**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.
87 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 ideia 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 pudéssemos 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 uma problemática hora ajudando a resolver barreiras as quais não conseguimos sozinhos.
E ainda a UnB por todo seu corpo docente que sem este essa jornada não seria concluída com excelência, em especial ao professor 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 Abreviaturas** **vii**

**1 Introdução** **1**
1.1 Introdução . . . 1
1.2 Objetivos . . . 1
1.3 Metodologia . . . 2

**2 Fundamentação** **3**
2.1 Evolução Linguagem Java . . . 3
2.2 Engenharia de Linguagens de Software . . . 5
2.3 Representação Intermediária . . . 8
2.4 Refactoring . . . 9
2.5 Análise estática . . . 11

**3 Suporte Ferramental para Minerar Padrões de Uso de Construções da Linguagem Java** **12**
3.1 Definição dos Projetos a Serem Analisados . . . 13
3.2 Análise da Representação Intermediária . . . 15
3.2.1 Descrição dos Visitors . . . 16
3.2.2 Extensibilidade para Inclusão de Novos Visitors . . . 18
3.3 Exportação dos Dados . . . 18

**4 Resultados** **20**
4.1 Adoção de Java Generics . . . 22
4.2 Adoção de Java Lambda Expression . . . 23
4.3 Análises adicionais . . . 26
4.3.1 Oportunidades para uso da construção `multi-catch` . . . 26
4.3.2 Try Resource . . . 28
4.3.3 Switch String . . . 30

**5 Considerações Finais e Trabalhos Futuros** **32**

**Referências** **33**

# Lista de Figuras


2.1 Resumo cronológico da linguagem Java. . . . . 4
2.2 Portabilidade da liguagem Java. . . . . 4
2.3 Fases de aplicações com linguagens. . . . . 6
2.4 Fase do pipiline do FindBugs. . . . . 6
2.5 Ferramentas necessárias para construção do analisador estático. . . . . 7
2.6 Representação da uma frase. . . . . 8

3.1 Visão geral da arquitetura do analisador estático . . . . . 13
3.2 Classe que representa o programa principal do analisador estático . . . . . 14
3.3 Implementação do método `analyse`, na classe `ProjectAnalyser`. . . . . 15
3.4 Classes usadas para capturar declarações de enumerações. . . . . 16
3.5 Exportação de dados usando o mecanismo de introspecção de código. . . . 19

4.1 Oportunidades de `multi-catch` nos projetos. . . . . 27
4.2 Adoção de `Try-Resource` nos projetos. . . . . 29
4.3 Oportunidades de *refactoring* em `if-then-else` por sistema. . . . . 31

# Lista de Tabelas

3.1 Estimativa da complexidade de desenvolvimento de cada *visitor*. . . . . 17

4.1 Projetos. . . . . 21
4.2 Resumo dos tipos agrupados por idade e do tipo dos projetos. . . . . 22
4.3 Tipo declarado X Número de instância . . . . . 22
4.4 Ocorrências de Expressões Lambda. . . . . 24
4.5 Classes concorrentes que *extends Thread* ou implementam *Runnable*. . . . . 26
4.6 Oportunidades de `multi-catch` por tipo do sistema. . . . . 28
4.7 Adoção Try-Resource por tipo do sistema. . . . . 30
4.8 Adoção `Switch String` por tipo do sistema. . . . . 30
4.9 Oportunidade de aplicar `switch` por tipo de sistema. . . . . 31

# 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

**CC** Complexidade Ciclomática

# 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 [11]. Conforme explicado por Jeffrey L. Overbey e Ralph E. Johnson. [16], 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 realização 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 [7, 9, 11, 12, 16, 17, 21]. 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 [1]). Posteriormente, foi iniciada uma fase de implementação dos analisadores estáticos usando padrões de projetos típicos para essa finalidade como *visitor* e *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 [6, 8, 10, 16, 19, 21, 22, 22, 23, 25].

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 [13] usando o *spring framework* [3]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

# Capítulo 2

# Fundamentação

Conforme mencionado no capítulo anterior, o principal objetivo deste trabalho de conclusão de curso é identificar oportunidades de evolução de código em projetos que utilizam recursos anteriores a Java 7 e Java 8, algo necessário para o contexto de reestruturação de código que visa adequar um código existente para usar construções mais atuais de uma determinada de linguagem de programação (no caso, a linguagem Java). Importante destacar que as versões da linguagem Java mencionadas anteriormente introduziram novos recursos, tais como: `multi-catch`, `try-with-resourse`, `switch-string` e `lambda expressions`; e que esse tipo de evolução constitui uma nova perspectiva de *refactoring*, que se caracteriza por uma transformação de código que preserva comportamento e que passa a usar novas construções da linguagem de programação (conforme defendido por Overbey and Johnson [16]).

Para atingir o objetivo do trabalho de conclusão de curso, foi necessário estudar temas relacionados à evolução da linguagem Java, engenharia de linguagens de software (ou no Inglês Software Language Engineering) e refatoramento de código (*code refactoring*). Para dar mais clareza ao leitor sobre essas temáticas, esse capítulo apresenta uma visão geral sobre esses temas. Note que não foi objetivo deste trabalho implementar um mecanismo de transformação de código, mas sim construir um suporte ferramental efetivo para compreender como os desenvolvedores usam as construções existentes na linguagem Java e **identificar oportunidades de melhoria de código**, algo essencial para permitir a atualização de um código existente que usa construções ultrapassadas de uma linguagem de programação.

Muito das ocorrência de código ultrapassao acontece devido a compatibilidade que a linguagem Java mantém em todas entre suas versões, o que pode vir a ser um problema na manutenção de um software dado diversas variáveis que vai desde o conhecimento do desenvolvedor até o sistema em que vai executar o software.

## 2.1 Evolução Linguagem Java

Java é uma linguagem orientada a objetos de proposito geral projetada pela SUN Microsystems para conter poucas dependências na implementação de um software, atualmente e pertence a Oracle (2009). Existem algumas semenlhanças em sua sintaxe com `C/C++` pois estas linguagens influênciaram no desenvolvimento da linguagem Java. Tendo em vista alguns pontos no importantes durante a evolução da linguagem Java a Figura: 2.1

exibe o características da linguagem que contém maior relevância para o desenvolvimento deste trabalho de conclusão. Ressaltando ainda que durante a realização deste trabalho a última versão da linguagem é Java 8.

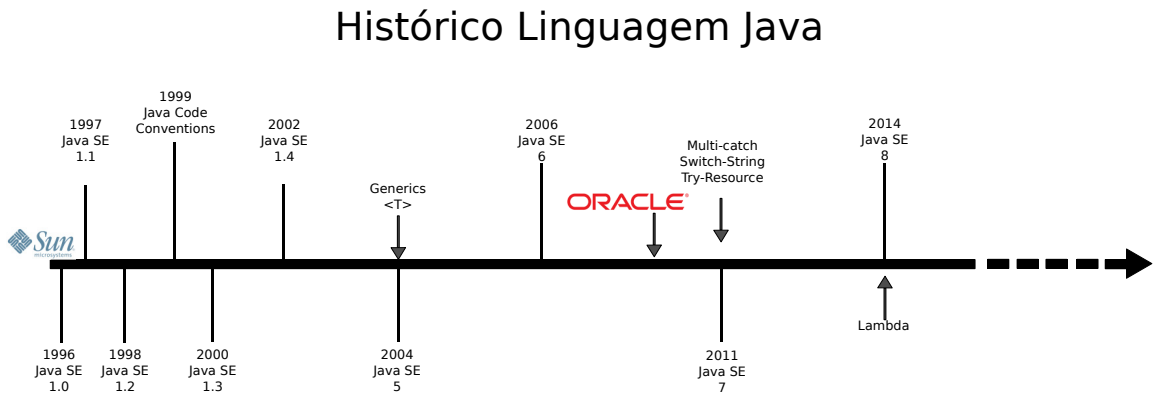


Figura 2.1: Resumo cronológico da linguagem Java.

Relativo a independência de plataforma a linguegem Java se destaca por permitir que programas desenvolvidos possam ser executados em qualque plataforma tendo em vinda que até hoje a portabilidade entre plataformas para alguns software é de um problema. Java permite tal flexibilidade entre plataformas devido uma máquina virtual **JVM!** que executa o código compilado conhecido com *bytecode*, e este pode ser executado em qualquer plataforma que contenha a máquina virtual. A Figura: 2.2 exemplifica a principal característica da linguagem Java, portabilidade.

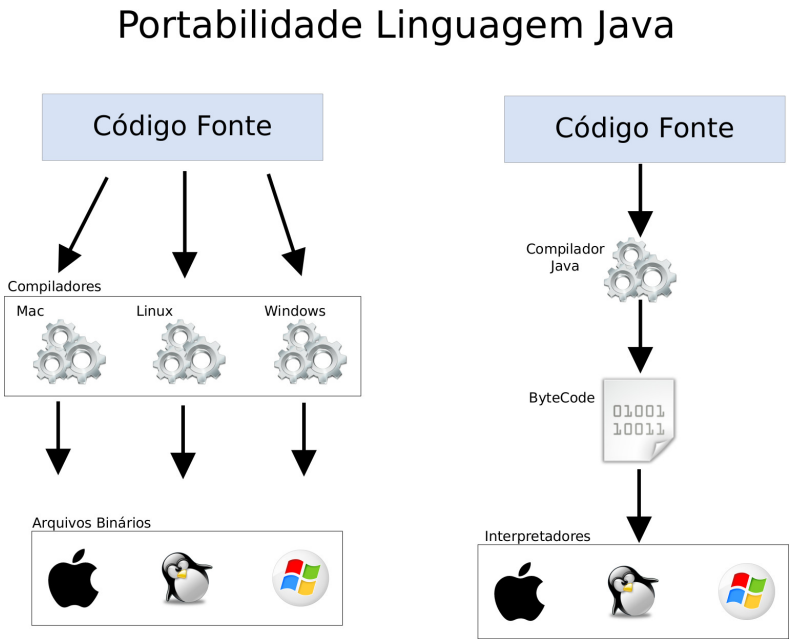


Figura 2.2: Portabilidade da liguagem Java.

No início dos anos noventa, um grupo de engenheiros da Sun Microsystems, chamados de *Green Team*, acreditava que a próxima grande área da computação seria a união de equipamentos eletroeletrônicos com os computadores. O *Green Team*, liderado por

James Gosling, especificou a linguagem de programação Java, inicialmente proposta para dispositivos de entretenimento como aparelhos de TV a cabo. Por outro lado, apenas em 1995, com a massificação da Internet, a linguagem Java teve sua primeira grande aplicação: a construção de componentes de software para o navegador Netscape.

Na sua primeira versão publicamente disponível (JDK 1.0.2), existiam apenas oito bibliotecas presentes na especificaçã Java, tais como `java.lang`, `java.io`, `java.util`, `java.net`, `java.awt` e `java.applet`; onde as três últimas favoreciam a construção de soluções envolvendo mobilidade de código: um componente (um *applet* Java) poderia ser transferido de um servidor para um cliente e, dessa forma, ser executado em um navegador Web compatível. As características de independência de plataforma e a aproximação com a Web fez com que a linguagem Java se tornasse bastante popular, passando a ser usada em outros domínios (como o desenvolvimento de software para cartões inteligentes, para jogos eletrônicos e para ambientes corporativos) e a ter uma evolução natural com a melhoria de desempenho da **JVM!** e a incorporação de um conjunto significativo de bibliotecas.

Apesar de toda essa evolução, que trouxe uma rápida aceitação da linguagem, mudanças significativas na especificação da semântica da linguagem só se tornaram publicamente disponíveis em 2004, com o lançamento da versão intitulada Java 5.0 (*Java Language Specification 1.5*). As principais contribuições para a semântica da linguagem afetavam diretamente a produtividade dos desenvolvedores e incluiam implementações mais eficientes de bibliotecas existentes (como as bibliotecas de IO e as bibliotecas para programação concorrente). Relacionadas à perspectiva semântica, as principais contribuições da especificação Java 5.0 introduziram o suporte a polimorfismo parametrizado (Java Generics) e enumerações; o uso de construções `foreach` para iterar sobre coleções; a possibilidade de definição de múltiplos argumentos com a construção `varargs` (suportados em linguagens como C); e o uso do mecanismo intitulado *autoboxing* para converter tipos primitivos nas classes Java correspondentes. As versões da linguagem Java 7 e Java 8 também trouxeram, em maior ou menor grau de significância, extensões sintáticas e semânticas bastante aguardadas pela comunidade de desenvolvedores, tais como:

**Java 7** introduziu em 2011 facilidades como (a) suporte ao tipo `String` em sentenças condicionais `switch`, (b) inferência de tipos na instanciação de classes genéricas e (c) captura de múltiplos tipos de exceção.

**Java 8** introduziu em 2014 o suporte a expressões lambda e a implementação de métodos *default* em interfaces Java. O suporte a expressões lambda pode ser compreendido como uma evolução da linguagem tão significativo quanto a introdução de Java Generics, na versão Java 5. Isso porque uma série de novos idiomas (baseadas em *streaming* para programação concorrente) estão sendo propostos para a linguagem com base em tal construção.

## 2.2 Engenharia de Linguagens de Software

*... ou engenharia de software para linguagens de programação (no Inglês, Software Language Engineering).*

A manipulação de artefatos escritos em uma linguagem de programação (ou em linguagens de software) é uma tarefa desafiadora, mas que permite o desenvolvimento de

software aplicável a diferentes cenários, mas que permitem, por exemplo, manipular arquivos `XML`, transformar informações e scripts presentes em bancos de dados legados, efetuar a tradução de programas escritos em uma versão desatualizada de uma linguagem.

Por envolver diferentes estágios, o desenho desse tipo de solução requer, geralmente, um estilo arquitetural baseado em um *pipeline*, onde cada estágio necessário à manipulação de uma linguagem é implementado como um componente de software. Quando combinados, tais componentes proporcionam um programa capaz de realizar a tarefa desejada para realizar o processamento de uma ou mais linguagens. A Figura: 2.3 exibe uma organização típica de componentes para o processamento de artefatos escritos em uma linguagem de programação, onde a cada estágio do *pipeline*, um componente utiliza os resultados do estágio anterior para gerar uma saída para o componente que realiza o processamento no estágio posterior.

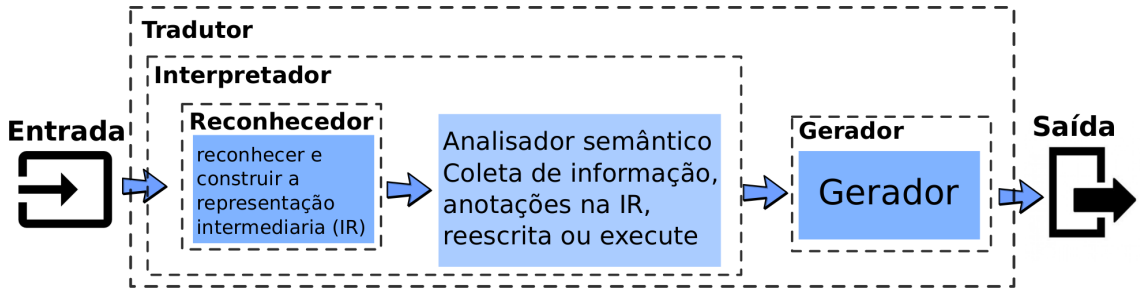


Figura 2.3: Fases de aplicações com linguagens.

Existe um esforço considerável para tratar a engenharia de linguagens de programação como sendo o desenvolvimento de um software comum. Algumas aplicações típicas deste domínio são `reconhecedores`, `interpretadores`, `tradutores` e `geradores`, conforme menciona Terrance Parr [18].

Além dessas aplicações típicas, ferramentas para a identificação estática de *bugs*, por exemplo, também são comumente implementadas usando uma organização como a representada na Figura 2.3. A ferramenta *FindBugs* [2] serve como um exemplo de solução para identificação de possíveis erros em programas escritos na linguagem Java, a partir do *bytecode* resultante do processo de compilação. Note na Figura: 2.4 a semelhança arquitetural com as abordagens típicas para o processamento de artefatos de linguagens de programação.

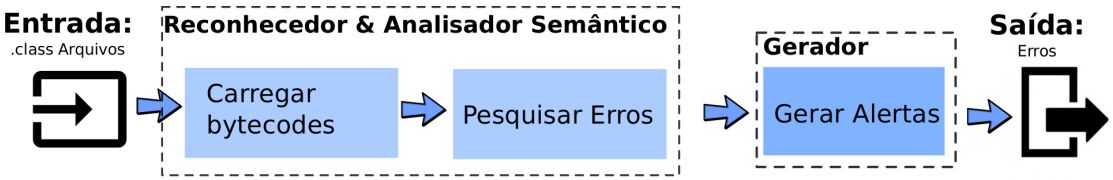


Figura 2.4: Fase do pipiline do FindBugs.

Especificamente no caso deste trabalho, percebeu-se a necessidade de construção de um software que realiza a análise estática de código para identificar tanto o uso quanto as oportunidades do uso de construções sintáticas / semânticas da linguagem Java. Em termos arquiteturais, a Figura: 2.5 ilustra, em um alto nível de abstração, os principais

componentes que formam o *pipeline* do analisador estático implementado nesse trabalho e cujos detalhes de implementação são apresentados no próximo capítulo.

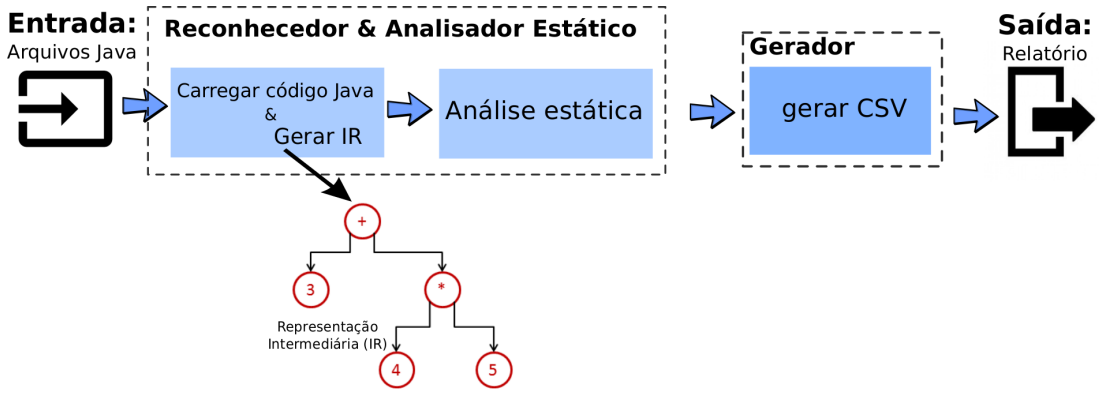


Figura 2.5: Ferramentas necessárias para construção do analisador estático.

O analisador desenvolvido neste trabalho é considerado um *grammarware* pois é um software que depende da gramática da linguagem Java para seu funcionamento pois tais softwares devem possum sólido conhecimento da gramática que manipulam. Como exemplos de arquiteturas que possuem tal conhecimento tem-se os programas de conversão de dados, processadores de `XML` e os softwares que efetuam *parses*.

Paul Klint at.al. [15] vai mais além para caracterizar um *grammarware* como sendo um software comum, afirmando que este deve adotar técnicas contemporâneas a engenharia de software tendo em vista que tais técnicas são utilizadas diariamente no desenvolvimento não sendo nenhuma novidade. A contemporaneidade deste analisador da-se por utilizar técnicas para injeção de dependência, controle de depêndencia e repositório para manter gerenciamento do software.

Gramáticas e software dependentes de gramática não são invenções recentes e este conceito faz-se presente em áreas da computação como engenharia reversa, desenvolvimento orientado a aspectos, tranformações de programas e metamodelagem.

Alguns cenários favorecem o desenvolvimento de softwares *grammarware*, onde dentre os diversos cenários pode-se destacar uma aplicação que necessite importar perfis de usuários para promover a transição da versão antiga para uma versão atual. Esta transição deve ser robusta e provavelmente necessitará de adptação o que em muitos casos necessita de um *parser* para partes que necessitam ser adaptadas.

Um outro cenário real é desenvolvimento de aplicações de banco de dados onde é necessário adotar uma nova linguagem de definição para um ambiente específico. De forma que automatizar esta solução requer o uso de um *parser* o qual será responsável por identificar os inputs de entrada para efetuar o mapeamento correto para a saída desejada no formato mais atual.

Outro tipo de *software* que manipula a gramática de uma linguagem, são os que realizam *refactoring*. Dentre diversas caracterísitas explicadas por Martin Fowler at.al. [5], fica claro que este trabalho contribui com o intuíto de identificar construções obsoletas e oportunidades para alguma evolução e não efetuando *refactoring* de modo automático mas sim sugerindo ao desenvolvedor escolher pela evolução ou não.

A evolução do código para um mais atual é mais que procurar por *bad smell* onde são ocorrências pontuais já estabelecidas. A oportunidade de evoluir para uma versão mais atual é muito mais complexa que meramente melhorar o desgin do *software* e por isso essa sugestão deve ser apreciada pelo desenvolvedor com cuidado pois pode ser um questão mais profunda.

Mesmo sem a implementação de *refactoring* uma contribuição que este trabalho faz é a identifição a capacidade de encontrar código duplicado como por exemplo ao identificar `catch` repetidos, ou até mesmo melhorar o desempenho com no caso da da utilização do `switch-string` ao invés do `if-string` pois oracle afirma que o desempenho é melhor de a implementação otimizada do `switch-string` na documentação [4].

## 2.3 Representação Intermediária

Conforme introduzido anteriormente na Figura: 2.5, é necessário uma representação intermediária que represente a linguagem Java e com isso é possível extrair informações da linguagem contidas nesta representação. E para criar tal representação é necessário reconhecer sentenças e símbolos da linguagem. Terrance Parr em [18], explica que o ato de reconhecer sentenças em computação é denominado *parser*.

Analogamente a estrutura do *parser* pode ser comparada a sentenças da língua portuguesa, identificando verbos, nomes, substantivos, etc... Desse modo em uma linguagem de programação é necessário reconhecer o vocabulário de símbolos (`Tokens`) da linguagem. E isto possibilita a identificação de regras e subsequências de `Tokens` como expressões da linguagem.

A representação do *parser* é dada por uma árvore sintática composta de `Tokens` nas folhas e utilizando esta estrutura é possível extrair toda informação necessária sobre sintaxe e estruturas que compõem as sentenças da linguagem. A Figura: 2.6 exibe a representação do *parser* como uma árvore.

O principal motivo da representação intermediária ser uma árvore é por possuir uma estrutura regular e os nós preservam a hierarquia, devido essa regularidade é possível automatizar esta tarefa utilizando ferramentas como no caso deste trabalho a biblioteca Eclipse JDT a qual possui classes especializadas em gerar e analisar as representações intermediárias dos código fonte da linguagem Java.

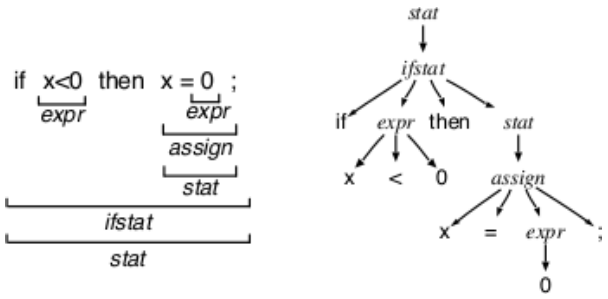


Figura 2.6: Representação da uma frase.

Existem muitos padrões de *parses* pois algumas linguagens são mais complexas que outras. Tendo em vista o desafio que é trabalhar e analisar uma linguagem de programação, alguns padrões adotados para este contexto facilitam esta tarefa e com isso este

trabalho abordará os quatro conceitos mais importantes segundo Terence Parr em [18] para prover o *parser* de acordo com a necessidade.

- **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 padrão 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 padrão 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.

## 2.4 Refactoring

Por definição o *refactoring* é mudança interna do software sem alterar seu comportamento tornando assim seu entendimento mais claro. Visando evitar que sejam perdidas diversa horas para identificar possíveis oportunidades de classes onde possam ser evoluídas, este trabalho de conclusão identifica trechos de código dentro das classes que pode ser evoluídos.

Muitas modificações podem ser feitas em um software mas segundo M.Fowler at al [5] somente é considerado *refactoring* mudanças que facilitam o entendimento do software.

Contrastando esta visão existem mudanças com objetivo de melhorar o desempenho do software onde somente são alterada as estruturas internas permanecendo inalterado o comportamento do software. Entretanto a melhoria na performance do software geralmente eleva o grau de dificuldade para sua compreensão, o que faz com que algumas dessas evoluções visando desempenho não sejam caracterizadas como *refactoring* dado a definição.

Dentre refatorar para facilitar o entendimento, para tornar o programa mais rápido, para encontrar bugs e para melhorar/atualizar o design, motivos apresentados por M.Fowler at. al. [5], este trabalho concentrou-se no último motivo para identificar possíveis casos onde o design do software pode ser evoluído por substituir funcionalidades de versões anteriores da linguagem Java.

Um software que não é refatorado tem o seu desgin deteriorado, o que leva a dificultar o entendimento do código. Um design ultrapassado tem mais código que o necessário para realizar a mesma tarefa. O que leva a um aspecto crucial para a melhoria, que é código duplicado. Vale ressaltar que reduzir a quantidade de código não implíca necessariamente na melhora do desempenho do software mas sim em ter um design mais atual.

O Listing: 2.1, exemplifica de forma empírica um *filter* em um *collection* onde ocorre uma redução significativa de código além de um design mais atual por utilizar expressões lambda que foram adicionadas em Java 8. Vale destacar que o Listing: 2.1 continua com mesmo comportamento após o *refactoring*, desta forma nenhum usuário ou desenvolvedor pode alegar que o software foi modificado.

Listing 2.1: The Exmplo de Filter Pattern

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

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

Conforme explica M.Fowler at. al. [5] algumas vezes não se deve ser refatorar o código. Um desses casos é quando existir a necessidade de reescrever todo o código, um outro caso é a necessidade de manter um código de fácil entendimento para os programadores iniciantes. O que é uma decisão difícil, no caso do Listing: 2.1 é possível fazer uma evolução com a parte funcional de Java 8 entretanto alguns desenvolvedores podem não possuir conhecimento adequado da parte funcional e com isso é recomendado que não seja evoluído o código.

Tendo em vista que aplicar um *refactoring* demanda tempo isto torna uma tarefa custosa para empresas, este fator é determinante para que programadores não refatorem seu códigos em muitos casos. Com esse cenário é imprecindível o uso de ferramentas que refatorem ou auxiliem nesta tarefa. Auxiliando nesta tarefa este trabalho identifica possibilidades de refatoração, ao utilizarem estas ferramentas torna mais acessível ao programador e a emrpesa refatorar pois o trabalho é direcionado fazendo com que tempo seja poupado.

## 2.5 Análise estática

Em computação análise estática é a referência a qualquer processamento realizado em código fonte sem a necessidade de executá-lo, com isto a análise estática torna-se uma poderosa técnica por permitir rápidas considerações por possibilitar uma larga exploração em um projeto podendo evitar erros triviais e simular alguns cenãrios para tal análise sem a necessidade do projeto ser executado.

Ferramentas que auxiliem a análise estática tem grande chance de ser um poderoso auxílio no desenvolvimento do software tendo em vista que pode reduzir a quantidade de erros e diminuir a quantidade de *refactoring* o qual tem um custo elevado para os projetos de software.

É nesse contexto que este trabalho faz sua contribuição por utilizar a análise estática para verificar não possibilidade de falhas ou *bad smell*, mas sim de identificar chances reais de evoluir para últimas *features* da linguagem Java sem interferir no comportamento interno do programa conforme preconiza M.Fowler at. al. [5].

A linguagem Java proporciana duas maneiras de realizar análise estática, a primeira é através código fonte, *.java* e a segunda através do *bytecode*, *.class*. Este trabalho foca em realizar análise no código fonte, entre tanto nada impede que o trabalho seja realizado da segunda maneira. Existem programas renomados que realizam tal análise utilizando os *bytecodes* e um destes programas é o FindBugs [2].

Para obter sucesso através nas análises realizadas, é necessário determinar padrões para encontrar características que dejesam ser evoluídas para a última versão da linguagem Java. Estes padrões são estabelecidos em uma estrutura que seja capaz de pesquisar nos nós da árvore da representação intermediária para extrair as informações pertinentes.

A técnica utilizada para pesquisar nos nós das árvores foi utilizar o padrão de projeto *Visitor* proposto por Gamma et. al. [13], pois este possibilitar que seja realizada uma operação sobre todos os elementos de uma estrutura, neste caso a operação é a pesquisa e a estrutura a representação intermediária.

A verificação de software possibilita a detecção de falhas de maneira precoce durante as fases de desenvolvimento entretanto este não é o objetivo deste trabalho pois existem ferramentas consolidadas que realizam tal análise de maneira excepcional. Aqui o objetivo principal é alertar ao desenvolvedor a possibilidade de usar o que há de mais recente na linguagem Java.

# Capítulo 3

# Suporte Ferramental para Minerar Padrões de Uso de Construções da Linguagem Java

Com o intuito de obter uma compreensão sobre o uso das construções da linguagem Java, tornou-se necessária a implementação de uma ferramenta de análise estática de propósito bem específico, por outro lado com capacidade de ser extensível para extrair diferentes tipos de informações. A Figura 3.1 apresenta uma visão geral 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, realiza 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 ferramentas e bibliotecas de programação que auxiliam a construção de analisadores estáticos, 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* [1] (Eclipse JDT). O Eclipse JDT 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 às construções sintáticas 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 cinco classes que trabalham exclusivamente com métodos referenciados e seis classes exclusivas que tratam os tipos declarados, como classes, interfaces e enumerações em Java. O Eclipse JDT [1] disponibiliza ainda um *parser* para a linguagem Java

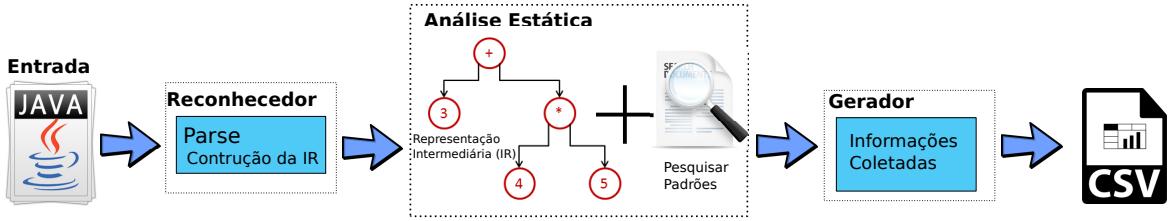


Figura 3.1: Visão geral da arquitetura do analisador estático

que atende a especificação Java 8 da linguagem e que produz a representação intermediária baseada no conjunto de classes Java mencionado anteriormente e que corresponde a uma AST do código fonte. A plataforma também oferece uma hierarquia de classes para travessia na AST, de acordo com o padrão de projeto *visitor* [13], e que facilita a análise estática de código fonte.

O padrão de projeto *Visitor* [13] 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, torna-se relativamente simples adicionar novas funcionalidades em um *visitor* existente ou criar um novo *visitor*. 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 travessia nos nós das ASTs. Essa flexibilidade foi alcançada com a utilização de introspecção de código que em Java é conhecido como *reflection*. O restante desse capítulo apresenta mais detalhes sobre a arquitetura e implementação do analisador estático, descrevendo as principais decisões de projeto relacionadas às cinco fases do analisador estático.

## 3.1 Definição dos Projetos a Serem Analisados

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 pasta no sistema de arquivos contendo o código fonte do projeto e a quantidade de linhas de código previamente computadas (conforme ilustrado na Figura: 3.1). As informações contidas no arquivo CSV são processadas por um conjunto de classes utilitárias que percorrem 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 são recuperados os arquivos contendo código fonte Java, que são convertidos para uma representação intermediária (por meio de um parser existente); processados e analisados com uma infraestrutura de *visitors*, e os resultados das análises são ,por fim, exportados.

Conforme mencionado, essa fase do analisador estático realiza o processamento de cada arquivo Java dos projetos analisados. Sob a perspectiva de usabilidade, o usuário deve executar o programa principal informando, na linha de comando, o caminho para o

arquivo CSV contendo as definições dos projetos. Isso produz uma lista contendo todos os projetos a serem processados (ver Linha 25 na Figura 3.2). Após a lista de projetos ter sido carregada, cada projeto é analisado com o uso da classe `ProjectAnalyser`, que possui um método (`analyse`) com a lógica necessária para processar a base de código fonte de cada projeto. A Figure 3.3 apresenta a implementação do método `analyse`, que recebe como parâmetro um projeto. Note na linha seis da Figura 3.3 que o resultado do *parse* em um arquivo fonte produz uma instaância da classe `CompilationUnit` que pertence a plataforma Eclipse JDT e que representa a AST de um determinado arquivo Java. Essa classe possui um método `accept`, conforme o padrão de projeto *visitor*, que é usado para pela ferramenta de análise estática para coletar as informações do uso de construções Java por meio de uma análise da representação intermediária. Isso é feito considerando cada um dos *visitors* aplicados em uma análise específica.

```
public class Main {

  public static void main(String[] args) {

    String pathCsv = ``'';

    if(args.length == 1) {
      System.out.println(``Args: ''+ args[0].toString());
      pathCsv = args[0];
    }else {
      System.out.println(``Error: inform a valid csv file!!!\nEXIT'');
      System.exit(0);
    }

    ReadCsv rcsv = new ReadCsv(pathCsv);

    List<String> errors = rcsv.getError();

    errors.forEach(e -> System.out.println(``Error in '' + e));

    ApplicationContext ctx = CDI.Instance().getContextCdi();

    ProjectAnalyser pa = ctx.getBean(``pa'', ProjectAnalyser.class);

    List<Project> projects = rcsv.readInput();

    try {
      projects.stream().forEach(project -> pa.analyse(project));
    }catch(Exception t) {
      t.printStackTrace();
    }
  }
}
```

Figura 3.2: Classe que representa o programa principal do analisador estático

```
public void analyse(Project p)  {
  CompilationUnit compilationUnit = null;
  List<String> fs = IO.list(p.getPath(), new String[] { ``java'' });

  for (String file : fs) {
    compilationUnit = Parser.Instance().parse(new File(file));

    for(IVisitor visitor : listVisitors){
      visitor.getCollectedData().setProject(p);
      visitor.setFile(file);
      visitor.setUnit(compilationUnit);

      compilationUnit.accept((ASTVisitor) visitor);
    }
  }
  exportData();
}
```

Figura 3.3: Implementação do método `analyse`, na classe `ProjectAnalyser`.

## 3.2 Análise da Representação Intermediária

Conforme mencionado na seção anterior, o resultado do *parser* em um arquivo fonte produz uma instância da classe `CompilationUnit`, que corresponde a uma AST com todas as definições de tipo e implementação de comportamento presentes em um módulo Java. A plataforma Eclipse JDT oferece uma infraestrutura de classes para realizar a traversia em uma AST, usando o padrão de projeto *visitor*. Dessa forma, foi feita uma implementação de biblioteca de *visitors*, para extrair as informações presentes na representação intermediária.

No contexto deste projeto, e objetivando um maior grau de reuso, toda classe *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{ ...}`) que representar as informações de interesse associadas ao uso de uma construção 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 Figura 3.4 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;

  //constructor + getters and setters.
}

public class EnumDeclarationVisitor extends Visitor<EnumDeclaration> {

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

    EnumDeclaration dec = new EnumDeclaration(...);

    collectedData.addValue(dec);

    return true;
  }

}
```

Figura 3.4: Classes usadas para capturar declarações de enumerações.

### 3.2.1 Descrição dos Visitors

Os *visitors* implementados neste projeto são brevemente descritos a seguir, enquanto que a Tabela 3.1 apresenta duas métricas relacionadas à complexidade de implementação, em termos de complexidade ciclomática e total de linhas de código fonte. A complexidade ciclomática é dada pela quantidade de caminhos independentes em um trecho de código, enquanto que a quantidade de linhas de código foi computada ignorando comentários e linhas em branco. Vale ressaltar que a complexidade ciclomática dos *visitors* varia entre um e oito (com a média igual a 2.5). A quantidade média de linhas de código necessária para escrever um *visitor* é 47.

- `AIC`: Coleta informações relacionadas a declaração de *Anonymous Inner Classes*. Tal informação é útil para estimar oportunidades de uso de expressões lambda.
- `ExistPattern`: Coleta informações de laços `foreach` que iteram sobre uma coleção com o intuito de verificar se um determinado objeto está presente na coleção. Tal informação é útil para estimar oportunidades de uso de expressões lambda.
- `FieldAndVariableDeclaration`: Coleta informações relacionadas a declarações de atributos e variáveis, com o intuito de extrair informações sobre a adoção de Java Generics.
- `FilterPattern`: Coleta informações de laços `foreach` que iteram sobre uma coleção com o intuito de filtrar elementos presentes na coleção. Tal informação é útil para estimar oportunidades de uso de expressões lambda.

- `ImportDeclaration`: Coleta informações relacionadas à importação de bibliotecas, sendo útil para estimar a adoção de bibliotecas voltadas para programação concorrente ou integração com linguagens de scripting, por exemplo.
- `LambdaExpression`: Coleta informações relacionadas à adoção de expressões lambda.
- `Lock`: Verifica se os métodos utilizam algum dos mecanismos de *lock* suportados diretamente pela linguagem Java, como `Lock`, `ReentrantLock`, `ReadLock` ou `WriteLock`.
- `MapPattern`: Coleta informações de laços `foreach` que iteram sobre uma coleção com o intuito de aplicar alguma operação sobre os elementos presentes na coleção. Tal informação é útil para estimar oportunidades de uso de expressões lambda.
- `MethodCall`: Coleta informações relacionadas às chamadas de método, sendo útil para estimar o uso da API de introspecção de código, por exemplo.
- `MethodDeclaration`: Coleta informações relacionadas às declarações de métodos, sendo útil para identificar padrões de uso de Java Generics, por exemplo.
- `ScriptEngine`: Coleta informações relacionadas ao uso da API Java para integração com linguagens de scripting.
- `SwitchStatement`: Coleta informações relacionadas ao uso de sentenças `switch-case`, com o intuito principal de identificar o uso de `strings` nesse tipo de sentença.
- `SwitchString`: Coleta informações associadas às oportunidades de reestruturação de código para usar sentenças `switch-case` com `strings`.
- `TryStatement`: Coleta informações relacionadas ao uso de blocos `try-catch`, em particular para estimar o uso da construção `try-with-resources`.
- `TypeDeclaration`: Coleta informações sobre os tipos declarados (classes, interfaces, enumerações), com o intuito, por exemplo, de estimar a adoção de Java Generics.

Tabela 3.1: Estimativa da complexidade de desenvolvimento de cada *visitor*.

| Visitor | CC | **LoC!** |
|---|---|---|
| AIC | 6 | 64 |
| ExistPattern | 28 | 116 |
| FieldAndVariableDeclaration | 8 | 81 |
| FilterPattern | 43 | 168 |
| ImportDeclaration | 1 | 63 |
| LambdaExpression | 1 | 33 |
| Lock | 12 | 75 |
| MapPattern | 27 | 103 |
| MethodCall | 2 | 22 |
| MethodDeclaration | 5 | 45 |
| ScriptingEngine | 5 | 39 |
| SwitchStatement | 3 | 35 |
| SwitchStringOpportunities | 3 | 50 |
| TryStatement | 11 | 80 |
| TypeDeclaration | 2 | 41 |

### 3.2.2 Extensibilidade para Inclusão de Novos Visitors

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 [3], 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].

## 3.3 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 [20]. 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 utiliza 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 Figura 3.5 apresenta o uso desse mecanismo para generalizar a exportação dos dados.

```
public class CSVData<T> implements Data<T> {

  @Override
  public void export() {
    StringBuffer str = new StringBuffer("");

    if(data == null) { return; }

    for(T value : data) {
      //reflection code...
      for(Field f: value.getClass().getDeclaredFields()){

        String fieldName = f.getName();
        String prefix = "get";

        if(f.getType().isPrimitive() &&
           f.getType().equals(Boolean.TYPE)) {
          prefix = "is";
        }

        String methodName = prefix +
        Character.toUpperCase(fieldName.charAt(0)) +
        fieldName.substring(1);

        Method m = value.getClass().getDeclaredMethod(methodName);
        str.append(m.invoke(value));
        str.append(";");

        writer.append(str.toString());
        writer.append("\n");

        writer.flush();
      }
    }
  }
```

Figura 3.5: Exportação de dados usando o mecanismo de introspecção de código.

# 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 [17]. Adicionalmente a contribuição deste trabalho é abordar outras quatro características da linguagem Java além de *Java Generics* que são *Java Lambda Expression, Multi-catch, Try-Resource e Switch-String* 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 [17] 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 46 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, [12, 17, 24], e também fora separados pela natureza da projeto, aplicações, bibliotecas e servidores/banco de dados conforme tabela: 4.1 o que totalizou mais de 9M de LOC detalhados na Tabela: 4.1.

Para a investigação foram utilizados os *visitors* criados exibidos 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 MapPatternVisitor, TryStatementVisitor, SwitchStringOpportunitiesVisitor e SwitchStatementVisitor.*

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 |

## 4.1 Adoção de Java Generics

Relacionado com a adoção de *Java Generics*, a maioria dos projetos apresentam um porção significativa 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.

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 | 0.97 |
| Aplication | No | 16148 | 744 | 4.61 |
| Library | Yes | 21537 | 1198 | 5.56 |
| Library | No | 22639 | 947 | 4.18 |
| Server/Database | Yes | 18038 | 552 | 3.10 |
| Server/Database | No | 11790 | 760 | 6.45 |

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. [17] 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 de uso entre os tipos genéricos. No entanto com 730.720 métodos, apenas entre 6157, 0.84%, são *métodos parametrizados.*

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 |

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: 4.1, 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*.

Listing 4.1: Declaração não trivial de Generics.

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

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. [17] 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

Tabela 4.4: Ocorrências de Expressões Lambda.

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

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 ideia 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 usado principalmente em pequenos projetos e experimentais. Isso pode contrastar com os resultados de [12], 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 [14], que investigou as mesmas questões porém eu um número de inferior de projetos. Existem dois cenários típicos para *refactoring* utilizando Expressões Lambda: *Annonymous In-*

[1] Última pesquisa realizada em Novembro 2015

*ner 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 [14]. Neste trabalho foi utilizado uma abordagem mais conservadora para considerar se é possível refatorar*Enhanced for Loops* 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: 4.2, 4.3 e 4.4.

Listing 4.2: EXIST PATTERN.

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

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

Listing 4.3: FILTER 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 4.4: MAP PATTERN.

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

//pode ser refatorado para:

collection.stream().forEach(e->{
        e.foo();
        e = blah();
        otherCollection.add(e);
});
```

Mesmo com uma 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.

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 |

## 4.3 Análises adicionais

Os principais resultados dessa monografia estavam relacionados à investigação discutida nas seções anteriores. Por outro lado, a infraestrutura construída durante a realização desse trabalho favorece a investigação de outras construções da linguagem Java. Conforme discutido por Jeffrey L. Overbey at al [16], desenvolvedores Java mantém construções ultrapassas ao longo histórico de versões de um software o que de fato é possível devido a compatibilidade mantida entre as versões da linguagem. Tais construções somente seriam evitadas caso ocorresse uma ruptura desta filosofia de compatibilidade da linguagem tal como ocorreu na linguagem Fortan conforme explicado por Jeffrey L. Overbey at al [16] quando foi introduzindo o paradigma de orientação a objeto e levando a quebra de compatibilidade com versões anteriores da linguagem Fortan.

Baseado nesta assertiva, foi feita uma investigação adicional que visa minerar o uso de construções obsoletas em código fonte existente e encontrar possíveis casos de trechos de código que poderiam ter evoluído ao longo das versões da linguagem Java. Essas situações caracterizam cenários potenciais de melhoria de código e 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 evoluções da linguagem as características investigadas, esse trabalho foca em descobrir oportunidades de adoção das características `multi-catch, try resource e switch com string` e se as mesmas etão sendo adotadas.

### 4.3.1 Oportunidades para uso da construção `multi-catch`

O mecanismo de tratamento de exceção sempre foi presente na linguagem Java entre tando em Java 7 forneceu uma evolução elegante que foi a opção do desenvolvedor utilizar `multi-catch` que possibilita a concatenação de `catchs` iguais ou similares. Com isso a adoção de recurso permite a redução da lógica duplicada em `catchs` distintos de uma construção `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 95%

dos projetos pesquisados possuem oportunidades reais para aplicação de `multi-catch` e foram encontrados 1474 blocos `try` que possuem *catchs* repetidos. Estas ocorrências estão distribuídas em 1028 arquivos e totalizando 30 936 LOC. Importante observar que o teste de similaridade entre os blocos `catch` foi realizado através de uma chamada a um método externo que verifica 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. A Tabela: 4.6 exibe a distribuição de oportunidades de `multi-catch` pela natureza do sistema.

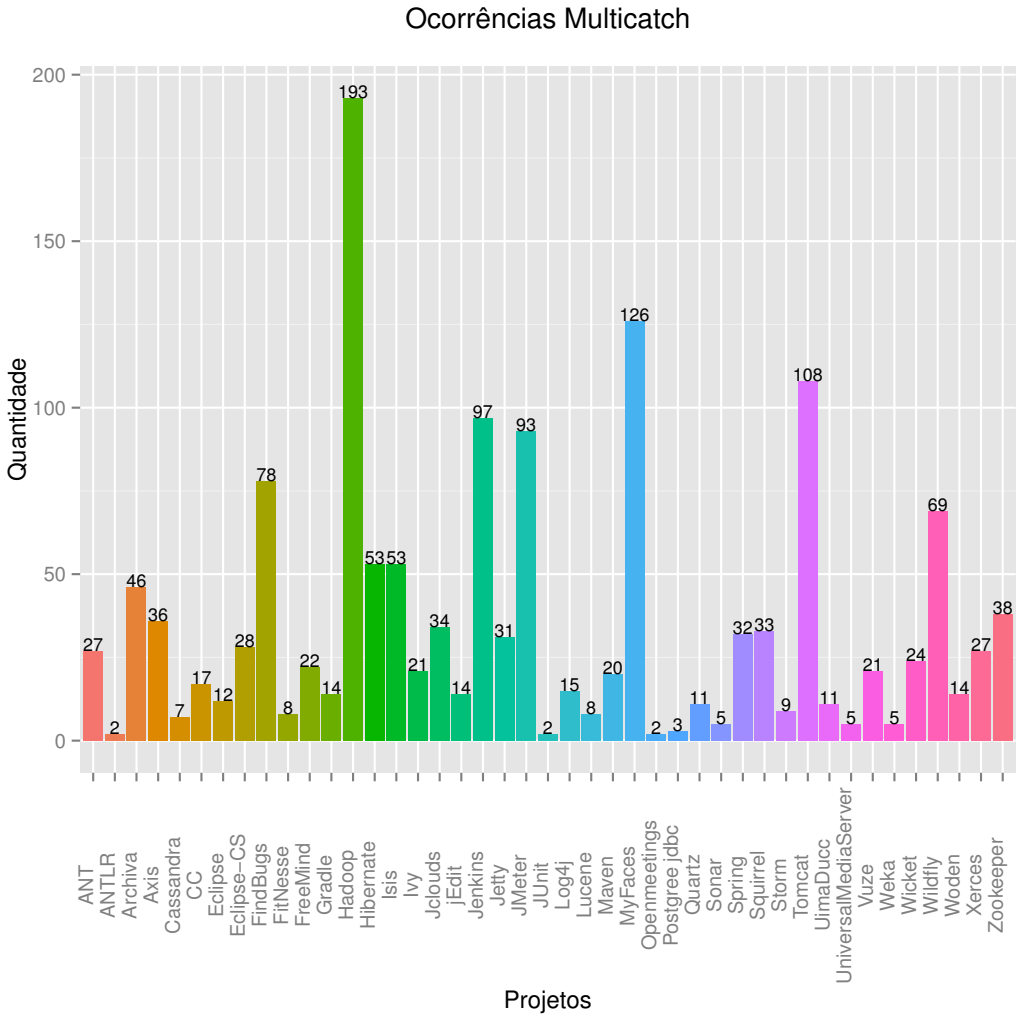


Figura 4.1: Oportunidades de `multi-catch` nos projetos.

A implantação de `multi-catch` em um projeto pode ser iniciada com substituição de blocos `catch` aninhados por este recurso tornado evidente a redução significativa das LOC duplicadas. A classe `AbstractNestablePropertyAccessor` do projeto *Spring 4.2.0.RC2* contém `catch` duplicado conforme a listagem: 4.5. Entretanto após um *refactoring* na listagem: 4.5 para implantação do recurso `multi-catch` pode-se verificar na listagem: 4.6 a redução significativa de código duplicado na ordem de 42% o que torna este recurso muito útil sem causar grandes mudanças no software.

Tabela 4.6: Oportunidades de `multi-catch` por tipo do sistema.

| Natureza | Ocorrências |
|---|---|
| Application | 551 |
| Library | 464 |
| Servers - Database | 459 |
| Total | 1474 |

Listing 4.5: Código sem adoção de multi-catch

```
try {
        ...
}catch (ConverterNotFoundException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject,
      this.nestedPath + propertyName, oldValue, newValue);
  throw new ConversionNotSupportedException(pce, td.getType(), ex);
}catch (ConversionException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject,
      this.nestedPath + propertyName, oldValue, newValue);
  throw new TypeMismatchException(pce, requiredType, ex);
}catch (IllegalStateException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject,
      this.nestedPath + propertyName, oldValue, newValue);
  throw new ConversionNotSupportedException(pce, requiredType, ex);
}catch (IllegalArgumentException ex) {
  PropertyChangeEvent pce = new PropertyChangeEvent(this.rootObject,
      this.nestedPath + propertyName, oldValue, newValue);
  throw new TypeMismatchException(pce, requiredType, ex);
}
```

Listing 4.6: Refactoring com uso de multi-catch

```
 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);
 }
```

### 4.3.2 Try Resource

Como continuação da evolução do mecanismo de exceção de Java 7 introduziu `try-resource` onde o `resouce` é um objeto que implemente `java.lang.AutoCloseable` ou `java.io.Closeable` e com isso o mecanismo garante o encerramento do recurso com encerramento do `statement` o que antes de recurso só era capaz utilizando um bloco `finally`[2].

[2] `try-resource` pode utilizar `catch` e `finally` de forma igual ao `try-catch`

Listing 4.7: Try sem adoção de resource.

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException
                                                       {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Listing 4.8: Try adotando resource.

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

Este trabalho contribui com a pesquisa para verificar como é a adoção deste recurso, após a realização das análise foram encontrados 1616 ocorrências de utilização onde a Figura: 4.2 demonstra a distribuição por projetos e somente 13 projetos o utilizam totalizando apenas 28% dos projetos verificados.

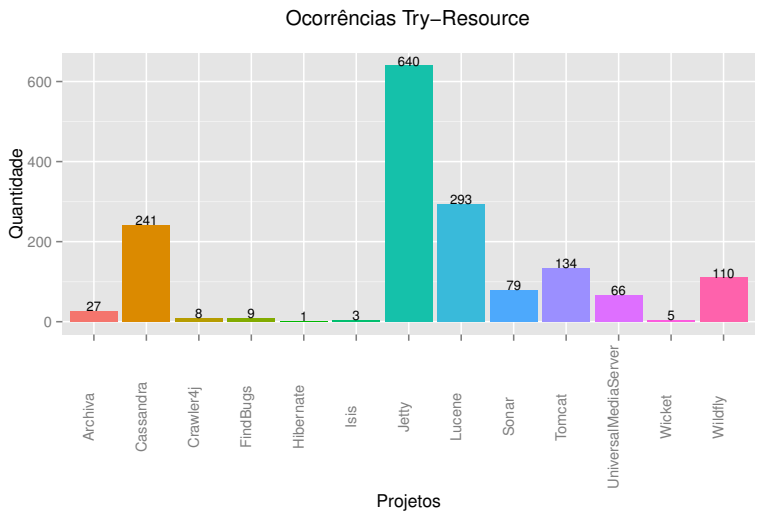


Figura 4.2: Adoção de `Try-Resource` nos projetos.

A Tabela: 4.7 exibe a distribuição deste recurso por tipo de sistema. Onde pode-se constatar que os `Servers-Database` realizaram uma adoção significativa de 91% em relação aos demais tipos.

Tabela 4.7: Adoção Try-Resource por tipo do sistema.

| Natureza | Ocorrências |
|---|---|
| Application | 115 |
| Library | 17 |
| Servers - Database | 1484 |
| Total | 1616 |

### 4.3.3 Switch String

O suporte de `String` em `Switch` foi permitido em Java 7 este recuso ajuda a preservar um padrão de codificação tendo em vista que atualmente é permitido operações de seleção em um conjunto de constantes de `String`. Além de possuir desempenho superior ao `if-then-eslse` conforme a documentação [4] informa pois o gerado para `switch` que utilizam `string` é mais eficiente que o `if-then-else`.

Como contribuição deste trabalho foi feita uma pesquisa para verificar se tal recurso é adotado tendo em vista os benefícios que pode trazer para o projeto além de ser uma característica de fácil implementação. De um total de 6827 utilizações de `switch` nos sistemas apenas 66 fazem uso de `string` o que corresponde a menos de 1% do total o que leva entender que tal característica não é aproveitada em sua plenitude. A Tabela: 4.8 exibe os projetos em houve ocorrência.

Tabela 4.8: Adoção `Switch String` por tipo do sistema.

| Sistema | Ocorrências |
|---|---|
| Cassandra | 14 |
| FindBugs | 3 |
| Jetty | 16 |
| Lucene | 2 |
| Sonar | 1 |
| Spring | 2 |
| Tomcat | 8 |
| UniversalMediaServer | 18 |
| Wicket | 1 |
| Wildfly | 1 |
| Total | 66 |

Com intuito de encontrar oportunidades de aplicar este recurso, foi verificado a existência de `if-then-else` que invoquem na sua expressão um método e verificando se este método utiliza o `equals()` comparando `String` conforme demonstrado na Listagem: 4.9 o qual é a oportunidade possível de efetuar um *refectoring* para o `switch`.

Listing 4.9: Modelo para aplicação de `Switch` com `String`.

```
if (String.equals("...")) {
...
}
```

Direcionado pelo padrão da Listagem: 4.9 foram encontrados o total de 4940 oportunidades destribuidas em 45 dos 46 projetos, o que confirma que este recurso não está sendo adotado em sua plenitude. A Figura: 4.3 exibe a quantidade de oportunidades em cada projeto verificado, e a Tabela: 4.9.

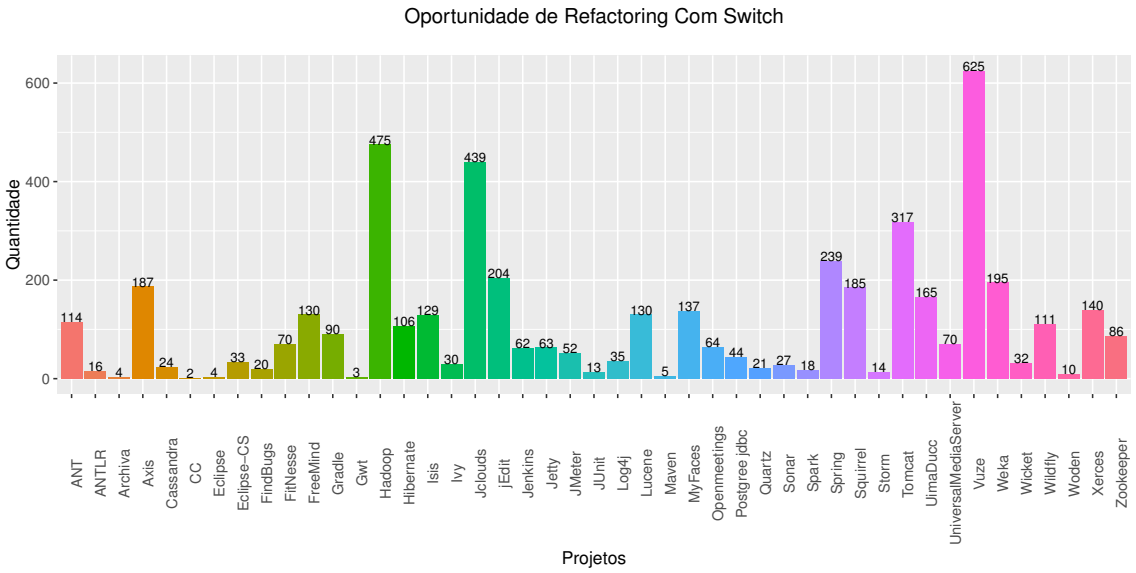


Figura 4.3: Oportunidades de *refactoring* em `if-then-else` por sistema.

Tabela 4.9: Oportunidade de aplicar `switch` por tipo de sistema.

| Natureza | Ocorrências |
|---|---|
| `Application` | 1773 |
| `Library` | 1881 |
| `Servers - Database` | 1286 |
| `Total` | 4940 |

# Capítulo 5

# Considerações Finais e Trabalhos Futuros

Este trabalho de conclusão concluiu com êxito o desenvolvimento do analisador estático proposto e este foi utilizado para comprovar a existência de uma grande quantidade de código duplicado/obsoleto em projetos open-source. Atentando em descobrir o real motivo de tais contruções de código, foi realizado contato via e-mail com algumas comunidades de desenvolvimento porém somente duas responderam, *Cassandra* e *Maven*, onde pode ser constatado que os desenvolvedores preferem não adotar novas *features* pois utilizam geralmente uma versão mais antiga da JDK relatado em: (http://goo.gl/h0uloY). A alteração/evolução é realizada em áreas que são trabalhandas na no dia a dia e não reservando um tempo específico para realizar uma evolução isto foi relatado em: (http://goo.gl/GQ4Ckn). Pode-se concluir que a existência de código duplicado é dada ao fato das equipes de desenvolvimento não adotarem uma *feature* e ainda por não reservarem tempo para revisar e evoluir um código desenvolvido anteriormente, sendo realizado apenas uma refatoração nas parte que consequentemente venham a ser retrabalhadas, isto pode ter como consequência direta techos de código congelados.

Como projeto futuro é proposto o aproveitamento da arquitetura atual para generalizar este analisador para outras linguagens como `C++` em um primeiro momento. Ainda com intuíto de tornar automatizar a refatoração pode se acoplar uma linguagem de metaprogramação, `Rascal MPL`, pois estas linguagens são adequadas para realizar o enriquecimento semântico na realização da refatoração.

# Referências


[1] Eclipse java development tools (jdt) @ONLINE. http://www.eclipse.org/jdt/. Accessed: 2015-07-06. 2, 12

[2] Findbugs in java programs @ONLINE. http://findbugs.sourceforge.net/. Accessed: 2015-07-06. 6, 11

[3] Spring framework reference documentation @ONLINE. http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/. Accessed: 2015-06-06. 2, 18

[4] Strings in switch statements @ONLINE. http://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html. Accessed: 2015-07-06. 8, 30

[5] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 7, 9, 10, 11

[6] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, September 2008. 2

[7] Rodrigo Bonifácio, Tijs van der , and Jurgen Vinju. The use of c++ exception handling constructs: A comprehensive study. 2

[8] Gilad Bracha, Martin Odersky, and David Stoutamire. Gj: Extending the javatm programming language with type parameters. 2

[9] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, October 1998. 2

[10] Alan Donovan, Adam Kiežun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. *SIGPLAN Not.*, 39(10):15–34, October 2004. 2

[11] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. A large-scale empirical study of java language feature usage. 2013. 1, 2

[12] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features.

In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM. 2, 20, 24

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

[14] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013. 24, 25

[15] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005. 7

[16] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, October 2009. 1, 2, 3, 26

[17] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 3–12, New York, NY, USA, 2011. ACM. 2, 20, 22, 23

[18] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009. 6, 8, 9

[19] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java's concurrent programming constructs. *J. Syst. Softw.*, 106(C):59–81, August 2015. 2

[20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. 18

[21] Jeffrey L. Schaefer and Ralph E. Johnson. Regrowing a language: Refactoring tools allow programming languages to evolve. *SIGPLAN Not.*, 44(10):493–502, October 2009. 2

[22] Max Schaefer and Oege de Moor. Specifying and implementing refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010. 2

[23] Daniel von Dincklage and Amer Diwan. Converting java classes to use generics. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 1–14, 2004. 2

[24] A Ward and D Deugo. Performance of lambda expressions in java 8. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 119. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015. 20

[25] Ba Wichmann, Aa. Canning, D. L. Clutterbuck, L A Winsborrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 1995. 2