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

Análise estática para detectar a evoulução da linguagem java em projetos open source

Thiago Gomes Cavalcanti
Vinícius Correa de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Wilson Henrique Veneziano

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador) — CIC/UnB
Prof. Dr. Genaina Nunes Rodriges — CIC/UnB
Prof. Dr. Edson Alves da Costa Junior — FE/UnB-Gama

CIP — Catalogação Internacional na Publicação

Cavalcanti, Thiago Gomes.

Análise estática para detectar a evoulução da linguagem java em projetos open source / Thiago Gomes Cavalcanti, Vinícius Correa de Almeida. Brasília : UnB, 2015.
75 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. análise estática, 2. evolução, 3. evolução de linguagens de programação linguagens, 4. language design, 5. software engeneering, 6. language evolution, 7. refactoring, 8. java

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Análise estática para detectar a evoulução da linguagem java em projetos open source

Thiago Gomes Cavalcanti
Vinícius Correa de Almeida

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

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

Prof. Dr. Genaina Nunes Rodriges
CIC/UnB

Prof. Dr. Edson Alves da Costa Junior
FE/UnB-Gama

Prof. Dr. Wilson Henrique Veneziano
Coordenador do Curso de Computação — Licenciatura

Brasília, 31 de março de 2015

Dedicatória

Dedicamos este trabalho a nossa família e ao departamento de Ciência da Computação da UnB. Que este seja apenas uma idéia inicial e que que futuros alunos possam ajudar a enriquecer ainda mais este projeto para que a Universidade tenha sua própria ferramenta de análise de código e que sirva de modelo para outras Universidade.

Agradecimentos

Com imensa dificuldade de agradecer a tantas pessoas que de certo modo nos ajudaram nessa conquista, hora em momentos calmos hora apreensivos. Em especial a toda nossa família por dar todo suporte necessário para que pudessemos concluir essa etapa em nossas vidas, também aluna Daniela Angellos pelo seu desdobramento e conhecimento para nos ajudar a criar essa ferramenta.
Em especial ao professor dr. Rodrigo Bonifácio que nos inseriu nesse imenso mundo da Engenharia de Software, hora apresentando um problemática hora ajundando a resolver barreiras as quais não conseguimos sozinhos.
E ainda a UnB por todo seu corpo docente que sem este essa jornada não seria concluida com excelência, em especial aoprofessor dr. Edson Alves da Costa Júnior por se deslocar da UnB-Gama para nos ajudar.

Resumo

Atualmente encontrar blocos de código específicos tem sido de grande importância para atualizar esse trechos por um mais moderno ou mais eficiênte e assim ter os projetos utilizando sempre o que há de mais recente disponibilizado por cada *feature* das linguagem no caso deste trabalho Java.

Com isso o principal objetivo deste trabalho é criar um analisador estático com o objetivo de encontrar construções específicas na linguagem Java, contruções que podem ser código ultrapassado ou até mesmo modificações de um *foreach* por uma expressão lambda. Tais contruções após encontradas farão parte de um relatório de saída para que possa ser tomada a decisão se tais contruções serão refatoradas ou não.

Visando a maior flexibilidade possível na contrução deste analisador, a parte responsável por encontrar código fonte pré-determinado é flexível fazendo com que a qualquer momento que seja necessário possam ser criados novos visitantes sem causar impacto na estrutura do analisador. Os relatórios gerados também são flexíveis e automático podendo a qualquer momento ser modificado a geração de arquivos CSV na saída por um banco de dados caso seja de interesse do desenvolvedor.

Palavras-chave: análise estática, evolução, evolução de linguagens de programação linguagens,language design, software engeneering, language evolution, refactoring, java

Abstract

Search to specific code has been very important from update to a more actual or efficient and with the projetc has every the least release of a language at this work Java.

Therefore the main goal of this project is develop a static analysis with objective to find specifics constructions of Java language, where this constructions can be older code or a update a block to another better such as foreach for a lambda expression. After find this code the place in source code is saved to write a output file for future evaluation and decide if this will be updated or not.

With focus in a flexibility the project the party responsible for visitors that find source code previously determined is the highest flexible that make easy in any time the developer create their own visitor and insert in the system without impacts in architecture. The output reports are flexible and automatics that provide in any time a possibility of chance the actuals CSV files to another form such as database.

Keywords: static analysis,language design, software engeneering, language evolution, refactoring, java

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas

LOC Linhas de Código

AST Árvore de sintaxe abstrata

IDE Ambiente de Desenvolvimento Integrado

JDBC Java Database Connectivity

JDK Java Development Kit

AWT Abstract Window Toolkit

RMI Invocação de Método Remoto

API Aplicações de Programação Interfaces

JNI Java Native Interface

GUI Interface Gráfica do Usuário

JDT Java Development Tools

ACDP Java Platform Debugger Architecture

JCP Java Community Process

EFL Enhanced for loop

AIC Annonymous Inner Class

DI Dependency Injection

IoC Inversion of Control

CSV Comma separated values

Capítulo 1

Introdução

1.1 Introdução

Uma premissa na Engenharia de Software é a *natureza evolutiva* do software, e, com isso, custos significativos são relacionados com as atividades de manutenção. De forma semelhante, as linguagens de programação evoluem, com o intuito de se adaptarem as novas demandas e trazerem benefícios relacionados a produtividade e a melhoria da qualidade dos softwares construídos. Entretanto, um desafio inerente é a evolução de sistemas existentes em direção a adoção de novas construções disponibilizadas nas linguagens [**?**]. Conforme explicado por Jeffrey L. Overbey e Ralph E. Johnson. [**?**], tal evolução faz com que características obsoletas sejam mantidas e raramente são removidas de uma linguagem o que acarreta em um aumento da complexidade, aprendizagem e da manutenção do software. Isso naturalmente aumenta a dificuldade de desenvolvimento o que resulta em um aumento de dificuldade de aprendizagem de determinada versão já ultrapassada de uma linguagem e faz com que a equipe alterne entre propriedades atuais e antigas as quais passam a ser quase um dialeto da linguagem implicando no aumento de tempo para conceber um projeto e consequentemente gerindo aumento no custo final projeto.

Uma decisão não tão simples é manter uma porção do código congelado, sem evolução, ao longo projeto devido alguma restrição técnica. O que infelizmente acarreta em uma estagnação de todo um sistema pois não é somente o projeto afetado, mas sim uma toda infraestrutura como compiladores, banco de dados e sistema operacional e que se de alguma forma vierem a ser atualizados com esta porção código estagnado pode ocasionar problemas como uma queda significativa de desempenho ou até mesmo o sistema parar de funcionar. Devido a esses problemas de código não atualizado, com as versões com estruturas mais atuais, a proposta da realização de refatoração através de ferramentas a ser desenvolvidas que visem atacar esse gargalo deixado por código obsoleto.

1.2 Objetivos

O principal objetivo deste trabalho é analisar a adoção de construções da linguagem de programação Java em projetos open-source, com o intuito de compreender a forma típica de utilização das construções da linguagem e verificar a adoção ou não das *features* mais recentemente lançadas. Especificamente, os seguintes objetivos foram traçados:

- implementar um ambiente de análise estática que recupera informações relacionadas ao uso de construções da linguagem Java.
- avaliar o uso de construções nas diferentes versões da linguagem Java, considerando projeto open-source.
- realizar um *survey* inicial para verificar o porque da não adoção de algumas construções da linguagem nos projetos.
- contrastar os resultados das nossas análises com trabalhos de pesquisa recentemente publicados, mas que possivelmente não analisam todas as construções de interesse deste trabalho, em particular a adoção de construções recentes na linguagem (como Expressões Lambda).

1.3 Metodologia

A realizaçao deste trabalho envolveu atividades de revisão da literatura, contemplando um estudo de artigos científicos que abordam a adoção de novas características da linguagem Java ao longo do lançamento das diferentes versões para a comunidade de desenvolvedores [**? ? ? ? ? ? ?**]. Com isso, foi possível compreender a limitação dos trabalhos existentes e, dessa forma, definir o escopo da investigação.

Posteriormente, foi necessário buscar uma compreensão sobre como implementar ferramentas de análise estática, e escolher uma plataforma de desenvolvimento apropriada (no caso, a plataforma Eclipse JDT [**?**]). Posteriormente, foi iniciada uma fase de implementação dos analisadores estáticos usando padrões de projetos típicos para essa finialidade: visitor, dependency injection, ...

Finalmente, foi seguida uma estrategia de Mineração em Repositórios de Software, onde foram feitas as análises da adoção de construções da linguagem Java em projetos open-source, de forma similar a outros artigos existentes [**? ? ? ? ? ? ? ? ? ?**].

Após tal entendimento sobre adoção de novas características, fora realizado um estudo sobre análise estática em códigos escritos na linguagem java o que se torna a base deste trabalho. E logo após a consolidação deste conhecimento, foi realizado a escolha de projetos Java de maior relevância na comunidade open-source.

Em seguida foi estudada a melhor arquitetura para a elaboração do analisador estático proposto de modo que esta no tivesse um fraco acoplamento entre os módulos necessários e facilitasse a pesquisa de outras características através da injeção do visitors [**?**] usando o *spring framework* [**?**]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

1.4 Problema a ser Atacado

Nos últimos anos sistemas computacionais ganharam cada vez mais espaço no mercado o que acarretou na dedicação de profissionais para manter a qualidade elevada tanto no desenvolvimento como na manutenção destes a fim de proporcionar tanto a multiplataforma quanto que qualquer equipe seja capaz de desenvolvem em qualquer local a qualquer tempo.

Com isso a produção de software tornou-se uma tarefa desafiadora de altíssima complexidade que pode acarretar no aumento da possibilidade de surgimento de problemas. Outro fator de grande relevância é que cada vez mais o bom desempenho do software depende da capacidade e qualificação dos profissionais que compõem a equipe de desenvolvimento. Um desses problemas é manter o desenvolvimento com partes ultrapassadas de uma linguagem o que torna um sistema obsoleto e com a chance de conter *bugs* e vulnerabilidades que podem comprometer a segurança de todo o sistema.

A atuação de equipes que desenvolvem utilizando códigos obsoletos continua sendo um grande problema no desenvolvimento de software ao longo de suas releases, mesmo com a evolução da linguagem. Códigos mais atuais tornam-se cada vez mais necessário pois evitam, corrigem falhas e vulnerabilidades além do mesmo tornar-se mais atual. Tais códigos não evoluem podem ser por falta de suporte da IDE, por falta conhecimento da equipe de desenvolvedora ou pelo simples fato de não possuir uma analisador estático que aborde estas construções lançadas nas novas versões das linguagens, especificamente java.

Após toda release uma linguagem demora um certo tempo de maturação para que comunidade de desenvolvedores adote novas características lançadas ou simplesmente não a utilizem, porém java possui uma filosofia de manter suporte a todos legado já desenvolvido por questão de portabilidade o que beneficia tanto IDE's quanto equipes a não ter a necessidade de se atualizarem para as ultimas versões da linguagem o que torna a construção de software com uma linguagem ultrapassada confortável porém existe a possibilidade do software possuir vulnerabilidades.

Um bom exemplo a ser lembrado é FORTRAN quando adicionou orientação objetos em sua na sua versão do ano de 2003 forçando a evoulução de seus compiladores os quais não forneciam mais suporte a versões anteriores conforme relata Jeffrey L. Overbey e Ralph E. Johnson em [**?**], que como consequência forçou toda comunidade desenvolvedora a se atualizar. E ainda havia a possiblidade de certos trechos de código sofrer um refectoring em tempo de compilação por um código mais atual e equivalente.

A processo de utilizar um analisador estático em um projeto antes de sua compilação pode vir a impactar na melhora da confiança do software pois pode detectar vulnerabilidades de maneira prematura além de reduzir o retrabalho caso estas não fossem detectadas. Tais vulnerabilidades são falhas que podem vir a ser exploradas por usuários maliciosos, estes podem desde obter acesso ao sistema, manipular dados ou até mesmo tornar todo serviço indisponível. Neste trabalho a criação de um analisador estático terá o intuito de pesquisar trechos de código ultrapassado.

A implementação de *refectoring* na grande parte das modernas IDEs mantem suporte para um simples conjunto de código onde o comportamento é intuitivo e fácil de ser analisado, quando características avançadas de uma linguagem com o java são usados descrever precisamente o comportamento de tarefas é de extrema complexidade além da implementação do refectoring ficar complexa e de difícil entendimento segundo Max Schäfer e Oege de Moor em [**?**]. Modernas IDEs como ecplise realizam complexos refectoring através da técnica de *microrefectoring* que nada mais é que a divisão de um bloco de código complexo em pequenas partes para tentar encontrar códigos mais intuitivos a serem modificados.

O analisador estático proposto nesse trabalho tem o objeto de identificar construções ultrapassadas e porções de código congelados que são utilizadas ao logo do desenvolvimento do software verificando o histórico do lançamento das *releases* de *software* livres desenvolvidos em especialmente usando a linguagem java. Ainda caberá ao desenvolve-

dor tomar a decisão caso existam construções ultrapassadas nas releases se adotará o *refectoring* ou manterá o código congelado expondo o mesmo a usuários maliciosos.

Capítulo 2

Fundamentação

Um ponto crítico quanto a análise de código fonte é o *parser* da linguagem, onde é necessário reconhecer uma frase para efetuar a interpretação ou fazer a tradução. Inicialmente é necessário identificar se a frase que será tratada é um *assignment* ou uma chamada de função.

Reconhecer uma frase acarreta em duas coisas, distingui-la de outras construções e identificar os elementos e as subestruturas que compõem esta frase. Por exemplo se uma frase for reconhecida como um *assignment*, pode-se identificar as variáveis a esquerda do operador = e uma expressão que é a subestrutura a direita. Este ato de reconhecer uma frase é denominado *Parse*.

2.1 Parse

Para conceber uma ferramenta de análise de código é necessário gerar um *parse* do código fonte o que torna isto uma tarefa complexa e desafiadora. Entretanto existem alguns padrões e neste será aborda os 4 mais importantes segundo Terence Parr em [**?**].

- **Mapping Grammars to Recursive-Descent Recognizers**
  Sua proposta é traduzir uma gramática para uma recursão descendente para reconhecer frases e sentenças em uma linguagem especificada por uma gramática. Este pradrão identifica o núcleo do fluxo de controle para qualquer recursão descendente e é utilizado nos 3 padrões seguintes. Para construir um reconhecedor léxico ou *parsers* manualmente o melhor ponto de início é a gramática, com isso este padrão fornece uma maneira simples de construir reconhecedores diretamente de sua gramática.

- **LL(1) Recursive-Descent Lexer**
  O objetivo deste pradrão é para emitir uma sequência de símbolos. Cada símbolo tem dois atributos primários: um tipo de *token*(símbolo da categoria) e o texto associado por exemplo no português, temos categorias como verbos e substantivos, bem como símbolos de pontuação, como vírgulas e pontos. Todas as palavras dentro de uma determinada categoria são do mesmo tipo de *token*, embora o texto associado seja diferente. O tipo de nome do *token* representa o categoria identificador. Então precisamos tipos de *token* para o vocabulário *string* fixa símbolos como também lidar com espaços em branco e comentários.

- **LL(1) Recursive-Descent Parser**
  Esse é o mais conhecido padrão de análise descendente recursiva. Ele só precisa a olhar para o símbolo de entrada atual para tomar decisões de análise. Para cada regra de gramática, existe um método de análise no analisador. Este padrão analisa a estrutura sintática da sequência sinal de uma frase usando um único *token lookahead*. Este analisador pertence à LL(1) classe do analisador de cima para baixo, em especial, porque usa um único sinal de verificação à frente (daí o "1"no nome). É o principal mecanismo de todos os padrões de análise subsequentes. Este padrão mostra como implementar as decisões de análise que utilizam um símbolo único da visão antecipada. É a forma mais fraca de descendente recursivo parser, mas o mais fácil de compreender e aplicar.

- **LL(k) Recursive-Descent Parser**
  Este padrão utiliza a o modo *top-down* para percorrer um árvore semântica com o auxílio de expressões booleanas que ajudam na tomada de decisão e estas expressões são conhecidas como predicados semânticos.

2.2 Análise estática

Análise estática é uma técnica automática no processo de verificação de software realizado por algumas ferramentas sem a necessidade de que o software tenha sido executado. Para Java exitem duas possibilidades de realizar tal análise na qual uma das técnicas realiza análise no código fonte e a outra a realiza no *bytecode* do programa segundo [**?**]. Neste trabalho ser utilizada a pesquisa baseada no código fonte sem que tenha sido executado devido a flexibilidade e infraestrutura consolidada encontrada no eclipse AST.

Um fato importante é que tal análise somente obtém sucesso se forem determinados padrões ou comportamento para que sejam pesquisados no software. Neste projeto o tais comportamentos são determinados por *visitors* conforme explica Gamma et. al. [**?**] devido a toda infraestrutura a qual as ferramentas do eclipse fornecem facilidade para que seja realizada uma análise baseada em padrões.

Devido a este trabalho de verificação de software é possível detectar falhas de forma precoce nas fases de desenvolvimento evitando que bugs e falhas sejam introduzidas e até mesmo postergados e isso é uma vantagem existe a economia de tempo com falhas simples, *feedback* rápido para alertar a equipe devido as falhas ocorridas e pode-se ir além de simples casos de testes podendo aprimorar estes para que fiquem mais rigorosos pois a partir do momento que o analisador encontrar uma falha é possível criar um teste de caso para que esta seja testada aumentando a confiabilidade do software.

Existe limitações nestes verificadores estáticos como em software desenvolvidos sem qualquer uso de padrões ou sem arquiteturas consolidadas, criado por equipes composta de desenvolvedores inexperientes o qual a ferramente poderá apontar erros que são falsos positivos que são erros detectados que não existem pois o analisador pesquisa por padrões e estruturas consolidadas. Tais problemas são desagradáveis porém não oferecem riscos ao desenvolvimento, podem afetar outras áreas como a de *refactoring* a qual poderá encontrar dificuldade em melhorar um código que não segue padrão. Vale ainda ressaltar que a penalidade de encontrar um falso positivo é a perda de tempo em fazer uma inspeção no código para comprovar se é ou não uma falha. Também há a possibilidade de falsos

negativos o que cabe ao programador verificar para evitar que tais limitação do analisador não se propague durante o ciclo de desenvolvimento.

2.3 Análise léxica

Ferramentas que operam em código-fonte conforme [**?**] começam por transformar o código em um série de *tokens*, descartando recursos sem importância de o texto do programa, tais como espaços em branco ou comentários ao longo do caminho. A criação do fluxo de sinal é chamado de análise lexical. Regras léxicas muitas vezes usam expressões regulares para identificar fichas. Observa-se que a maioria dos *tokens* são representados inteiramente por seu tipo, mas para ser útil, o *tokens* de identificação requer uma peça adicional de informação: o nome do identificador. Para habilitar o relatório de erro útil mais tarde, os *tokens* devem transportar pelo menos um outro tipo de informação com eles: a sua posição no texto-fonte (geralmente um número de linha e um número de coluna). Para as mais simples ferramentas de análise estática, o trabalho está quase concluído neste ponto. Se toda a ferramenta tem que fazer é combinar os nomes de funções, o analisador pode ir através do fluxo de *tokens* procurando identificadores, combiná-los com uma lista de nomes de funções, e relatar o resultados.

2.4 Parser

Um analisador de linguagem usa uma gramática livre de contexto (CFG) indicado por [**?**] para coincidir com os *tokens* correntes. A gramática é composta por um conjunto de produções que descrevem os símbolos (elementos) na língua. No Exemplo é enumerado um conjunto de produções que são capazes de analisar o fluxo de *tokens* de amostra.

```
1  stmt := if_stmt | assign_stmt
2  if_stmt := IF LPAREN expr RPAREN stmt
3  expr := lval
4  assign_stmt := lval EQUAL expr SEMI
5  lval = ID | arr_access
6  arr_access := ID arr_index+
7  arr_idx := LBRACKET expr RBRACKET
8
```

O analisador executa uma derivação, combinando o fluxo de sinal contra as regras de produção. Se cada símbolo é ligado a partir da qual o símbolo foi derivado, uma árvore de análise é formada. Na Figura: 2.1 mostra uma árvore de análise criada, usando as regras de produção do exemplo anterior. Omiti-se terminais de símbolos que não carregam nomes *(IF, LPAREN, RPAREN, etc.)*, para fazer o principais características da árvore de análise mais óbvia.

2.4.1 Paser JDT Eclipse

No caso do *parser* provido pela infraestrutura *JDT* do eclipse, a classe *ASTParser* contida na biblioteca *org.eclipse.jdt.core.dom* permite a criação de uma árvore de sintaxe

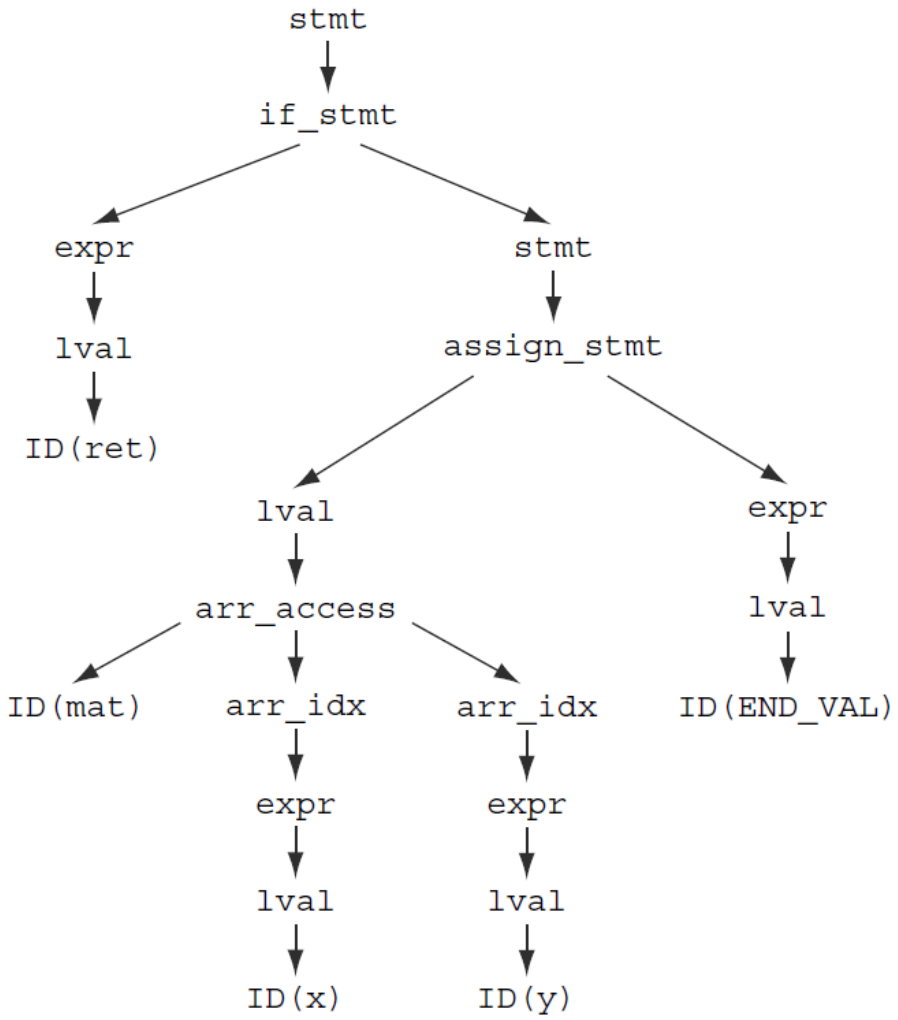

Figura 2.1: Árvore de parser.

abstrata.

Este procedimento é realizado em todos os aquivos *.java* contido em um projeto e com isso cada um possui uma referência de *CompilationUnit* o qual permite acesso ao nó raiz árvore sintática de cada arquivo. O parse é gerado conforme as últimas definições da linguagem utilizando *AST.JLS8*.

```
1   ASTParser parser = ASTParser.newParser(AST.JLS8);
2
3   Map<String, String> options = JavaCore.getOptions();
4   options.put(JavaCore.COMPILER_COMPLIANCE, JavaCore.VERSION_1_8);
5   options.put(JavaCore.COMPILER_CODEGEN_TARGET_PLATFORM, JavaCore.
    VERSION_1_8);
6   options.put(JavaCore.COMPILER_SOURCE, JavaCore.VERSION_1_8);
7
8   parser.setKind(ASTParser.K_COMPILATION_UNIT);
9   parser.setCompilerOptions(options);
10  parser.setSource(contents);
11
12  final CompilationUnit cu = (CompilationUnit) parser.createAST(null);
```

```
13      return cu;
14
```

Neste, o *parser* é realizado através de uma classe denominada de mesmo nome, a qual é instanciada um única vez no projeto através do padrão *singleton* [**?**].

2.5 Sintaxe abstrata

É possível fazer uma análise significativa em uma árvore de parser, e certos tipos de checagem estilísticas são mais bem executadas em uma árvore de análise, pois contém mais representações diretas do código assim como o programador escreve. No entanto, executar análise complexa em uma árvore de análise pode ser inconveniente. Os nós da árvore são derivados diretamente das regras de produção da gramática, e essas regras podem-se introduzir símbolos não terminais que existem apenas para fins de fazer a análise mais fácil e menos ambígua, ao invés de para o objetivo de produzir uma facilmente compreendido a árvore. É geralmente melhor para abstrair ambos os detalhes da gramática e as estruturas sintáticas presente no código fonte do programa. Uma estrutura de dados que faz estas coisas é chamado de uma árvore de sintaxe abstrata (AST). O objectivo da AST é fornecer uma versão padronizada do programa adequado para posteriores análises. A AST é normalmente construída associando código construção árvore com regras de produção da gramática. A Figura: 2.2 mostra uma AST. Observa-se que a instrução *if* agora tem uma outra ramificação vazia, o predicado testado pelo caso é agora uma comparação explícita para zero (o comportamento exigido pelo C), e acesso à matriz é uniformemente representada como uma operação de binário.

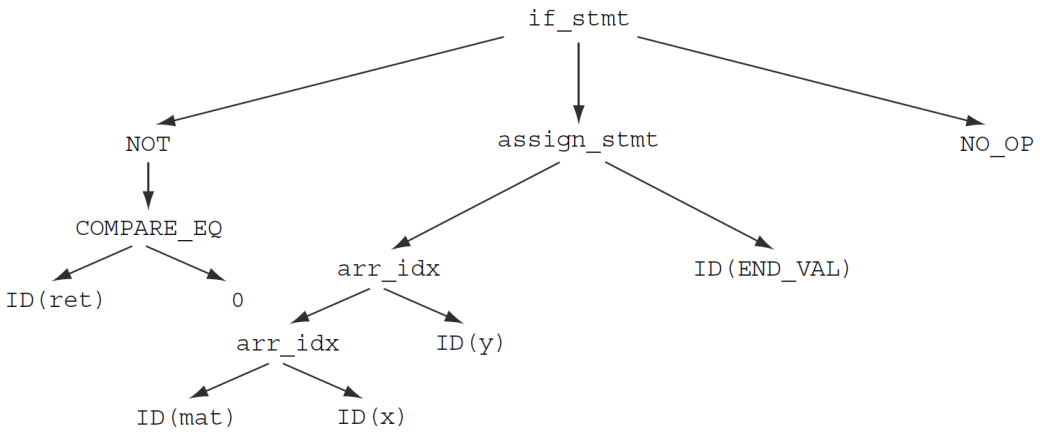

Figura 2.2: Árvore AST.

Capítulo 3

Suporte Ferramental

A Figura: 3.1 apresenta uma visão abstrata dos elementos que compõem o analisador estático desenvolvido durante a condução deste trabalho de graduação. Em linhas gerais, tal suporte ferramental recupera do sistema de arquivos todos os arquivos contendo código fonte escrito na linguagem Java, reliza o *parse* desses arquivos gerando uma representação intermediária correspondente, mais adequada para as análises de interesse deste projeto, aplica uma série de mecanismos de análise estática para coletar as informações sobre o uso das características da linguagem de programação e, por fim, gera os resultados no formato apropriado para as análises estatísticas (no contexto deste projeto, foi feita a opção pelo formato CVS).

Atualmente existem diversas tecnologias capazes de prover ferramentas para implementar um analisador estático conforme as nossas necessidades. Entretanto, devido a maior experiência dos participantes do projeto com uso da linguagem Java, foi feita a opção por se utilizar a infraestrutura da plataforma *Eclipse Java Development Tools* [**?**] (Eclipse JDT). O EclipseJDT [**?**] fornece um conjunto de ferramentas que auxiliam na construção de ferramentas que permitem processar código fonte escrito na linguagem de programação Java.

A plataforma Eclipse JDT é composta por 4 componentes principais *APT, Core, Debug* e *UI*. Neste projeto a plataforma foi usada essencialmente através do *JDT Core*, que dispõe de uma representação Java para a navegação e manipulação dos elementos de uma árvore sintática AST gerada a partir do código fonte, onde os elementos da representação correspondem aos elementeos sintáticos da linguagem (como pacotes, classes, interfaces métodos e atributos).

A AST provida pelo JDT é composta por 122 classes, como por exemplo existem 22 classe para representar sentenças como *IF-Than-Else, Switch, While, BreakStatement* entre outras. Exitem 5 classes que trabalham exclusivamente com métodos referenciados e 6 classes exclusiva que tratam os tipos declarados como classes, interfaces e enumerações em Java.

O Eclipse JDT [**?**] disponibiliza ainda um *Parser* para a linguagem Java que atende a especificação Java 8 da linguagem (a mais atual com lançamento público) e que produz a representação intermediária baseada no conjunto de classes Java mencionado anteriormente e que correspondem a uma AST do código fonte. A plataforma também oferece uma hierarquia de classes para traversia na AST, de acordo com o padrão de projeto *Visitors* [**?**], e que facilita a análise estática de código fonte.

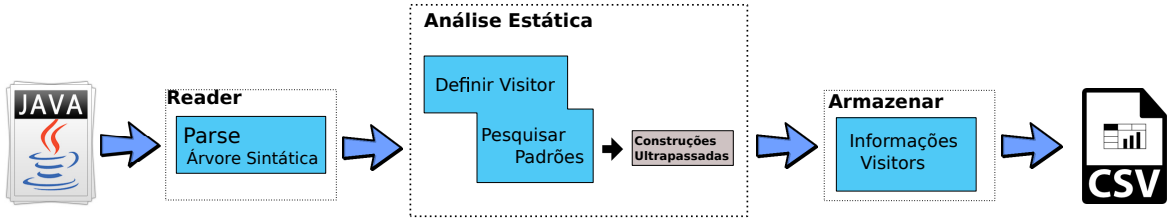

Figura 3.1: Alto nível de funcionamento do analisador estático.

O padrão de projeto *Visitor* [**?**] é um padrão de projeto de característica comportamental que representa uma operação a ser realizada sobre elementos de uma árvore de objetos. Neste caso a operação a ser realizadas é visitar nós de interesse da AST Java (como os nós que representam o uso de uma expressão Lambda em Java). Cada *visitor* permite que uma nova operação seja criada sem que a estrutura da árvore de objetos sofra alterações. Com isso é trivial adicionar novas funcionalidades em um *visitor* existente ou criar um novo.

Por outro lado, a biblioteca Eclipse JDT não fornece mecanismos para extração e exportação de dados. Entretanto, no contexto deste projeto, foi implementado um conjunto de classes que visam obter maior facilidade e flexibilidade na exportação das informações coletadas durante a traversia nos nós das ASTs. Essa flexibilidade foi alcançada com a utilização de introspecção de código que em Java é conhecido como *reflection*. O restante dessa seção apresenta mais detalhes sobre a arquitetura e implementação do analisador estático.

3.1 Entrada de dados

O analisador estático recebe como entrada um arquivo CSV (comma-separated values) que contém informações sobre os projetos a serem analisados, como nome do projeto, caminho absoluto para uma entrada no sistema de arquivos com o código fonte do projeto e a quantidade de linhas de código previamente computadas (conforme ilustrado na Figura: 3.2). As informações contidas no arquivo CSV são processadas por um conjunto de classes utilitária que varrem os diretórios de um determinado projeto e seleciona todos os arquivos fonte da linguagem Java. Os códigos fontes Java encontrados servem então como a entrada descrita na representação abstrata do analisador estático (Figura: 3.1). Ou seja, para cada projeto é feita uma varredura dos arquivos contendo código fonte Java, que são convertidos para uma representação intermediária (por meio de um parser existente); processados e analisados com uma infra-estrutura de *visitors*, e os resultados das análises são por fim exportados.

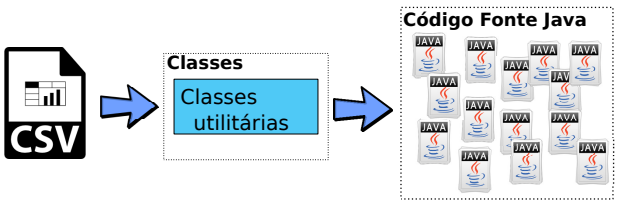

Figura 3.2: Input analisador estático.

3.2 Análise da Representação Intermediária

Após todos os código fontes Java identificado é dado início a verificação destes arquivos onde são processados e gerado um *parser* para que os *visitors* pesquisam padrões previamente estabelecidos onde a pesquisa elaborada com o principal objetivo de reconhecer elementos e sua subestruturas contidos no código fonte. Com isso a representação intermediária deste analisador é o processamento do código fonte para converte-lo em um *parser* para que os *visitor* realizem sua pesquisa. A Figura: 3.3 demonstra o mais alto nível do funcionamento deste projeto.

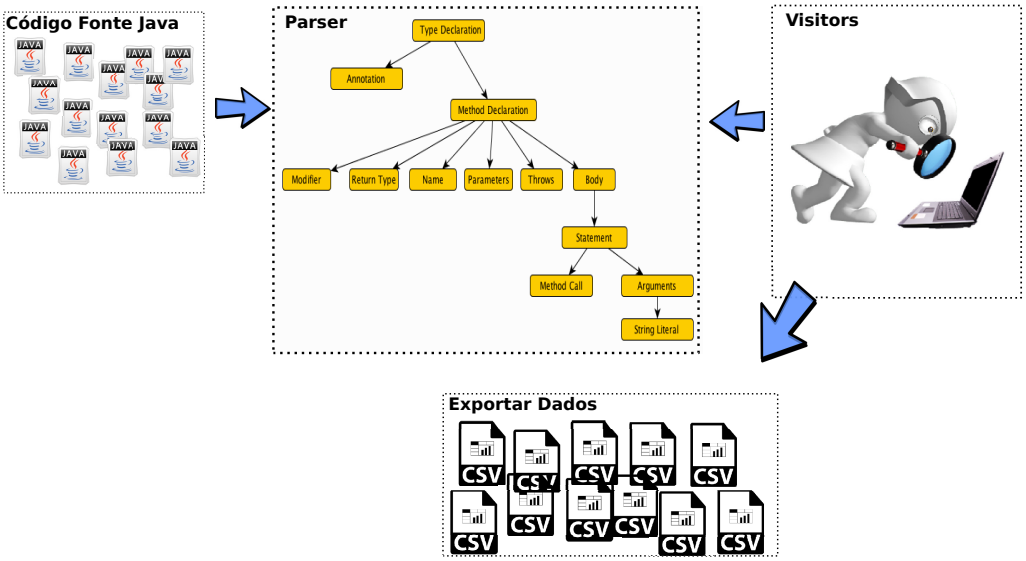

Figura 3.3: Funcionamento analisador estático.

rbonifacioparece que ocorre uma transição muito abrupta de uma discussao alto nivel para uma discussao muito rica em detalhes.

De forma mais técnica, a representação intermediária, para cada arquivo fonte o analisador estático realiza a coleta de dados utilizando uma infraestrutura de *visitors*. No contexto deste projeto, e objetivando um maior grau de reuso, todo *visitor* precisa herdar de uma classe abstrata e parametrizada em relação a um tipo **T**, a classe ***Visitor<T>***, onde o tipo **T** deve corresponder a classe usada para armazenar as informações coletadas pelo *visitor*. O parâmetro de tipo **T** faz referência a uma classe composta basicamente por atributos e por operações de acesso (*getters* e *setters*), que serve para representar os dados extraídos. Em geral, de acordo com a arquitetura do analisador estático proposto, para cada construção que se deseja identificar o perfil de adoção nos projetos, são criadas duas classes: uma classe (`public class C{ ...}`) para representar as informações de interesse associadas ao uso de uma construção particular da linguagem Java e uma classe (`public class ConstVisitor extends Visitor<C> { ...}`) que *visita* a construção de interesse na árvore sintática abstrata.

Por exemplo, a Listagem 24 apresenta o código necessário para visitar e popular informações relacionadas a declaração de enumerações. A classe `public class Visitor<T> { ...}` possui uma coleção de objetos do tipo parametrizado, sendo possível adicionar instâncias desses objetos com a chamada `collectedData.addValue()`. Note que o exemplo

apresentado corresponde a um dos mais simples *visitors* implementados. Outros *visitors* possuem uma lógica mais elaborada, como por exemplo os *visitors* que identificam oportunidades para usar construções como *multi-catch* ou *lambda expressions*.

```java
public class EnumDeclaration {
  private String file;
  private int startLine;
  private int endLine;

  //constructos + getters and setters.
}

public class EnumDeclarationVisitor extends Visitor<EnumDeclaration> {

  @Override
  public boolean visit(org.eclipse.jdt.core.dom.EnumDeclaration node) {

    EnumDeclaration dec = new EnumDeclaration(this.file, unit.getLineNumber
    (node.getStartPosition()), unit.getLineNumber(node.getLength() - node.
    getStartPosition()));

    collectedData.addValue(dec);

    return true;
  }

}
```

Listing 3.1: Classes usadas para capturar declaração de enumerações.

O diagrama da Figura: 3.4 exibe de maneira técnica o procedimento de criar um *Visitor* que detecte e colete informações dos tipos declarados no sistema, basta criar uma classes modelo *TypeDeclaration.java* e setar o parâmetro <**T**> como <*TypeDeclaration*>, com isso os dados serão extraídas pelo *Visitor, TypeDeclarationVisitor.java*, que identifica as informações pertinentes. refletir se esse parágrafo eh necessario

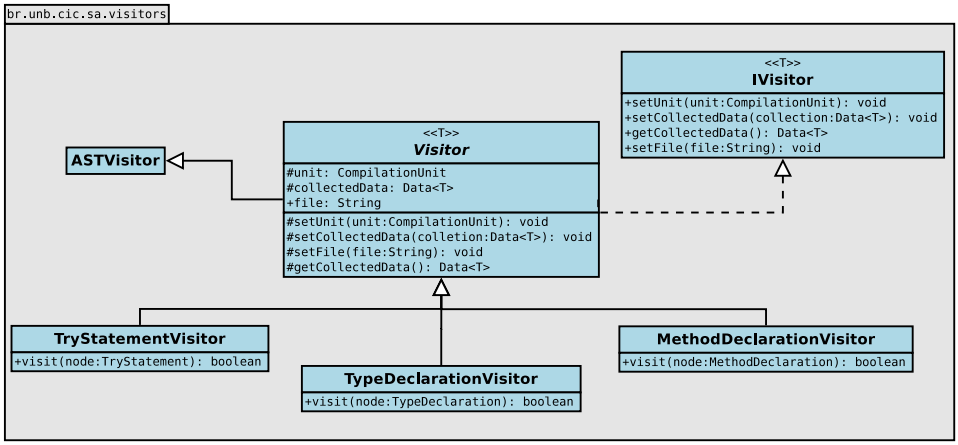

Figura 3.4: Funcionamento analisador estático.

A Tabela: 3.1 exibe todos os *Visitors* criados neste projeto com sua respectiva descrição. A complexidade ciclomática dos *Visitors* a qual é dada pela quantidade de caminhos

independentes em um trecho de código, vale ressaltar que a complexidade dos *Visitors* varia de 1 a 8 sendo a média 2,5 e a quantidade média de linha de código total necessária para escrever cada *Visitor* é 47.

importante incluir mais informações aqui, como complexidade ciclomática e linhas de código fonte de cada visitor.

Tabela 3.1: Tabela de Visitors criados com suas respectivas atribuições

Visitor	**Atribuição**
AICVisitor	Pesquisar *Anonymous Inner Class* declaradas.
EnumDeclarationVisitor	Pesquisa por *Enums* declarados.
ExistPatternVisitor	Pesquisa *EnhancedFor* que iteram sobre uma coleção procurando qualquer ocorrência nessa coleção.
FieldAndVariableDeclarationVisitor	Lista todos as variáveis declaradas como os respectivos tipos.
FilterPatternVisitor	Lista todos os *EnhancedFor* que iteram uma coleção filtrando elementos desta mesma coleção.
ImportDeclarationVisitor	Lista todos os *imports*.
LambdaExpressionVisitor	Pesquisa casos de utilização da expressões lambda.
LockVisitor	Verifica se nos métodos declarados existe alguma variável chamada Lock, ReentrantLock, ReadLock ou WriteLock.
MapPatternVisitor	Pesquisa *EnhancedFor* que iteram sobre uma coleção onde seja aplicado algum método sobre os itens desta coleção.
MethodCallVisitor	Verifica onde esta sendo utilizado reflection no projeto.
MethodDeclarationVisitor	Coleta informações sobre os métodos declarados nos projetos.
ScriptingEngineVisitor	Verifica se o projeto faz chamada a algum *Scripting*.
SwitchStatementVisitor	Pesquisa *Switchs* que utilizam *String* como parâmetro.
SwitchStringOpportunitiesVisitor	Pesquisa *If-Else* aninhados onde no *If* contenha *String*, caracterizando uma possibilidade de adoção de *Switch* com *String*.
TryStatementVisitor	Pesquisa *trys* que utilizar *resource*, adoção de *multicatch* e *trys* que possuem *catchs* aninhados.
TypeDeclarationVisitor	Pesquisa todos os tipos declarados.

Para tornar a solução mais extensível, foram utilizados os mecanismos de *Injeção de Dependência* e introspecção de código. Injeção de dependência DI, é um mecanismo de extensibilidade mais conhecido como um padrão de projeto originalmente denominado de inversão de controle IoC. De acordo com esse mecanismo, a sequência de criação dos objetos depende de como os mesmos são solicitados pelo sistema. Ou seja, quando um sistema é iniciado, os objetos necessários são instanciados e injetados de forma apropriada, geralmente de acordo com arquivo de configurações.

O mecanismo de injeção de dependência foi incorporado na arquitetura com o uso do *framework* Spring [**?**], o que não causou nenhum impacto significativo na solução inicialmente proposta e que não fazia uso de tal mecanismo— os *visitors* eram instanciados de maneira *programática*. O uso do mecanismo de injeção de dependência serviu para flexibilizar não apenas a incorporação de novos *visitors*, mas também para definir, de forma mais flexível, a estratégia de exportação dos dados coletados. Graças ao mecanismo de injeção de dependência, o desenvolvedor pode concentrar seu esforço na criação de *visitors*, fazendo como que estes implementem a lógica necessária para extrair as informações. Para

que novos *visitors* se conectem a plataforma, tornou-se necessário declarar o *visitor* no arquivo com a definição dos objetos gerenciados pelo Spring [**?**].

3.2.1 Exportação dos Dados

Na versão atual do suporte ferramental desenvolvido nessa monografia, os dados coletados pelo analisador estático são exportados exclusivamente no formato CSV. Esse formato facilita as análises estatísticas usando o ambiente e linguagem de programação R [**?**]. Também com foco na extensibilidade do sistema, os componentes envolvidos na geração de relatórios utilizam os mecanismos de injeção de dependência, mencionado na seção anterior, e introspecção de código, via a a API *Reflection* da linguagem de programação Java. Tal mecanismo oferece aos programadores a capacidade de escreverem componentes que podem observar e até modificar a estrutura e o comportamento dos objetos em tempo de execução.

A geração dos relatórios ocorre utilizando a classe `public class CSVData<T> { ... }` onde o tipo parametrizado <**T**> é o mesmo utilizado para representar os dados coletados pelos *visitors*. Os dados são obtidos através dos métodos de acesso (*getters*) destas classes e exportados para arquivos CSV.

O método *export()* da classe **CSVData<T>** descobre quais dados são armazenados nos objetos do tipo <**T**>, usando o mecanismo de introspecção de código. Com isso, é possível generalizar a implementação e simplificar a exportação de dados coletados a partir de *visitors* específicos. Ou seja, após a descoberta dos dados coletados pelos *visitors* usando introspecção, é possível recuperar os mesmos assumindo a existência de métodos de acesso (*getters* de acordo com a especificação Java Beans) e, como isso, exportá-los em arquivos CSV de saída. A Listagem: 50 demonstra o uso desse mecanismo para generalizar a exportação dos dados.

```
1  public class CSVData<T> implements Data<T>{
2    ...
3    @Override
4    public void export() {
5      try (FileWriter writer =
6        new FileWriter(this.makeCsv(head), true)){
7
8        StringBuffer str = new StringBuffer("");
9
10       if(data == null) { return;  }
11
12       for(T value : data) {
13         //reflection code...
14         for(Field f: value.getClass().getDeclaredFields()){
15
16           String fieldName = f.getName();
17           String prefix = "get";
18
19           if(f.getType().isPrimitive() &&
20             f.getType().equals(Boolean.TYPE)) {
21             prefix = "is";
22           }
23
24           String methodName = prefix +
```

```
25              Character.toUpperCase(fieldName.charAt(0)) +
26              fieldName.substring(1);
27
28            try {
29              Method m = value.getClass().getDeclaredMethod(methodName);
30              str.append(m.invoke(value));
31              str.append(";");
32            }catch(NoSuchMethodException |  IllegalAccessException |
33                  IllegalArgumentException | InvocationTargetException e) {
34              throw new RuntimeException("Type " +
35                         value.getClass().getName() +
36                         " must have a method named " + methodName);
37            }
38          }
39          writer.append(str.toString());
40          writer.append("\n");
41
42        }
43        writer.flush();
44
45      }catch(Exception e) {
46        e.printStackTrace();
47      }
48    }
49  }
```

Listing 3.2: Reflection na geração de relatório.

Capítulo 4

Resultados

Neste capítulo foi investigado empiricamente a adoção de *features* Java e *standard libraries* por replicação de um estudo existente de Parnin et al [**?**]. Adicionalmente a contribuição deste trabalho é abordar duas características da linguagem Java, *Java Generics* e *Java Lambda Expression* onde a questão que direcionou esta contribuição foi **RQ1: Qual o típico uso de Java Generics e Java Lambda Expressions?**

A replicação do estudo de *Java Generics* ocorreu através de projetos opensource. Onde adicionalmente além da replicação do trabalho estudo de Parnin et al [**?**] este adicionou a compreensão de como *Java Generics* se correlaciona com a *release* inicial dos projetos selecionados.

Também foi investigado empiricamente como esta ocorrendo a adoção de *Java Lambda Expression*, vale ressaltar para melhor conhecimento não há estudo empírico que investigou tal questão até o presente momento. Ainda em relação a *Java Lambda Expression* foi levantando um questionamento nas comunidade *opensources* para descobrir qual o comportamento adotado pelas equipes de desenvolvedores após o lançamento desta *feature*.

Para a realização a investigação os 47 projetos opensource escolhidos foram separados em 3 grupos **G1** projetos iniciados antes do lançamento de *Java Generics*, **G2** projetos iniciados após o lançamento de *Java Generics* e **G3** projetos com a última *release* em 2015. Alguns destes projetos são os mesmos utilizados em, [**?** **?** **?**], e também fora separados pela natureza da projeto, Aplicações, Bibliotecas e Servidores/Banco de dados conforme tabela: 4.1 o que totalizou mais de 8.5M de LOC detalhados na Tabela: 4.1.

Para a investigação foram utilizados os *visitors* criados exbibidos na Tabela: 3.1. A conclusão do estudo de *Java Generics* foram utilizados os seguintes *visitors*:*MethodCallVisitor, FieldAndVariableDeclarationVisitor e TypeDeclarationVisitor* e para o *Java Lambda Expression* estes: *AICVisitor, ExistPatternVisitor, FilterPatternVisitor, LambdaExpressionVisitor, LockVisitor e MapPatternVisitor*.

4.1 Adoção de Java Generics

Relacionado com a adoção de *Java Generics*, a maioria dos projetos apresentam um porção significantiva entre a quantidade de tipos genéricos e a quantidade total de tipos declarados em média(5.31% e 12.31%). Pode-se comprovar que em 16% dos sistemas não declaram nenhum tipo genérico e que o projeto *Commons Collections* é o sistema que

com a relação mais expressiva de tipos parametrizados: 75% de todos os tipos declarados são genéricos.

Também foi investigado a relação entre tipos genéricos declarados e todos os tipos considerando os tipos e idade dos sistemas. Tabela: 4.2 apresenta um resumo desta observação onde é possível comprovar que o uso típico de *Java Generics* não muda significativamente entre os tipos de projetos Java, embora essa proporção seja mais baixa para Aplicações e servidores/bancos de dados com versões anteriores ao lançamento do Java SE 5.0.

Existe um número expressivo de atributos e variáveis declaradas como instâncias de tipos genéricos. A partir de 925.925 variáveis e atributos declarados em todos os projetos, 84.880 são instâncias de tipos genéricos, 10 % de todas as declarações. Além disso, a partir destes atributos e variáveis declaradas como instância de tipos genéricos, quase 17% são instâncias dos tipos presentes na Tabela: 4.3. Note que, em um trabalho anterior, Parning et al. [**?**] apresenta `List<String>` com quase 25% de todos os genéricos. O que pode ser confirmado que `List<String>` ainda é o tipo com maior frequência deuso entre os tipos genéricos. No entanto com 730.720 métodos, apenas entre 6157, 0.84%, são *métodos parametrizados.*

Também fora investigado o uso mais avançado de *Java Generics*, incluindo construções que fazem polimorfismo parametrizado. Com este recurso é possível criar classes paramétricas que aceitam qualquer tipo **T** como argumento, uma vez que um tipo **T** satisfaça um determinada pré-condição isto é, o tipo **T** deve ser um qualquer um subtipo (usando o modificador *extends*) ou um super-tipo (usando o modificador *super*) de um determinado tipo existente. Estes modificadores pode ser usado tanto na declaração de novos tipos, bem como na declaração de campos e variáveis em combinação com o *wildcard* **?**). A partir de 4355 tipos genéricos declarados em todos os sistemas, descobriu-se que 1.271, quase 30% usam alguns desses modificadores, extends, super, ou ?. Notavelmente, o modificador *extends* é o mais comum, e está presente em todos os tipos genéricos que usam os modificadores *?* e *super*. Alguns casos de uso são combinações de modificadores, como no exemplo da Listing: 4, onde a classe IntervalTree (projeto CASSANDRA) é parametrizado de acordo com três parâmetros de tipo (C, D e I). Com relação aos campos e declarações de variáveis, quase 13% de todos os casos genéricos usam o *?* *wildcard* e 3,13% usam o *extends*.

```
1 public class IntervalTree<C extends Comparable<? super C>, D, I extends
    Interval<C, D>> implements Iterable<I>{
2   //...
3 }
```

Listing 4.1: Declaração não trivial de Generics.

Os resultados mostram que *Java Generics* é uma *feature* em que corresponde a 5% de todos os tipos declarados dos sistemas, portanto, uma grande quantidade de código repetido e tipo coerções (moldes) foram evitado usando tipos genéricos. Além disso, a partir desses tipos genéricos, quase 30% usam um recurso avançado (como extends e super envolvendo parâmetros de tipo). Também foi descoberto que quase 10% de todos os atributos e variáveis declaradas são tipos genéricos, embora a maior parte são instâncias de tipos genéricos da biblioteca *Java Collection*. Finalmente, embora Parnin et ai. [**?**] argumentam que uma classe como *StringList* pode cumprir 25% das necessidades de desenvolvedores entretanto, o uso de *Java Generics* não deve ser negligenciada devido aos benefícios que são incorporados ao sistema.

4.2 Adoção de Java Lambda Expression

Considerando os sistemas pesquisados, o uso de Java Lambda Expression ainda é muito limitado, independente das expectativas e reivindicações sobre os possíveis benefícios dessa construção. Na verdade, apenas cinco projetos adotam este recurso conforme a Tabela: 4.4, embora o cenário de uso (quase 90%) está relacionado com testes unitários. O que em um primeiro momento leva a indagar se algum *framework* de teste unitário conduz o desenvolvedor para o emprego deste recurso no teste. Entretanto após analisar manualmente o código fonte não é encontrado nenhum indício da adoção de *Java Lambda Expresssion* para testes unitários o que pode-se concluir que tais testes ocorreram de forma *ad-hoc* através de esforços individuais de cada desenvolvedor. Ou seja, a partir de milhares de casos de testes unitários no Hibernate, apenas poucos testes para uma biblioteca específica (relacionados com cache) usam *Java Lambda Expression*. Este pequeno uso de *Java Lambda Expression* pode ser principalmente motivado por uma decisão estratégica do projeto para evitar a migração do código fonte ultrapassado para a versão mais atual.

Foi enviado mensagens para grupos do desenvolvedores sobre o assunto, e algumas respostas esclarecem a atual situação da adoção de *Java Lambda Expression*. Primeiro de tudo, para os sistemas estabelecidos, as equipes de desenvolvedores muitas vezes não podem assumir que todos os utilizadores são capazes de migrar para uma nova versão do *Java Runtime Environment*. Por exemplo, o seguinte *post* explica uma das razões para não adotar algumas construções adicionada a linguagem Java: "É, sobretudo, para permitir que as pessoas que estão vinculados (por qualquer motivo) a versões mais antigas do JDK utilizem nosso software. Há um grande número de projetos que não são capazes de usar novas versões do JDK. Eu sei que este é um tema controverso e acho que a maioria de gostaria de usar todos esses recursos. Mas não devemos esquecer as pessoas usando nosso software em seu trabalho diário"(http://goo.gl/h0uloY).

Além disso, um abordagem inicial utilizando uma nova característica da linguagem é mais oportunista. Ou seja, os desenvolvedores não migram todo o projeto, mas em vez disso as modificações que introduzem estas novas construções de linguagem ocorrem quando eles estão implementando novas funcionalidades. Duas respostas a estas perguntas deixam isso claro: "Nós tentamos evitar reescrever grandes trechos de código base, sem uma boa razão. Em vez disso, tirar proveito dos novos recursos de linguagem ao escrever novo código ou refatoração código antigo."(https://goo.gl/2WgjVG) e "Eu, pessoalmente, não gosto da idéia de mover todo o código para uma nova versão Java, eu modifico áreas que atualmente trabalho."(http://goo.gl/ GQ4Ckn). Observe que não se pode generalizar estas conclusões com base nessas respostas, uma vez que não foi realizado um inquérito mais estruturado. No entanto, estas respostas podem apoiar trabalhos contra a adoção antecipada de novos recursos de linguagem por sistemas estabelecidos com uma enorme comunidade de usuários.

Também foi efetuada uma busca no *STACK OVERFLOW* tentando descobrir se *Java Lambda Expression* é um tema discutido atualmente ou não [1], utilizando *tags* Java e Lambda. Foi encontrada mais de 1000 questões respondidas. Este número é bastante expressivo, quando considerou-se uma busca por questões marcadas com as *tag* de Java Generics levou-se a um número próximo de 10 000 perguntas, embora *Generics* tenha sido introduzido há mais de dez anos. Possivelmente, *Java Lambda Expression* está sendo

[1] Última pesquisa realizada em Novembro 2015

usado principalmente em pequenos projetos e experimentais. Isso pode contrastar com os resultados de [?], que sugerem uma adoção antecipada de novos recursos da linguagem (mesmo antes de lançamentos oficiais). Com base nesses resultados, pode-se comprovar com este trabalho que a adoção antecipada de novos recursos da linguagem ocorre em projetos pequenos e experimentais.

Outra investigação foi se existia a oportunidade de adoção de *Java Lambda Expression* nos projetos estudados. Desta forma, foi complementado um testou maior [?], que investigou as mesmas questões porém eu um número de inferior de projetos. Existem dois cenários típicos para *refactoring* utilizando Expressões Lambda: *Annonymous Inner Classes* (AIC) e *Enhanced for Loops* (EFL). É importante notar que nem todas as AICs e EFLs podem ser reescritas utilizando *Java Lambda Expression*, e existem rígidas precondições que são detalhadas em [?]. Neste trabalho foi utilizado uma abordagem mais conservadora para considerar se é possível refatorar Enhanced for loop para *Java Lambda Expression* o que evita falsos positivos. Entretanto, foi considerado somente oportunidades de refatorar EFL para *Java Lambda Expression* em 3 casos particular: EXIST PATTERN, BASIC FILTER PATTERN e BASIC MAPPING PATTERN de acordo com os Listing: 11, 12 e 15.

```
//...
for(T e : collection){
  if(e.pred(args)){
    return true;
  }
}
return false;

//pode ser refatorado para:
return collection.stream().anyMatch(e->pred(args));
```

Listing 4.2: EXIST PATTERN.

```
//...
for(T e : collection){
  if(e.pred(args)){
    otherCollection.add(e);
  }
}

//pode ser refatorado para:
collection.stream().filter(
    e->pred(args).forEach(e->otherCollection.add(e)
  );
```

Listing 4.3: FILTER PATTERN.

```
//...
for(T e : collection){
  e.foo();
  e = blah();
  otherCollection.add(e);
}

//pode ser refatorado para:
```

```
9
10 collection.stream().forEach(e->{
11   e.foo();
12   e = blah();
13   otherCollection.add(e);
14 });
```

Listing 4.4: MAP PATTERN.

Mesmo com um abordagem conservadora, foram encontrada 2496 casos em que poderia ser efetuado *refactoring* EFL para Expressão Lambda. Atualmente, a maior parte destes casos 2190 correspondem ao MAP PATTERN.

Também foi investigado o típico uso de características de concorrência em Java. Foi encontrado que 39 de 43 dos sistemas declarados classes que herdam de *Thread* ou implementam a interface *Runnable*. A Tabela: 4.5 apresenta a relação destas declarações quando considerado o número total de tipos declarados, agrupados projetos estudados. Note que o uso de classes que herdam de *Thread* ou implementam *Runnable* é elevado considerando os casos de servidores e database.

4.3 Análises adicionais

Os principais resultados dessa monografia estavam relacionados á investigação discutida nas seções anteriores. Por outro lado, a infraestrutura construída durante a realização desse trabalho de conclusão de curso permitiu investigar outros mecanismos e oportunidades que surgiram com a evolução da linguagem Java. Conforme citado por Jeffrey L. Overbey e Ralph E. Johnson [**?**], Java mantém construções ultrapassas ao longo das versões do desenvolvimento do software o que de fato acontece devido a compatibilidade mantida entre as versões da linguagem. Tais construções somente seriam evitadas caso ocorresse uma adaptação da linguagem mais impactante, como ocorreu na linguagem Fortan [**?**], introduzindo o paradigma de orientação a objeto e levando a quebra de compatibilidade com a versões anteriores da linguagem.

Baseado nesta assertiva, foi feita uma investigação adicional que visa minerar o uso de construções obsoletas em código fonte Java existente e encontrar possíveis casos de trechos de código que poderiam ter sido evoluídos ao longo das versões de Java. Essas situações caracterizam cenários potenciais de melhoria de código e que preservam o comportamento do sistema (tipicamente um *refactoring*), com o objetivo único de usar construções introduzidas nas versões 7 e 8 da linguagem Java. Dentre as característica investigadas, as seguintes estão sendo reportadas no restante desse capítulo.

4.3.1 Oportunidades para uso da construção `multi-catch`

O uso da construção `multi-catch` permite reduzir qualquer lógica duplicada existente em blocos `catch` distintos de uma contrução `try-catch`. Com as análises realizadas, foi possível identificar uma quantidade significativa de oportunidades de uso dessa construção, conforme exibido na Figura: 4.1. Ao todo, foram encotrados 10 368 blocos `try` que possuem *catchs* repetidos. Essas ocorrências estão distribuídas em 7297 arquivos e totalizamo 97 347 LOC. Importante observar que o teste de similaridade entre os blocos `catch` foi realizado através de uma chamada a um método exteno que verificava a igualdade da

árvore sintática. Apesar dessa abordagem não fazer uso de uma estratégia de análise de similaridade de código mais robusta, a mesma pode ser facilmente alterada de acordo com algum algoritmo existente.

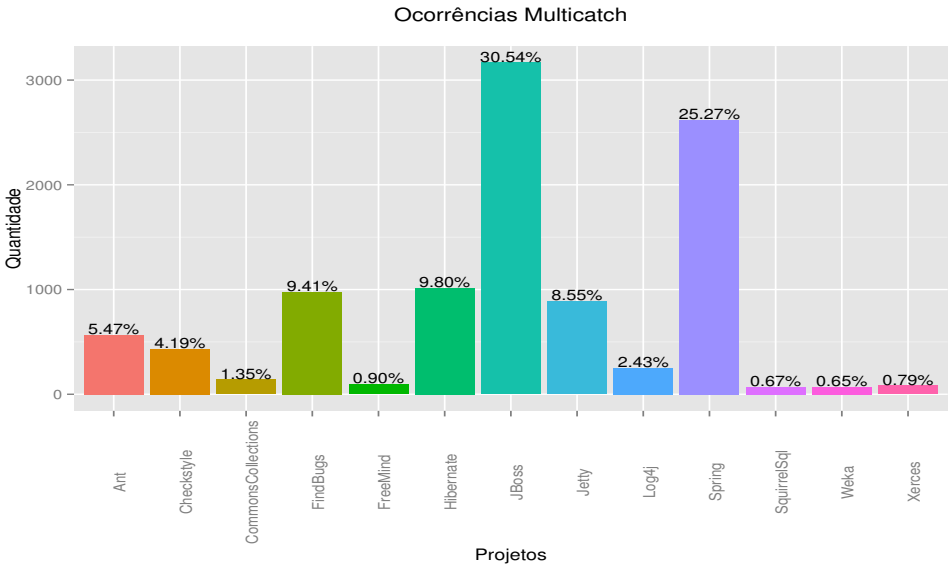

Figura 4.1: Oportunidades de *Multicatch* nos projetos.

Considere os exemplos nas listagens X e Y, encontrados na classe `AbstractNestablePropertyAccess` do projeto *Spring 4.2.0.RC2*. Neste caso, é possível reestruturar o código para usar a construção `multi-catch`, o que levaria a uma redução de 40% para esse trecho de código. Um simples *refactoring* unindo todos os blocos que potencialmente se beneficiariam do uso de blocos `multi-catch` levaria a uma redução de 68063 LOC, tornando essa construção útil para reduzir a quantidade de linhas de código em duplicidade de um projeto.

```
1  // Sem uso de Multicacth 17 LOC
2  try {...}
3  catch (ConverterNotFoundException ex) {
4    PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
       nestedPath + propertyName, oldValue, newValue);
5    throw new ConversionNotSupportedException(pce, td.getType(), ex);
6  }catch (ConversionException ex) {
7    PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
       nestedPath + propertyName, oldValue, newValue);
8    throw new TypeMismatchException(pce, requiredType, ex);
9  }catch (IllegalStateException ex) {
10   PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
       nestedPath + propertyName, oldValue, newValue);
11   throw new ConversionNotSupportedException(pce, requiredType, ex);
12 }catch (IllegalArgumentException ex) {
13   PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
       nestedPath + propertyName, oldValue, newValue);
14   throw new TypeMismatchException(pce, requiredType, ex);
15 }
```

```java
// Com uso de Multicatch 10 LOC
try {...}
catch (ConverterNotFoundException ex | IllegalStateException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
  nestedPath + propertyName, oldValue, newValue);
  throw new ConversionNotSupportedException(pce, td.getType(), ex);
}catch (ConversionException ex | IllegalArgumentException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject, this.
  nestedPath + propertyName, oldValue, newValue);
  throw new TypeMismatchException(pce, requiredType, ex);
}
```

A figura: 4.2 exibe as ocorrências nos projetos mais numerosos do estudo. Onde todas as ocorrências totalizam 17100 LOC e após um simples *refactoring*, conforme o exemplo anterior, obtem-se 5955 LOC o que acarreta uma redução da ordem de 65% de código duplicado em blocos `catch`.

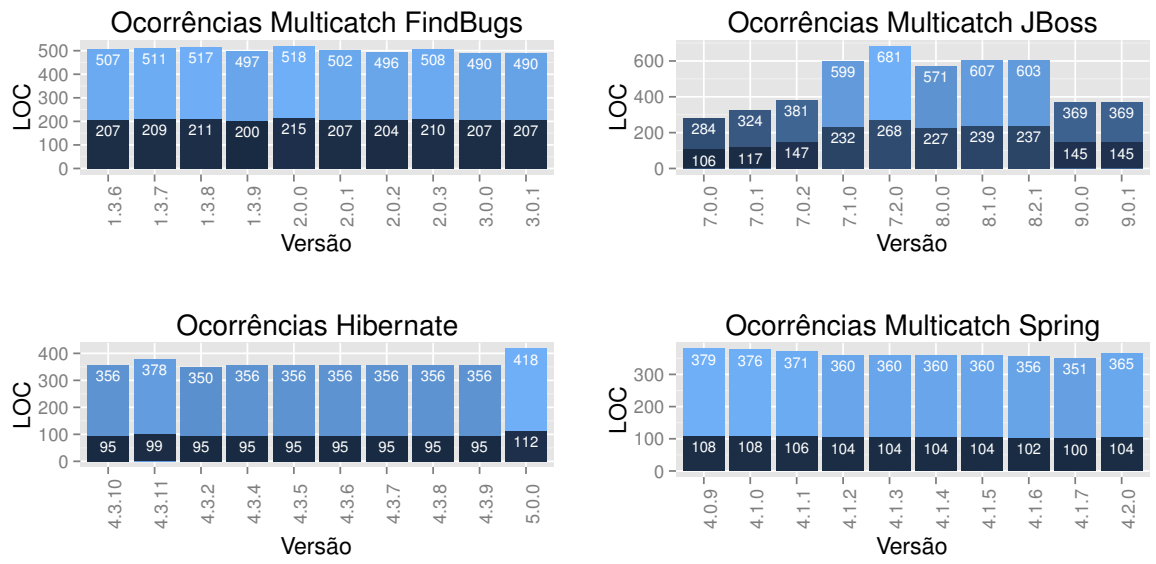

Figura 4.2: Oportunidades de *Multicatch* nos projetos.

4.3.2 Try Resource

O visitor responsável por detectar a adoção desta *feature* pesquisou por padrões como o código abaixo onde não foi pesquisado oportunidades de *refactoring* mas sim onde esta característica estava sendo adotada.

```java
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Com o advento do Java 7 foi introduzido o *Try with Resource* onde um *resource* é um objeto que pode ser fechado antes do programa ser encerrado. Com isso promoveu uma maior autonomia e flexibilidade ao programador. Foi encontrado no total 284321 *trys* nos quais esta *feature* totaliza 1.8%, ou seja, 5186 casos. Conforme exibido na figura: 4.3 exibe a distribuição desta *feature* entre os projetos onde pode-se constatar que somente 4 projetos fizera a adoção. Com exceção do projeto *Jetty* que totaliza 87% dos casos os demais projetos não aderiram de forma massiva esta caracterísitica.

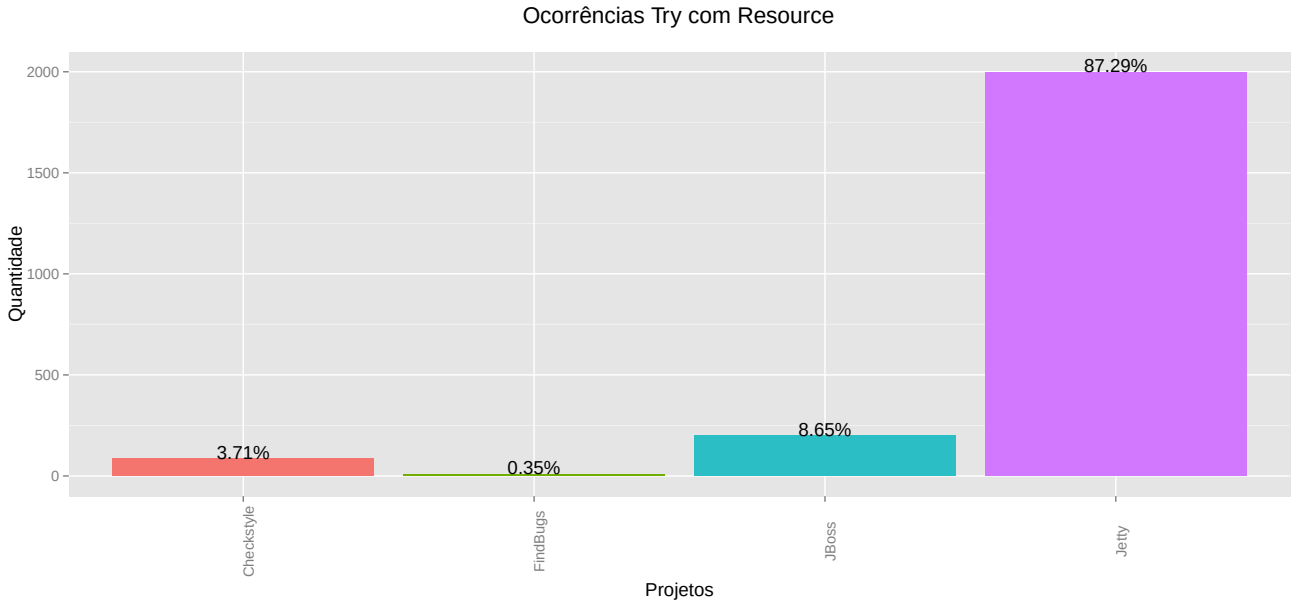

Figura 4.3: Oportunidades de *Try with Resource* nos projetos.

4.3.3 Switch String

O *Switch with String* veio no Java 7 e trouxe este recurso que porém pequeno é efetivamente útil porque ajuda a escrever o código mas legível e além do mais o compilador irá gerar o *codebyte* com mais eficiência em comparação com *if-then-else*. Nos projetos analisados foram somente encontrados 77 oportunidades de migração para *switch with string*.

Conforme exibido na imagem acima das diferentemente do *switch with string* o uso desse recurso seria mais bem empregado no projeto o *Weka* pois teria uma redução de 31 *if - else* anhinhados comparando *Strings* onde seria mais elegante a evolução sem grande impacto pois teria um código mais elegante e atual.

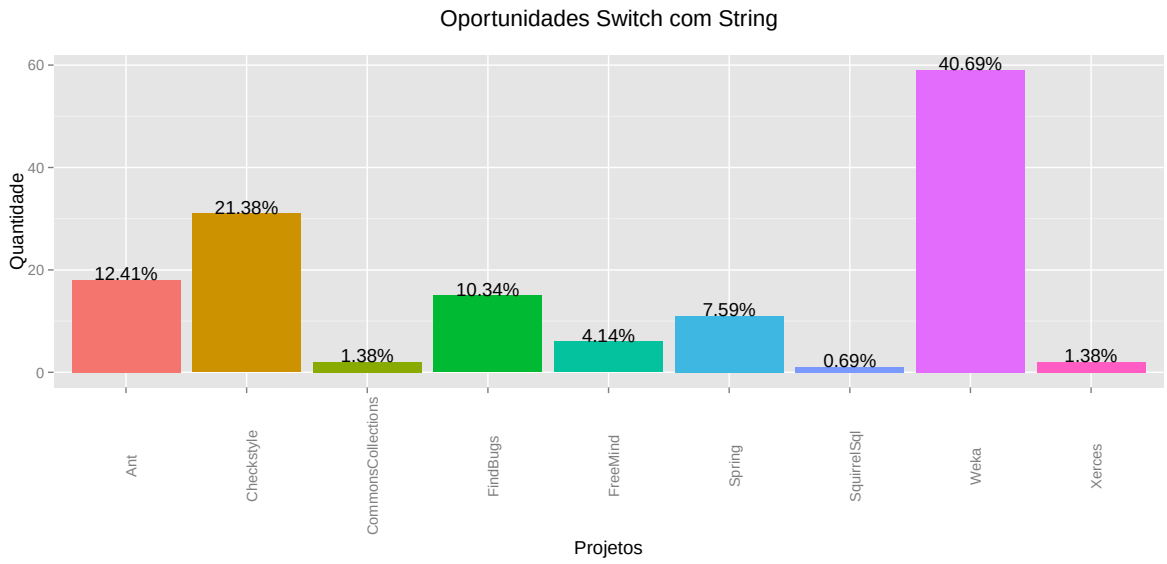

Figura 4.4: Oportunidades de *Switch with String* nos projetos.

Tabela 4.1: Projetos.

	System	Release	Group	LOC
Application	ANT	1.9.6	G1	135741
	ANTLR	4.5.1	G1/G3	89935
	Archiva	2.2.0	G2/G3	84632
	Eclipse	R4_5	G1	13429
	Eclipse-CS	6.9.0	G1	20426
	FindBugs	3.0.1	G1/G3	131351
	FitNesse	20150814	G2/G3	72836
	Free-Mind	1.0.1	G1	67357
	Gradle	2.7	G2	193428
	GWT	2.7.0	G2	15421
	Ivy	2.4.0	G2/G3	72630
	jEdit	5.2.0	G1	118492
	Jenkins	1.629	G2/G3	113763
	JMeter	2.13	G1/G3	111317
	Maven	3.3.3	G1/G3	78476
	Openmeetings	3.0.6	G2/G3	50496
	Postgree JDBC	9.4.1202	G1/G3	43596
	Sonar	5.0.1	G2/G3	362284
	Squirrel	3.4.0	G1	252997
	Vuze	5621-39	G1	608670
	Weka	3.6.12	G1	274978
Library	Axis	1.4	G2	121820
	Commons Collections	4.4.0	G1	51622
	Crawler4j	4.1	G2/G3	3986
	Hibernate	5.0.1	G1/G3	541116
	Isis	1.9.0	G2	262247
	JClouds	1.9.1	G2/G3	301592
	JUnit	4.1.2	G1/G3	26456
	Log4j	2.2	G1/G3	69525
	MyFaces	2.2.8	G2/G3	222865
	Quartz	2.2.1	G2	31968
	Spark	1.5.0	G2/G3	31282
	Spring-Framework	4.2.1	G1/G3	531757
	Storm	0.10.0	G2/G3	98344
	UimaDucc	2.0.0	G2	96020
	Wicket	7.0.0	G2/G3	211618
	Woden	1.0	G2/G3	29348
	Xerces	2.11.0	G1	126228
Servers - Databases	Cassandra	2.2.1	G2/G3	282336
	Hadoop	2.6.1	G2/G3	896615
	Jetty	9.3.2	G1	299923
	Lucene	5.3.1	G1	506711
	Tomcat	8.0.26	G1/G3	287897
	UniversalMedia Server	5.2.2	G3	54912
	Wildfly	9.0.1	G1/G3	392776
	Zookeeper	3.4.6	G3	61708

Tabela 4.2: Resumo dos tipos agrupados por idade e do tipo dos projetos.

Tipo de Projeto	Antes Java SE 5.0	Tipo	Tipo Genérico	Ratio(%)
Aplication	Yes	18168	177	0.99
Aplication	No	16148	744	5.39
Library	Yes	21537	1198	5.26
Library	No	22639	947	4.36
Server/Database	Yes	18038	552	2.97
Server/Database	No	11790	760	6.06

Tabela 4.3: Tipo declarado X Número de instância

Tipo	Número de Instância
`List<String>`	4993
`Class<?>`	3033
`Set<String>`	2872
`Map<String,String>`	2294
`Map<String,Object>`	1554

Tabela 4.4: Ocorrências de Expressões Lambda.

Sistema	Ocorrências Expressões Lambda
`Hinernate`	168
`Jetty`	2
`Lucene`	11
`Spark`	77
`Spring-framework`	121

Tabela 4.5: Classes concorrentes que *extends Thread* ou implementam *Runnable*.

Tipo Sistema	Relação dos Tipos de Concorrência
`Applications`	0.69
`Libraries`	0.34
`Serves and database`	1.52

Capítulo 5

Considerações Finais e Projeto Fututos

Com este projeto pode-se comprovar que existe muito código duplicado em projetos renomados tais códigos existe pelo fato da equipe de desenvolvimento não adotar uma *feature* ou uma característica lançada como no caso de *multicatch* para eliminar *catchs* aninhados com tratamento iguais ou até mesmo utilizar expressões lambada para eliminar alguns *foreach* o que proporcionaria uma evolução do software junto com a evolução da linguagem.

5.1 Projeto Futuro

Como projeto futuro a especialzação deste trabalho para efetuar o *refectoring* de maneira automática seria de grande importância na evolução de código congelado. Com os *visitors* existentes é possível indentificar oportunidades reais de empregar *multicatch* e *swicht com String* o que seria uma evolução.

Visando o enrriquecimento para tornar esta ferramenta robusta acoplar uma linguagem especializada em metaprogramação para efeturar este trabalho seria de suma importância tanto para evoulução dos alunos quanto da ferramenta. Com isso a linguagem Rascal MPL se destaca tendo em vista que é uma linguagem que atenderia perfeitamente esta evolução dada sua fácil integração com Java e por ser uma linguagem de metaprogamação.