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

Uma Implementação do Move Method usando a Linguagem Rascal-MPL

Renan R. Reboredo

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

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2022

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

Uma Implementação do Move Method usando a Linguagem Rascal-MPL

Renan R. Reboredo

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

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

Prof.a Edna Dias Canedo
CIC / UnB

Walter Lucas
doutorando do PPGI / UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 16 de maio de 2022

Dedicatória

Dedicamos este trabalho a todos que acreditaram em nós e nos impulsionaram a frente. E a família, a Débora e a todos os melhores amigos, um lugar especial neste trabalho e na jornada a todos vocês.

Agradecimentos

Primeiramente agradecer ao Prof. Rodrigo Bonifácio por nos ajudar e acreditar no trabalho desenvolvido aqui, e ao Walter Lucas pelo suporte desde o início. Sem vocês definitivamente o trabalho não teria saído desta forma.

Agradecer amplamente também aos amigos da jornada da graduação. CJR, Central. Sem vocês a caminhada seria extremamente mais dolorosa e muito menos divertida.

Agradecer aos verdadeiros professores do CIC dos quais aprendemos bastante. E como não podia faltar, a Carolina Alvez Okimoto por tornar a vida dentro da universidade mais organizada e tranquila.

Por fim, agradecer a todos que nos apoiaram nessa reta final, familiares, namoradas, trabalho e amigos no geral que permitiram-nos ver a luz no fim do túnel.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho implementa a refatoração Move Method e apresenta uma abordagem nova para a extração e análise de pré-condições para executar refatorações em projetos Java, comparando com resultados encontrados em estudo comparativo da evolução de projetos open-source.

Palavras-chave: LaTeX, metodologia científica, trabalho de conclusão de curso

Abstract

This work implements the refactoring technique Move Method and presents a new approach to extraction and analysis of Preconditions for the well execution of refactorings in general, comparing with results found in a comparative study of evolution in open-source projects.

Keywords: LaTeX, scientific method, thesis

Sumário

1 Introdução **1**

1.1 Objetivos Gerais 1

1.2 Organização do Trabalho 2

2 Fundamentação Teórica **3**

2.1 Transformação de Programas 3

2.2 Refatoração 3

2.2.1 Exemplo - Extrair Classe 4

2.2.2 *Mover Método* 6

2.2.3 Pré-condições 7

2.3 Ferramentas e Linguagens para Transformação de Programas 8

2.3.1 *Stratego/XT* 8

2.3.2 *Spoofax* 9

2.3.3 Rascal MPL 10

2.4 Trabalhos Relacionados 16

3 Proposta de Solução **18**

3.1 Parser 19

3.2 Motor de Pré-condições 20

3.2.1 Estruturas de Dados 20

3.2.2 Extração de anotações 23

3.2.3 Queries 27

3.3 Move Method 28

4 Resultados **31**

4.1 Limitações e considerações sobre os testes 32

5 Conclusão **33**

Referências **34**

Lista de Figuras

2.1 Fluxo *Stratego/XT* [1]. 9
2.2 Captura de tela do ambiente integrado do Spoofax [2]. 10
2.3 Camadas do Design da linguagem RASCAL [3]. 12

3.1 Pipeline da Aplicação. 18

Lista de Tabelas

2.1 Tipos básicos do Rascal [3] 12

4.1 Resultados obtidos 32

Lista de Abreviaturas e Siglas

SCAM Source Code Analysis and Manipulation.

Capítulo 1

Introdução

Desenvolvimento de software envolve a conciliação de diferentes restrições, como qualidade, custos e prazo de desenvolvimento, tentando ao máximo manter a estrutura e qualidade do software de forma coerente e satisfatória. Mas nem sempre é possível obter a melhor qualidade possível durante a primeira iteração no processo de desenvolvimento. Nesse contexto que está inserida a prática de *refatoração*, que consiste no "processo de mudar um sistema de software de tal forma que não altere o comportamento externo mas aprimora a estrutura interna do projeto do software". [4]

Conforme a prática de *refatoração* se tornou cada vez mais presente no contexto de desenvolvimento de software, e, de certa forma, mais necessária em se tratando da evolução saudável da base de código, diversos mecanismos foram criados para a melhor utilização dessa prática. Entre elas, a implementação automática de diversos tipos de refatoração por ambientes integrados de desenvolvimento, ou IDEs, de tal forma a tentar minimizar o erro humano na hora de executar tais alterações.

1.1 Objetivos Gerais

O objetivo geral deste trabalho envolve explorar a linguagem de metaprogramação Rascal para implementar uma refatoração específica, chamada *Move Method*. Esta refatoração é relevante pois envolve uma série de pré-condições que precisam ser verificadas e constitui uma parte fundamental do processo de refatorações como um todo [5]. Essas pré-condições são características das classes e métodos alvos de uma refatoração que precisam ser verificadas para garantir a preservação do comportamento do código sendo refatorado [6].

Mais especificamente, para alcançar o objetivo geral, os seguintes objetivos específicos tiveram que ser endereçados:

- Condução de uma revisão não sistemática da literatura sobre refatoramento e transformações de programas.
- Condução de uma revisão não sistemática da literatura sobre pré-condições para aplicação de refatoramentos (em particular voltadas para o *Move Method*
- Escolha e estudo de uma linguagem de meta-programação para implementar as verificações de pré-condições e a transformação do *Move Method*.
- Implementação, testes e validação da implementação do refatoramento *Move Method*.

1.2 Organização do Trabalho

Este trabalho está dividido em 4 capítulos:

1. Capítulo 2: Apresenta uma base teórica sobre Transformações de Programas, Refatoração e Ferramentas e Linguagens de Transformações de Programas
2. Capítulo 3: Faz uma descrição da ferramenta refatoração *Move Method*.
3. Capítulo 4: Apresenta os resultados da ferramenta na aplicação específica na técnica de refatoração Move Method e principais limitações deste estudo.
4. Capítulo 5: Perpassa as principais conclusões obtidas a partir deste trabalho.

Capítulo 2

Fundamentação Teórica

Esse capítulo visa apresentar a fundamentação teórica para contextualização dos temas abordados neste trabalho. Os conceitos abordados são sobre Refatoração e Ferramentas; Transformação de Programas e Linguagens utilizadas para realizar transformação em programas.

2.1 Transformação de Programas

O termo transformação de programas pode ser associado a dois significados: tanto o ato de transformar um programa em outro, seja traduzindo-o para uma linguagem diferente da original ou refraseando o programa para a mesma linguagem, aplicando algum tipo de modificação; ou ainda se referir ao algoritmo que implementa uma mudança no programa em questão [7].

Podemos encontrar a transformação de programas em várias áreas de engenharia de software, como no campo de construção de compiladores, transformando uma determinada linguagem de programação em código de máquina, em visualização de software, transformando o programa original em uma linguagem mais visual, em geração de documentação, extraindo sentido do programa em questão e gerando documentação, em linguagem padronizada, de forma automática, e também em renovação automática de software, como administração de versões de bibliotecas externas e detecção de falhas de seguranças conhecidas.

2.2 Refatoração

Segundo Fowler, refatoração é o processo de "transformar a estrutura interna do software de forma a torná-lo mais fácil de compreender e mais barato para modificar sem alterar partes do design do sistema" [4]. É uma série de pequenos passos para melhorar a melhorar

o design de um programa atual de forma controlada e existem vários motivos pelos quais um engenheiro de software gostaria de empregá-los: melhorar o projeto do *software*, torná-lo de mais fácil compreensão, ajudar a encontrar *bugs* de forma mais facilitada e ajuda a codificar de forma mais rápida [4].

Conforme a engenharia de software foi avançando, padrões foram sendo encontrados e algumas refatorações foram sendo compiladas para resolver problemas repetitivos. A seguir seguem alguns exemplos retirados de [4]:

2.2.1 Exemplo - Extrair Classe

A refatoração de extração de classe é utilizada quando existe alguma classe na base de código realizando o papel que deveria ser de duas classes distintas. Para isso ser resolvido, é necessário criar uma nova classe e mover as partes relevantes da classe antiga para a nova classe; afim de manter o sentido e funcionamento originais da combinação das duas classes. Considere o exemplo, retirado de [4]:

Temos uma classe *Person* que possui lógica para tratar de um *TelephoneNumber*

```
1  class Person {
2      public String getName() {
3          return _name;
4      }
5      public String getTelephoneNumber() {
6          return ("(" + _officeAreaCode + ") " + _officeNumber);
7      }
8      String getOfficeAreaCode() {
9          return _officeAreaCode;
10     }
11     void setOfficeAreaCode(String arg) {
12         _officeAreaCode = arg;
13     }
14     String getOfficeNumber() {
15         return _officeNumber;
16     }
17     void setOfficeNumber(String arg) {
18         _officeNumber = arg;
19     }
20     private String _name;
21     private String _officeAreaCode;
22     private String _officeNumber;
23 }
```

Portanto, podemos extrair a lógica de *TelephoneNumber* para uma nova classe e adicionar uma referência a ela na classe de origem *Person*.

```
class Person {
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }
    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
}

class TelephoneNumber {
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
}
```

2.2.2 *Mover Método*

A refatoração *Mover Método* pode ser considerada uma das refatorações mais básicas [4], sendo interessante em diversos casos como quando classes estão carregadas de comportamento ou quando estão altamente acopladas. Um exemplo de que *Mover Método* é uma transformação importante é que para executar a transformação *Extrair Classe* , algumas transformações *Mover Método* foram realizadas.

A transformação consiste em criar um novo método com corpo similar ao anterior na classe que mais o utiliza. Dependendo da situação, transforma-se o antigo método em um *delegate* para o novo método na nova classe ou se remove o antigo método completamente. Eis um exemplo do *Mover Método*:

Tome como base uma classe chamada *Account*

```
1   class Account{
2       double overdraftCharge() {
3           if (_type.isPremium()) {
4               double result = 10;
5
6               if (_daysOverdrawn > 7) result += (_daysOverdrawn - 7) * 0.85;
7               return result;
8           }
9           else return _daysOverdrawn * 1.75;
10      }
11
12      double bankCharge() {
13
14          double result = 4.5;
15
16          if (_daysOverdrawn > 0) result += overdraftCharge();
17
18          return result;
19      }
20
21      private AccountType _type;
22      private int _daysOverdrawn;
23  }
```

Mas imagine que possa existir mais de um tipo de *Account*. Nesse cenário, pode-se criar uma nova classe chamada *AccountType* para receber o método *overdraftCharge*. Por fim, as classes ficarão assim

```
1   class Account{
```

```
        double overdraftCharge() {
            return _type.overdraftCharge(_daysOverdrawn);
        }
        double bankCharge() {

            double result = 4.5;

            if (_daysOverdrawn > 0) result += overdraftCharge();

            return result;
        }
        private AccountType _type;
        private int _daysOverdrawn;
    }

    class AccountType {
        double overdraftCharge(int daysOverdrawn) {

            if (isPremium()) {

                double result = 10;

                if (daysOverdrawn > 7) result += (daysOverdrawn- 7) * 0.85;
                return result;
            }
            else return daysOverdrawn * 1.75;
        }
    }
```

Importante ressaltar que a refatoração *Mover método* é apenas uma das etapas da transformação ocorrida no código acima, pois para concluir é necessário extrair o método da origem, alterar partes do programa de tal forma que a remoção do código não introduza erros e, por fim, realizar o *Mover método* para enviar para outra classe.

2.2.3 Pré-condições

Como mencionado anteriormente, desenvolvedores refatoram um programa com o intuito de melhorar o design de um programa sem alterar seu funcionamento. De forma a corretamente aplicar essas refatorações afim de preservar o comportamento inicial, é necessário checar uma série de pré-condições específicas para cada refatoração, e não constitui uma

tarefa essencialmente trivial. Essas pré-condições são características a serem consideradas pra cada refatoração que, dado uma configuração específica, permite decidir se é seguro aplicar a refatoração e, caso prossiga com a técnica, não introduzir *bugs* ou comportamentos diferentes dos observados anteriormente.

As pré-condições consideradas para este trabalho referentes a refatoração *Move Method* são as seguintes [6]:

- Métodos com os modificadores `abstract` e `native` ou algum método dentro de uma interface não podem ser movidos pois são métodos que definem uma espeficação e podem gerar efeitos colaterais se movidos.
- O método construtor da classe não pode ser movido por ser parte intrínseca a sua classe de origem.
- Ao referenciar um tipo não local ao método selecionado, como por exemplo um tipo advindo de uma classe genérica, por motivos de não introduzir um erro ao desreferenciar `null`, o *Move Method* não pode ser realizado.
- Uma chamada a esse método com passagem de `null` dentro do destino desejado ao método existe: caso exista, para não desreferenciar `null`, não é possível realizar a operação.
- Para um método polimórfico ou um método que possua uma chamada a sua super classe serem movido, é necessária que uma estrutura chamada *delegate* [6] fique no seu lugar. Caso essa condição não seja satisfeita, o *Move Method* não pode ser realizado.
- Para o método observado ser movido para uma interface, é imprescindível que nenhuma variável que este método receba como parâmetro seja inicializada com algum valor no corpo do método. Caso essa condição não seja satisfeita, não pode ser realizado o *Move Method* pois a interface não pode conter um método que seja uma implementação.

2.3 Ferramentas e Linguagens para Transformação de Programas

2.3.1 *Stratego/XT*

Stratego/XT é uma linguagem e um ferramental para transformação de programas. Combina *Stratego* [8], uma linguagem para implementação de transformações baseada no pa-

radigma de estratégias programáveis de reescrita com o *XT*, uma coleção de componentes reutilizáveis e ferramentas para o desenvolvimento de sistemas de transformação [1].

A Figura 2.1 exemplifica a organização de *Stratego/XT* no ponto de vista de reusabilidade, dividido em 5 camadas [1]:

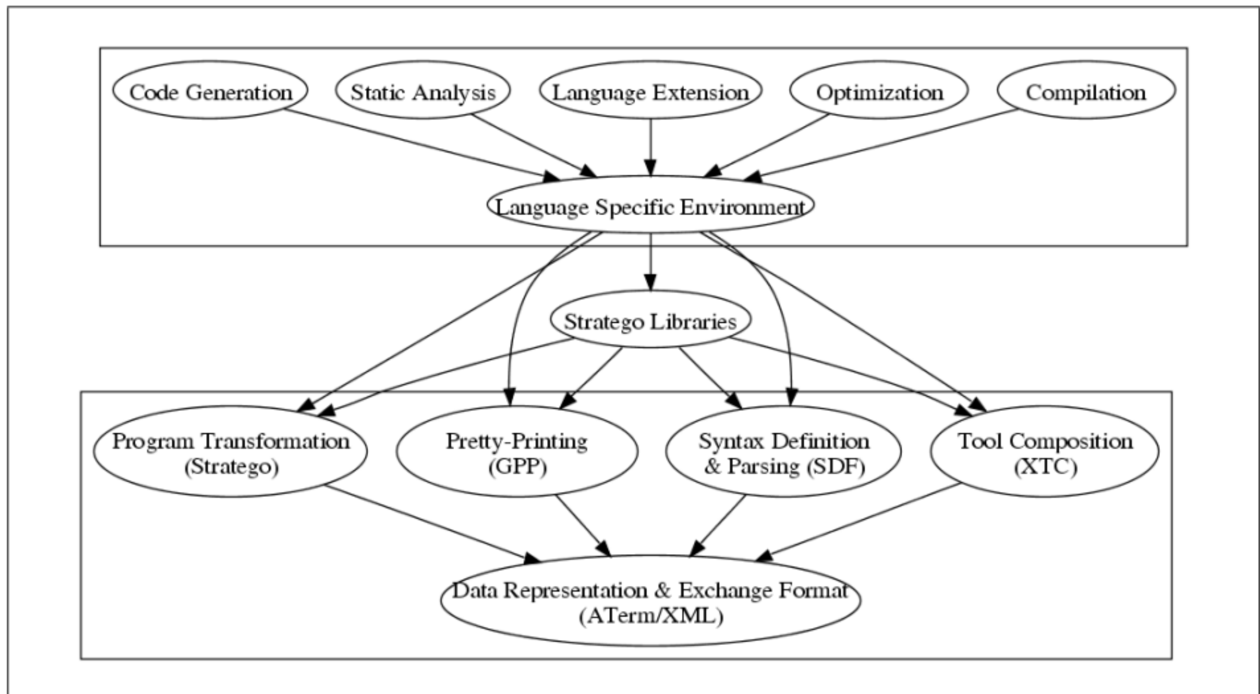

Figura 2.1: Fluxo *Stratego/XT* [1].

A linguagem *Stratego* é direcionada para o uso numa alta gama de aplicações de processamento de linguagens incluindo transformação fonte-a-fonte, geração de aplicações, otimização de programas, compilação e geração de documentação, mas não se aplica a transformação interativa de programas e nem prova de teoremas [8].

2.3.2 *Spoofax*

Spoofax é um ambiente de desenvolvimento baseado em Eclipse [1] extensível e interativo para desenvolver sistemas de transformação de programas utilizando *Stratego/XT* [9]. *Spoofax* se torna um ambiente de suporte à tecnologia *Stratego/XT* , provendo realce de sintaxe, código autocompletável, navegação de código fonte e *rebuild* automático e incremental de código. Além disso, integra ao ambiente de desenvolvimento extensibilidade por *scripts* criados por usuário na própria linguagem *Stratego* que permite análises em tempo real e transformações de código sob desenvolvimento, entregando, enfim, o objetivo de

[1] https://www.eclipse.org/downloads/

Spoofax de tornar um ambiente mais integrado para o desenvolvimento de *Stratego/XT* em ambiente Eclipse. A Figura 2.2 mostra uma captura de tela do ambiente integrado *Spoofax* dentro da plataforma Eclipse.

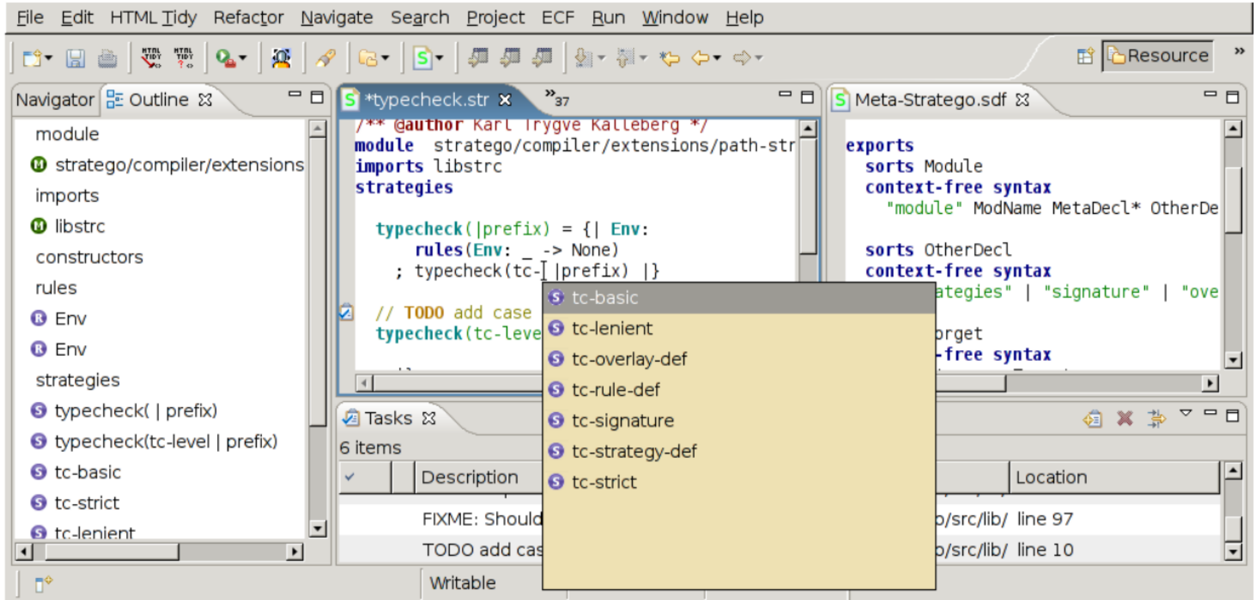

Figura 2.2: Captura de tela do ambiente integrado do Spoofax [2].

2.3.3 Rascal MPL

Rascal é uma linguagem específica de domínio (DSL) construída especialmente para prover para o campo da análise e manipulação de código fonte (SCAM) uma integração de alto nível da própria análise e manipulação, perpassando os níveis conceitual, sintático, semântico e técnico [3].

Por almejar uma perspectiva mais voltada para a engenharia de software e não a um viés mais formal, a linguagem alicerçou sua implementação em alguns pilares importantes:

- **Expressividade** Este pilar se manifesta através da criação de primitivas que atendem tanto a análise quanto a transformação de código, dois contextos importantes pro meio de SCAM. Por intermédio dessas primitivas, é possível expressar operações complexas como percorrer um programa a procura de uma estrutura de código específica com poucas linhas de código.

- **Confiança** Para trazer mais confiança ao lidar com projetos complexos no campo SCAM, foi implementado no RASCAL um sistema de tipos estáticos que reforça a imutabilidade e boa formação de um programa. Ainda, como a manipulação de

código se dá através de árvores tipadas anotadas com as informações do código fonte, isso garante uma maior fidelidade ao realizar transformações de código.

- **Usabilidade** Para tornar a barreira de entrada mais acessível, RASCAL preza por se integrar a linguagem Java, uma linguagem mais usual, através de ambiente integrado que permite utilizar tanto Java quanto RASCAL, na forma de um plugin no Eclipse e também provém na sua biblioteca padrão para manipulação de programas Java [2]. Provém ainda arcabouço de testes unitários e ferramentas para estender a linguagem conforme necessário for para quaisquer necessidades mais específicas do domínio enfrentado, duas características que tornam o desenvolvimento em RASCAL mais facilitado.

- **Performance** Tendo em vista que o meio de SCAM costuma envolver grandes quantidades de arquivos de uma só vez, é interessante que a linguagem seja capaz de lidar com essa carga de tal forma que não tenha impacto significativo na performance da aplicação e, considerando que RASCAL é usado para construir aplicações do domínio SCAM, a performance da ferramenta não deve impactar profundamente no resultado da aplicação final, pois poderia afetar a adoção mais ampla de RASCAL nesse domínio.

O design da linguagem Rascal é organizado em camadas, conforme Figura 2.3, de tal forma que a camada mais interna serve de alicerce para as camadas mais externas. Portanto, é de se concluir que o caráter imperativo no seu cerne, bem como seus dados imutáveis, são imprescindíveis para a boa execução da linguagem.

Outra grande força de RASCAL são seus tipos de dados primitivos criados pensando em operações comuns durante a manipulação de programas. Iremos focar mais precisamente em dois, `node` e `rel`, embora uma lista dos tipos básicos está descrita na Tabela 2.1. O tipo `node` [3] é um nó da árvore de *parse* não tipada que permite manipular e trabalhar com as estruturas do código, de tal modo que é possível realizar operações diretamente na árvore do programa, tornando manipulações mais naturais para o usuário da linguagem. O tipo `rel` [4], por ter um papel mais relevante para este trabalho, é discutido mais a fundo no Capítulo 3.

Por fim, outras características-chave, melhor detalhadas no Capítulo 3, de RASCAL são seu casamento de padrões e suas primitivas de `switch` e `visit`, que permitem fazer inferências e percorrer as estruturas do código em questão, respectivamente, de forma mais eficiente e simplificada.

[2] java.com/en/
[3] https://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Expressions/Values/Node/Node.html
[4] https://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Expressions/Values/Relation/Relation.html

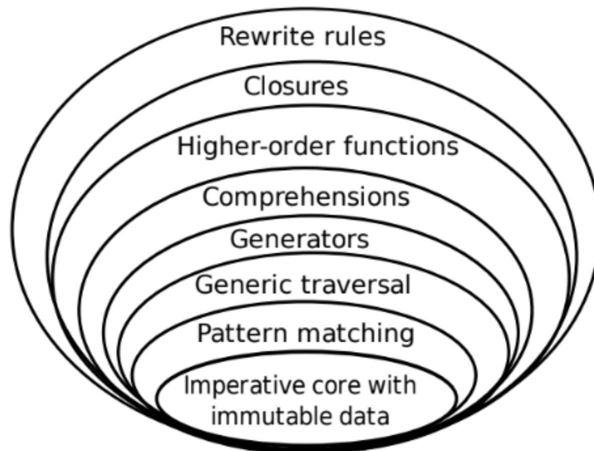

Figura 2.3: Camadas do Design da linguagem RASCAL [3].

Tipos	Exemplos literais
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
str	”abc”, ”first\nnext”
loc	!file:///etc/passwd
tuple[$t_1, \ldots, t_n$]	<1,2>,<”john”,43,true>
list[t]	[ ], [1], [1,2,3], [true, 2, ”abc”]
set[t]	{ },1,2,3,5,7,”john”,4.0
rel[$t_1, \ldots, t_n$]	<1,2>,<2,3>,<1,3>,<1,10,100>,<2,20,200>
map[t, u]	( ),(1 : true,2 : true),(6 :1,2,3,6,7 :1,7)
node	f, add(x, y), g(”abc”, [2, 3, 4])

Tabela 2.1: Tipos básicos do Rascal [3]

A linguagem RASCAL possui vários benefícios para trabalhar com a transformação de programas e tomaremos esta seção para explicitar algumas das que trouxeram maior impacto na decisão pela linguagem.

Visitors

A ação de percorrer uma estrutura complexa e poder aferir informações sobre ela são um aspecto imprescindível para o domínio SCAM [3], e para este presente trabalho, pois a operação primordial para construção do motor de pré-condições, descrito em detalhes na Seção 3.2, é percorrer uma árvore de *parse* para extrair as informações necessárias. A

estrutura se assemelha a um `switch` presente em linguagens como Java ou C [5], e provém uma estratégia que vai percorrendo uma estrutura de dados complexa em questão e pode ou não parar em algum `case` específico, que executa um código apropriado. Ainda, o *Visitor* permite a transformação da estrutura ao mesmo tempo que a percorre, que é posta em prática na construção do Move Method. Eis um exemplo de uso de `visit` para poder calcular a complexidade ciclomática de um método, valendo-se também do casamento de padrões em cima da sintaxe concreta do Java. [6].

```
module CalculateCC

import lang::java::\syntax::Java15;

int cyclomaticComplexity(MethodDec m) {
  result = 1;
  visit (m) {
    case (Stm)`do <Stm _> while (<Expr _>);`: result += 1;
    case (Stm)`while (<Expr _>) <Stm _>`: result += 1;
    case (Stm)`if (<Expr _>) <Stm _>`: result +=1;
    case (Stm)`if (<Expr _>) <Stm _> else <Stm _>`: result +=1;
    case (Stm)`for (<{Expr ","}* _>; <Expr? _>; <{Expr ","}*_>) <Stm _>` :
        result += 1;
    case (Stm)`for (<LocalVarDec _> ; <Expr? e> ; <{Expr ","}* _>) <Stm _>`:
        result += 1;
    case (Stm)`for (<FormalParam _> :<Expr _>) <Stm _>` :result += 1;
    case (Stm)`switch (<Expr _> ) <SwitchBlock _>`: result += 1;
    case (SwitchLabel)`case <Expr _> :` :result += 1;
    case (CatchClause)`catch (<FormalParam _>) <Block _>` :result += 1;
  }
  return result;
}
```

Código 2.1: Exemplo de Visitor

A parte importante a ser considerada neste código é a construção `visit`. A construção percorre uma unidade do tipo node e checa casos que ocorrem no código fonte para poder incrementar o contador de complexidade. Por fim, retorna a quantidade de *branches* presentes no código. Aliada ao casamento de padrões, visto mais a frente, o *Visitor* consegue realizar inferências sobre o código fonte, de forma expressiva, de tal modo que consegue agir em relação a essas informações, como por exemplo determinar,

[5] https://www.iso.org/standard/74528.html
[6] https://www.rascal-mpl.org/#_Metrics

em determinado projeto JAVA, quais são os dez métodos com maior complexidade como um todo(Código 2.2) [7].

```
module FindComplexFiles

import List;
import Exception;
import ParseTree;
import util::FileSystem;
import lang::java::\syntax::Disambiguate;
import lang::java::\syntax::Java15;

import CalculateCC;

lrel[int cc, loc method] findComplexFiles(loc project, int limit = 10) {
  result = [*maxCC(f) | /file(f) <- crawl(project), f.extension == "java"];
  result = sort(result, bool (<int a, loc _>, <int b, loc _>) { return a < b;
      });
  return head(reverse(result), limit);
}

set[MethodDec] allMethods(loc file)
  = {m | /MethodDec m := parse(#start[CompilationUnit], file)};

lrel[int cc, loc method] maxCC(loc file)
  = [<cyclomaticComplexity(m), m@\loc> | m <- allMethods(file)];
```

Código 2.2: Encontre arquivos complexos

Casamento de padrões

A principal estratégia para determinar em RASCAL diferentes casos da Estrutura em 2.1 é o casamento de padrões. A técnica é possível em *strings* utilizando expressões regulares, em listas e *sets*, bem como em tipos abstratos de dados e sintaxe concreta definidos na linguagem, responsáveis por grande parte das operações envolvendo código-fonte. Unindo casamento de padrões a *Visitors* , obtém-se uma estrutura poderosa que permite fazer inferências no código fonte e agir de acordo. O exemplo a seguir mostra essas duas estruturas em um código para fazer a transformação de estilo de um código em Java.

[7]https://www.rascal-mpl.org/#_Metrics

O que o Código 2.3, extraído do site oficial da linguagem RASCAL [8], está realizando é uma transformação de cláusulas de "if else"para uma forma considerada mais idiomática, de tal forma que o resultado é mais enxuto e legível, ainda preservando o sentido do código original.

```
module Idiomatic

import lang::java::\syntax::Java15;
import IO;
import ParseTree;

CompilationUnit idiomatic(CompilationUnit unit) = innermost visit(unit) {
   case (Stm) `if (!<Expr cond>) <Stm a> else <Stm b>` =>
        (Stm) `if (<Expr cond>) <Stm b> else <Stm a>`

   case (Stm) `if (<Expr cond>) <Stm a>` =>
        (Stm) `if (<Expr cond>) { <Stm a> }`
     when (Stm) `<Block _>` !:= a

   case (Stm) `if (<Expr cond>) <Stm a> else <Stm b>` =>
        (Stm) `if (<Expr cond>) { <Stm a> } else { <Stm b> }`
     when (Stm) `<Block _>` !:= a

   case (Stm) `if (<Expr cond>) { return true; } else { return false; }` =>
        (Stm) `return <Expr cond>;`
};
```

Código 2.3: Exemplo de Casamento de Padrões

Outra estrutura nativa da linguagem RASCAL que auxilia a construção de bons programas são os testes unitários, pois seu uso é bastante facilitado utilizando um modificador de função de tal forma que não é necessário utilizar nenhuma biblioteca externa para começar a escrever testes unitários. Utilizando essa estrutura, facilita averiguar se as transformações que estão sendo executadas preservam o mesmo sentido, como visto no exemplo em 2.4 [9].

```
test bool example() {
    code = (CompilationUnit) `class MyClass { int m() { if (!x) println("x");
        else println("y"); if (x) return true; else return false; } }`;
    return idiomatic(code)
```

[8]https://www.rascal-mpl.org/#_Transformations
[9]https://www.rascal-mpl.org/#_Transformations

```
4           ==
5           (CompilationUnit) ‘class MyClass { int m() { if (x) { println("y");
               } else { println("x"); } return x; } }‘ ;
6   }
```

Código 2.4: Exemplo de teste em Rascal

Relations

Relations são utilizadas neste contexto para representação da base de dados referente ao projeto sendo avaliado. São uma notação facilitada para sets de tuplas, ou seja, todos os elementos em uma Relation possuem tuplas de mesmo tipo [10], pois é muito utilizada no contexto de transformação de programas e os projetistas da linguagem implementaram ela como um tipo base do Rascal. Além do caráter único de cada item dentro de uma Relation, algumas funções são fornecidas pela biblioteca padrão do Rascal para poder estender a utilidade das *Relations*, permitindo fazer algumas operações relevantes na pesquisa e comparação de dados como *join* de Relations, produto cartesiano entre duas Relations, e as *queries* são realizadas em forma de perguntas se tal estrutura está presente ou não na Relation, de tal forma a lembrar uma linguagem de programação lógica como Prolog [11].

Um exemplo básico da utilização de Relations é saber se uma determinada tupla está presente na Relation. Para isso, se aproveita que a Relation é uma especificação de Set e utiliza o operador `in`, como no Código 2.5, para aferir sobre a presença ou não de determinado valor.

```
1   rel[str className, str method] classMethods;
2   classMethods = {<"Classe", "metodo">};
3   "metodo" in classMethods["Classe"]; // true
4   "naoMetodo" in classMethods["Classe"]; // false
```

Código 2.5: Exemplo de uso de Relations

2.4 Trabalhos Relacionados

O trabalho em [10] posta uma solução análoga à implementada neste trabalho e detalhada na Seção 3 que é a de apresentar um sistema de verificação de pré-condições para realizar transformações em programas. Nesse trabalho, é implementado um banco de dados local para armazenar informações das classes em Java para posteriormente fazer pesquisas nesse

[10] http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Expressions/Values/Relation/Relation.html

[11] https://www.cos.ufrj.br/ ines/courses/LP/prolog1/

banco de dados e aferir se dada transformação é possível ou não. A abordagem apresenta pontos interessantes por criar uma base de implementação da solução em Rascal e a utilização do banco de dados local evita múltiplas passagens no código para geração das consultas para aferir as pré-condições.

Outro trabalho relacionado é o presente em [6], que foi inspiração ao trabalho em [10], e utiliza a mesma estratégia de banco de dados relacional para construir uma base de conhecimento sobre as classes avaliadas de forma a conseguir aferir de forma mais assertiva se determinada transformação é viável. Em especial, a análise de pré-condições são amplamente exploradas no trabalho, pois sem o auxílio do banco de dados relacional toda vez que uma operação de refatoração é acionada, todas as checagens deveriam ser revisitadas e remontadadas, de forma que teria um impacto significativo no desempenho das refatorações. O detalhamento das pré-condições levantadas pelo trabalho auxiliaram a construção das pesquisas a serem realizadas neste trabalho.

Por fim, o trabalho [11], que forneceu a massa de testes para nossa aplicação, criou uma estratégia para detectar refatorações entre várias versões diferentes do código fonte nas linguagens Java, Javascript e C para que pudessem fazer um estudo sobre a evolução de um software através dessas refatorações. Um dos subprodutos do trabalho foi uma massa de dados de refatorações coletadas através do histórico de vários projetos *open source*, que foram utilizados para comparar nossos resultados tanto em termos de assertividade do nosso motor de pré-condições, bem como da eficácia da aplicação em realizar as refatorações propostas, neste caso específico, Move Methods.

Capítulo 3

Proposta de Solução

Neste capítulo serão aprofundados aspectos importantes da solução implementada, detalhando as decisões tomadas na construção de ferramenta para produzir a transformação Move Method com o suporte de uma base de coleta e decisão em cima de pré-condições necessárias para realização desta transformação em códigos Java.

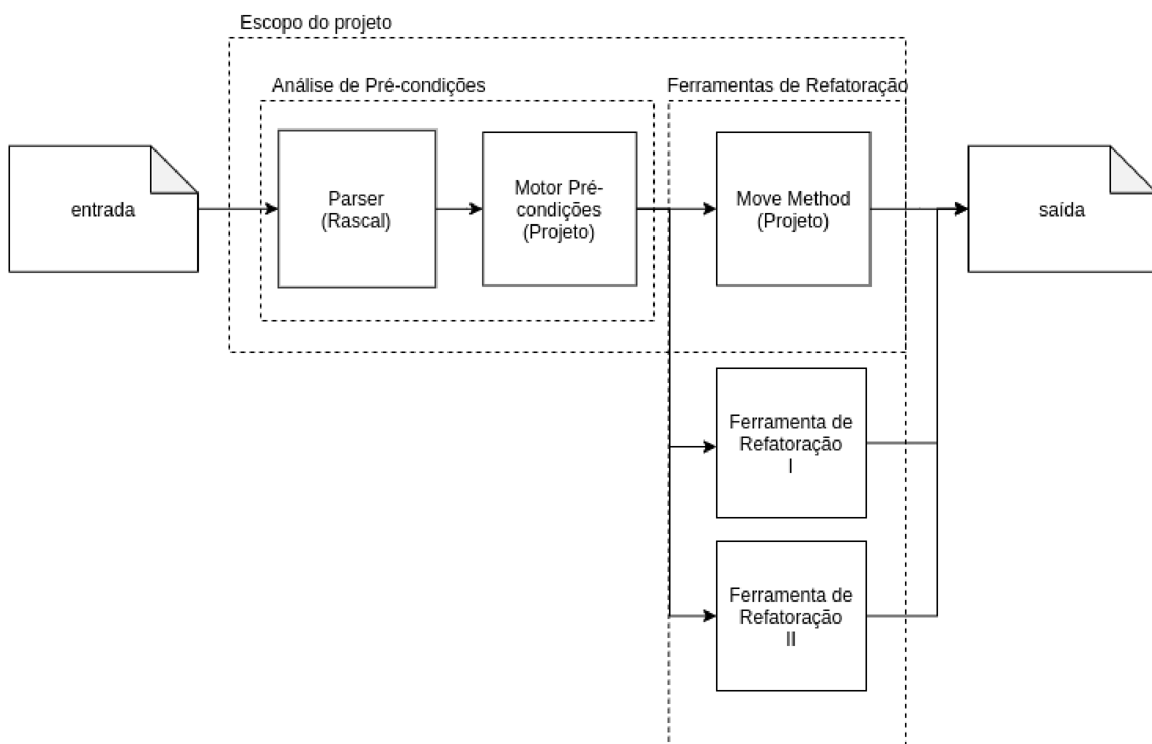

Figura 3.1: Pipeline da Aplicação.

A solução apresentada neste trabalho é acessada via linha de comando do Rascal, carregando o módulo Parser e invocando a função `main`, passando como argumentos `project, source, method, target`, sendo:

- `project` o caminho relativo do projeto a ser analisado.
- `source` a classe de origem do método a ser movido.
- `target` é a classe de destino para a qual o método será movido.
- `method` a assinatura completa do método a ser movido.

Todas as `CompilationUnit` são então armazenadas em uma tabela *hash* onde a chave é o nome da classe e o conteúdo de cada linha da tabela é uma tupla contendo a `CompilationUnit`, o caminho relativo da classe dentro do projeto e uma variável booleana indicando se a classe foi modificada ou não durante a execução da ferramenta, importante quando os arquivos serão gravados novamente após sofrerem a transformação MOVE METHOD.

Durante a fase de preenchimento da tabela *hash*, também é gerada a *Relation*, estrutura de RASCAL, responsável por armazenar as pré-condições e permitir queries em cima dessas características para que seja determinado se a refatoração MOVE METHOD pode ser realizada. Nessa *Relation* são armazenadas informações que tangem tanto a classe quanto o método, e essa operação é realizada para todas as classes e todos os métodos presentes no projeto.

Por fim a tabela *hash* e a *Relation* referente às pré-condições são repassadas para o módulo que realizará o MOVE METHOD, para que possa decidir se a operação é possível e, em caso afirmativo, realizar a transformação. Caso o MOVE METHOD seja realizado e duas ou mais classes sejam modificadas em decorrência desse fato, as classes modificadas são salvas na tabela *hash* para ao final da operação serem regravadas nos seus arquivos especificados.

Cada parte integrante do *pipeline* da solução é descrito em mais detalhes nas sessões posteriores.

3.1 Parser

O módulo PARSER encapsula a funcionalidade de ser a entrada da solução através de sua função `main` e também construir as `CompilationUnit` através do seu código-fonte, e o realiza para todas as classes presentes no projeto Java em questão.

Uma `CompilationUnit` é uma estrutura da sintaxe concreta do JAVA definida em RASCAL que representa uma classe que pode ser devidamente manipulada pelas ferramentas disponíveis na linguagem.

Para poder construir as `CompilationUnit` do projeto inteiro, o módulo utiliza o *Parser*[1] fruto do trabalho do Grupo *Program Analysis and Manipulation* da Universidade de

[1] https://github.com/PAMunb/rascal-Java8

Brasília e consiste em uma implementação da versão 8 do JAVA na definição de sintaxe concreta entregue pelo RASCAL. Nela, cada pedaço da sintaxe do JAVA 8 é descrita em forma de regras da seguinte forma:

```
start syntax CompilationUnit = PackageDeclaration? Imports TypeDeclaration*
    LAYOUT?;
syntax PackageDeclaration = PackageModifier* "package" {Identifier "."}+ ";" ;
syntax Imports = ImportDeclaration*;
syntax TypeDeclaration = ClassDeclaration ";"*
                       | InterfaceDeclaration ";"*
                       ;
```

Código 3.1: Início de definição do parser de Java8

Este exemplo (Código 3.1), que configura o início da descrição da sintaxe concreta de JAVA utilizando RASCAL, define construções sintáticas que compõem a linguagem. No exemplo, identificamos a descrição da declaração de um pacote JAVA, que pode importar outros pacotes, além de poder corresponder a uma classe ou a uma interface. O exemplo corresponde a um recorte da implementação, mas graças ao RASCAL, através da gramática descrita da linguagem através de regras como no exemplo, é possível fazer o parse de programas, nesse caso escritos em JAVA, e que resultam em uma árvore sintática abstrata. O Código 3.2 demonstra o ponto de entrada da geração da árvore abstrata após a leitura do arquivo da classe a ser *parseada*. Primeiramente, é lido o conteúdo do arquivo `s`, resultando em uma `string` com o conteúdo do arquivo. Posteriormente, o conteúdo do arquivo é repassado ao *parser*, resultando nas `CompilationUnit`.

```
sourceContent = readFile(s);
source = parse(#CompilationUnit, sourceContent);
```

Código 3.2: Parsing de uma classe

3.2 Motor de Pré-condições

3.2.1 Estruturas de Dados

`MoveOperation` (Código 3.3) é uma representação abstrata de uma operação básica de MOVE METHOD para o projeto, contendo o *full qualified name* da classe de origem, o nome do método a ser movido e o *full qualified name* da classe de destino.

```
alias MoveOperation = tuple[str source, str method, str target];
```

Código 3.3: Estrutura de dados que descreve o Move Method

`Class` (Código 3.4) define a estrutura da tupla de uma informação de uma determinada classe do projeto. Não ser confundido com a classe em si, mas cada entrada na Relation `Class` representa um conjunto associado entre o nome de uma classe, um `Maybe` [12] que pode ou não conter o nome de um método e uma informação semântica sobre aquele método ou classe. O `Maybe` ajuda a reutilizar a mesma estrutura para anotar o método, caso o `Maybe` contenha o nome do método, e para anotar a classe, caso o `Maybe` esteja vazio.

```
alias Class = rel[
                str className,
                Maybe[str] method,
                SemanticInformation info
            ];
```

Código 3.4: Descrição do tipo Class

`CompilationUnitWithLoc` (Código 3.5) é uma tupla contendo a localização do arquivo no projeto e a sua Unidade de Compilação correspondente. Para a biblioteca criada para manipular arquivos Java, uma `CompilationUnit` é a unidade representativa de uma classe Java e seu código fonte, manipulável pelo programa Rascal.

```
alias CompilationUnitWithLoc = tuple[
                                CompilationUnit unit,
                                loc location
                            ];
```

Código 3.5: Estrutura CompilationUnitWithLoc

`ClassTable` (Código 3.6) é uma tabela *hash* responsável por armazenar as unidades de compilação de cada classe dentro do projeto para serem tanto pesquisadas quanto manipuladas e reescritas. Sua chave primária é o *full qualified name* da classe, e mapeia à uma tupla que contém `CompilationWithLoc`, que é a unidade compilada e sua localização no projeto e um booleano que representa se a classe foi modificada ou não, importante para o momento de reescrita dos arquivos no final da execução.

```
alias ClassTable = map[
                    str name,
                    tuple[CompilationUnitWithLoc unitWithLoc,
                    bool modified] unitDetails
                ];
```

Código 3.6: Estrutura de Dados Classtable

`SemanticInformation` constitui um tipo abstrato de dados que representa as anotações semânticas que compõe as pré-condições às transformações em refatorações, descritas

em maior profundidade na Seção 2.2.3. Cada uma está associada a alguma informação e representa uma característica atribuída ou a uma classe ou a um método.

- `targetHasMethod()`: a classe de destino do Move Method já possui um método com o mesmo nome do método de origem da refatoração Move Method.
- `isAbstract()`: Método é do tipo `abstract`.
- `isNative()`: Método é do tipo `native`.
- `isConstructor()`: Método é um construtor.
- `isInsideInterface()`: Método está definido em uma interface.
- `isPolymorphic()`: Método é polimórfico. Considerado apenas polimorfismo por *Overload*.
- `nonLocalTypeReference()`: Há alguma referência a um tipo não local ao método, como um tipo abstrato da classe.
- `duplicateTypeParameter()`: O método utiliza um tipo genérico já referenciado na classe destino.
- `nullHomeValue()`: O método possui uma chamada passando null
- `superReference()`: Há uma referência à *keyword* super.
- `superClass(str c)`: Classe tem como Super Classe a classe cujo nome é c.
- `methodCalled(str m)`: Há uma chamada ao método m
- `noSuperClass()`: Classe não possui uma Super Classe correspondente.
- `conflictingMethod()`: Classe possui um método que causará conflitos ao realizar a operação.
- `none()`: anotação nula, usada apenas para demonstrar que alguma classe ou método existem, mas não há nenhum tipo de anotação relevante a ser computada.

```
1 data SemanticInformation = targetHasMethod()
2                          | isAbstract()
3                          | isNative()
4                          | isConstructor()
5                          | isInsideInterface()
6                          | isPolymorphic()
7                          | nonLocalTypeReference()
```

```
8                               | duplicateTypeParameter()
9                               | nullHomeValue()
10                              | superReference()
11                              | superClass(str c)
12                              | methodCalled(str m)
13                              | noSuperClass()
14                              | conflictingMethod()
15                              | none()
16                              ;
```

Código 3.7: Informações Semânticas

3.2.2 Extração de anotações

Para cada classe presente no projeto em análise, é feita uma travessia utilizando um `visit` para extrair todas as informações necessárias presentes na classe.

O Código 3.8 analisa e separa o nome do pacote da classe, que posteriormente será concatenado com o nome da classe para configurar a chave na tabela *hash* que contém a unidade compilada e sua localização.

```
1 case (PackageDeclaration) `<PackageModifier* m> package <{Identifier "."}+ id>
      ;`: {
2     packageName = unparse(id);
3 }
```

Código 3.8: Declaração de Classe

Aqui no Código 3.9 estão as análises de todas as declarações de interface e classe e extração das informações pertencentes a esse nível de profundidade. Nesta fase, são extraídas o nome da classe/interface, e se possui e qual é seu tipo abstrato associado, e se pertence ou não a uma `SuperClasse`.

```
1 case (NormalClassDeclaration) `<ClassModifier* m> class <Identifier id>
      <TypeParameters? typeArgs> <ClassBody body>`: {
2     abstractType = unparse(typeArgs);
3     abstractType = extractType(abstractType);
4     className = packageName + "." + trim(unparse(id));
5     table += addUnitToMap(className, unitWithLoc);
6     decls = decls + [ <nothing(), noSuperClass()> ];
7 }
8
```

```
9  case (NormalClassDeclaration) ‘<ClassModifier* m> class <Identifier id>
       <TypeParameters? typeArgs> extends <ClassType classType> <ClassBody
       body>‘: {
10     abstractType = unparse(typeArgs);
11     abstractType = extractType(abstractType);
12     className = packageName + "." + trim(unparse(id));
13     table += addUnitToMap(className, unitWithLoc);
14     str cls = trim(unparse(classType));
15     decls = decls + [ <nothing(), superClass(cls)> ];
16 }
17
18 case (NormalClassDeclaration) ‘<ClassModifier* m> class <Identifier id>
       <TypeParameters? typeArgs> implements <ClassType classType> <ClassBody
       body>‘: {
19     abstractType = unparse(typeArgs);
20     abstractType = extractType(abstractType);
21     className = packageName + "." + trim(unparse(id));
22     table += addUnitToMap(className, unitWithLoc);
23     str cls = trim(unparse(classType));
24     decls = decls + [ <nothing(), superClass(cls)> ];
25 }
26
27 case (NormalInterfaceDeclaration) ‘<InterfaceModifier* m> interface
       <Identifier id> <TypeParameters? typeArgs> <InterfaceBody body>‘: {
28     abstractType = unparse(typeArgs);
29     abstractType = extractType(abstractType);
30     className = packageName + "." + trim(unparse(id));
31     table += addUnitToMap(className, unitWithLoc);
32     precon = precon + [ isInsideInterface() ];
33     decls = decls + [ <nothing(), noSuperClass()> ];
34 }
35
36 case (NormalInterfaceDeclaration) ‘<InterfaceModifier* m> interface
       <Identifier id> <TypeParameters? typeArgs> extends <InterfaceType
       interfaceType> <InterfaceBody body>‘: {
37     abstractType = unparse(typeArgs);
38     abstractType = extractType(abstractType);
39     className = packageName + "." + trim(unparse(id));
40     table += addUnitToMap(className, unitWithLoc);
41     precon = precon + [ isInsideInterface() ];
```

```
42     str cls = trim(unparse(interfaceType));
43     decls = decls + [ <nothing(), superClass(cls)> ];
44 }
```

Código 3.9: Extração de informações de Declarações de Classes e Interfaces

O método construtor da classe não pode ser movido para outra classe, portanto possui uma anotação específica e é extraída no Código 3.10.

```
1 case (ConstructorDeclarator) `<TypeParameters? tp> <Identifier id> (
     <FormalParameterList? fp> )`: {
2     decls = decls + [ <just(unparse(id)), isConstructor()> ];
3 }
```

Código 3.10: Extração de construtor

```
1 case (InterfaceMethodDeclaration) `<InterfaceMethodModifier* modifier> <Result
     r> <MethodDeclarator m> <Throws? t> <MethodBody b>`: {
2     decls = decls + extractSemanticInformationMethod(b, precon, modifier, m,
         abstractType, "");
3     precon = [];
4 }
5
6 case (InterfaceMethodDeclaration) `<InterfaceMethodModifier* modifier>
     <TypeParameters types> <Annotation* an> <Result r> <MethodDeclarator m>
     <Throws? t> <MethodBody b>`: {
7     decls = decls + extractSemanticInformationMethod(b, precon, modifier, m,
         abstractType, extractType(unparse(types)));
8     precon = [];
9 }
10
11 case (MethodDeclaration) `<MethodModifier* modifier> <Result r>
     <MethodDeclarator m> <Throws? t> <MethodBody b>`: {
12     decls = decls + extractSemanticInformationMethod(b, precon, modifier, m,
         abstractType, "");
13     precon = [];
14 }
15
16 case (MethodDeclaration) `<MethodModifier* modifier> <TypeParameters types>
     <Annotation* an> <Result r> <MethodDeclarator m> <Throws? t> <MethodBody
     b>`: {
17     decls = decls + extractSemanticInformationMethod(b, precon, modifier, m,
         abstractType, extractType(unparse(types)));
```

```
18      precon = [];
19  }
```

Código 3.11: Extração de informação na declaração de métodos

No que tange a declaração dos métodos (Código 3.11), é neste momento que as informações dos métodos são retiradas através da função contida no Código 3.12.

Primeiramente, verifica se o método é ou `abstract` ou `native`, ambos impossibilitados de mover. Posteriormente prossegue para extração de informações importantes no corpo do método como quais métodos são chamados dentro daquele método e se há uma referência a chamada de `super`. Em seguida, extrai a assinatura do método, bem como os parâmetros separadamente. E, por fim, verifica se há alguma referência a tipos dentro do método que possam ocasionar uma duplicação de tipo ou o tipo não ser local ao método, causando uma possível complicação ao mover o método futuramente.

```
1   list[tuple[Maybe[str], SemanticInformation]]
2   extractSemanticInformationMethod(MethodBody body, list[SemanticInformation]
        precon, MethodModifier* modifier, MethodDeclarator m, str classType, str
        methodType) {
3   if (contains(unparse(modifier), "abstract")) {
4       precon = precon + [ isAbstract() ];
5   }
6
7   if (contains(unparse(modifier), "native")) {
8       precon = precon + [ isNative() ];
9   }
10
11  precon += extractMethodBodySemantics(body);
12
13  <name, params> = extractMethodDetails(m);
14
15  if (classType != "") {
16      if (checkTypeInBody(body, classType)) {
17          precon = precon + [ nonLocalTypeReference() ];
18      }
19  }
20
21  if (!isEmpty(params)) {
22      if (checkTypeInBody(body, methodType) && checkTypeInBody(body,
            last(params))) {
23          precon = precon + [ duplicateTypeParameter() ];
```

```
24      }
25  }
26
27  if (precon == []) {
28      precon = [ none() ];
29  }
30  }
```

Código 3.12: Extração de informações sobre o método

Aqui (Código 3.13) é anotada se, na chamada de um método, alguma vez é passado `null`. Essa verificação é importante para analisar a pré-condição `nullHomeValue()` posteriormente.

```
1  case (MethodInvocation) `<MethodName name> ( <ArgumentList args> )`: {
2      if (lastIndexOf(split(",", unparse(args)), "null") != -1 ) {
3          decls = decls + [ <just(unparse(name)), nullHomeValue()> ];
4      }
5  }
```

Código 3.13: Extração de informação de invocação do método com passagem de null

3.2.3 Queries

Condensados todas as anotações necessárias para serem feitas inferências sobre o código presente nas classes, são construídas *queries*, com o suporte da API do Rascal para o cômputo das características de uma Relation, para cada pré-condição que se deseja averiguar. Dependendo de como a pré-condição se relaciona com alguma característica objetiva de uma classe ou método, como a presença do método a ser movido já presente na classe de destino, ou alguma característica sensível ao contexto, a complexidade das *queries* aumenta ou diminui. Por exemplo, se quiser saber se o método observado existe na classe de destino, fazemos:

```
1  bool checkTargetHasMethod(Class class, MoveOperation move) =
       !isEmpty(class[move.target, just(move.method)]);
```

Código 3.14: Query simples para determinar se método existe em dada classe

Lembrando que `class` é uma relation que associa uma classe, um possível método e uma anotação, e `move` uma tupla que possui a classe de origem do método em análise, o nome do método em si e a classe de destino desejada para esse método, basta avaliar se o retorno da pesquisa `class[move.target, just(move.method)]` não retorna lista vazia, o que significa que pelo menos alguma anotação existe relacionando aquele método e

aquela classe. Por conta disso a anotação `none()` foi criada, para salientar a existência do método nas anotações, mesmo que não apresente nenhuma outra característica relevante para a transformação.

Uma *query* um pouco mais complexa (Código 3.15) aparece para averiguar se determinada configuração nas classes observadas pode vir a alterar a amarração de referências ao método caso o mesmo seja movido. É preciso, então, avaliar as super classes em questão para procurar por possíveis conflitos ao mover o método.

```
1    str unwrapSuperClass(superClass(x)) = x;
2    str unwrapSuperClass(_) = "";
3    bool checkConflictingMethod(Class class, MoveOperation move) {
4        classes = [ unwrapSuperClass(an) | an ← class[move.target, nothing()],
             !isEmpty(unwrapSuperClass(an))];
5        return !isEmpty([ c | c ← classes, !isEmpty(class[c,
             just(move.method)])]);
6    }
```

Código 3.15: Detecta métodos conflituosos em super classe

3.3 Move Method

Os passos anteriores do Parser (Seção 3.1) e Motor de pré-condições (Seção 3.2) são necessários para construir a API de refatoração do tipo Move Method. Dado uma tabela *hash*, construída no passo anterior, e a tupla `move`, remove-se o método da classe origem e o realoca na classe de destino. Neste passo não é mais necessário apurar se tal operação é possível, visto que essa averiguação foi realizada na parte da extração e verificação das pré-condições.

A forma como o programa lida com a mudança é simples: percorre a classe de origem e remove-se o método, gerando uma nova `CompilationUnit` e salvando-a na tabela. Posteriormente, percorre-se a lista de declarações de métodos dentro da classe destino e concatena-se o novo método nesta lista. Por fim, reescreve as classes marcadas como alteradas na tabela *hash* no projeto.

```
1  ClassTable moveMethod(MoveOperation move, ClassTable table) {
2      if (isEmpty(table)) {
3          return ();
4      }
5
6      <newSource, method> = removeMethodFromSource(table, move);
7
```

```
8       table[move.source].unitWithLoc.unit = newSource;
9       table[move.source].modified = true;
10
11      table[move.target].unitWithLoc.unit = addMethodToTarget(table, move,
            method);
12      table[move.target].modified = true;
13
14      return table;
15  }
```

Código 3.16: Move Method

```
1   CompilationUnit addMethodToTarget(ClassTable table, MoveOperation move,
        ClassBodyDeclaration method) =
2       top-down visit(table[move.target].unitWithLoc.unit) {
3           case (ClassBody) `{ <ClassBodyDeclaration* decls> }`: {
4               list[ClassBodyDeclaration] methodsList = [];
5               classBody = mergeTrees([method] + [ d | d ← decls]);
6               insert (ClassBody) `{ <ClassBodyDeclarationList classBody> }`;
7           }
8       };
```

Código 3.17: Adiciona método à lista de declarações na classe destion

```
1   tuple[CompilationUnit newSource, ClassBodyDeclaration method]
        removeMethodFromSource(ClassTable table, MoveOperation move) {
2       ClassBodyDeclaration method;
3       foundMethod = false;
4
5       CompilationUnit newSource = top-down
            visit(table[move.source].unitWithLoc.unit) {
6           case (ClassBody) `{ <ClassBodyDeclaration* decls> }`: {
7               if(methods := [ d | d ← decls, checkIfIsMethod(move.method, d)]
8                   && size(methods) == 1
9                   && newDecls := [ d | d ← decls, !checkIfIsMethod(move.method,
                        d)]) {
10
11                 method = head(methods);
12                 foundMethod = true;
13                 classBody = mergeTrees(newDecls);
14                 insert (ClassBody) `{ <ClassBodyDeclarationList classBody> }`;
15             }
```

```
16          }
17      };
18
19      if (foundMethod) {
20          return <newSource, method>;
21      } else {
22          throw "Tried to remove non existant method \"<move.method>\" from
                class \"<move.source>\"";
23      }
24  }
```

Código 3.18: Remove o método da origem

Capítulo 4

Resultados

Em um primeiro momento, foram realizados testes simplificados utilizando os exemplos apresentados em [6] para validar as extrações das pré-condições.

Posteriormente, foram realizados testes valendo-se dos dados obtidos em [11] disponibilizados em [1] para aprovar as refatorações realizadas pela ferramenta presente neste trabalho. O interessante de utilizar o trabalho de [11] é que, como é um trabalho exploratório e descritivo das refatorações já realizadas em determinado projeto, o passo de verificar se as operações são ou não possíveis é pulada, e podemos assumir que, caso nossa ferramenta acuse que algum MOVE METHOD descrito não pode ser realizado, estamos diante então de um resultado equivocado.

Para a realização dos testes foram selecionados seis projetos dentre os dados para averiguar as transformações. Os projetos são NEO4J [2], CLOSURE COMPILER [3], AWS SDK JAVA [4], MOCKITO [5] , SPRING BOOT [6] e ANTLR4 [7]. Todas as realizações de *Move Method* possíveis foram escolhidas dentre os seis projetos, totalizando 62 testes, sendo que o espaço total contando com os demais projetos não escolhidos era de 319 testes possíveis.

O repositório NEO4J foi o primeiro a nos alertar sobre o impacto da quantidade de arquivos nas operações. Como a maioria dos outros testes foram realizados em menos arquivos ao mesmo tempo, foi utilizado uma parte menor do projeto para poder realizar as refatorações, sem prejuízo final no resultado, mas no caso do NEO4J isso não foi possível, o que impactou diretamente nos resultados obtidos.

[1]https://github.com/aserg-ufmg/RefDiff/blob/master/refdiff-evaluation/data/java-evaluation/evaluation-data-public.xlsx
[2]https://github.com/icse18-refactorings/neo4j
[3]https://github.com/icse18-refactorings/closure-compiler
[4]https://github.com/icse18-refactorings/aws-sdk-java
[5]https://github.com/icse18-refactorings/mockito
[6]https://github.com/icse18-refactorings/spring-boot
[7]https://github.com/icse18-refactorings/antlr4

Projeto	Nº testes	Nº testes bem-suscedidos	Nº testes com apoio manual
Neo4j	12	0	6
Spring Boot	18	9	4
Mockito	10	9	2
AWS SDK Java	6	6	1
Closure Compiler	16	16	1
Antlr4	11	11	2

Tabela 4.1: Resultados obtidos

Como RASCAL possui uma taxa de sucesso na compilação de arquivos JAVA de mais de 95%, podemos assumir que a imensa maioria dos arquivos do projeto serão compilados, e, ainda, que os arquivos relevantes não serão afetados na maioria dos casos. Um caso que isto não ocorreu foi no caso de falha em MOCKITO e SPRING BOOT. No caso MOCKITO, ocasionou em apenas um erro e, no caso de SPRING BOOT, ocasionou em metade, pois nove operações de MOVE METHOD eram realizadas em apenas um arquivo.

4.1 Limitações e considerações sobre os testes

Como as alterações descritas em [11] são avaliadas a posteriore, algumas modificações não são passíveis de serem executadas exatamente da mesma forma com nossa API atual. Exemplo mais comum é um método `public` na origem que é alterado para `private` no destino ou a mudança de um tipo abstrato relacionado a algum parâmetro dentro do método. Como essas alterações carecem de ação humana para serem realizadas e não estão diretamente ligadas ao MOVE METHOD em si, nossa aplicação apenas desconsidera essas pequenas incrongruências.

Além disso, outro problema decorrente dos dados serem de análises a posteriore é que as refatorações são analisadas no antes e depois de ocorrerem no código. Porém, a aplicação assume, quando vai realizar as transformações, que ambas as classes já existem posteriormente no momento do pedido da mudança. Essa condição nem sempre é verdade nos projetos observados. Para contornar este problema, são criadas as classes na mão com a exceção do método que será movido e é executado o teste a partir daí.

Por fim, uma última limitação é o tamanho dos projetos observados. Foi observado que, acima de 2000 arquivos no mesmo projeto, a ocorrência de estouro da *heap* do JAVA se tornaram frequentes. É aconselhado, enquanto não é feita uma otimização do uso da memória no projeto, realizar transformações em projetos com número menor de arquivos a 2000.

Capítulo 5

Conclusão

Neste trabalho foi apresentado uma estratégia para implementação da refatoração MOVE METHOD, se valendo de checagem de pré-condições em programas JAVA utilizando a linguagem RASCAL e suas poderosas *queries* utilizando RELATIONS.

Para poder averiguar os resultados encontrados, foi utilizada um *dataset* disponibilizado publicamente para comparar operações de MOVE METHOD realizadas em seis projetos open-source e comparado com rodadas de testagem do programa criado com os resultados obtidos. Os resultados foram promissores e apresentam uma boa perspectiva da extração de pré-condições utilizando esta abordagem.

Com poucas alterações, o motor de pré-condições pode servir como base para muitas ferramentas de refatoração. Além disso, a transformação MOVE METHOD pode se tornar mais robusta, e essa mesma biblioteca pode tanto evoluir quanto se juntar a outros trabalhos relacionados para poder aumentar a capacidade da solução para além de uma operação apenas.

Possíveis pontos de evolução futuros estão na ordem de melhorias de performance, principalmente de caráter de uso de memória, para poder trabalhar de forma mais otimizada com projetos maiores. Ainda, é possível vislumbrar uma interface de pesquisa das pré-condições com bibliotecas externas, de tal modo que a lógica das pré-condições não dependa da implementação atual para ser utilizada em outros contextos.

Referências

[1] Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas e Eelco Visser: *Stratego/XT 0.17. A language and toolset for program transformation.* Science of Computer Programming, 72(1-2):52–70, 2008, ISSN 01676423. viii, 9

[2] Kalleberg, Karl Trygve e Eelco Visser: *Spoofax : An Extensible , Interactive Development Environment for Program Transformation with Stratego / XT.* Language, 2007. viii, 10

[3] Klint, Paul: *R ASCAL : a Domain Specific Language for Source Code Analysis and Manipulation.* viii, ix, 10, 12

[4] Fowler, Martin: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999, ISBN 0-201-48567-2. 1, 3, 4, 6

[5] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*, capítulo 7, páginas 115–137. Addison-Wesley, Boston, MA, USA, 1999, ISBN 0-201-48567-2. 1

[6] Kim, Jongwook, Don Batory, Danny Dig e Maider Azanza: *Improving refactoring speed by 10X.* Proceedings - International Conference on Software Engineering, 14-22-May-:1145–1156, 2016, ISSN 02705257. 1, 8, 17, 31

[7] Visser, Eelco: *A survey of rewriting strategies in program transformation systems.* Electronic Notes in Theoretical Computer Science, 57:109–143, 2001, ISSN 1571-0661. WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming. 3

[8] Visser, Eelco: *Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5.* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2051 LNCS(May):357–361, 2001, ISSN 16113349. 8, 9

[9] Kalleberg, Karl Trygve e Eelco Visser: *Spoofax: An interactive development environment for program transformation with Stratego/XT.* páginas 47–50, March 2007. 9

[10] Barcelos, Uriel De e Conceição Silva: *Evoluções na Biblioteca de Transformações RJTL.* 2018. 16, 17

[11] Silva, Danilo, Joao Silva, Gustavo Jansen De Souza Santos, Ricardo Terra e Marco Tulio O. Valente: *RefDiff 2.0: A Multi-language Refactoring Detection Tool.* IEEE Transactions on Software Engineering, XX(X):1–1, 2020, ISSN 0098-5589. 17, 31, 32

[12] Meijer, Erik e Johan Jeuring: *Merging monads and folds for functional programming.* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 925:228–266, 1995, ISSN 16113349. 21