoramento, tendo em vista que as mesmas buscam preservar o comportamento e trazer uma simplificação do código e conformidade a novos idiomas da linguagem Java.

2.2.2 Suporte Ferramental para Transformações de Programas

Existem diversas ferramentas para suportar transformações de programas, entre elas StrategoXT, Spoofax e Rascal. StrategoXT combina uma linguagem específica do domínio (*Domain Specific Language*, DSL) voltada para transformações de programas (Stratego) com um conjunto de ferramentas XT que segue a metáfora arquitetural de um *pipeline*, onde os componentes envolvidos no processo de transformação podem ser *combinados* sequencialmente [8]. Stratego foi uma das DSLs pioneiras para a implementação de transformações de programas. Baseada no paradigma de reescrita de estratégias programáveis, Stratego é oferece mecanismos avançados para a a execução de travessias e transformações em árvores. O elemento básico de um programa em Stratego é denominado termo, que, por sua vez, pode ser compreendido como uma (sub) árvore sintática. Os termos podem ser agrupados em forma de árvores sintáticas abstratas ou concretas, e podem ser modificados por regras de reescrita estáticas, dinâmicas ou condicionais. O conjunto de ferramentas XT auxiliam a construção de meta programas e transformações de programas, incluindo suporte à definição da sintaxe de linguagens e utilitários para gerar uma infraestrutura base de transformações a partir das definições sintáticas (como *parsers* e *pretty-printers*). A principal limitação do StrategoXT está relacionada á dificuldade de se configurar o ambiente para sua utilização, algo que vai de encontro a uma tendência do desenvolvimento de *languages workbenches* [9], que suportam a geração de toda uma infraestrutura de disponibilização de ferramentas—incluindo *IDEs* ricas.

Spoofax é um projeto que surgiu em 2007, motivado pela necessidade de um *IDE* para desenvolvimento de *DSLs* utilizando as linguagens Stratego e SDF (*Syntax Definition Formalism*). Na verdade, esse esforço tinha como objetivo mitigar um problema relacionado à falta de ferramentas mais amigáveis para apoiar a implementação de ferramentas de metaprogramação. O ambiente Spoofax foi concebido como um conjunto de plugins para a plataforma Eclipse, incorporando um editor para Stratego e SDF, um interpretador para essas linguagens, e características como realce de sintaxe personalizável, análise semântica do código escrito, navegação pelo código e completar automaticamente elementos de sintaxe [10]. Spoofax pode também ser estendido por meio de scripts escritos em Stratego para adição de novas funcionalidades ao *IDE* e utilizados para criar novos plugins para *DSLs* desenvolvidas no próprio ambiente Spoofax [11].

RASCAL é uma *DSL* para metaprogramação, voltada para resolução de problemas relacionados a análise e manipulação de código [12]. O desenho da linguagem tem como objetivo auxiliar a implementação de ferramentas para análise e transformações de programas. A linguagem RASCAL possui um conjunto extenso de tipos algébricos, suporte a compreensão de listas, expressões regulares, parser de árvores sintáticas abstratas e concretas, elementos de paradigma funcional e regas de reescrita. Seu sistema de tipos suporta mecanismos de inferência tipicamente encontrados em linguagens com HASKELL e ML, incluindo conceitos como imutabilidade de dados e verificação de construções bem formadas. Esses recursos fornecem maiores garantias de corretude durante a implementação de transformações de programas. Por último, por seguir uma sintaxe próxima a da linguagem Java, que é amplamente difundida e de fácil aprendizagem, e estar integrada em um IDE baseada na plataforma Eclipse, RASCAL torna-se uma alternativa interessante para a implementação de meta programas.

Durante o desenvolvimento desta pesquisa, e com base em uma experiência em uma disciplina de transformações de programas, foi possível perceber que a linguagem e o ambiente de programação RASCAL estão estáveis e apresentando recursos bastante avançados para a manipulação de linguagens. Em particular, a implementação do padrão de projeto *Visitor* [13] como construção de alto nível da linguagem, e a possibilidade de casamento de padrões com elementos da sintaxe concreta das linguagens alvo da transformação, simplificam a escrita de transformações de programas. Dessa forma, em um momento inicial, foi possível implementar alguns cenários de refatoramento, de forma declarativa, e usando poucas linhas de código. Tais experiências nos levaram a optar pelo uso de RASCAL no desenvolvimento das transformações de código descritas no próximo capítulo.

Um exemplo de uso de RASCAL pode ser exemplificado pela implementação da transformação na Listagem 2.1, que transforma sentenças `if-then-else` em um único comando `return stmt`. Tal transformação, ilustrada na Figura 2.1, é útil para lidar com exemplos de código que podem ser trivialmente simplificados. Tal implementação de transformação espera como argumento uma unidade de compilação (`CompilationUnit`), que representa uma árvore sintática abstrata (AST) resultante da aplicação de um *parser* de um arquivo fonte na linguagem Java. A expressão `visit(unit)` realiza uma *travessia* nos nós da árvore, e, sempre que ocorre um casamento de padrão definido nas cláusulas `case`, uma transformação (descrita no lado direito do operador `=>`) é aplicada. No exemplo, duas cláusulas `case` estão definidas, representando as duas situações de interesse representadas na Figura 2.1. A transformação retorna uma nova unidade de compilação que contem as transformações de código eventualmente aplicadas no código fonte.

A Listagem 2.1 apresenta alguns aspectos relevantes da linguagem RASCAL. Por exemplo, a transformação correspondente faz uso da construção `visit` da linguagem.

Essa construção é simulada em linguagens orientadas a objetos com o uso do padrão de projeto *Visitor* [13]. Em mais detalhes, a construção `visit` corresponde a uma expressão, ou seja, possui um tipo (no caso o tipo é uma `CompilationUnit`) e, após sua aplicação, retorna um valor. O exemplo também ilustra o uso de *sintaxe concreta* para casamento de padrões. As cláusulas `case` da expressão `visit`, neste exemplo particular, fazem uso de exemplos válidos de sintaxe concreta para sentenças `if-then-else` na linguagem Java. Esse recurso, que permite lidar com a sintaxe concreta durante a travessia de uma AST simplifica, consideravelmente, a escrita de algumas transformações. Mais ainda, o suporte ferramental de Rascal indica situações em que um trecho de código não corresponde a uma sintaxe concreta na linguagem Java. Essa verificação é possível porque a especificação de uma sintaxe concreta é precedida de um elemento sintático da linguagem (no caso, o elemento sintático é `Statement`). Na definição da sintaxe da linguagem Java em Rascal, deve existir uma regra gramatical que permite derivar um `Statement` a partir de uma sintaxe concreta como:

```
1 if(<Expression cond>) { return true; }
2 else { return false; }
```

conforme descrita na primeira cláusula `case` da transformação implementada na Listagem 2.1. Outro recurso da linguagem Rascal é o suporte a *meta-variáveis* declaradas em um trecho de uma sintaxe concreta. A condição de uma sentença `if-then-else` em Java deve ser uma expressão. Tal expressão (`cond`) é usada tanto no lado esquerdo quanto no lado direito da transformação. Por essa razão, foi usada a declaração da *meta-variável* `<Expression cond>` na realização dos casamentos de padrão da transformação `TIfThenElse`, de tal forma que a mesma pudesse ser referenciada tanto do *lado esquerdo* quanto do *lado direito* da transformação.

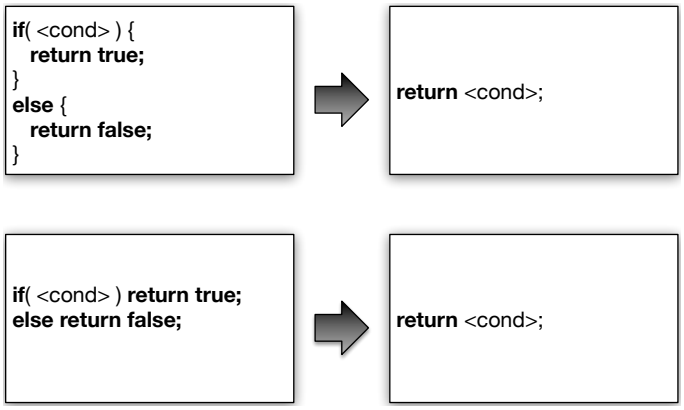

Figura 2.1: Exemplo de transformação envolvendo a construção `if-then-else`.

Listagem 2.1: Transformação de sentenças `it-then-else`

```
/**
 * Transform naive if statements. A quite simple
 * transformation based on the Rascal documentation.
 */
CompilationUnit TIfStatement(CompilationUnit unit) = visit(unit) {
  case (Statement) `if (<Expression cond>) { return true; }
                   else { return false; }`
       => (Statement) `return <Expression cond>;`
  case (Statement) `if (<Expression cond>)  return true;
                   else return false;`
       => (Statement) `return <Expression cond>;`
};
```

2.3 Métodos Empíricos em Engenharia de Software

Além da implementação de transformações de programas para apoiar os desenvolvedores a migrarem um código existente em direção a novas construções da linguagem de programação Java, esta dissertação também objetiva investigar, empiricamente, qual a aceitação das transformações por parte de desenvolvedores de projetos *open-source*. Esta seção descreve alguns estilos de estudos empíricos tipicamente encontrados em Engenharia de Software, usando como base o trabalho de Easterbrook et al. [14]— que apresentam um guia para seleção de métodos empíricos para pesquisa em Engenharia de Software. Além disso, esta seção caracteriza a avaliação empírica discutida no Capítulo 4.4

2.3.1 Experimentos Controlados

Método empírico reducionista que objetiva / permite estabelecer relações de *causa e efeito* entre variáveis que são rigorosamente controladas usando técnicas relacionadas a desenho de experimentos (incluindo aleatorização, replicação e bloqueio). Para obter o rigor necessário de controle das variáveis, experimentos são tipicamente realizados em sessões de laboratório utilizando tarefas que podem ser executadas em intervalos de tempo relativamente curtos (com aproximadamente duas horas). Em geral, experimentos são usados para identificar o efeito da adoção de alguma técnica, como por exemplo identificar se o uso de preprocessamento não disciplinado impacta no tempo para conduzir tarefas de manutenção de código [15]; se o uso de práticas ágeis, como programação em pares ou o desenvolvimento dirigido a testes, melhora a qualidade do software [16]; ou se o uso de abordagens composicionais favorecem a evolução de especificações de linhas de produtos de software [17]. Experimentos controlados se caracterizam (a) pelos custos razoavelmente altos para planejamento, execução e análise dos dados, (b) pela possibilidade de estabelecer formalmente relações da causa e efeito, e (c) por apresentar limitações relacionadas a generalização dos resultados.

2.3.2 Estudos de Caso

Método empírico observacional que, diferentemente dos experimentos controlados, são executados em um contexto real durante um intervalo de tempo que envolve tipicamente algumas semanas ou meses. Geralmente estudos de casos são conduzidos a medida que um projeto real de desenvolvimento é executado, e Easterbrook et al. consideram dois cenários típicos para condução de estudos de caso: (a) como investigação inicial de algum fenômeno, com o intuito de derivar novas hipóteses e construir teorias; e como uma investigação confirmatória, com o intuito de testar a validade de uma determinada teoria [18]. Estudos de caso apresentam como principais características a falta de controle para estabelecer relações entre causa e efeito e a possibilidade de generalizar os resultados para situações mais realistas. Importante ainda destacar que a terminologia *estudo de caso* muitas vezes é usada no sentido de um exemplo em que uma técnica é aplicada. Tal perspectiva é refutada pela comunidade de engenharia de software empírica. A abordagem conhecida como *pesquisa-ação* é considerada por alguns autores como um tipo particular de estudos de caso.

2.3.3 *Surveys*

Método empírico que tem como objetivo identificar as percepções de uma população sobre um objeto de estudo específico. Tipicamente, surveys são conduzidos utilizando questionários ou entrevistas semi-estruturadas. A escolha de uma amostra representativa da população e as técnicas de análise de dados usadas para tentar generalizar os resultados (como *Grounded Theory*) são dois desafios comumente citados durante a condução desse método de pesquisa. Um outro desafio tipicamente encontrado com a condução de surveys em pesquisas na área de Engenharia de Software é a baixa taxa de respostas. Por outro lado, mesmo considerando os desafios supracitados, a condução de surveys possibilita aos pesquisadores: (a) analisar de uma forma mais independente e sob a perspectiva de profissionais da área os custos / benefícios de uma determinada técnica; e (b) obterem novas percepções sobre um determinado problema. Por exemplo, por meio da condução surveys, Medeiros et al. analisaram as razões que levam os desenvolvedores C/C++ a usarem a técnica de preprocessamento, mesmo estando cientes dos problemas inerentes ao uso de tal tecnologia [19]; enquanto que Bonifácio et al. conseguiram identificar razões que inicialmente não tinham sido concebidas para fazer com que desenvolvedores evitassem o uso dos mecanismos de tratamento de exeções em C++ [20].

2.3.4 Método empírico a ser usado

Primeiramente, é importante destacar que essa pesquisa não tem interesse em estabelecer causa e efeito entre diferentes variáveis. Com isso, a estratégia de condução de experimentos controlados nesta pesquisa foi descartada. Além disso, não se pretende conduzir a pesquisa associada ao desenvolvimento de um projeto real de software—o que poderia introduzir riscos desnecessários. A primeira questão de pesquisa dessa dissertação é mais facilmente respondida com a realização de um survey baseado em pedidos de inclusão de código que foram reestruturados para usar novas construções da linguagem de programação Java. Quanto mais pedidos aceitos, melhor a percepção dos desenvolvedores quanto a reestruturação de código. Os pedidos serão feitos usando os mecanismos de *Pull-Request* disponíveis em ambientes como GITHUB e BITBUCKET. A segunda questão de pesquisa deve ser respondida comparando a completude do suporte ferramental implementado nesta dissertação com o suporte ferramental desenvolvido por um outro grupo de pesquisa. Curiosamente, esse método de pesquisa, mais baseado em mineração de repositório de software, não é discutido como um método empírico em referências como [18, 21]—apesar de estar se tornando bem explorado na literatura.

2.4 Trabalhos Relacionados

Esta dissertação está relacionada com trabalhos existentes que estudam o impacto da evolução de linguagens de programação em projetos de software existentes e como o suporte a transformações de programa pode auxiliar a evolução de sistemas legados para que os mesmos se mantenham mais atualizados em relação a novas construções de uma linguagem de programação.

Neste contexto, Overbey e Johnson discutem que linguagens de programação bem sucedidas evoluem naturalmente com o passar do tempo [1], se tornando mais complexas e com algumas características se tornando obsoletas e raramente usadas (como o velho estilo de se iterar sobre coleções usando a interface `Iterator` em Java). Em geral, quando uma linguagem de programação evolui, novas características são adicionadas, mas raramente removidas, fazendo com que as linguagens se tornem mais complexas e difíceis de se compreender. Neste contexto, Overbey e Johnson sugerem ferramentas de refatoramento automatizadas para eliminar o uso de construções e idiomas antigos de código fonte existente.

As necessidades em se atualizar um código legado para se adequar á evolução das linguagens de programação possuem certa relação com as necessidades em se manter um projeto atualizado em relação a novas versões de bibliotecas de software. Apesar de não existir um estudo que indique qual o percentual das mudanças de código, em

direçãp a novas construções de linguagens de programação, que poderia ser feito via refatoramentos automatizados, Dig e Johnson reportam que 80% das mudanças de API podem ser expressas como refatoramento passíveis de automação e que o código cliente (que faz uso das bibliotecas) pode ser também atualizado usando scripts e refatoramentos automatizados [22]. Dessa forma, alguns trabalhos apresentam ferramentas para migrar um código legado Java para usar *thread-safe collections* [23] ou para migrar uma base de testes unitários em JUnit 3 para JUnit 4 [24].

Trabalhos recentes descrevem técnicas de refatoramento automatizado para facilitar a migração de um código Java em direção ao uso de novas construções da linguagem. Por exemplo, Khatchadourian et al. apresentam uma estratégia para migrar constantes presentes em código legado Java (descritas como atributos estáticos e *final*) para enumeraões [25], introduzidas na versão 5 da linguagem Java (JSE 1.5). Gyori et al. descrevem um conjunto de heurísticas que devem ser observadas durante o refatoramento de *anonymous inner classes* e *enhanced for loops* para o uso de expressões lambda e composições de *streams* e discutem algumas decisões de projeto e implementação da ferramenta LambdaFicator, que automatiza esses tipos de refatoramento. Algumas das transformações desenvolvidas no contexto dessa dissertação (e descritas no próximo capítulo) observam essas heurísticas.

Finalmente, Khatchadourian e Masuhara discutem a mecânica de uma estratégia de refatoramento que possibilita introduzir *default methods* em interfaces Java, reduzindo a necessidade do uso do *skeletal implementation pattern* [26]. Os autores deste trabalho avaliam a aceitação de 19 *pull-requests* submetidos ao GitHub, com mudanças de código referentes à aplicação deste tipo particular de refatoramento. Um total de 20% dos *pull-requests* foram aceitos (50% não foram aceitos e 30% foram ignorados). A principal razão para a não aceitação, de acordo com os autores, é a necessidade de se manter a base de código dos projetos aderente a versões anteriores da linguagem Java.

Capítulo 3

Conjunto de Transformações Implementadas

Este capítulo apresenta detalhes de implementação de um conjunto de transformações de programas escritas usando Rascal. Essas transformações, ainda podem servir de base para que novos desenvolvedores não familiarizados com o Rascal poderem desenvolver as suas próprias transformações ou expandir as existentes. Conforme mencionado na Seção 2.1, uma série de construções foram adicionadas nas versões 7 e 8 da linguagem Java. Tais construções são o alvo desta pesquisa. Inicialmente, a Seção 3.1 apresenta um resumo das transformações, servindo como uma visão geral para simplificar a leitura do restante deste capítulo. Em seguida, na Seção 3.2, algumas das transformações implementadas são documentadas de acordo com um padrão que contempla o propósito da transformação, um exemplo real de aplicação, uma descrição em formato de *template* de transformação e uma enumeração das pré-condições e limitações da implementação atual. Finalmente, na Seção 3.3, algumas considerações relacionadas às transformações são apresentadas. Importante destacar que este capítulo apresenta algumas oportunidades de uso de transformações de código usando um projeto *open-source* como exemplo (o banco de dados não relacional Cassandra). As oportunidades de uso das transformações foram identificadas com o uso de um suporte ferramental de análise estática desenvolvido também no contexto desta pesquisa.

3.1 Resumo das Transformações Implementadas

Na essência, uma parte substancial dos resultados desta pesquisa envolve a escrita de transformações para suportar a evolução de programas Java em direção ao uso de construções recentes da linguagem, as quais não estavam sendo usadas porque, provavelmente, o trecho de código correspondente havia sido escrito antes da versão da linguagem que

tornou disponível a construção ou porque a equipe de desenvolvimento não possuía familiaridade com uma construção específica da linguagem Java.

Conforme discutido no capítulo anterior, os recursos de meta-programação da linguagem Rascal fizeram com que a mesma fosse escolhida para suportar a implementação das transformações. Por outro lado, no momento da escrita das transformações, só existia disponível publicamente a definição sintática da versão *Tiger* (JSE 1.5) da linguagem Java em Rascal. Dessa forma, **como primeira contribuição** dessa pesquisa, foi implementada uma nova definição de sintaxe para a versão 8 da linguagem Java (JSE 1.8) em Rascal. Como estratégia de desenho, em vez de reusar a definição anterior da sintaxe da linguagem Java, foi feita a opção por se criar a versão correspondente da sintaxe Java 8 tendo como base as especificações existentes do ANTLR e da ORACLE. Isso permitiu a construção da sintaxe em um período relativamente curto (aproximadamente 15 horas), mas que não faz uso de todos os recursos de desambiguação suportados pela linguagem Rascal. De qualquer forma, é importante destacar que a definição sintática criada como fruto deste trabalho de pesquisa reconhece aproximadamente 99.7% do código fonte de projetos não triviais (como o banco de dados textual Lucene). Como **segunda contribuição** alcançada deste trabalho, um conjunto de seis transformações para suportar a evolução de código Java foi implementado. São elas:

(T01) `SwitchString`: Essa transformação reestrutura um conjunto `if-else-if` que usa valores `string` como condição em uma sentença `switch`, cujas cláusulas referenciam os valores `string` correspondentes através de um teste de igualdade usando o métodos `equals`. Para essa transformação, a estrutura precisa fornecer ao menos um `if-else-if`. Como resultado, é retornado um bloco do tipo `switch-case` adicionado uma cláusula `default`, caso o *statement* original seja finalizada com um `else` simples (e não com um `else-if`).

(T02) `VariadicArgs`: Transformação que tem como objetivo explorar o uso da construção `varargs` em métodos que possuem como último parâmetro um `array` de objetos. Dessa forma, como premissa para aplicar essa transformação, é necessário que o último parâmetro do contrato do método seja um `array`. O resultado da transformação leva a uma construção similar aos `varargs` da linguagem C (tal construção não era suportada nas versões iniciais da linguagem Java).

(T03) `DiamondOperator`: Transformação que busca reduzir instanciações que referenciam explicitamente tipos parametricados, substituindo por um mecanismo (relativamente simples) de inferência de tipos suportado a partir da versão JSE 1.7. Para realizar a transformação, é necessário localizar atribuições que utilizam va-

riáveis parametrizadas como alvo e como fonte a instanciação de uma classe não parametrizada.

(T04) `MultiCatch:` Transformação cujo objetivo é agrupar cláusulas `catch` com corpo similares pertencentes a um mesmo bloco `try-catch`. Para isso, é necessário que as definições internas aos blocos `catch` sejam iguais (ou similares), podendo haver variação das classes utilizadas, desde que elas acessem um método definido por um ancestral em comum.

(T05) `AIC2Lambda:` Essa transformação converte a definição de *Anonymous Inner Classes* (AIC) em expressões *Lambda*. A aplicação dessa transformação requer uma análise rigorosa de seus pré-requisitos; sendo necessário verificar se o tipo referenciado na AIC é uma interface que possui apenas um método (ou seja, corresponde a uma *functional interface*), não pode ter membros estáticos e, caso haja alguma atributo declarado, deve ser `final`. Como resultado, obtém-se uma expressão *Lambda* que invoca o bloco interno ao método a ser implementado.

(T06) `Foreach2Lambda:` Conjunto de três transformações que convertem laços `foreach` em expressões lambda usando `streams`. De forma generalizada, esse tipo de transformação requer um maior rigor quanto a análise dos pré-requisitos envolvidos. No contexto das implementações atuais, estão sendo considerados apenas casos específicos relacionados a padrões recursivos típicos, como *exist*, *filter* e *map*.

3.2 Implementação das Transformações

Esta seção apresenta algumas decisões relacionadas à implementação das transformações usando a linguagem de meta-programação Rascal. O principal objetivo é apresentar as decisões de projeto mais relevantes.

3.2.1 Transformação `SwitchString`

Essa transformação tem o objetivo de transformar uma sequência de sentenças `if-else-if` que possuem como condição valores do tipo `String` por um comando condicional `switch-case`. O suporte ao tipo `String` em casamento de padrões envolvendo `switch-cases` foi introduzido na especificação JSE 1.7 – apesar de ser algo esperado há muito tempo, como pode ser observado por postagens no StackOverflow [27], por exemplo:

> "(`switch string` was introduced in Java 1.7), at least 16 years after they were first requested. A clear reason for the delay was not provided, but it likely had to do with performance."

Exemplo real de aplicação

No projeto Cassandra, o resultado de uma análise estática de código identificou 24 oportunidades para aplicar essa transformação, que juntas totalizam aproximadamente 500 linhas de código fonte (média = 22, mediana = 11 e desvio padrão = 28.7), mas com um exemplo particular envolvendo mais de 100 linhas de código. Para esse caso específico, os blocos das cláusulas `if-else-if` possuem uma quantidade significativa de linhas de código, levando a um cenário que mesmo a aplicação dessa transformação não leva a uma melhoria significativa quanto à facilidade de compreensão de código. A Listagem 3.1 apresenta um exemplo de código do projeto Cassandra que é passível dessa transformação; enquanto que a Listagem 3.2 apresenta o respectivo resultado da transformação.

Listagem 3.1: Exemplo de código passível da transformação `SwitchString`

```
1  if (compression.equals("snappy")) {
2     if (SnappyCompressor.instance == null)
3        throw new ProtocolException(UNKNOWN_ALGORITHM);
4     connection.setCompressor(SnappyCompressor.instance);
5  }
6  else if (compression.equals("lz4")) {
7     connection.setCompressor(LZ4Compressor.instance)
8  }
9  else {
10    throw new ProtocolException(UNKNOWN_ALGORITHM)
11 }
```

Listagem 3.2: Resultado após a aplicação da transformação `SwitchString`

```
1  switch (compression) {
2  case "snappy":
3     if (SnappyCompressor.instance == null)
4        throw new ProtocolException(UNKNOWN_ALGORITHM);
5     connection.setCompressor(SnappyCompressor.instance);
6     break;
7  case "lz4":
8     connection.setCompressor(LZ4Compressor.instance)
9     break;
10 default:
11    throw new ProtocolException(UNKNOWN_ALGORITHM)
12 }
```

Mecânica da Transformação `SwitchString`

A transformação `SwitchString` visita os nós de uma unidade de compilação com o objetivo de identificar sentenças `if-else-if` que possuem como condição uma expressão `<var>.equals(<string_value_01>)`. Caso todas as sentenças `else` subsequentes possuam apenas comandos `if(<var>.equals(<string_value>))` como sentenças, a transformação pode ser aplicada. Exceto o último `else`, que pode ter qualquer tipo de sentença dando origem a um caso `default`. De forma declarativa, a transformação é aplicada quando ocorre um casamento de padrões com uma sentença de acordo com a Listagem 3.3.

Listagem 3.3: *Template* de código que deve existir uma correspondência para a aplicação da transformação `SwitchString`.

```
1 if(<var>.equals(<string_value_1>)) { <block_1> }
2 else if(<var>.equals(<string_value_2>) { <block_2> }
3 else if(<var>.equals(<string_value_3>) { <block_3> }
4 //...
5 else if(<var>.equals(<string_value_n>) { <block_n> }
6 else <block>
```

Um trecho de código que realiza o casamento com o *template* da Listagem 3.3 deveria ser transformado de acordo com o *template* de transformação da Listagem 3.4. Nesta dissertação, metavariáveis são definidas entre os símbolos `<` e `>`.

Listagem 3.4: *Template* da transformação `SwitchString`

```
1 switch(<var>) {
2  case <string_value_1>: <block_1> break;
3  case <string_value_2>: <block_2> break;
4  //...
5  case <string_value_n>: <block_n> break;
6  default : <block>;
7 }
```

3.2.2 Transformação `VariadicArgs`

Introduzido na especificação JSE 1.5, o recurso *variadic args* (`varargs`) é um "*açúcar sintático*" que visa tornar mais flexível a definições de métodos que possuem como último parâmetro um `array` de elementos, de tal forma que o método passe a aceitar uma quantidade arbitrária de argumentos (que precisam ser do mesmo tipo base do `array` original). Para isso, é necessário que o último parâmetro da definição de um método seja um `array` de objetos.

Exemplo real de aplicação

Utilizando análise estática no código fonte do projeto CASSANDRA foram detectadas 49 oportunidades de refatoração para essa transformação. A listagem 3.5 apresenta um trecho de código que pode ser transformado por tal transformação, enquanto que a Listagem 3.6 mostra o resultado.

Listagem 3.5: Exemplo de código passível da transformação VariadicArgs

```
1 static TypeCodec<Object>[] codecsFor(DataType[] dataType)
2 {
3     TypeCodec<Object>[] codecs = new TypeCodec[dataType.length];
4     for (int i = 0; i < dataType.length; i++)
5         codecs[i] = codecFor(dataType[i]);
6     return codecs;
7 }
```

Listagem 3.6: Exemplo da transformação VariadicArgs

```
1 static TypeCodec<Object>[] codecsFor(DataType... dataType)
2 {
3     TypeCodec<Object>[] codecs = new TypeCodec[dataType.length];
4     for (int i = 0; i < dataType.length; i++)
5         codecs[i] = codecFor(dataType[i]);
6     return codecs;
7 }
```

Mecânica da transformação

Uma transformação para *variadic args* visita todos os nós de uma unidade de compilação buscando identificar definições de métodos cujo último parâmetro é um `array` de objetos. Quando essa pré-condição é validada, ocorre a aplicação da transformação. Declarativamente, a transformação é realizada quando ocorre um casamento de padrão de acordo com a Listagem 3.7, obtendo um código que segue a definição do padrão da Listagem 3.8.

Listagem 3.7: *Template* de padrão para aplicação da transformação VariadicArgs

```
1 <MethodDecl> <id> ( <pmtList> "," <type>[] <id> ) {
2   <BlockStmt>
3 }
```

Listagem 3.8: *Template* da transformação VariadicArgs

```
1 <MethodDecl> <id> ( <pmtList> "," <type>... <id> ) {
2   <BlockStmt>
3 }
```

3.2.3 Transformação `DiamondOperator`

Essa transformação tem o objetivo de transformar instanciações de objetos que explicitam tipos parametrizados em instanciações que fazem uso do mecanismo de inferência de tipos – a partir do tipo da variável que está sendo instanciada. O suporte ao uso do operador diamante `<>` foi introduzido na especificação JSE 1.7 Neste trabalho, são consideradas apenas declaração com atribuição na mesma instrução. Caso a variável seja declarada anteriormente, a transformação não é aplicada.

Exemplo real de aplicação

No projeto Cassandra, o resultado de uma análise estática de código identificou 72 oportunidades para aplicar essa transformação, sendo que algumas dessas oportunidades possibilitam simplificar de maneira significativa o entendimento de código e reduzir a quantidade de código duplicado para especificar os tipos concretos na instanciação de classes parametrizadas. A Listagem 3.9 apresenta um exemplo de código do projeto

Gradle que é passível dessa transformação; enquanto que a Listagem 3.10 apresenta o respectivo resultado do refatoramento.

Listagem 3.9: Exemplo de código passível da transformação `DiamondOperator`

```
1  Map<Class<? extends BuildOutcome>, BuildOutcomeComparator<?, ?>>
       comparators = new HashMap<Class<? extends BuildOutcome>,
       BuildOutcomeComparator<?, ?>>();
```

Listagem 3.10: Resultado após a aplicação da transformação `DiamondOperator`

```
1  Map<Class<? extends BuildOutcome>, BuildOutcomeComparator<?, ?>>
       comparators = new HashMap<>();
```

Mecânica da Transformação `DiamondOperator`

Em um alto nível, a transformação `DiamondOperator` visita os nós de uma unidade de compilação com o objetivo de identificar instanciações de construtores genéricos em uma declaração que possuam o tipo definido no construtor. Caso a variável declarada seja genérica com o tipo definido explicitamente no lado direto da atribuição, então a transformação pode ser aplicada. De forma declarativa, a transformação é aplicada quando ocorre um casamento de padrões com uma sentença de acordo com a Listagem 3.11.

Listagem 3.11: *Template* de código que deve existir uma correspondência para a aplicação da transformação `DiamondOperator`.

```
1  <Modifier?><Identifier><TypeArgs1><var> =
2     new <AnoType*><TypeArgs2>(<Args?>)
```

Um trecho de código que realiza o casamento com o *template* da Listagem 3.11 deveria ser transformado de acordo com o *template* de transformação da Listagem 3.12.

Listagem 3.12: *Template* da transformação `DiamondOperator`

```
1  <Modifier?><Identifier><TypeArgs1><var> =
2     new <AnoType*>\<\>(<Args?>)
```

3.2.4 Tranformação `Multicatch`

Essa transformação tem o intuito de transformar blocos `catch` similares em uma construção `multi-catch`, suportada a partir da versão 7 da linguagem Java. Esse recurso possibilita a redução de código duplicado.

Exemplos Reais de Aplicação

Realizando análise estática no código fonte do CASSANDRA foram identificadas 18 oportunidades de aplicação da transformação, sendo uma delas exemplificada na Listagem 3.13.

Listagem 3.13: Exemplo de oportunidade de aplicação da transformação MULTICATCH

```
1  private Object newInstance(Constructor<?> constructor, Object...
       args) {
2    try {
3      return constructor.newInstance(args);
4    }
5    catch (InstantiationException e) {
6      throw new IllegalArgumentException(e);
7    }
8    catch (IllegalAccessException e) {
9      throw new IllegalArgumentException(e);
10   }
11   catch (InvocationException e) {
12     throw new IllegalArgumentException(e);
13   }
14 }
```

Listagem 3.14: Aplicação da transformação multicatch no exemplo listado

```
1  private Object newInstance(Constructor<?> constructor, Object...
       args) {
2    try {
3      return constructor.newInstance(args);
4    }
5    catch (InstantiationException|IllegalAccessException|
         InvocationException e) {
6      throw new IllegalArgumentException(e);
7    }
8  }
```

Mecânica da Transformação

Todos os nós de uma unidade de compilação são verificados por um *visitor* que busca identificar blocos `try-catch`. Uma vez encontrado um bloco, é feita uma passagem pelos seus blocos `catch` para popular um `mapeamento` onde a chave é o conteúdo do bloco com a substituição do identificador usado na captura da exceção pelo identificador `e` e o valor corresponde ao conjunto de classes de exceções capturadas. Finalizada a visita do bloco `try-catch`, se algum valor das entradas do mapeamento tiver tamanho maior que um, o bloco é transformado para em multicatch.

Listagem 3.15: Estrutura de código capturada para aplicar a transformação.

```
1      (catch(<class> <identifier>\) <block>)+
```

Listagem 3.16: Map utilizado para realizar a análise

```
1  {
2      "<Block1>":{<class11>, <class12> ... ,<class1N>}
3      //...
4      "<BlockM>":{<classM1>, <classM2> ... ,<classMN>}
5  }
```

Listagem 3.17: Resultado da transformação

```
{
    result = 0
    foreach x | x <- {1..M} =>
      result ++
      ((catch(<classx1> | <classx2> ... | <classxN> e) <blockx>) |
      (catch (classx <identifierx>) <block2>)){1,M}
}
```

3.2.5 Transformação `AIC2Lambda`

Introduzido na versão 8 da linguagem Java (JSE 1.8), as expressões *Lambda* fornecem oportunidades para simplificar idiomas relativamente verbosos usados nas versões anteriores da linguagem, o que pode levar a trechos de código mais enxutos atualmente. O uso dessa transformação possibilita diminuir a necessidade de implementação de *Anonymous Inner Classes* (AICs) em algumas situações, desde que os pré-requisitos para a transformação `AIC2Lambda` sejam satisfeitos.

Exemplo real de aplicação

Utilizando recursos de análise estática de código no projeto CASSANDRA, foi possível detectar 63 oportunidades de aplicação dessa transformação. A listagem 3.18 apresenta um trecho de código desse projeto que é passível de transformação enquanto que a Listagem 3.19 apresenta o resultado correspondente da aplicação.

Listagem 3.18: Exemplo de código passível de transformação AIC para Lambda

```
ScheduledExecutors.optionalTasks.schedule(new Runnable()
{
  public void run()
  {
    convertLegacyData();
  }
}, AuthKeyspace.SUPERUSER_SETUP_DELAY, TimeUnit.MILLISECONDS);
```

Listagem 3.19: Exemplo de transformação AIC para Lambda

```
ScheduledExecutors.optionalTasks.
  schedule(() -> convertLegacyData();),
            AuthKeyspace.SUPERUSER_SETUP_DELAY, TimeUnit.
               MILLISECONDS);
```

Mecânica da transformação

De uma maneira geral, a transformação `AIC2Lambda` visita os nós de uma unidade de compilação com o objetivo de identificar instanciação de classes anônimas em chamadas de métodos. Uma vez identificada uma instanciação sob essas condições, é verificado se

a classe instanciada é uma interface, possui apenas um método, não possui declarações estáticas, utiliza apenas variáveis locais e todas as suas declarações são finais (constantes). Caso todas as premissas sejam verdadeiras, a transformação é aplicada. De forma declarativa, a transformação é aplicada quando ocorre um casamento de padrões conforme a Listagem 3.20 e as premissas são satisfeitas.[1]

Listagem 3.20: *Template* de código para aplicação da transformação AIC2Lambda

```
1 <expression>.<method>(<pmtList1> new <Interface>()
2   {
3     <method declaration>(<method arguments>) <methodBlock>
4   }
5 }, <pmtList2>);
```

Um trecho de código que realiza o casamento com o *template* da Listagem 3.20 e se adequa a todos os pré-requisitos é transformado de acordo com o *template* da Listagem 3.21.

Listagem 3.21: *Template* de transformação `AIC2Lambda`

```
1 <expression>.<method>(<pmtList1> (<method arguments>) -> <
    methodBlock>, <pmtList2> );
```

3.2.6 Tranformação `Foreach2Lambda`

A transformação `Foreach2Lambda` objetiva simplificar laços de repetição `foreach` em padrões recursivos que podem ser implementados como expressões *Lambda* combinadas com a interface de iteração Java Stream (que suporta vários padrões recursivos). Possíveis vantagens dessa transformação envolve a padronização do código fonte—uma vez que se espera um aumento no uso desse tipo de construção, e o suporte à concorrência durante a iteração nos elementos de uma coleção. Apesar de existirem diferentes estratégias para transformar laços de repetição `foreach` em expressões *Lambda* [5], no contexto dessa dissertação, foi tomada uma decisão inicial de suportar um subconjunto das situações em que essa transformação pode ser aplicada. Tal subconjunto compreende os seguintes padrões:

ExistPattern: varre uma coleção de objetos e, caso um predicado seja verdadeiro, a operação retorna `true`. Caso contrário, retorna `false`.

FilterPattern: filtra os objetos de uma coleção, de tal forma que os objetos remanescentes satisfaçam um determinado predicado. A operação não altera os elementos da coleção original e o resultado é uma nova coleção.

[1]Note que a Listagem 3.20 não apresenta a verificação das precondições.

MapPattern: aplica uma função (tipicamente expressa como uma expressão Lambda) a todos os elementos de uma determinada coleção. A operação não altera os elementos da coleção original e o resultado é uma nova coleção.

Exemplos Reais de Aplicação

No projeto Cassandra foram encontradas vinte e duas ocorrências de código que casam com o padrão de código do `ExistPattern` (Listagem 3.22). Juntas, tais ocorrências totalizam pouco mais de 100 linhas de código que (a) poderiam ser simplificadas significativamente e (b) levando a uma maior padronização do código. A Listagem 3.22 apresenta um exemplo de código do projeto Cassandra que é passível dessa transformação; enquanto que a Listagem 3.23 apresenta o respectivo resultado da transformação.

Listagem 3.22: Exemplo de código correspondente ao padrão `ExistPattern`

```
1 public boolean containsBindMarker() {
2    for (Term t : terms) {
3       if (t.containsBindMarker())
4          return true;
5    }
6    return false;
7 }
```

Listagem 3.23: Resultado após a aplicação da transformação `Foreach2Lambda` aplicada ao padrão `ExistPattern`

```
1 public boolean containsBindMarker() {
2    return terms.stream().anyMatch(t ->  t.containsBindMarker())
3 }
```

Também no projeto Cassandra, foram encontradas nove ocorrências de código que casam com o padrão de código do `FilterPattern` (Listagem 3.24) . Juntas, tais ocorrências totalizam pouco mais de 25 linhas de código. A Listagem 3.24 apresenta um exemplo de código do projeto Cassandra que é passível dessa transformação; enquanto que a Listagem 3.25 apresenta o respectivo resultado da transformação.

Listagem 3.24: Exemplo de código correspondente ao padrão `FilterPattern`

```
1 for (File snapshot : snapshotDirs) {
2   if (snapshot.isDirectory())
3      snapshots.add(snapshot);
4 }
5 return snapshots;
```

Listagem 3.25: Resultado após a aplicação da transformação `Foreach2Lambda` aplicada ao padrão `FilterPattern`

```
1 snapshotDirs.stream()
2             .filter(snapshot.isDirectory())
3             .collect(Collectors.toList())
```

Foram identificadas 67 ocorrências do padrão `MapPattern` no projeto Cassandra. Juntas essas ocorrências totalizam aproximadamente 228 linhas de código. A Listagem 3.26 apresenta um exemplo de código do projeto Cassandra que é passível dessa transformação; enquanto que a Listagem 3.27 apresenta o respectivo resultado da transformação.

Listagem 3.26: Exemplo de código correspondente ao padrão `MapPattern`

```
1 private List<String> stringify(Iterable<InetAddress> endpoints) {
2   List<String> stringEndpoints = new ArrayList<>();
3   for (InetAddress ep : endpoints) {
4     stringEndpoints.add(ep.getHostAddress());
5   }
6   return stringEndpoints;
7 }
```

Listagem 3.27: Exemplo de código correspondente ao padrão `MapPattern`

```
1 private List<String> stringify(Iterable<InetAddress> endpoints) {
2   return endpoints.stream()
3                   .map(ep -> ep.getHostAddress())
4                   .collect(Collectors.toList());
5 }
```

Mecânica da Transformação `Foreach2Lambda`

A transformação `Foreach2Lambda` visita os nós de uma unidade de compilação com o objetivo de identificar sentenças `foreach` que realizam o casamento com um dos três padrões recursivos apresentados anteriormente. De forma declarativa, a transformação é aplicada quando ocorre um casamento de padrões com uma sentença de acordo com a Listagens 3.28, 3.29 e 3.30.

Listagem 3.28: `Exist Pattern`

```
1 //...
2 for(<T e>: <collection>) {
3    if(<e.pred(args)>) {
4       return true;
5    }
6 }
7 return false;
```

Listagem 3.29: The Filter Pattern

```
1 for(<T e>: <collection>) {
2    if(<e>.<pred(args)>) {
3       <otherCollection>.add(e);
4    }
5 }
```

Listagem 3.30: The Map Pattern

```
for(<T e>: <collection>) {
   <stmts>
   <otherCollection>.<method>(e);
}
```

Um trecho de código que realiza o casamento com o *template* das Listagens 3.28, 3.29 e 3.30 deveria ser transformado de acordo com os *templates* de transformação exemplificadas nas próximas listagens Listagens 3.31, 3.32 e 3.33 .

Listagem 3.31: `Exist Pattern`

```
//might be replaced by:
return collection.stream().anyMatch(e->pred(args));
```

Listagem 3.32: The Filter Pattern

```
//might be replaced by:
collection.stream().filter(e->pred(args).forEach(e ->
    otherCollection.add(e));
```

Listagem 3.33: The Map Pattern

```
//might be replaced by:
collection.stream().forEach(e -> {
  <stmts>
  otherCollection.<method>(e);
});
```

3.3 Considerações Finais

As transformações discutidas neste capítulo foram implementadas em módulos `Rascal` que juntos totalizam aproximadamente 400 linhas de código, incluindo algumas funções utilitárias para realizar o *parser* de todos os arquivos fonte escritos na linguagem Java, por exemplo[2]. Algumas simplificações das transformações foram assumidas. Por exemplo, nem todas as oportunidades de transformação de laços `foreach` para expressões Lambda estão sendo consideradas. Essa simplificação leva a `falsos negativos` que foram investigados para responder a segunda questão de pesquisa deste trabalho. Uma simplificação como essa, leva a um código da transformação mais enxuto e mais fácil de compreender, por outro lado potencialmente menos expressivo. Mas, caso essa possível falta de expressividade ainda permita cobrir uma quantidade significativa de situações, reduzindo a quantidade de falsos negativos, então isso implica que uma redução da expressividade, não compromete a sua aplicação, além de facilitar a sua escrita da transformação.

[2]O código fonte do projeto está disponível em https://github.com/refactoring-towards-language-evolution/rascal-Java8

Capítulo 4

Avaliação Empírica

O objetivo do estudo empírico descrito neste capítulo é investigar a relevância de ferramentas de transformação de código para apoiar a evolução de linguagens de programação (conforme sugerido em [1]). Este capítulo primeiro apresenta uma descrição dos procedimentos seguidos, algo importante para permitir a replicação do objeto desta investigação no futuro uma melhor compreensão sobre a obtenção dos dados e sobre os resultados obtidos. O estudo foi concebido em três estágios:

Planejamento: Momento em que as questões de pesquisa e os procedimentos de investigação foram definidos.

Execução: Momento em que, no caso desta pesquisa, as transformações foram aplicadas e os resultados de aceitação foram colhidos.

Análise: Momento em que os resultados foram analisados e classificadas para responder às questões de pesquisa.

4.1 Questões de Pesquisa e Procedimentos

Conforme discutido, o principal objetivo desta investigação é compreender a importância de suporte ferramental de transformação de código para apoiar em atividades de evolução de sistemas legados para que os mesmos passem a usar construções mais atuais de linguagens de programação.

Visando analisar a importância de se usar ferramentas que apoiem a evolução de linguagem por meio de transformações de código fonte, as seguintes questões de pesquisa foram propostas.

(QP1) Desenvolvedores de projetos *Open Source* aceitam os resultados das transformações propostas em código legado?

(QP2) Quais razões levam desenvolvedores de projetos *Open Source* a rejeitarem os resultados das transformações de código propostas?

(QP3) Quais transformações propostas são mais aceitas pelos desenvolvedores de projetos *Open Source*?

(QP4) Qual a aplicabilidade das transformações propostas em projetos *Open Source* de alta relevância?

A primeira pergunta busca compreender como mantenedores lidam com o refatoramento de código em suas aplicações. Correção de código com defeitos e inserção de novas funcionalidades têm boa aceitação nas comunidades de projetos *Open Source*, mas não está claro na literatura qual a percepção e aceitação dos desenvolvedores de projetos *Open Source* quanto a refatoramentos.

Após submetida a transformação, ela será aceita? Senão, por quê? Que razões os mantenedores têm para não aceitar seus códigos evoluírem junto com a linguagem? Transformações localizadas ao invés de globais são mais facilmente aceitas? Ou esse fator não é importante?

Uma vez obtida as informações de aceitação, bem como seus motivos, pode-se indagar quais tipos de transformação são mais frequentemente aceitas. As submissões de transformações foram realizadas por tipo de transformação. Algumas são de fácil compreensão, como a que introduz o `DiamondOperator`, enquanto outras, a princípio, levam a um código mais enxuto (como `Foreach2Lambda`), mas podem prejudicar o entendimento do código.

Além do aspecto de aceitação de transformações, existe uma discussão sobre o uso de verificações mais rigorosas (que podem levar a falsos negativos) e verificações menos rigorosas (que podem levar a falsos positivos) das pré-condições de transformações de programas. As transformações implementadas levaram em conta dois fatores: (a) custos e prazos de desenvolvimento e (b) maior rigor na verificação das transformações. Ou seja, para equilibrar *segurança das transformações* com os recursos de custo e prazo de desenvolvimento, foi feita a opção por trabalhar de uma forma mais conservativa. Isso faz com que a aplicação das transformações implementadas e discutidas no capítulo anterior levem a uma quantidade significativa de falsos-negativos. Por essa razão, esse estudo investiga ainda se essa estratégia de conciliar custos e prazos com expressividade reduz, de forma significativa, a aplicabilidade das transformações; levando a uma quarta questão de pesquisa: *(QP5) o uso de abordagens mais conservativas de transformação leva a uma grande quantidade de falso-negativos?*

Para responder a essa última questão de pesquisa, foi feita uma comparação entre as transformações `Foreach2Lambda` e a implementação da ferramenta *Lambdaficator* [5], quanto ao potencial de aplicação em projetos *Open Source*.

4.1.1 Objetos de estudo

Para responder as questões de pesquisa estabelecidas, foram escolhidos 100 projetos da plataforma *GitHub* desenvolvidos em Java que possuíssem mais de 50.000 KB (o *GitHub* não trabalha com mega-bytes como unidade de medida) e outros 10 projetos adicionais que tivessem até esse tamanho para ser objeto das transformações *Foreach2Lambda*. Todos os projetos foram ordenados pela quantidade de *estrelas* que possuíam, sendo essa a unidade indicadora de popularidade do *GitHub*. Desta população, foram aleatoriamente selecionados 40 projetos para a aplicação das transformações, sendo importante observar que essa seleção foi realizada em maio de 2017. A relação dos projetos selecionados pode ser vista no anexo I.

Buscando identificar a aplicabilidade das transformações foram executadas as seis transformações em três projetos relevantes da comunidade *Open Source*: Cassandra, ElasticSearch e CoreNLP.

4.1.2 Procedimentos

Para investigar a aceitação das transformações por partes dos desenvolvedores de repositórios *Open Source*, o procedimento consistiu em realizar o refatoramento nos projetos e, em seguida, submeter as alterações via o mecanismo de *(*Pull-Request), que permite o desenvolvimento colaborativo de um sistema habilitando colaboradores a submeter as suas contribuições aos projetos. Por meio desta abordagem, foram aplicadas **2371 transformações em 40 projetos, totalizando 3053 linha de código modificadas em 1187 arquivos**. A relação completa dos *(*Pull-Requests) pode ser vista no anexo II.

Uma vez submetido o conjunto de alterações, a equipe de mantenedores decide se as aceitam ou não. Durante esse processo, podem ser discutidas melhorias que podem ser agregadas à requisição para ela ser aceita ou informado o motivo de rejeição. Caso aceita, a contribuição é integrada ao código original. Fazendo uso desse mecanismo, é possível ter uma visão de como é a aceitação das transformações que visam evoluir o código legado de um sistema, assim como os motivos pelos quais elas são rejeitadas. É também com o suporte do mecanismo de *pull request* que se torna possível elencar as transformações mais aceitas, assim como os motivos que as levam à aceitação.

Por fim, a expressividade das transformações envolvendo *enhanced for loops* to *lambda expressions* foi computada usando a quantidade de falso-positivos e falso-negativos, com a aplicação das transformações propostas e das transformações implementadas no Lambdaficator, considerando 5 dos projetos alvos para as transformações. Aqui, considera-se um falso-positivo uma transformação aplicada usando as transformações propostas e não aplicada pelo Lambdaficator. Diferentemente, um falso-positivo ocorre quando uma

transformação proposta não é aplicada; mas é aplicada pelo LAMBDAFICATOR. Falso-positivos pode revelar a existência de um erro nas implementações descritas nessa dissertação; enquanto que falsos-negativos revelam um maior rigor na implementação.

4.2 Execução da Pesquisa

Conforme discutido nos capítulos anteriores, as versões 7 e 8 da linguagem Java introduziram diversas construções com potencial de refatoramento em código legado. Elencadas algumas dessas construções, foi iniciado o processo de construção das respectivas transformações. Em seguida, foi realizada a condução do estudo empírico, que, após a definição das questões de pesquisa e dos procedimentos, envolveu a identificação e recuperação dos projetos que caracterizam a população de interesse. Por meio de um *script Python* que recupera as informações dos projetos no GitHub, foram *clonados* os 100 projetos considerados mais populares na linguagem Java e que tinham um tamanho maior que 50.000 KB. Usando um outro *script Python*, foi gerada uma amostra de 40 projetos da população original, e definidas as taxas de aplicação de transformação de (valores 10%, 30%, 50%, 70% ou 90%) em relação às oportunidades encontradas. O resultado foi então exportado para um arquivo CSV*(Comma-Separated Values)* que serve como *input* para um módulo RASCAL (`Driver`) que guia todo o processo de aplicação das transformações e coleta métricas (como a quantidade de transformações e o tempo de processamento) das execuções.

Para a verificação de aplicabilidade, foram executadas todas as transformações de maneira separada para cada projeto com taxa de aplicação fixada em 90%. Isso possibilitou ter uma ideia mais geral sobre a utilização dos novos recursos da linguagem além de poder averiguar a taxa de falsos positivos gerados a partir deles. Levando em consideração os falsos positivos, eles são contabilizados quando emergem erros ao ser executada a construção do projeto seguindo as diretivas apontadas pelo repositório do projeto para efetuar a sua compilação. Tal abordagem é possível porque todos os erros acarretados pelas transformações desenvolvidas neste trabalho podem ser verificados em tempo de compilação.

Após a aplicação das transformações, tornou-se necessário efetuar uma atividade de correção de possíveis erros, antes da submissão dos *pull-request*—uma vez que a implementação atual das transformações ainda leva a falsos positivos, o que pode resultar inclusive em erros de compilação do projeto. Em situações como essa, a correção envolve, por exemplo, restaurar o código fonte para o estado anterior à aplicação da transformação. Essa etapa de melhoria é repetida até que os resultados das transformações não apresen-

tem falsos positivos. O procedimento é repetido para todos os tipos de transformação e pode ser melhor compreendido considerando a Figura 4.1

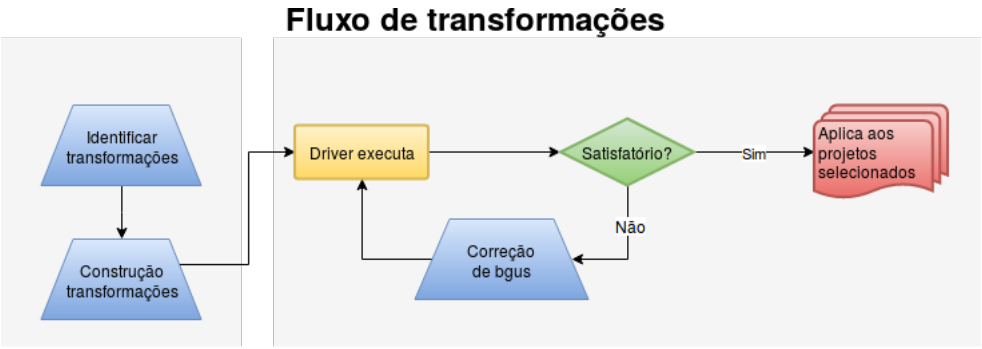

Figura 4.1: Fluxo de aplicação das transformações desenvolvidas.

A aplicabilidade das transformações é verificar analisando a quantidade de transformações com o número de erros. Feita essa relação, é possível identificar as transformações que necessitam ser aperfeiçoadas.

Para análise da expressividade das transformações Foreach2Lambda, foram considerados os projetos: *Hystrix, Interviews, java-design-patterns, retrofit e Dubbo*. Para sua execução são comparadas as transformações efetuadas pelas ferramentas *Foreach2Lambda* e *Lambdaficator* e comparados os resultados das duas aplicações, onde são consideradas as transformações aplicadas por ambas as ferramentas, e exclusivamente por uma das delas.

4.3 Resultados

Esta seção apresenta os resultados obtidos a partir das respostas aos *pull-requests*, com o intuito de responder às questões de pesquisa. As respostas aos *pull-requests* podem ser classificados de 3 maneiras diferentes:

- Submissões Aceitas
- Submissões Ignoradas
- Submissões Rejeitadas

O restante desta seção apresenta considerações sobre os *pull-requests* conforme tal classificação.

4.3.1 Submissões Aceitas

Representam as transformações que foram aceitas pela comunidade e incorporadas ao projetos aos quais foram submetidas. Foram aceitas 10 requisições, o que totaliza 336

transformações com 373 linhas modificadas em 132 arquivos; sendo 8 delas transformações envolvendo o operador diamante e 2 relacionadas à transformação *MultiCatch*. Como exemplo de uma submissão aceita, temos uma transformação de *MultiCatch* na Listagem 4.1, onde 2 cláusulas catch são unidas em uma única instrução.

Listagem 4.1: Exemplo de transformação aceita

```
1  //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> In
2  try {
3      name = URLDecoder.decode(matcher.group(1),"8859_1");
4      value = URLDecoder.decode(matcher.group(2),"8859_1");
5      cell[0] = name;
6      cell[1] = value;
7      listPair.add(cell);
8  } catch (UnsupportedEncodingException | IllegalArgumentException e
       ) {
9      // ZAP: Log the exception
10     logger.error(e.getMessage(), e);
11 }
12 //==============================
13 try {
14     name = URLDecoder.decode(matcher.group(1),"8859_1");
15     value = URLDecoder.decode(matcher.group(2),"8859_1");
16     cell[0] = name;
17     cell[1] = value;
18     listPair.add(cell);
19 } catch (UnsupportedEncodingException e) {
20     // ZAP: Log the exception
21     logger.error(e.getMessage(), e);
22 } catch (IllegalArgumentException e) {
23     // ZAP: Log the exception
24     logger.error(e.getMessage(), e);
25 }
26 //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Out
```

4.3.2 Submissões Ignoradas

Representam as transformações que foram ignoradas pela comunidade, motivadas pelo fato do projeto ter sido descontinuado ou pela falta de interesse dos mantenedores em incorporar as transformações. Foram ignoradas 9 requisições, o que totaliza 252 transformações com 371 linhas modificadas em 154 arquivos—sendo 3 delas transformações do operador diamante, 2 de funções Lambda, 4 de MultiCatch. Dentre os projetos transformados, um projeto foi descontinuado e oito não tiveram interesse nos refatoramentos, uma vez que houve integração de código oriundo de outros *pull-requests* aceitos em datas próximas a desse trabalho.

Como exemplo de uma submissão ignorada, tem-se uma transformação de *Foreach2Lambda* na Listagem 4.2, onde 2 cláusulas *catch* são unidas em um único bloco.

Listagem 4.2: Exemplo de Transformação ignorada

```
1  //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> In
```

```
for(ArrayList<Integer> subset : allsubsets) {
    ArrayList<Integer> newsubset = new ArrayList<Integer>();
    newsubset.addAll(subset);
    newsubset.add(item);
    moresubsets.add(newsubset);
}
//==============================
allsubsets.stream().map(subset -> {
    ArrayList<Integer> newsubset = new ArrayList<Integer>();
    newsubset.addAll(subset);
    return newsubset;
}).map(newsubset -> {
    newsubset.add(item);
    return newsubset;
}).forEach(newsubset -> {
    moresubsets.add(newsubset);
});
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Out
```

4.3.3 Submissões Rejeitadas

Correspondem às submissões que não foram aceitas, mas que ao, contrário das ignoradas, houve um discussão com a equipe de mantenedores do projeto. Nessa categoria, foram realizadas transformações em 21 projetos, totalizando 1783 transformações nas quais houveram 2309 linhas modificadas em 901 arquivos. Como exemplo de uma transformação rejeitada, temos a Listagem 4.3, onde é adicionado o operador diamante e o tipo dos objetos instanciados é definido de forma implícita.

As razões pelas quais essas requisições foram rejeitadas podem ser agrupadas em 3 grupos: *pull-requests* sumariamente rejeitado pela equipe de desenvolvimento, *pull-requests* rejeitados porque o projeto utiliza uma versão anterior do Java (talvez devido as suas dependências), e *pull-requests* que foram deixados em aberto para tentativa de integração futura, quando o projeto tenha evoluído até a versão onde os recursos transformados possam ser aceitos.

Listagem 4.3: Transformação ignorada

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> In
    Map<ResolvableArtifact, TransformArtifactOperation>
        artifactResults = new ConcurrentHashMap<>();
    Map<File, TransformFileOperation> fileResults = new
        ConcurrentHashMap<>();
//==============================
public Completion startVisit(BuildOperationQueue<
    RunnableBuildOperation> actions, AsyncArtifactListener listener
    ) {
    Map<ResolvableArtifact, TransformArtifactOperation>
        artifactResults = new ConcurrentHashMap<ResolvableArtifact,
         TransformArtifactOperation>();
    Map<File, TransformFileOperation> fileResults = new
        ConcurrentHashMap<File, TransformFileOperation>();
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Out
```

Submissões deixadas em aberto para integração futura

Nesse caso, a equipe de desenvolvimento se mostrou aberta a discussão e não encerrou o pedido de mudanças via *pull-request*. Assim, a requisição continua disponível para integração quando o projeto estiver utilizando uma versão compatível com as mudanças acarretadas pela transformação. Esse tipo de resposta foi identificada em 3 requisições (*pull-request*) que resultaram em 40 transformações e 129 linhas de código divididas em 38 arquivos.

Submissões sumariamente rejeitadas

Nesse caso, a equipe de desenvolvimento não se mostrou aberta a discussão e encerrou a requisição de mudanças. Os principais motivos que levaram a essa decisão foram:

- Refatoramento em arquivos onde era indesejado pela equipe, como arquivos de testes. Um dos mantenedores do projeto *CheckStyle*, com login *romani* no GitHub, escreveu: "Arquivos de teste devem permanecer inalterados. Quanto mais feio, melhor."
- Incorporação do projeto em um projeto maior, como o projeto *Storm*, que foi migrado para o repositório do *Apache*, e foi dito que pedidos de *pull-request* só seriam analisados se submetidos para o novo repositórios.
- Burocracia – Alguns dos projetos exigiam que possíveis contribuidores executassem diversos procedimentos externos à ferramenta para que as requisições fossem analisadas.

Esse tipo de resposta foi identificado em 5 requisições (*pull-requests*) que resultaram em 197 transformações em 324 linhas espalhadas em 88 arquivos.

Submissões rejeitadas devido a dependências

Nesse caso, as submissões são encerradas devido o projeto ter dependências com versões anteriores da linguagem Java, mas sem previsão de atualização para versões mais recentes. Em alguns dos casos, o mecanismo de compilação de testes relata erro por forçar construções específicas da versão especificada, mas, em outros casos, todos os testes internos do projeto são executados com sucesso, mas mesmo assim é mantida apenas código legado devido a equipe mantenedora acreditar que as novas construções da linguagem Java não agregam valor ao projeto, como é o caso do mantenedor *EugenHotaj* do projeto *Interviews*. Foram realizadas 13 submissões (*pull-requests*) que resultaram em 1546 transformações em 1856 linhas espalhadas em 775 arquivos que retornaram esse quadro.

4.3.4 Análise de aplicabilidade das transformações

Em resposta à QP4 foram executadas as seis transformações nos projetos Cassandra, ElasticSearch e CoreNLP totalizando 2365 transformações, em 6157 linhas de 1178 arquivos, onde 2025(85,66%) foram efetuadas sem erros enquanto 339(14,34%) geraram código quebrado(falsos positivos). Verificando os dados das transformações para cada um dos projetos temos que no Cassandra, conforme Tabela 4.1 é possível visualizar uma excelente taxa de transformações realizadas com sucesso para DI, MC e VA (100%) enquanto que SS, AC e FUNC possuem uma taxa normalizada de apenas 48% de sucesso.

Tabela 4.1: Transformações no projeto Cassandra

Transformação	Linhas	Arquivos	Sucesso	Fracasso
DI	287	106	287	0
MC	31	5	9	0
VA	130	89	130	0
SS	29	2	2	1
AC	752	52	104	24
FUNC	875	73	96	30

Já quando considerando o ElasticSearch, embora o resultado de DI, MC e VA seja mantido(100%), SS, AC e FUNC apresentam um queda ainda maior em relação ao Cassandra(48%), segundo Tabela 4.2 1

Tabela 4.2: Transformações no projeto ElasticSearch

Transformação	Linhas	Arquivos	Sucesso	Fracasso
DI	23	18	23	0
MC	23	6	6	0
VA	219	117	219	0
SS	277	18	21	4
AC	87	4	13	0
FUNC	828	75	97	140

Para CoreNLP o resultado foi obtida 98% de sucesso nas transformações DI, MC e VA, ao passo que para SS, AC e FUNC, esse valor permeia os 60%, conforme Tabela 4.3

4.3.5 Análise de expressividade das transformações `Foreach2Lambda`

Para verificar a relevância das transformações desenvolvidas foi realizada uma comparação dos resultados da aplicação das transformações `Foreach2Lambda` e Lambdaficator, levando em consideração quando ela era efetuada apenas pelo `Foreach2Lambda` (falso-positivo), apenas pelo Lambdaficator (falso-negativo) e quando era realizada pelas

Tabela 4.3: Transformações no projeto CoreNLP

Transformação	Linhas	Arquivos	Sucesso	Fracasso
DI	134	46	132	0
MC	96	16	24	20
VA	678	117	678	0
SS	122	18	7	17
AC	83	4	16	30
FUNC	1483	75	161	73

duas ferramentas. A Listagem 4.4 mostra um caso onde ambas as ferramentas identificaram uma oportunidade de refatoramento de um bloco *enhanced for loop* para expressões lambda. Foi observado que o Lambdaficator pode sugerir um arquivo em branco como transformação, mas isso pode ser resolvido descartando a sugestão e inspecionando o código fonte novamente.

Listagem 4.4: transformação desenvolvida e Lambdaficator com resultado semelhante

```
1 //============================== Foreach2Lambda
2           subscribers.forEach(subscriber -> {
3             subscriber.assertNoEvents();
4           });
5 //============================== Lambdaficator
6           subscribers.stream().forEach((subscriber) -> {
7               subscriber.assertNoEvents();
8             });
9 //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Entre as aplicações das transformações por parte do `Foreach2Lambda` e Lambdaficator foram verificados os resultados obtidos para os projetos *Retrofit*, *Java-design-patterns*, *Interviews*, *Hystrix* e *Dubbo*. Os resultados podem ser consultados na Tabela 4.4.

Tabela 4.4: Total de transformações por projeto e tipo.

	Ambas	Foreach2Lambda	Lambdaficator	Total
Retrofit	6	1	2	9
Java-design-patterns	11	4	20	35
Interviews	0	3	3	6
Hystrix	17	2	79	98
Dubbo	16	1	310	327
Total	50	11	414	475

É possível observar uma quantidade significativa de *falsos negativos* nos projetos *Hystrix* e *Dubbo* e que o número de *falsos positivos* é relativamente baixo. Isso é um reflexo direto da opção conservativa, que reduz os custos de desenvolvimento e leva a aplicação das transformações para um conjunto de casos bem definidos, sugerindo que existe um espaço significativo de melhorias nas implementações das transformações.

4.4 Consolidação dos Resultados

Para responder as questões de pequisa, foi realizada a análise dos dados inicialmente discutidos nas seções anteriores. Analisando cada uma das transformações, foi possível identificar que a transformação do operador diamante obteve uma melhor aceitação das submissões (aproximadamente 50% dos *pull-requests* foram aceitos, conforme apresentado na Figura 4.2). Para essa transformação, não foi registrada nenhuma submissão em aberto para integração futura. Esse é um aspecto positivo, uma vez que uma requisição deixada para integração posterior pode ser completamente inviabilizada devido a possíveis mudanças no código fonte.

Figura 4.2: Resultado de submissões da transformação Operador Diamante

Considerando a transformação `MultiCatch`, embora não haja qualquer submissão rejeitada sumariamente, uma quantidade significativa de *pull-requests* submetidos foram ignorados, o que sugere que a comunidade não considere esse tipo de transformação relevante, e, embora tenha havido pouca aceitação, elas ainda representam o dobro dos motivos de rejeição (Figura 4.3).

De forma um tanto surpreendente, os pedidos de *pull-request* envolvendo transformações para utilização de expressões Lambda foram todos rejeitados, onde não aconteceu integração desse tipo de transformação e o resultado mais favorável foi deixar a requisição em aberto para possível integração futura. Mesmo assim, a situação de integração futura chega a 15% do total de requisições submetidas (Figura 4.4). Com um total de rejeição superior a 80%, pode-se considerar que essa transformação é completamente ignorada pela comunidade.

Analisando o resultado das transformações em conjunto, conforme as Figuras 4.5 e 4.6, é possível observar que a de aceitação de refatoramento de código fonte com objetivo de

Figura 4.3: Resultado de submissões da transformação *Multicatch*

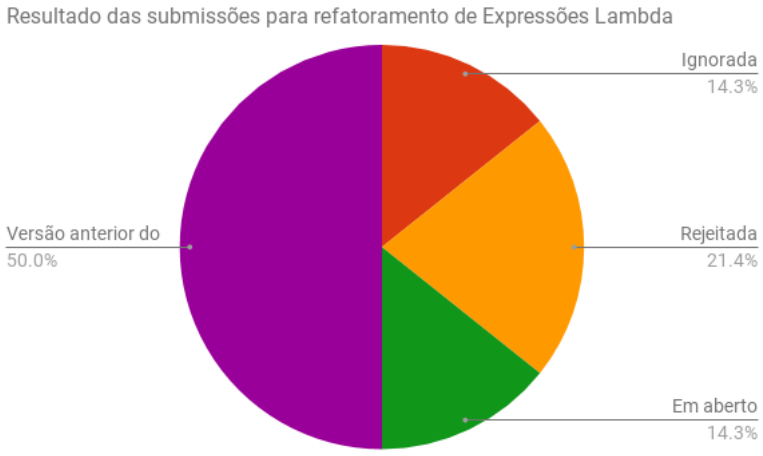

Figura 4.4: Resultado de submissões da transformação *Forech2Lambda*

evoluir código legado, ou seja, sem adição de novas funcionalidades, está longe de se comparar à adição de novas funcionalidades ou correção de falhas no sistema.

Uma possível justificativa é que existe certa resistência à atualização da versão alvo da linguagem em que é gerado o código objeto das aplicações. Durante a pesquisa, foi identificado que vários projetos (todos os que foram rejeitados devido à dependência de uma versão mais antiga da linguagem) configuram as ferramentas de *build* (como o *Maven* ou o *Gradle* com o suporte a versão da linguagem Java restrito às versões JSE 1.5 ou JSE 1.6, impedindo o uso das construções mais recentes da linguagem Java.

Dentre os motivos levantados por essa rejeição está a incompatibilidade com as dependências do projeto. Esse é um aspecto curioso, pois desde a segunda versão da linguagem Java, não houve remoção de construções na linguagem, apenas adições e recomendações para seguir as novas maneiras de desenvolvimento. Além disso, como essas aplicações tem compilação assistida, se as dependências fossem atualizadas, o código objeto gerado seria atualizado de maneira transparente para os desenvolvedores e usuários finais.

Outro possível motivo para a rejeição das transformações é a insegurança por parte dos mantenedores com construções mais recentes, sobretudo em código fonte para automação dos testes. Aceitação de modificações em testes não foram vistas com a mesma apreciação que em outras partes de código. Finalmente, um último motivo que pode afetar a aceitação de uma transformação é o tamanho da submissão. De todas as submissões enviadas, apenas 1 com mais de 50 transformações foi aceita pela comunidade, e esse valor aumenta significativamente a medida que o tamanho das transformações é reduzido, conforme Tabela 4.5. Isso responde a QP1 quanto aceitação das transformações e QP2, que busca os motivos de rejeição. Quanto a QP3, a transformação com maior índice de aceitação foi a *Diamond Operator*, seguida da *MultiCath* e *Foreach2Lambda*.

Esses resultados sugerem que a aceitação de uma transformação está relacionada com a dificuldade de compreensão e o tamanho do *pull-request*. Ou seja, um desenvolvedor inexperiente pode aplicar facilmente o recurso do operador diamante, ao passo que apenas programadores mais experiente devem conseguir utilizar expressões Lambda em código fonte Java. De forma semelhante, um *pull-request* que afeta vários arquivos fonte é difícil de se compreender, e consequentemente aprovar.

Quando considerada a aplicabilidade das transformações, como poder ser visto pela Figura 4.7, é possível inferir que DI e VA tem uma excelente aplicabilidade e MC razoável. Entretanto SS, AC e FUNC necessitam de melhorias em suas transformações haja visto que a taxa de falsos positivos para elas está muito elevada(superior a 27% para todos os casos).

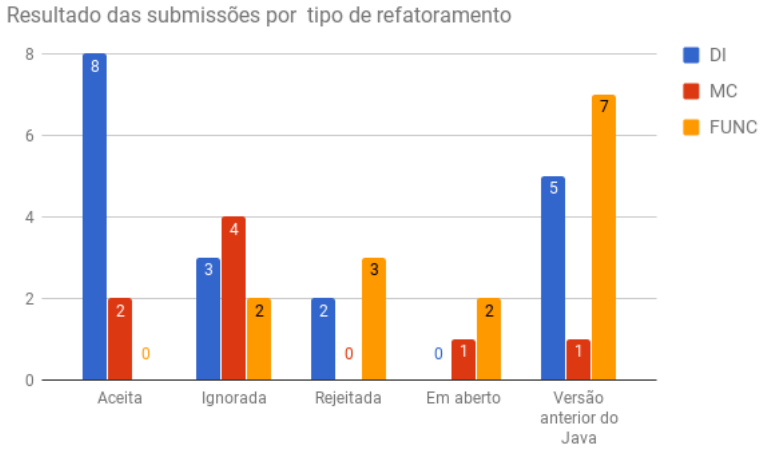

Figura 4.5: Resultado de submissões da refatoramento

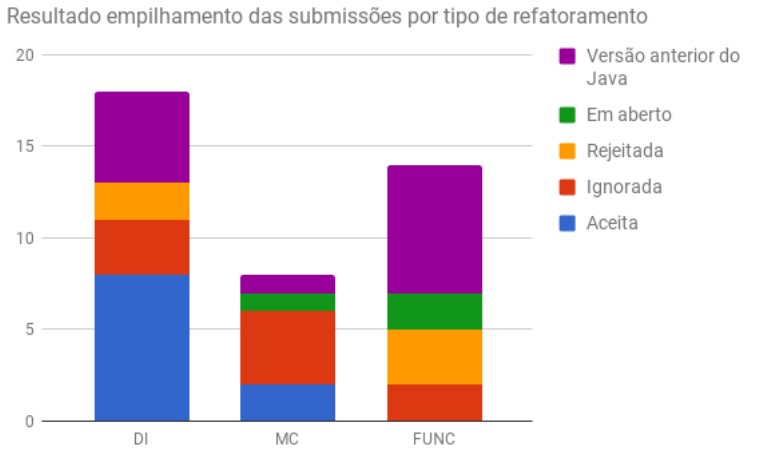

Figura 4.6: Resultado acumulado de submissões da refatoramento

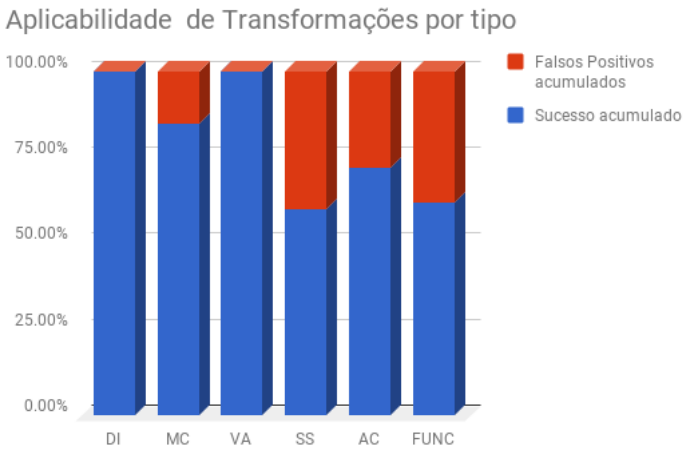

Figura 4.7: Aplicabilidade acumulada por tipo de transformação

Tabela 4.5: Aceitação por quantidade de transformações.

Quantidade de transformações	quantidade de requisições aceitas
Mais que cem transformações	1
Mais que cinquenta	2
Mais que vinte	4
Mais que dez	5
Mais cinco	9
Mais que uma	10

Capítulo 5

Conclusão

Este capítulo apresenta algumas considerações finais relacionadas à condução desta pesquisa, destacando as principais contribuições deste trabalho (Seção 5.1), as limitações e ameaças da pesquisa (Seção 5.2) e algumas sugestões de trabalhos futuros (Seção 5.3).

5.1 Contribuições

O foco dessa pesquisa é relacionado à implementação de transformações de código para apoiar a migração de código legado em direção ao uso de novas construções da linguagem Java. Neste cenário, as principais contribuições desse são:

- A escrita de uma nova gramática Java (aderente à especificação JSE 1.8) na linguagem Rascal, que reconhece mais de 99% do código fonte de projetos não triviais.
- Um conjunto de transformações em Rascal para apoiar na evolução de código fonte legado. A linguagem Rascal se mostrou muito efetiva para lidar com transformações que lidam exclusivamente com elementos sintáticos. Quando uma transformação requer análise de tipos (por exemplo), os benefícios de Rascal não se tornam muito evidentes.
- A condução de um estudo empírico com intuito de responder se a comunidade de projetos *open-source* é receptiva a transformações de programas que evoluem o código fonte. Diferentemente de outros trabalhos de pesquisa [28, 29], que aplicavam uma única transformação via *pull-request* por projeto, aqui foram aplicadas várias transformações por projeto. Essa estratégia levou a uma baixa quantidade de aceitações de *pull-requests*. Duas possibilidades emergem: ou esse tipo de transformação não é relevante—o que vai de encontro à perspectiva de Overbey e Johson [**?**]- ou a estratégia de avaliação, aplicando várias transformações no mesmo projeto e com *pull-requests* relativamente complexos, não foi adequada.

Outra contribuição é a comparação da expressividade de um conjunto de transformações implementadas nesta dissertação com a de outro software (Lambdaficator). Foi possível perceber um aspecto razoável de melhoria na transformação *Foreach2Lambda*. Embora as transformações dessa dissertação capturem casos que o Lambdaficator não detecta (exemplo na Listagem 5.1), a quantidade de falsos negativo foi significativa. Apenas como observação, não foi possível identificar com exatidão o porque do Lambdificator não conseguir capturar alguns dos cenários reais de transforamção. Finalmente, dada a estrat'egia conservativa das implementações desta dissertação, algumas transformações conseguiram atingir uma taxa próxima a zero de falsos positivos.

Listagem 5.1: Transformação aplicada apenas por Foreach2Lambda

```
1  //============================== Foreach2Lambda
2  cand.stream().map(str -> {
3      cur.add(str);
4      return _item;
5  }).map(_item -> {
6      helper(ret, cur, matched+1, total, map);
7      return _item;
8  }).forEach(_item -> {
9      cur.remove(cur.size()-1);
10 });
11 //============================== Lambdaficator
12 for(String str:cand){
13     cur.add(str);
14     helper(ret, cur, matched+1, total, map);
15     cur.remove(cur.size()-1);
16 }
17 //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

5.2 Limitações do Trabalho

As transformações implementadas nessa dissertação seguiram uma estratégia conservativa de desenvolvimento, uma vez que apenas durante a realização da pesquisa se obteve uma melhor compreensão do suporte à metaprogramação do Rascal. Devido a complexidade da análise e a correção das transformações envolver diversos problemas com pré-condições quando contruídos de maneira generalizada, esta dissertação tomou como decisao de projeto a simplificação das transformações(particularmente em relação as pré-condições). Com isso, algumas transformações, em situações específicas, introduzem erros de compilação no código fonte resultante. Dessa maneira, ainda existe a necessidade de refinamento das transformações implementadas. Considerando o estudo empírico de expressividade, foi realizada apenas uma comparação com a ferramenta Lambdaficator, que foi introduzida como um componente de

refatoração do *IDE NetBeans* e que contou com financiamento da Oracle para o seu desenvolvimento. Entretanto, o LAMBDAFICATOR não possui suporte a todos os refatoramentos discutidos no Capítulo 3.3. Seria necessário realizar um estudo mais abrangente, comparando as demais transformações com outras ferramentas existentes.

5.3 Trabalhos Futuros

Como sugestão de trabalhos futuros, é possível evoluir as transformações de modo e reduzir a quantidade de falsos negativos. Atualmente, o grupo de pesquisa vem trabalhando nessa linha e introduzindo suporte ferramental adicional para melhorar a formatação de código resultante das transformações e geração automática de testes unitários dos componentes alterados nas transformações. Também é interessante conduzir mais estudos empíricos, possivelmente usando estratégias complementares às propostas nessa dissertação.

Referências

[1] Overbey, Jeffrey L. e Ralph E. Johnson: *Regrowing a language: Refactoring tools allow programming languages to evolve.* Em *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, páginas 493–502, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-766-0. `http://doi.acm.org/10.1145/1640089.1640127`. 1, 2, 14, 29

[2] Opdyke, William F: *Refactoring object-oriented frameworks.* Tese de Doutoramento, University of Illinois at Urbana-Champaign, 1992. 1

[3] Fowler, Martin e Kent Beck: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. 1

[4] Dig, Danny, Can Comertoglu, Darko Marinov e Ralph Johnson: *Automated detection of refactorings in evolving components.* Em *European Conference on Object-Oriented Programming*, páginas 404–428. Springer, 2006. 2

[5] Gyori, Alex, Lyle Franklin, Danny Dig e Jan Lahoda: *Crossing the gap from imperative to functional programming through refactoring.* Em *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, páginas 543–553. ACM, 2013. 2, 3, 4, 25, 30

[6] Gosling, James, Bill Joy e Guy Steele: *Java Language Specification, First Edition.* Addison-Wesley Professional, 1996. 7

[7] Visser, Eelco: *A survey of rewriting strategies in program transformation systems.* Electronic Notes in Theoretical Computer Science, 57:109–143, 2001. 8

[8] Visser, Eelco: *Program Transformation with Stratego/XT*, páginas 216–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, ISBN 978-3-540-25935-0. 9

[9] Fowler, Martin: *Language workbenches: The killer-app for domain specific languages?* `https://martinfowler.com/articles/languageWorkbench.html`. Accessed: 2017-08-30. 9

[10] Kalleberg, Karl Trygve e Eelco Visser: *Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT.* Delft University of Technology, Software Engineering Research Group, 2012. 9

[11] Kats, Lennart C. L. e Eelco Visser: *The Spoofax language workbench. Rules for declarative specification of languages and IDEs.* Em Rinard, Martin (editor):

Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010), 2010. 9

[12] Klint, P., T. v. d. Storm e J. Vinju: *Rascal: A domain specific language for source code analysis and manipulation.* Em *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, páginas 168–177, Sept 2009. 10

[13] Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2. 10, 11

[14] Easterbrook, Steve, Janice Singer, Margaret Anne Storey e Daniela Damian: *Selecting empirical methods for software engineering researchs.* Em Shull, Forrest, Janice Singer e Dag IK Sjøberg (editores): *Guide to advanced empirical software engineering*, capítulo 11, páginas 285–311. Springer, 2008. 12

[15] Schulze, Sandro, Jörg Liebig, Janet Siegmund e Sven Apel: *Does the discipline of preprocessor annotations matter?: a controlled experiment.* Em *ACM SIGPLAN Notices*, volume 49, páginas 65–74. ACM, 2013. 12

[16] Lemos, Otávio Augusto Lazzarini, Fabiano Cutigi Ferrari, Fábio Fagundes Silveira e Alessandro Garcia: *Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming.* Em *Proceedings of the 34th International Conference on Software Engineering*, páginas 529–539. IEEE Press, 2012. 12

[17] Bonifácio, Rodrigo, Paulo Borba, Cristiano Ferraz e Paola Accioly: *Empirical assessment of two approaches for specifying software product line use case scenarios.* Software & Systems Modeling, páginas 1–27, 2015. 12

[18] Shull, Forrest, Janice Singer e Dag IK Sjøberg: *Guide to advanced empirical software engineering*, volume 93. Springer, 2008. 13, 14

[19] Medeiros, Flávio, Christian Kästner, Márcio Ribeiro, Sarah Nadi e Rohit Gheyi: *The love/hate relationship with the c preprocessor: An interview study.* Em *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 13

[20] Bonifacio, Rodrigo, Fausto Carvalho, Guilherme N. Ramos, Uirá Kulesza e Roberta Coelho: *The use of c++ exception handling constructs: A comprehensive study.* Em *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, páginas 21–30. IEEE, 2015. 13

[21] Wohlin, Claes, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell e Anders Wesslén: *Experimentation in software engineering.* Springer Science & Business Media, 2012. 14

[22] Dig, Danny e Ralph Johnson: *How do apis evolve? a story of refactoring.* Journal of Software Maintenance and Evolution: Research and Practice, 18(2):83–107, 2006, ISSN 1532-0618. `http://dx.doi.org/10.1002/smr.328`. 15

[23] Dig, Danny, John Marrero e Michael D. Ernst: *Refactoring sequential java code for concurrency via concurrent libraries.* Em *Proceedings of the 31st Interna-*

tional Conference on Software Engineering, ICSE '09, páginas 397–407, Washington, DC, USA, 2009. IEEE Computer Society, ISBN 978-1-4244-3453-4. http://dx.doi.org/10.1109/ICSE.2009.5070539. 15

[24] Tansey, Wesley e Eli Tilevich: *Annotation refactoring: Inferring upgrade transformations for legacy applications*. Em *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, páginas 295–312, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-215-3. http://doi.acm.org/10.1145/1449764.1449788. 15

[25] Khatchadourian, Raffi, Jason Sawin e Atanas Rountev: *Automated refactoring of legacy java software to enumerated types*. Em *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, páginas 224–233. IEEE, 2007. 15

[26] Bloch, Joshua: *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, 2ª edição, 2008, ISBN 0321356683, 9780321356680. 15

[27] *Why can't i switch on a string?* Stackoverflow. http://stackoverflow.com/a/338230/3935114, http://stackoverflow.com/a/338230/3935114. 18

[28] Malaquias, Romero, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia e Rohit Gheyi: *The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter*. Em *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, páginas 297–307, Piscataway, NJ, USA, 2017. IEEE Press, ISBN 978-1-5386-0535-6. https://doi.org/10.1109/ICPC.2017.41. 44

[29] Khatchadourian, Raffi e Hidehiko Masuhara: *Automated refactoring of legacy java software to default methods*. Em *Software Engineering, 2017. ICSE 2017. IEEE International Conference on*, 2017. 44

Anexo I

Relação de projetos selecionados

elasticsearch
guava
spring-framework
zxing
fresco
lottie-android
spring-boot
libgdx
material-dialogs
kotlin
Signal-Android
realm-java
BaseRecyclerViewAdapterHelper
deeplearning4j
bazel
vert.x
recyclerview-animators
presto
guice
uCrop
gradle
druid
mybatis-3
buck
OpenRefine
Apktool
titan
lucida
okhttp-OkGo
intellij-community
lombok
material-theme-jetbrains
storm
che
VitamioBundle
roboguice
cw-omnibus
neo4j
SimplifyReader
JieCaoVideoPlayer

freeline

pinpoint

BoomMenu

cassandra

openhab1-addons

processing

wechat

closure-compiler

hadoop

smile

gocd

actor-platform

CoreNLP

graylog2-server

robospice

hackpad

jna

orientdb

alluxio

jstorm

atmosphere

ansj-seg

zeppelin

GalleryFinal

remusic

heron

weiciyuan

NoHttp

DraggablePanel

cas

quasar

antlr4

hibernate-orm

zaproxy

SmarterStreaming

dbeaver

flink

RecyclerViewSnap

h2o-2

incubator-weex

bilibili-android-client

XCL-Charts

grails-core

litho

android

binnavi

FastDev4Android

Activiti

hazelcast

MagicIndicator

Terasology

gephi

h2o-3

VideoPlayerManager

UltimateAndroid

phonegap-facebook-plugin

aws-sdk-java

GeekNews

Genius-Android

checkstyle

SeleniumQuery

RxJava

Retrofit

okhttp

java-design-patterns

netty

Anexo II

Resultado das transformações

Tabela II.1: Aceitação de Transformações por Projeto.

projeto	Transformacao	Quantidade	Linhas	Arquivos	status
flink	MC	5	18	5	Aceita
UltimateAndroid	DI	6	6	4	Aceita
zaproxy	MC	9	32	7	Aceita
jna	MC	4	15	4	Versão antiga
orienteddb	MC	10	46	7	Em aberto
processing	DI	11	11	6	Ignorada
processing	MC	4	14	3	Ignorada
processing	FUNC	2	11	2	Ignorada
CoreNLP	MC	14	67	13	Ignorada
antlr	MC	60	60	24	Ignorada
VitamioBundle	MC	5	35	5	Ignorada
Spring-Framework	FUNC	2	12	2	Rejeitada
RxJava	FUNC	16	34	10	Versão antiga
retrotift	FUNC	7	17	7	Versão antiga
okhttp	FUNC	13	29	15	Versão antiga
java-design-patterns	FUNC	15	38	17	Em aberto
netty	FUNC	61	135	43	Versão antiga
Hystrix	FUNC	20	45	10	Versão antiga
dubbo	FUNC	17	83	10	Versão antiga
interviews	FUNC	3	16	3	Ignorada
fastjson	FUNC	11	57	10	Versão antiga
storm	FUNC	29	81	24	Rejeitada
zeppelin	DI	20	22	11	Ignorada
binnavi	DI	133	135	87	Ignorada
checkstyle	DI	19	24	6	Rejeitada
guava	DI	226	259	105	Versão antiga
Atmosphere	DI	23	23	10	Aceita
Gephi	DI	10	11	1	Aceita
jna	DI	77	77	36	Versão antiga
gradle	DI	702	708	365	Versão antiga
PinPoint	DI	176	176	83	Versão antiga
Hazelcast	DI	216	221	77	Versão antiga
cas	DI	1	1	1	Aceita
CoreNLP	DI	132	134	46	Rejeitada
Elasticsearch	DI	23	23	18	Aceita
Activiti	DI	192	192	52	Aceita
SeleniumQuery	DI	8	8	5	Aceita
Openrefine	DI	59	59	29	Aceita
SeleniumQuery	FUNC	15	73	10	Rejeitada
gephi	FUNC	15	45	14	Em aberto