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.
111 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 a nossa família

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

Este tem o objetivo analisar se o desenvolvimento do software evolui em conformidade a evolução das linguagens em específico java.

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

Abstract

This job has objective an analyze the software evolution and know how developers evolved software in accordance with evolution of languages specifically in java .

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

RSA

SQL3

JSE2

ACDP Java Platform Debugger Architecture

JAX-WS

JAXB

STAX

JAXP

JCP Java Community Process

JVM

DSA

MD5

RSA

RMI

CSV

CLOC

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 [14]. Conforme explicado por Jeffrey L. Overbey e Ralph E. Johnson. [17], 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 [9, 11, 14, 15, 17, 18, 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 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 [8, 10, 13, 17, 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 [16] usando o *spring framework* [7]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

1.4 História da linguagem

No começo da decada de 90 um pequeno grupo de engenheros da Oracle chamados de "Green Team" acreditava que a próxima onde de na area da computação seria a união de equipamentos eletroeletrônicos com os computadores. O "Green Team" liderado por James Gosling, demonstraram que a linguagem de programaçao Java, que foi desenvolvida pela equipe e originalmente era chamado de Oak, foi desenvolvida para dispositivos de

entretenimento como aparelhos de tv a cabo, porem não foi bem aceita no meio. Em 1995 com a massificação da Internet, a linguagem Java teve sua primeira grande aplicação o navegador Netscape.

Java é uma linguagem de programação de propósito geral orientada a objetos, concebida especificadademente para ter poucas dependencias de implementação que isso acarreta que uma vez que a aplicação fora desenvolvida ela poderá ser executada em qualquer ambiente computacional.

Na sua primeira versão chamada de Java 1 (JDK 1.0.2) haviam oito pacotes básicos do java como: java.lang, java.io, java.util, java.net, java.awt, java.awt.image, java.awt.peer e java.applet. Foi usado para o desenvolvimento de ferramentas populares na epoca como o Netscape 3.0 e o Internet Explorer 3.0.

Sua segunda versão foi o JDK1.1 [6] que trouxe ganhos em funcionalidades, desempenho e qualidade. Novas aplicações tambem surgiram como : JavaBeans, aprimoramento do AWT, novas funcionalidades como o JDBC, acesso remoto ao objeto RMI e suporte ao padrão Unicode 2.0.

A terceira versão Java 2 (JDK 1.2) ofereceu melhorias significativas no desempenho, um novo modelo de segurança, flexível e um conjunto completo de aplicações de programação interfaces API's. Os novos recursos da plataforma Java 2 incluiram:

- O modelo de "sandbox"foi ampliado para dar aos desenvolvedores, usuários e administradores de sistema a opção de especificar e gerenciar um conjunto de políticas de segurança flexíveis que governam as ações de uma aplicação ou applet que pode ou não ser executada.

- Suporte nativo a thread para o ambiente operacional Solaris. Compressão de memória para classes carregadas. Alocação de memória com mais desempenho e melhor para a coleta de lixo. Arquitetura de máquina virtual conectável para outras máquinas virtuais, incluindo a Java HotSpot VMNew. Just in Time (JIT). Java Native Interface JNI de conversão.

- O conjunto de componentes de projeto, GUI (Swing). API Java 2D que fornece novos recursos gráficos 2D e AWT, bem como suporte para impressão. O Java *look and fell*. Uma nova API de acessibilidade.

- Framework de entrada de caracteres (suporte a japonês, chinês e coreano). Complexo de saída usando a API do Java 2D para fornecer um *display* bi-direcional, de alta qualidade de japonês, árabe, hebraico e outras línguas de caracteres.

- Java Plug-in para navegadores da web, incluída na plataforma Java 2, fornecendo um tempo de execução totalmente compatível com a máquina virtual Java amplamente implantadas em navegadores.

- Invocação das operações ou serviços de rede remoto. Totalmente compatível com Java ORB e incluído no tempo de execução.

- JDBC que fornece um acesso mais fácil aos dados para consultas mais flexíveis. Melhor desempenho e estabilidade são promovidos por cursores de rolagem e suporte para SQL3 de tipos.

Em 8 de Maio de 2000 foi anunciado o Java 2 versão 1.3 que trouxe ganho de desempenho em relação a primeira versão da JS2E de cerca de 40% no tempo de *start-up*. Tambem trouxe novas funcionaliadades como:

- O Java HotSpot VM de cliente e suas bibliotecas atentando ao desempenho ao fazer o J2SE versão 1.3 a *realease* o mais rápido até à data.
- Novos recursos, como o *caching applet* e instalação do pacote opcional Java através da tecnologia Java *Plug-in* para aumentar a velocidade e a flexibilidade com que os *applets* e aplicativos baseados na tecnologia Java pode ser implantado. Java *Plug-in* tecnologia é um componente do ambiente de execução Java 2 que permite Java *applets* e aplicativos para a execução.
- O novo suporte para RSA assinatura eletrônica, gerenciamento de confiança dinâmico, certificados X.509, e verificação de arquivos o que significa o aumento das possibilidades que os desenvolvedores tem para proteger dados eletrônicos.
- Uma série de novos recursos e ferramentas de desenvolvimento da tecnologia J2SE versão 1.3 que permite o desenvolvimento mais fácil e rápido de aplicações baseadas na tecnologia *web* ou Java *standalone* de alto desempenho.
- A adição de RMI/IIOP e o JNDI para a versão 1.3, melhora na interoperabilidade J2SE. RMI/IIOP melhora a conectividade com sistemas de *back-end* que suportam CORBA. JNDI fornece acesso aos diretórios que suportam o populares LDAP Lightweight Directory Access Protocol, entre outros.

No ano de 2002 no dia 6 de Fevereiro, foi lançado a J2SE versão 1.4. Com a versão 1.4, as empresas puderam usar a tecnologia Java para desenvolver aplicativos de negócios mais exigentes e com menos esforço e em menos tempo. As novas funcionalidades como a nova I/O e suporte a 64 bits. A J2SE se tornou plataforma ideal para a mineração em grande escala de dados, inteligência de negócios, engenharia e científicos. A versão 1.4 forneceu suporte aprimorado para tecnologias padrões da indústria, tais como SSL, LDAP e CORBA a fim de garantir a operacionalidade em plataformas heterogêneas, sistemas e ambientes. Com o apoio embutido para XML, a autenticação avançada, e um conjunto completo de serviços de segurança, está versão forneceu base para padrões de aplicações Web e serviços interoperáveis. O J2SE avançou o desenvolvimento de aplicativos de cliente com novos controles de GUI, acelerou Java 2D, a performance gráfica, internacionalização e localização expandida de apoio, novas opções de implantação e suporte expandido para o até então Windows XP.

Com a chegada da JSE2 versão 1.5 (Java 5.0) em 30 de Setembro de 2004, impulsionou benefícios extensivos para desenvolvedores, incluindo a facilidade de uso, desempenho global e escalabilidade, monitoramento do sistema e gestão e desenvolvimento. O Java 5 foi derivado do trabalho de 15 componentes Java Specification Requests (JSRs) englobando recursos avançados para a linguagem e plataforma. Os líderes da indústria na época que participam no grupo de peritos J2SE 5.0 incluiram: Apache Software Foundation, Apple Computer, BEA Systems, Borland Software Corporation, Cisco Systems, Fujitsu Limited, HP, IBM, Macromedia, Nokia Corporation, Oracle, SAP AG, SAS Institute, SavaJe Technologies e Sun Microsystems.

Novas funcionalidades foram implementadas como:

- Facilidade de desenvolvimento: os programadores da linguagem Java pode ser mais eficiente e produtivos com os recursos de linguagem Java 5 que permitiram a codificação mais segura. Nesta versão surgiu o *Generics* [5, 10], tipos enumerados, metadados e autoboxing de tipos primitivos permitindo assim uma fácil e rápida codificação.
- Monitoramento e gestão: Um foco chave para a nova versão da plataforma, a aplicativos baseados na tecnologia Java *Virtual Machine* que passou a ser monitorado e gerenciado com o *built-in* de suporte para Java *Management Extensions*. Isso ajudou a garantir que seus funcionários, sistemas de parceiros do cliente permanecessem em funcionamento por mais tempo. Suporte para sistemas de gestão empresarial baseados em SNMP também é viável.
- Um olhar novo aplicativo, mais moderna, baseada na tecnologia Java padrão e proporciona uma sensação GUI para aplicativos baseados na tecnologia Java. A J2SE 5.0 teve suporte completo a internacionalização e também possuindo suporte para aceleração de hardware por meio da API OpenGL e tambem para o sistema operacional Solaris e sistemas operacionais da distribuição Linux.
- Maior desempenho e escalabilidade: A nova versão incluiu melhorias de desempenho, tais como menor tempo de inicialização, um menor consumo de memória e JVM auto ajustável para gerar maior desempenho geral do aplicativo e desenvolvimento em J2SE 5.0 em relação às versões anteriores.

Java 1.6 (Java 6) foi divulgado em 11 de dezembro de 2006. Tornou o desenvolvimento mais fácil, mais rápido e mais eficiente em termos de custos e ofereceu funcionalidades para serviços web, suporte linguagem dinâmica, diagnósticos e aplicações desktop. Com a chegada dessa nova versão do Java houve combinação com o NetBeans IDE 5.5 fornecendo aos desenvolvedores uma estrutura confiável, de codigo aberto e compatível, de alta performance para entregar aplicativos baseados na tecnologia Java mais rápido e mais fácil do que nunca. O NetBeans IDE fornece uma fonte aberta e de alto desempenho, modular, extensível, multi-plataforma Java IDE para acelerar o desenvolvimento de aplicações baseadas em software e serviços *web*. Novas funcionalidades foram implementadas como:

- O Java 1.6 ajudou a acelerar a inovação para o desenvolvedor, aplicativos de colaboração *online* e baseadas na *web*, incluindo um novo quadro de desenvolvedores API's para permitir a mistura da tecnologia Java com linguagens de tipagem dinâmica, tais como PHP, Python, Ruby e tecnologia JavaScript. A Sun também criou uma coleção de mecanismos de script e pré-configurado o motor JavaScript Rhino na plataforma Java. Além disso, o software inclui uma pilha completa de clientes de serviços web e suporta as mais recentes especificações de serviços *web*, como JAX-WS 2.0, JAXB 2.0, STAX e JAXP.
- A plataforma Java 1.6 forneceu ferramentas expandidas para o diagnóstico, gestão e monitoramento de aplicações e também inclui suporte para o novo NetBeans Profiler 5.5 para Solaris DTrace e, uma estrutura de rastreamento dinâmico abrangente que está incluído no sistema operacional Solaris 10. Além disso, o software Java SE

6 aumenta ainda mais a facilidade de desenvolvimento com atualizações de interface ferramenta para o Java Virtual Machine (JVM) e o Java Platform Debugger Architecture (ACDP).

Java 7 [4] foi lançado no dia 28 de julho de 2011. Essa versão foi resultado do desenvolvimento de toda a indústria envolvendo uma revisão de codigo aberto e extensa colaboração entre os engenheiros da *Oracle* e membros do ecossistema Java em todo o mundo através da comunidade *OpenJDK* e do *Java Community Process* (JCP). Compatibilidade com versões anteriores de Java 7 com versões anteriores da plataforma a fim de preservar os conjuntos de habilidades dos desenvolvedores de software Java e proteger os investimentos em tecnologia Java.

Com essa versão novas funcionalidades foram adicionadas:

- As alterações de linguagem ajudaram a aumentar a produtividade do desenvolvedor e simplificar tarefas comuns de programação, reduzindo a quantidade de código necessário, esclarecendo sintaxe e tornar o código com mais legibilidade.
- Melhor suporte para linguagens dinâmicas incluindo: Ruby, Python e JavaScript, resultando em aumentos substanciais de desempenho no JVM.
- Uma nova API *multicore-ready* que permite aos desenvolvedores para se decompor mais facilmente problemas em tarefas que podem ser executadas em paralelo em números arbitrários de núcleos de processador.
- Uma interface de I/O abrangente para trabalhar com sistemas de arquivos que podem acessar uma ampla gama de atributos de arquivos e oferecem mais informações quando ocorrem erros.
- Novos recursos de rede e de segurança. Suporte expandido para a internacionalização, incluindo suporte a Unicode 6.0. Versões atualizadas das bibliotecas padrão.

Com o lançamento do Java SE 8 em 18 de Março de 2014, permitiu uma maior produtividade e desenvolvimento de aplicativos significativos aumentos de desempenho através da redução de linhas de código, *collectons* melhoradas, modelos mais simples de programação paralela e uso mais eficiente de processadores multi-core modernos.As principais características do JDK 8 são o Projeto Lambda, Nashorn JavaScript Engine, um conjunto de perfis compactas e a remoção da "geração permanente"do HotSpot Java Virtual Machine (JVM). A JDK 8 alcançou desempenho recorde mundial para 4 sistemas de soquete em servidores baseados em Intel e NEC por 2 sistemas de soquete em servidores SPARC da Oracle T5, com uma melhoria de desempenho de 12% para 41% em comparação com o JDK 7 na mesma configuração de Oracle. O JDK 8 adicionou novas funcionalidades como:

- As expressões lambda são suportados pelas seguintes características: As referências a metodos são compactas, maior legibilidade expressões lambda para métodos que já têm um nome. Métodos padrão que permitem adicionar novas funcionalidades para as interfaces de suas bibliotecas e assegurar a compatibilidade binária com o código escrito para versões mais antigas dessas interfaces. Eles são os métodos de

interface que têm uma aplicação e a palavra-chave padrão no início da assinatura do método. Além disso, pode-se definir métodos estáticos em interfaces. Novos e aprimorados APIs que se aproveitam de expressões lambda e dos *streams* em Java 8 descrevem as classes novos e aprimorados que se aproveitam de expressões lambda e *streams*.

- O compilador Java aproveita digitação alvo para inferir os parâmetros de tipo de um método de invocação genérica. O tipo de destino de uma expressão é o tipo de dados que o compilador Java espera, dependendo de onde a expressão aparece. Por exemplo, você pode usar o tipo de destino de uma instrução de atribuição para o tipo de inferência em Java 7. No entanto, em Java 8, você pode usar o tipo de destino para a inferência de tipos em mais contextos.
- Anotações sobre tipos Java. Agora é possível aplicar uma anotação em qualquer lugar onde um tipo é usado. Utilizado em um conjunto com um sistema de tipo de conector, isso permite a verificação de tipo mais forte de seu código.
- Repetindo Anotações. Agora é possível aplicar o mesmo tipo de anotação mais de uma vez para a mesma declaração ou o tipo de utilização.

1.5 Aspectos evolutivos da liguagem Java

1.5.1 Java 2

A primeira versão do Java Security, disponível no JDK 1.1 [6], contém um subconjunto dessa funcionalidade, incluindo API's para:

- Assinaturas Digitais: Algoritmos de assinatura digital, como DSA ou MD5 com RSA. A funcionalidade inclui a geração de chaves público/privado , bem como assinatura e verificação de dados digitais.
- Gerenciamento de Chaves: Um conjunto de abstrações para o gerenciamento de "diretores" (entidades como usuários individuais ou grupos), suas chaves, e os seus certificados. Ele permite que aplicativos para projetar seu próprio sistema de gerenciamento de chaves, e para interoperar com outros sistemas em alto nível.
- Lista de controle de acesso: Um conjunto de abstrações para o gerenciamento de "diretores" e suas permissões de acesso.
- A obtenção de um objeto de assinatura:

```
1  import java.security.Signature;
2  import java.security.NoSuchAlgorithmException;
3
4  public class SignFile {
5    Signature signature;
6
7    private void init(String algorithm) throws NoSuchAlgorithmException{
8      signature = Signature.getSignature(algorithm);
9      }
10 }
```

- Em versões anteriores, Java suportava apenas *top-level* classes, que devem ser membros de pacotes. Na versão 1.1, o programador Java pode agora definir classes internas como membros de outras classes [10], localmente dentro de um bloco de instruções, ou (anonimamente) dentro de uma expressão.

```
1  public class FixedStack {
2    ...
3     public java.util.Enumeration elements() {
4         return new FixedStack$Enumerator(this);
5     }
6  }
7
8  class FixedStack$Enumerator implements java.util.Enumeration {
9    private FixedStack this$0;
10
11   FixedStack$Enumerator(FixedStack this$0) {
12     this.this$0 = this$0;
13     this.count = this$0.top;
14    }
15
16   int count;
17   public boolean hasMoreElements() {
18     return count > 0;
19   }
20
21   public Object nextElement() {
22     if (count == 0)
23       throw new NoSuchElementException("FixedStack");
24
25     return this$0.array[--count];
26   }
27 }
```

- Para escrever um objeto remoto RMI, escreve-se uma classe que implementa uma ou mais interfaces remotas.

```
1 package examples.hello;
2 public interface Hello extends java.rmi.Remote {
3   String sayHello() throws java.rmi.RemoteException;
4 }
```

- HelloImpl.java

```
1  package examples.hello;
2
3  import java.rmi.*;
4  import java.rmi.server.UnicastRemoteObject;
5
6  public class HelloImpl extends UnicastRemoteObject implements Hello{
7    private String name;
8
9    public HelloImpl(String s) throws RemoteException {
10     super();
11     name = s;
12   }
13
14   public String sayHello() throws RemoteException {
15     return  "Hello World!";
16   }
17
18   public static void main(String args[]){
19
20     System.setSecurityManager(new RMISecurityManager());
21
22     try {
23       HelloImpl obj = new HelloImpl("HelloServer");
24       Naming.rebind("//myhost/HelloServer", obj);
25       System.out.println("HelloServer bound in registry");
26     } catch (Exception e) {
27       System.out.println("HelloImpl err: " + e.getMessage());
28       e.printStackTrace();
29     }
30   }
31 }
```

1.5.2 Java 4

- *Assertion Facility* [2]. As *assertions* são expressões booleanas que o programador acredita ser verdade sobre o estado de um programa de computador. Por exemplo, depois de ordenar uma lista o programador pode afirmar que a lista está em ordem crescente. Avaliando as afirmações em tempo de execução para confirmar a sua validade é uma das ferramentas mais poderosas para melhorar a qualidade do código, uma vez que rapidamente se descobre equívocos do programador sobre o comportamento de um programa.

1.5.3 Java 5

- *Generics* [2, 5, 18]. Este novo recurso para o sistema de tipo permite que um tipo ou método operar em objetos de vários tipos, proporcionando em tempo de compilação tipo de segurança. Acrescenta em tempo de compilação um tipo de segurança para as *collections* e elimina o trabalho penoso de *casting*. Um exemplo do uso de *collections* e *generics* respectivamente:

```
1  static void expurgate(Collection c) {
2    for (Iterator i = c.iterator(); i.hasNext(); )
3      if (((String) i.next()).length() == 4)
4        i.remove();
5    }
6
7  static void expurgate(Collection<String> c) {
8    for (Iterator<String> i = c.iterator(); i.hasNext(); )
9      if (i.next().length() == 4)
10       i.remove();
11 }
```

- *For-Each Loop*. Esta nova estrutura de linguagem elimina o trabalho e erro de propensão de iteradores e variáveis de índice quando a iteração ocorre sobre coleções e arrays. Como a construção evoluiu com o advento dessa nova estrutura:

```
1  void cancelAll(Collection<TimerTask> c) {
2    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
3      i.next().cancel();
4  }
5
6  void cancelAll(Collection<TimerTask> c) {
7    for (TimerTask t : c)
8      t.cancel();
9  }
```

- *Varargs*. Esta nova estrutura tende a eliminar a necessidade de passagem manual de listas de argumentos em um array ao invocar métodos que aceitam de um comprimento variável de uma lista de argumentos. Nas versões anteriores, um método levava um número arbitrário de valores necessários a criar uma matriz e colocar os valores para a matriz antes de chamar o método.

```
public class Test {
  public static void main(String[] args) {
    int passed = 0;
    int failed = 0;
    for (String className : args) {
        try {
            Class c = Class.forName(className);
            c.getMethod("test").invoke(c.newInstance());
            passed++;
        } catch (Exception ex) {
            System.out.printf("%s failed: %s%n", className, ex);
            failed++;
        }
    }
    System.out.printf("passed=%d; failed=%d%n", passed, failed);
  }
}
```

- *Autoboxing/Unboxing*. Esta nova estrutura elimina o trabalho de conversão manual entre tipos primitivos (como *int*) e os tipos de classes *wrapper*

1.5.4 Java 6

Não ocorram mudanças ou introdução de novas estruturas na linguagem Java [2].

1.5.5 Java 7

- *Multi Catch* e lançamento de exceções com melhora na verificação de tipos. Um único bloco *catch* poderá lidar com mais de um tipo de exceção. Além disso, o compilador executa a análise mais precisa das exceções. Isso permite que o programador especifique tipos de exceção mais específicos na cláusula de uma declaração método. Um exemplo de como era as estruturas que usavam *cacths* e com a introdução de *multi catch* com o Java 7 [4], respectivamente.

```
catch (IOException ex) {
  logger.log(ex);
  throw ex;
}catch (SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

```
catch (IOException|SQLException ex) {
  logger.log(ex);
  throw ex;
}
```

- O *try-with-resouces*. A declaração *try-with-resouces* é uma instrução *try* que declara um ou mais recursos. Um recurso é um objeto que deve ser fechada após o programa terminar com ele. Essa declaração garante que cada recurso é fechada no final da declaração [3].

```

public static void writeToFileZipFileContents(
      String zipFileName, String outputFileName) throws java.io.
    IOException {

  java.nio.charset.Charset charset = java.nio.charset.StandardCharsets
    .US_ASCII;
  java.nio.file.Path outputFilePath = java.nio.file.Paths.get(
    outputFileName);

  try(
     java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName)
    ;
     java.io.BufferedWriter writer = java.nio.file.Files.
    newBufferedWriter(outputFilePath, charset)
  ){

    for (java.util.Enumeration entries = zf.entries(); entries.
    hasMoreElements();) {
        String newLine = System.getProperty("line.separator");
        String zipEntryName = ((java.util.zip.ZipEntry)entries.
    nextElement()).getName() + newLine;
        writer.write(zipEntryName, 0, zipEntryName.length());
     }
  }
}
```

- Inferência de tipos para criação de instâncias em *generics* [5, 10, 18]. Com o Java 7 pode-se substituir os argumentos de tipo necessários para invocar o construtor de uma classe genérica com um conjunto vazio de parâmetros de tipo (<>), desde que o compilador infira os argumentos de tipo a partir do contexto. Este par de colchetes angulares é informalmente chamado de diamante.

```
1  Map<String, List<String>> myMap = new HashMap<String, List<String>>();
2  Map<String, List<String>> myMap = new HashMap<>();
3
4  List<String> list = new ArrayList<>();
5  list.add("A");
6
7  list.addAll(new ArrayList<>());
8
9  class MyClass<X> {
10   <T> MyClass(T t) {
11    ...
12   }
13 }
```

1.5.6 Java 8

- Melhoria na inferência de tipos. O compilador Java aproveita digitação para inferir os parâmetros de tipo de uma invocação de método genérica. O tipo de destino de uma expressão é o tipo de dados que o compilador Java espera, dependendo de onde a expressão aparece. Por exemplo, pode-se usar o tipo de destino de uma instrução de atribuição para o tipo de inferência em Java 7. No entanto, em Java 8, pode-se usar o tipo de destino para a inferência de tipos em mais contextos. O exemplo mais proeminente está usando tipos de destino de um método de invocação para inferir os tipos de dados dos seus argumentos.

```
1  List<String> stringList = new ArrayList<>();
2  stringList.add("A");
3  stringList.addAll(Arrays.asList());
```

- Expressões lambda. Permitem encapsular uma única unidade de comportamento e passá-lo para outro código. Pode-se usar uma expressãos lambda, se quiser uma determinada ação executada em cada elemento de uma *collection*, quando o processo for concluído, ou quando um processo encontra um erro. [4]

```
public class Calculator {

  interface IntegerMath {
    int operation(int a, int b);
  }

  public int operateBinary(int a, int b, IntegerMath op) {
    return op.operation(a, b);
  }

  public static void main(String... args) {

    Calculator myApp = new Calculator();
    IntegerMath addition = (a, b) -> a + b;
    IntegerMath subtraction = (a, b) -> a - b;
    System.out.println("40 + 2 = " + myApp.operateBinary(40, 2, addition));
    System.out.println("20 - 10 = " + myApp.operateBinary(20, 10,
     subtraction));

  }
}
```

1.6 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 [17], 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 [22]. 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 desenvolvedor 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

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 [8]. 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. [16] 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.

2.1 Análise léxica

Ferramentas que operam em código-fonte conforme [25] 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.

2.2 Parser

Um analisador de linguagem usa uma gramática livre de contexto (CFG) indicado por [12] 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.

2.2.1 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 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

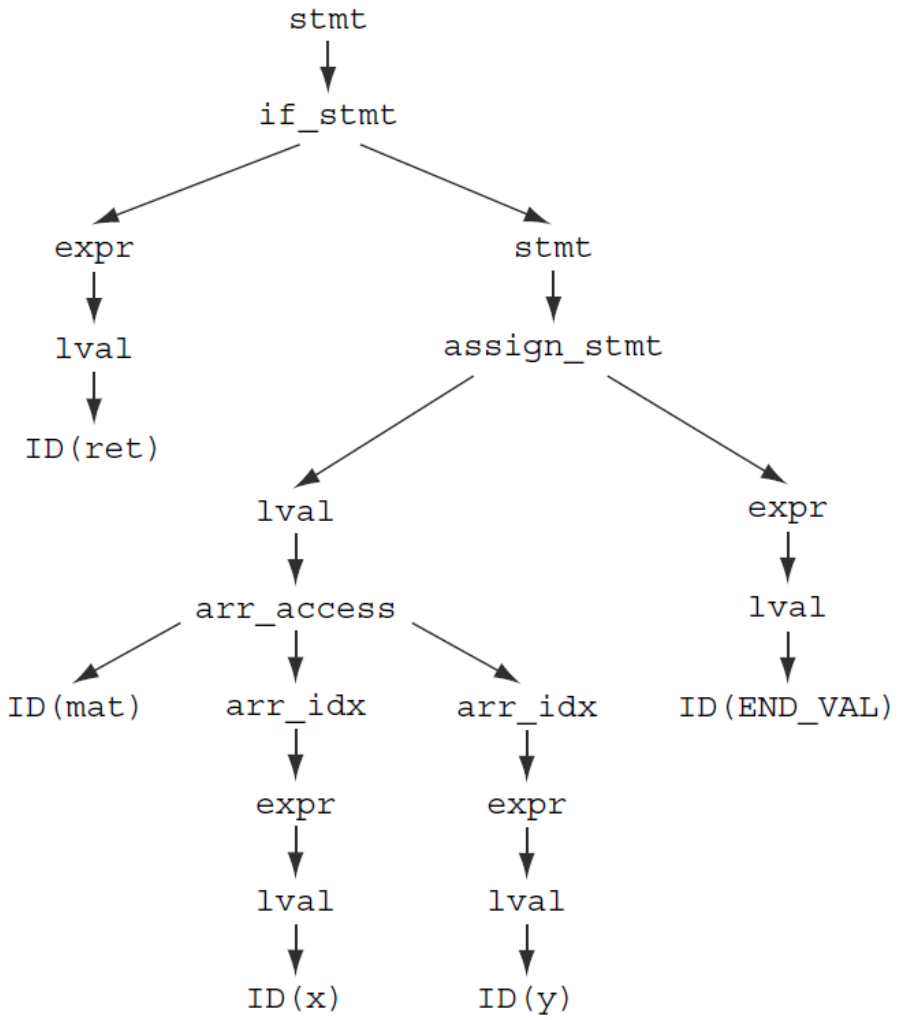

Figura 2.1: Árvore de parser.

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* [16].

2.3 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.

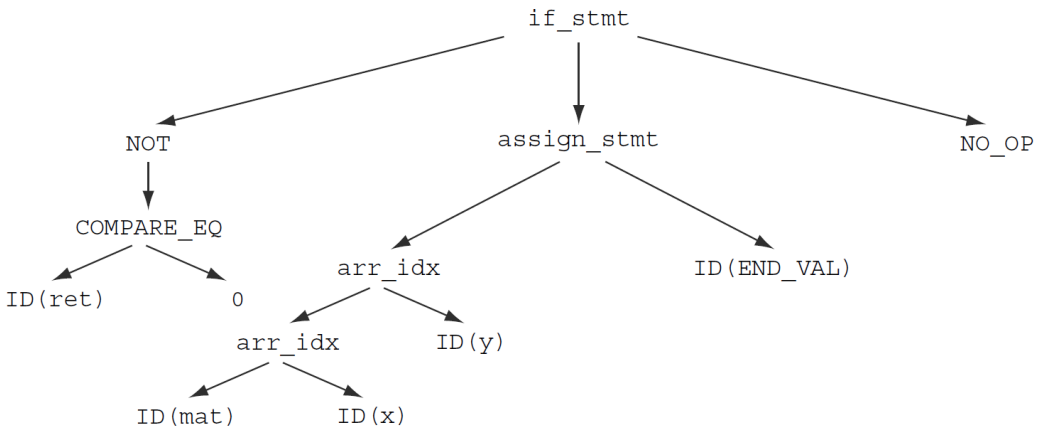

Figura 2.2: Árvore AST.

2.4 Análise semântica

Como a AST está sendo construída, a ferramenta cria uma tabela de símbolos ao lado dela. Para cada identificador no programa, a tabela de símbolos associa o identificador

com seu devido tipo e um ponteiro para a sua declaração ou definição. Com a AST e a tabela de símbolo, a ferramenta está agora equipado-se para realizar a verificação de tipo. A ferramenta de análise estática não pode ser obrigados a comunicar erros de checagem de tipo da maneira um compilador faz, mas informações de tipo é criticamente importante para a análise de uma linguagem orientada a objetos, porque o tipo de um objeto determina o conjunto de métodos que o objeto pode invocar. Além disso, é normalmente desejável para converter, pelo menos, as conversões do tipo implícito no código fonte para conversões de tipo explícitas no AST. Por estas razões, uma ferramenta de análise estática avançado tem a ver apenas como muito trabalho relacionado com a verificação de tipo como um compilador faz. No mundo do compilador, resolução de símbolo e verificação de tipo são referidos como análise semântica porque o compilador está atribuindo significado aos símbolos encontrada no programa. As ferramentas de análise estática que usam essas estruturas de dados têm uma vantagem distinta sobre ferramentas que não o fazem. Por exemplo, eles podem interpretar corretamente o significado dos operadores sobrecarregados em C++ ou determinar que um método em Java chamado doPost () é, na verdade, uma parte de uma implementação de HttpServlet.Estas capacidades permitem uma ferramenta para executar verificações úteis na estrutura deo programa. Após análise semântica, compiladores e a análise estática mais avançada ferramentas de formas de peça. Um compilador moderno usa a AST e o símbolo e o tipo informações para gerar uma representação intermediaria, uma versão genérico do código de máquina que é adequado para otimização e, em seguida, a conversão em específico da plataforma de código-objeto. O caminho para ferramentas de análise estática é menos clara. Dependendo do tipo de análise a ser realizada, uma ferramenta de análise estática pode executar transformações adicionais sobre a AST ou pode gerar a sua própria variedade de representação intermediária adequada às suas necessidades. Se uma ferramenta de análise estática usa sua própria representação intermediária, que, geralmente, permite a atribuição, pelo menos, ramificando, *looping*, e chamadas de função. A representação intermediária que uma ferramenta de análise estática usa é geralmente umvista de nível superior do programa do que a representação intermediária que um compilador usa. Por exemplo, um compilador de linguagem C, provavelmente, converter todas as referências a campos para estruturar deslocamentos em *byte* na estrutura pela sua representação intermediaria, enquanto uma ferramenta de análise estática mais provavelmente continuará para se referir a estrutura de campos, pelos seus nomes.

2.5 Checagem de tipo

A checagem de tipos é a forma mais utilizada de análise estática, e aquela que a maioria dos programadores estão familiarizados. As regras do "jogo"são tipicamente definida pela linguagem de programação e executadas pelo compilador, portanto, um programador que obtiver pouco a dizer quando a análise é executada ou como a análise funciona. Verificação de tipo elimina categorias inteiras de erros de programação. Por exemplo, ele impede programadores de atribuição acidentalmente valores integrais de oposição variáveis. Pela captura de erros em tempo de compilação, verificação de tipo de tempo de execução e impede erros. Verificação de tipo é limitado em sua capacidade de detectar erros, porém, sofre com falsos positivos e falsos negativos como todas as outras formas de análise estática. Curiosamente, os programadores raramente reclamar sobre

uma escreva imperfeições do verificador. As demonstrações de Java no exemplo não vai compilar porque nunca é legal para atribuir uma expressão do tipo int para uma variável do tipo short, mesmo que a intenção do programador é inequívoca. A checagem de tipo sofre de falsos negativos também. Um exemplo de Java será quando o programa passará a verificação de tipo e compilar sem problemas, mas será falhar em tempo de execução. Arrays em Java são covariante, o que significa que o verificador de tipos permite uma variável de matriz de objeto para manter uma referência a uma matriz String (porque a classe String é derivado da classe de objecto), mas no tempo de execução Java não vai permitir que a matriz String para conter uma referência a um objeto do tipo Objeto.

Um falso positivo de verificação de tipo: Estas declarações Java não satisfazem tipo regras de segurança, embora sejam logicamente correta.

```
1  short s = 0;
2  int i = s; /* o checador de tipos permite isso */
3  short r = i; /*causara um falso positivo em tempo de compilacao assim
     ocorrendo um erro de tipo.*/
4
```

2.6 Checagem de estilo

Verificadores de estilo também são ferramentas de análise estática. Eles geralmente impor um pickier e um conjunto de regras mais superficial do que um verificador de tipos. Verificadores puro estilo fazem cumprir as regras relacionadas com espaços em branco, nomeação, funções obsoletas, comentando, estrutura de programa, e semelhantes. Como muitos programadores estão ferozmente anexado a sua própria versão de um bom estilo, a maioria dos verificadores de estilo são bastante flexível sobre o conjunto de regras que impõem. Os erros produzidos pela verificadores estilo muitas vezes podem afetar a legibilidade e a manutenção do código, mas não indicam que um erro particular irá ocorrer quando o programa rodam. Com o tempo, alguns compiladores têm implementado verificações de estilo opcionais. Por exemplo, bandeira do gcc: -Wall fará com que o compilador para detectar quando um switch não leva em conta todos os valores possíveis de um *Enum* escrito.

2.7 Entendimento do código

Ferramentas do programa compreensão ajudam os programadores a entender o sentido do programa de uma grande base de código. Os ambientes de desenvolvimento integrado (IDEs) incluem pelo menos algumas funcionalidade compreensão programa. Exemplos simples incluem "encontrar tudo utiliza desse método "e/ou "encontrar a declaração dessa variável global". Uma análise mais avançada pode suportar funcionalidades automáticas programa de refatoração, como renomear variáveis ou dividir uma única função em múltiplos funções. De nível superior ferramentas compreensão programa de tentar ajudar os programadores ter uma visão sobre a forma como um programa funciona. Alguns tentam fazer engenharia reversa informações sobre a concepção do programa com base

em uma análise da implementação, dando assim o programador uma visão abrangente do programa. Isto é particularmente útil para programadores que precisam entender o programa fora de um grande corpo de código que eles não escreveram.

2.8 Verificação de programa

A verificação de programa é uma ferramenta que aceita uma especificação e um corpo de código e em seguida, as tentativas para demonstrar que o código é implementado fielmente com a especificação. A especificação é uma descrição completa de tudo o programa deveria fazer, a ferramenta de verificação de programa pode realizar equivalência verificar se o código e a especificação corresponder exatamente. Mais comumente as ferramentas de verificação de software contra um especificação parcial que detalha apenas uma parte do comportamento de um programa. Este esforço, por vezes, passa a verificação de propriedade de nome. A maioria das ferramentas de verificação tendem a trabalhar na aplicação de inferência lógica ou realizando verificação de modelos. Muitas ferramentas de verificação de propriedade concentram-se em propriedades de segurança temporais. A propriedade de segurança temporais especifica uma seqüência ordenada de eventos que um programa que não deve ser realizada. Um exemplo de uma propriedade de segurança temporal é. Um local de memória não deve ser lido depois de ser libertado."A maioria das ferramentas permitem aos programadores escrever suas próprias especificações para verificar as propriedades específicas do programa.

2.9 Verificação de propriedade

Uma ferramenta de verificação propriedade é dito ser de som com respeito à especificação se ele vai sempre relatar um problema se houver. Em outras palavras, a ferramenta nunca vai sofrer um falso negativo. A maioria das ferramentas que afirmam ser de som exigir que o programa que está sendo avaliado cumprir determinadas condições. Alguns não permitem ponteiros de função, enquanto outros não permitir recursão ou assumir que dois ponteiros nunca de alias (aponte para o mesmo local de memória). Para grandes quantidades de código, é quase impossível de satisfazer as condições estipuladas pela ferramenta, de modo a garantia de solidez não é significativo. Por esta razão, a solidez é raramente uma exigência do ponto de vista de um praticante.Em busca da solidez ou por causa de outras complicações, uma propriedade de verificação ferramenta pode produzir falsos positivos. No caso de um falso positivo, o contra-exemplo irá conter um ou mais eventos que não podia realmente ter lugar. Um exemplo é uma fuga de memória. O verificador de propriedade deu errado; ele não entende que, ao retornar NULL, malloc () é indicando que não há memória foi alocada. Isso pode indicar um problema com a forma como a propriedade for especificado,ou poderia ser um problema com o modo como o verificador propriedade funciona.

Capítulo 3

Ferramentas de Análise Estática

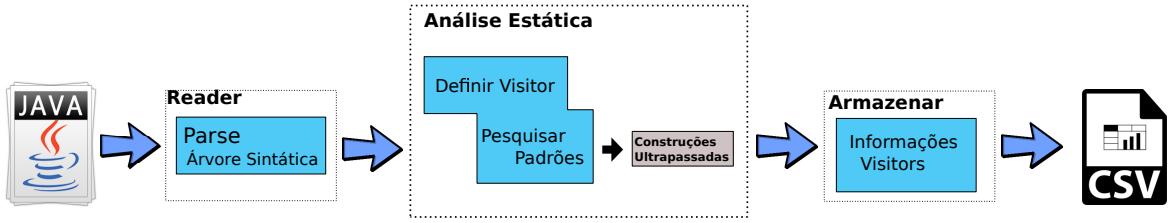

Figura 3.1: Alto nível de funcionamento do analisador estático.

No mais elevado nível de abstração do analisador estático a Figura: 3.1 demonstra seu funcionamento que é encontrar um código fonte Java, criar um *Parse* que é uma representação intermediária deste código fonte em seguida é aplicado uma série de mecanismos de análise estática para coletar as informações de interesse no código fonte e por fim é gerado relatórios CSV.

Atualmente existem diversas tecnologias capazes de prover ferramentas para implementar um analisador estático entretanto devido a maior experiência com uso da linguagem Java, neste projeto foi utilizado a infraestrutura da plataforma Eclipse JDT, *Eclipse Java Development Tools* [1]. O EclipseJDT [1] fornece um conjunto de ferramentas que contribuem com elaboração uma análise sobre o código Java.

A biblioteca JDT é composta 4 componentes *APT, Core, Debug* e *UI*, neste projeto a adoção deu-se através do *JDT Core* que dispõe de uma modelo Java para a navegação dos elementos de uma árvore sintática, AST, onde os elementos podem ser pacotes, tipos, métodos e atributos. Também existe API pronta para a manipulação de código fonte.

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

O Eclipse JDT [1] fornece para este projeto um *Parser* que produz uma representação intermediária baseada em um conjunto de classes Java que representam uma AST de um código fonte. Fornece ainda uma infraestrutura de *visitors* [16] que possibilitam a análise estática de código fonte.

Um *visitor* é um padrão de projetos proposto por Eric Gamma [16], este padrão de projeto tem sua característica comportamental representando uma operação a ser realizada sobre elementos da estrutura de um objetos. Neste caso operação a ser realizadas

nos nós de uma árvore sintática. Um *visitor* permite que uma nova operação seja criada sem que a os elementos operados sofram alterações. Com isso é trivial adicionar novas funcionalidades em um *visitor* existente ou criar um novo.

Projetar um analisador estático para extrair informações de softwares desenvolvido não é uma simples tarefa, entretanto EclipseJDT [1] prove uma abstração significativa tornado menos árdua este projeto. O que pode possibilitar projetar uma arquitetura mais robusta para este analisador onde o foco de qualquer desenvolvedor que deseje utilizá-lo concentre-se apenas na produção de seus *Visitors*.

Um ponto de extrema relevância foi tornar este analisador independente de qualquer plataforma e IDE foi um ponto vital para o concepção deste projeto que não tem como intuito ser plugin de qualquer IDE o que acarretaria na limitação do seu uso a um cenário específico mas sim uma ferramenta para apoiar no desenvolvimento de um software com construções atuais. Utilizando a portabilidade nativa entre as plataformas provido pela linguagem Java este trabalho foi concebido com intuito de atender ao mais diversos desenvolvedores quer utilizem Linux, Windows, Mac ou qualquer outro sistema operacional que tenha suporte para Java.

A análise tem como início a seleção de projetos, nesse caso seleção em repositórios públicos, onde após o download é informado analisador o diretório raiz onde estes estão localizados. Após é iniciado uma listagem dos arquivos fonte Java contidos nos projeto e contabilizados o total de LOC e criado uma árvore sintática para cada arquivo encontrado através de um *parser* provido pela biblioteca EclipseJDT [1]. Em seguida os *Visitors* são instanciados com o objetivo de percorrer os nós destas árvores para pesquisar por construções de código previamente determinadas.

Todas as construções pesquisadas quando encontradas pelos *Visitors* são armazenadas temporariamente enquanto o analisador verifica todo projeto, quando é verificado a última árvore sintática pelos *Visitors* é iniciado o processo de exportar os dados encontrados para um arquivo CSV o conteúdo relevante destes blocos, a Figura: 3.1 demostra de maneira clara o funcionamento do analisador. Após a exportação dos dados o analisador inicia todo processo novamente caso exista mais de um projeto.

Devido ao mecanismo de *reflection* proveniente da linguagem Java, o desenvolvedor não tem a necessidade de implementar código para exportação de dados tendo em vista que isto ocorre automaticamente através da introspecção realizada pelo analisador extraindo os dados armazenados pelos *Visitors*.

3.1 Arquitetura

3.1.1 Visitors

Conforme mencionado anteriormente, nos *Visitors* é onde deve ser concentrado todo esforço para extrair as informações com a maior confiabilidade possível. Efetuar a criação de novos *Visitor* quando necessário na atual arquitetura deste analisador requer que todos sejam obrigatoriamente estendidos da classe *Visitor.java* a qual por sua vez entende da API *ASTVisitor*, e implementa uma interface parametrizada que possui um coleção do tipo, $<\boldsymbol{T}>$, onde esta terá a responsabilidade de armazenar os dados minerados pelos *Visitors*.

Os *Visitors* criados necessitam somente que cada *Visitor* sobrescreva o método *public boolean visit* que é um método abstrato da classe *ASTVisitor* de acordo com sua necessidade.

```
1 package br.unb.cic.sa.visitors;
```

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.

```
2
3  import org.eclipse.jdt.core.dom.ASTVisitor;
4  import org.eclipse.jdt.core.dom.CompilationUnit;
5  import br.unb.cic.sa.model.Data;
6
7  public class Visitor<T> extends ASTVisitor implements IVisitor<T> {
8    protected CompilationUnit unit;
9    protected Data<T> collectedData;
10   protected String file;
11
12   public Visitor() {}
13
14   @Override
15   public void setUnit(CompilationUnit unit) { this.unit = unit; }
16
17   @Override
18   public void setCollectedData(Data<T> colletion) { this.collectedData =
       colletion; }
19
20   @Override
21   public void setFile(String file) {  this.file = file; }
22
23   @Override
24   public Data<T> getCollectedData() { return collectedData; }
25 }
```

A tabela: 3.1 detalha os 17 *Visitors* criados com a respectiva descrição do trabalho realizado. Como forma de exemplificar a criação de um *Visitors*, etapa a qual é composta de 4 passos quo serão demonstrados a seguir.

A investigação para saber como o tratamento de exceção Java tem sido utilizado acarretou na criação de um *Visitor* específico para este fim, *TryStatementVisitor*, responsável por pesquisar este mecanismo o qual pode contar com alguma evoluções ao logo do histórico da linguagem Java. A pesquisa é iniciada encontrando blocos *trys/catch*, onde destes será coletadas as informações referentes a adoção de *resources* e oportunidade de utilizar *multicatch* em blocos *trys* que contenham *catchs* iguais aninhados.

Criação do *Visitor TryStatementVisitor* com exemplo.

1. Inicialmente é necessário criar a classe modelo onde esta terá atribuição de receber os dados que serão extraídos pelos *Visitors*, basicamente as classes modelos são compostas de *getters* e *setters* e são declaradas no pacote ***br.unb.cic.sa.model***.

```
1  package br.unb.cic.sa.model;
2
3  public class TryStatementData  {
4    private String file;
5    private int startLine;
6    private int endLine;
7    private boolean tryWithResource = false;
8    private boolean multiCatch = false;
9
10   public TryStatementData(String file, int startLine, int endLine){
11     this.file = file;
12     this.startLine = startLine;
13     this.endLine = endLine;
14   }
15
16   //getters and setters
17 }
18
```

2. Em seguida, é concebida a criação do *Visitor* no pacote ***br.unb.cic.sa.visitors*** onde esta nova classe será estendida da classe parametrizada ***Visitor<TryStatementData>*** apresentada anteriormente onde a parametrização desta classe é o modelo criado anteriormente.

```
1  package br.unb.cic.sa.visitors;
2
3  import java.util.List;
4  import org.eclipse.jdt.core.dom.CatchClause;
5  import org.eclipse.jdt.core.dom.TryStatement;
6  import br.unb.cic.sa.model.TryStatementData;
7  import br.unb.cic.sa.similarity.BasicSimilarityChecker;
8  import br.unb.cic.sa.similarity.SimilarityChecker;
9
10 public class TryStatementVisitor extends Visitor<TryStatementData> {
11
12   SimilarityChecker similarity;
```

```
13
14    public TryStatementVisitor() {
15      similarity = new BasicSimilarityChecker();
16    }
17
18    @Override
19    public boolean visit(s node) {
20
21      TryStatementData t = new TryStatementData(this.file, unit.
      getLineNumber(node.getStartPosition()),
22          unit.getLineNumber(node.getStartPosition() + node.getLength())
      );
23
24      if (node.resources().size() > 0) {
25        t.setTryWithResource(true);
26      }
27
28      if (node.catchClauses().size() > 1) {
29        if (this.checkSimilarity(node.catchClauses())) {
30          t.setMultiCatch(true);
31        }
32      }
33
34      this.collectedData.addValue(t);
35
36      return super.visit(node);
37    }
38
39    private boolean checkSimilarity(List<CatchClause> catchClause) {
40      for (CatchClause cc : catchClause) {
41        for (CatchClause cn : catchClause) {
42          // To ignore the same catch in loops
43          if (!cc.equals(cn)) {
44
45          //Chamada externa para testar similaridade
46            if (this.similarity.checkSimilarity(cc.getBody(),
47                                cn.getBody())) {
48              return true;
49            }
50          }
51        }
52      }
53      return false;
54    }
55
56  }
57
```

Na linha24, é testada a condição para saber se este bloco fez adoção de *Resouce*. Na linha 30 é verificado que o bloco é um possível caso de *multicatch* pois existe mais de um bloco. Na linha 39 tem-se o método *checkSimilarity* o qual efetua a comparação dos blocos *Catch* aninhados, entretando de forma trivial na linha 46, pode ser utilizado um algoritmo mais sofisticado para testar por similaridade modificando

apenas o conteúdo do método ***checkSimilarity*** da classe ***SimilarityChecker*** o que não resulta em mudanças neste *Visitor*.

3. Em seguida deve-se declarar o cabeçalho no arquivo *resource/Beans.xml* do *Spring* que estará presente no CSV de saída. Onde este cabeçalho é composto pelo dados que serão extraídos pelos *Visitors* e armazenados no modelo criado.

```
<bean id="tryStatementData" class="br.unb.cic.sa.model.CSVData">
  <property name="outDir" value="output"/>
  <property name="fileName" value="tryStatement"/>
  <property name="head" value="typeProject, before, project, version,
    file, start, end, resource, multiCatch"/>
</bean>

```

4. Por fim declarar o *Visitor* como um *bean* do *Spring* para que este seja injetado no projeto e assim realize sua pesquisa.

```
<bean id="tryStatementVisitor" class="br.unb.cic.sa.visitors.
    TryStatementVisitor">
  <property name="collectedData" ref="tryStatementData"/>
</bean>

```

3.1.2 Exportar Dados

A exportação dos dados para CSV é realizado de forma automática utilizando o mecanismo de *reflection* fornecido por Java. A classe parametrizada ***CSVData<T>*** no pacote ***br.unb.cic.sa.model*** implementa a interface ***Data<T>*** onde ***<T>*** faz aos modelos declarados para as informações coletadas.

```
1 package br.unb.cic.sa.model;
2
3 public interface Data<T> {
4   public void setProject(Project project);
5   public void addValue(T value);
6   public void export();
7   public int size();
8   public void clean();
9 }
```

Onde o atributo *String[] head* é injetado pelo *Spring* pois fora declarado anteriormente na criação do visitor. O método export(), na linha 23, é o método responsável para exportar os dados em seus respectivos CSV pois este método recupera as informações contidas nas coleções populadas pelos *Visitors* e com *reflection* captura os campos declarados em especial para os métodos quais são pré-fixados com *get ou is* que terão seus valores capturados para que sejam exportados.

```java
package br.unb.cic.sa.model;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

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

  private Project project;
  private String fileName;
  private String outDir;
  private String[] head;
  private List<T> data;

  ...

  @Override
  public void export() {

    try (FileWriter writer = new FileWriter(this.makeCsv(head), true)){

      StringBuffer str = new StringBuffer("");

      if(data == null) {
        return;
      }

      for(T value : data) {
        str = new StringBuffer("");

        str.append(project.getTypeOfProject());
        str.append(";");
        str.append(project.getBefore());
        str.append(";");
        str.append(project.getProjectName());
        str.append(";");
        str.append(project.getProjectRevision());
        str.append(";");

        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 +
```

```
56                  Character.toUpperCase(fieldName.charAt(0)) +
57                  fieldName.substring(1);
58
59              try {
60                Method m = value.getClass().getDeclaredMethod(methodName);
61                str.append(m.invoke(value));
62                str.append(";");
63              }catch(
64                  NoSuchMethodException |
65                  IllegalAccessException |
66                  IllegalArgumentException |
67                  InvocationTargetException e) {
68
69                throw new RuntimeException("Type " +
70                           value.getClass().getName() +
71                           " must have a method named " +
72                            methodName);
73              }
74            }
75
76            writer.append(str.toString());
77            writer.append("\n");
78
79          }
80
81          writer.flush();
82
83        }
84        catch(Exception e) {
85          e.printStackTrace();
86        }
87      }
88  }
```

Capítulo 4

Investigação

Conforme afirmado por Jeffrey L. Overbey e Ralph E. Johnson em [17], 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 um grande e radical *refactoring* como ocorreu na linguagem Fortan [17], quando foi introduzido o paradigma de orientação a objeto que acarretou na quebra de compatibilidade com a versões anteriores.

Baseado nesta assertiva este trabalho tem como principal objetivo de encontrar construções obsoletas e encontrar possíveis casos de trechos de código que poderiam ter sido evoluídos ao longo das versões de Java e que não evoluíram. Neste caso esta sendo pesquisado caso onde deveriam ter sofrido o *refactoring* baseado nas características de Java 7 e Java 8. Dentre as característica mais importante a ser procuradas estão:

- Adoção e oportunidades de Multicatch.
- Adoção de Try com resource.
- Adoção e oportunidades de Switch com String.
- Adoção de Expressões Lambda.
- Oportunidades de exist - Expressões Lambda.
- Oportunidades de filter - Expressões Lambda.
- Oportunidades de map - Expressões Lambda.

Além da replicação do estudo de Parnin .et.al [18] e diferentemente do trabalho original, estender como objetivo compreender como a adoção de Generics se correlaciona com a data de lançamento inicial dos projetos selecionados.

Para a realização deste trabalho foram escolhidos 47 projetos open-source alguns sendo os mesmos existentes em, [15, 18, 24], divididos em 3 grupos, Aplicações, Bibliotecas e Servidores/Banco de dados conforme tabela: 4.1 o que totalizou mais de 8.5M de LOC.

Os dados contidos no arquivos CSV foram processados e analisados com o software R [20] versão (3.1.2) além da biblioteca ggplot [26] versão (1.0.1) para gerar gráficos mais consistentes.

Tabela 4.1: Projetos Escolhidos

	Sistema	Versão	LOC
Application	ANT	1.9.6	135741
	ANTLR	4.5.1	89935
	Archiva	2.2.0	84632
	Checkstyle	1.0	82163
	Eclipse	R4_5	13429
	Eclipse-CS	6.9.0	20426
	FindBugs	3.0.1	131351
	FitNesse	20150814	72836
	Free-Mind	1.0.1	67357
	Gradle	2.7	193428
	GWT	2.7.0	15421
	Ivy	2.4.0	72630
	jEdit	5.2.0	118492
	Jenkins	1.629	113763
	JMeter	2.13	111317
	Maven	3.3.3	78476
	Openmeetings	3.0.6	50496
	Postgree JDBC	9.4.1202	43596
	Sonar	5.0.1	362284
	Squirrel	3.4.0	252997
	Vuze	5621-39	608670
	Weka	3.6.12	274978
Library	Axis	1.4	121820
	Commons Collections	4.4.0	51622
	Crawler4j	4.1	3986
	Hibernate	5.0.1	541116
	Isis	1.9.0	262247
	JClouds	1.9.1	301592
	JUnit	4.1.2	26456
	Log4j	2.2	69525
	MyFaces	2.2.8	222865
	Quartz	2.2.1	31968
	Spark	1.5.0	31282
	Spring-Framework	4.2.1	531757
	Storm	0.10.0	98344
	UimaDucc	2.0.0	96020
	Wicket	7.0.0	211618
	Woden	1.0	29348
	Xerces	2.11.0	126228
Server - Database	Cassandra	2.2.1	282336
	Hadoop	2.6.1	896615
	Jetty	9.3.2	299923
	Lucene	5.3.1	506711
	Tomcat	8.0.26	287897
	UniversalMedia Server	5.2.2	54912
	Wildfly	9.0.1	392776
	Zookeeper	3.4.6	61708

Capítulo 5

Resultados

5.1 Lambda

Ao todo o visitor responsável por pesquisar a adoção de *Lambda* encontrou somente 824 caso de uso. Distribuidos em 365 arquivos o que leva a acreditar que tal característica não esta sendo tão relevante quanto sua espectativa de lançamento. Onde somente 2 casos não estão envolvidos em testes unitátos, os demais 824 estão sendo empregados no desevolvimento de testes unitários conforme exibido pela figura: 5.1. Vale resaltar que somente duas únicas ocorrências de lambda foram encontradas no projeto Jetty versão 9.3.2 nos fontes *PathMap.java* e *RegexSet.java*.

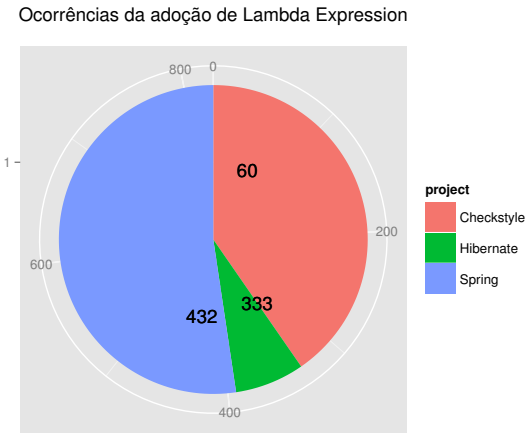

Figura 5.1: Adoção de *Lambda* em testes unitários.

5.1.1 Oportunidades de Aplicar Lambda

Entretanto foram encontradas 2618 caso de possível utilização de *Lambda Expression* distribuido em 1635 oportunidades sobre o padrão *exists* onde um *loop* que itera *Collections* para verificar se um elemento existe. E 983 que utilizam o padrão *filter* onde um *loop*

que itera uma *Collection* retornar uma outra *Collection* com os elementos dado um filtro. Onde em ambos os casos podem ser migrados para *Lambda* através da *Inteface Stream*.

A figura: 5.2, exibe a distribuição do padrão *exists* e *filter* encontrados pelos *visitors*.

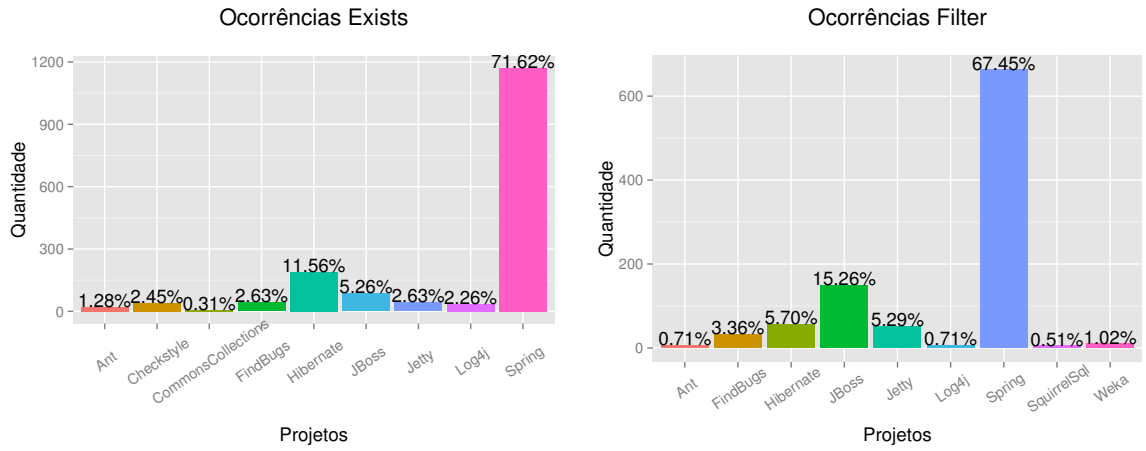

Figura 5.2: Ocorrências de Filter e Exists nos Projetos.

Uma ênfase mais minunciosa sobre o Spring para demosntrar como suas *releases* tem distribuido essas ocorrências pois pode-se constatar que com 67.45%, 663 casos, de *Filter* e com 71.62%, 1170 casos, de Exists é o projeto que contém mais ocorrências destas características a Figuras: 5.3 e 5.4.

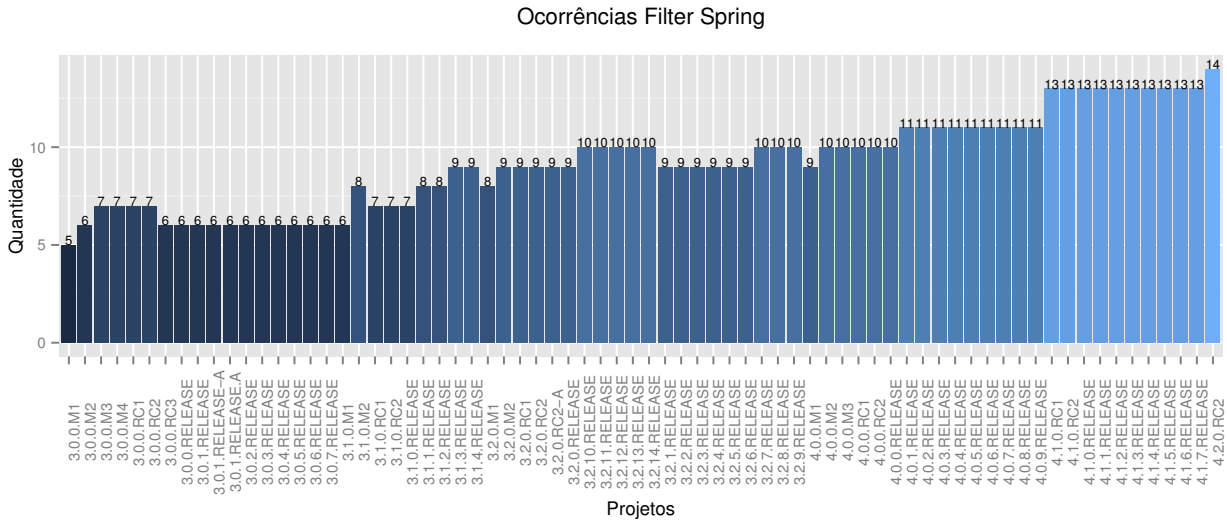

Figura 5.3: Ocorrências de Filter nas Versões do Spring.

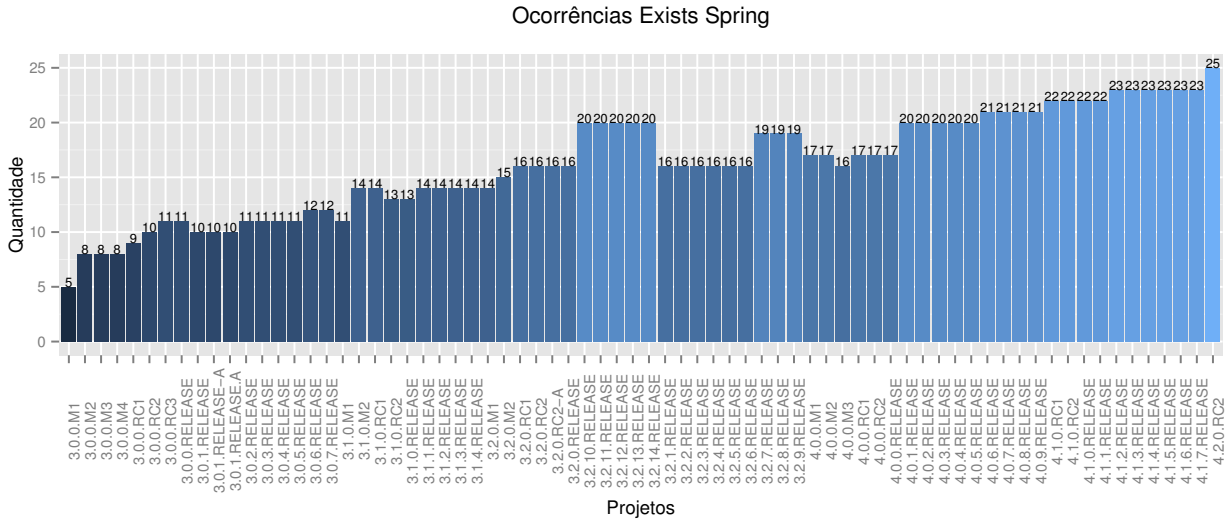

Figura 5.4: Ocorrências de Exists nas Versões do Spring.

5.2 MultiCatch

Aparentemente esta sendo um recurso pouco utilizado tendo em vista a grande quantidade de reais oportunidades mineradas neste estudo conforme exibido na Figura: 5.5. Foram encotrados ao todo 10368 *trys* que possuem blocos *catchs* repetidos essas ocorrências estão distribuídas em 7297 arquivos totalizando 97347 LOC, o teste de similaridade entre os *catchs* foi realizado através de uma chamada a um método exteno que verificava a igualdade podendo ser facilmente alterado para verificar a similaridade baseando em algum algoritmo existente.

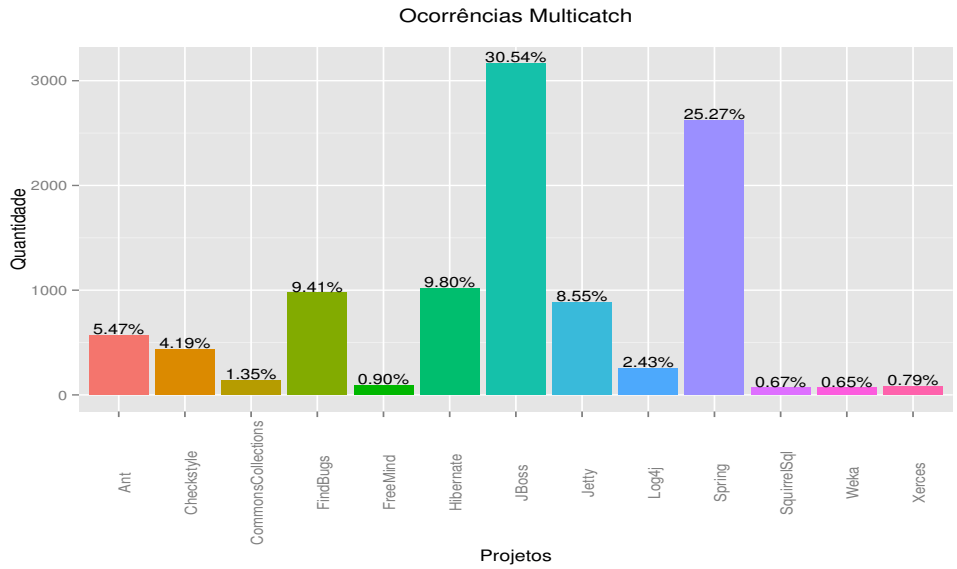

Figura 5.5: Oportunidades de *Multicatch* nos projetos.

```
1  //              Sem uso de Multicacth 17 LOC
2  try {...}
3  catch (ConverterNotFoundException ex) {
4    PropertyChangeEvent pce =
5      new PropertyChangeEvent(this.rootObject, this.nestedPath +
     propertyName, oldValue, newValue);
6    throw new ConversionNotSupportedException(pce, td.getType(), ex);
7  }catch (ConversionException ex) {
8    PropertyChangeEvent pce =
9      new PropertyChangeEvent(this.rootObject, this.nestedPath +
     propertyName, oldValue, newValue);
10   throw new TypeMismatchException(pce, requiredType, ex);
11 }catch (IllegalStateException ex) {
12   PropertyChangeEvent pce =
13     new PropertyChangeEvent(this.rootObject, this.nestedPath +
     propertyName, oldValue, newValue);
14   throw new ConversionNotSupportedException(pce, requiredType, ex);
15 }catch (IllegalArgumentException ex) {
16   PropertyChangeEvent pce =
17     new PropertyChangeEvent(this.rootObject, this.nestedPath +
     propertyName, oldValue, newValue);
18   throw new TypeMismatchException(pce, requiredType, ex);
19 }
```

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

Um simples *refactoring* unindo estes blocos semelhantes por igualdade acarretaria em redução de 68063 LOC o que reduz o código duplicado em 70%, 29284 LOC, tornando essa *feature* muito relevante na rescrita de um software. Como exemplo foi utilizado uma classe a *org.springframework.beans.AbstractNestablePropertyAccessor.java* do *Spring 4.2.0.RC2*, onde é possível implantar essa característica de maneira rápida e sem impacto na aplicação, reduzindo em 40% o trecho de código.

A figura: 5.6 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%.

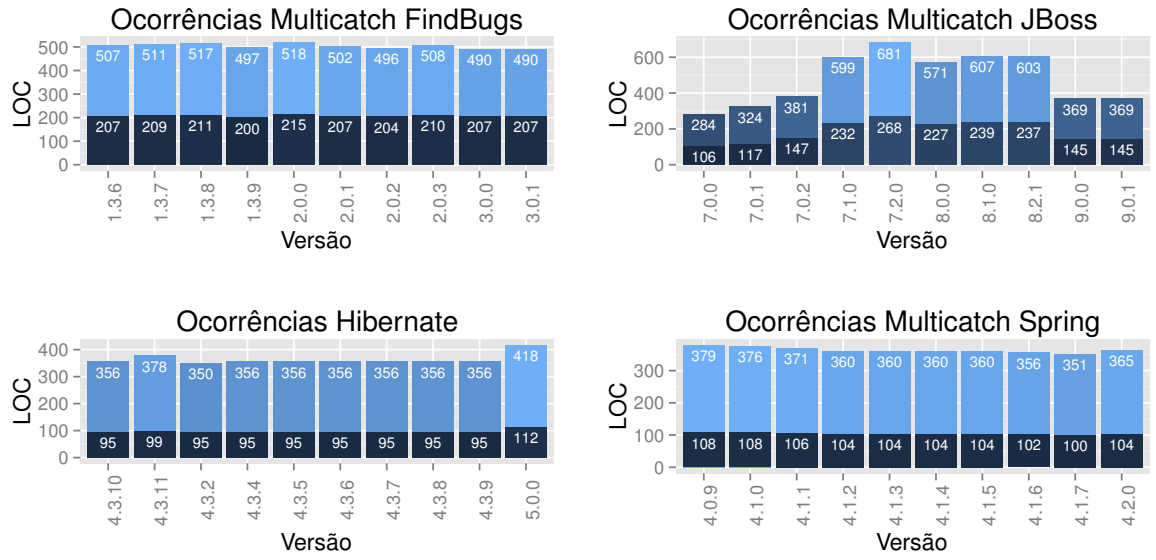

Figura 5.6: Oportunidades de *Multicatch* nos projetos.

5.3 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.

```
1 static String readFirstLineFromFile(String path) throws IOException {
2   try (BufferedReader br = new BufferedReader(new FileReader(path))) {
3     return br.readLine();
4   }
5 }
```

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: 5.7 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.

5.4 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*.

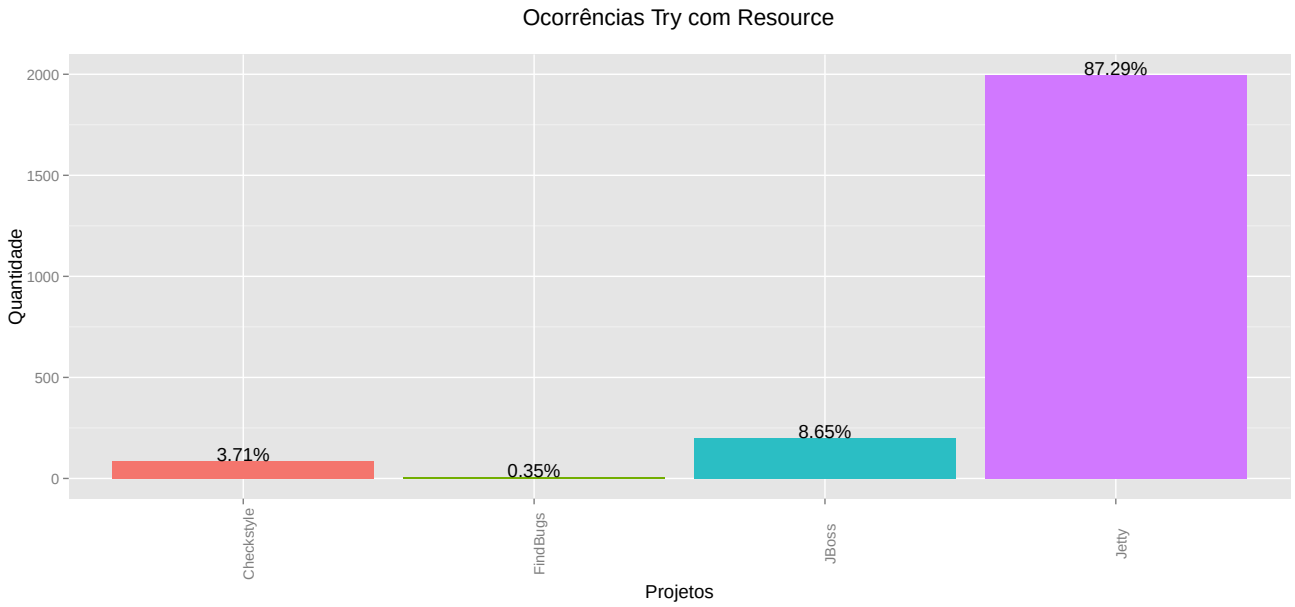

Figura 5.7: Oportunidades de *Try with Resource* nos projetos.

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.

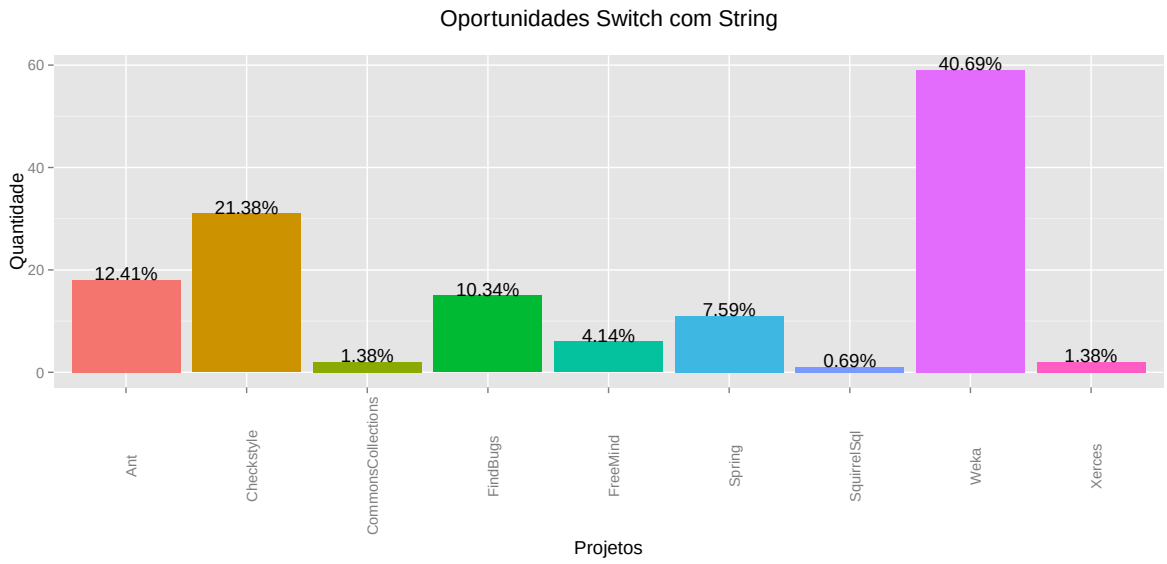

Figura 5.8: Oportunidades de *Switch with String* nos projetos.

Capítulo 6

Considerações Finais e Projeto Fututos

Dada uma redução significativa de código duplicado no caso da adoção de *multicatch* é impossível acreditar que projetos tão renomado não fazem uso desta *feature*. Ignorando uma característica que prove um código mais conciso e elegante conforme a proposta original desta *feature* pela Oracle em 2007. Entretanto vale ressaltar ainda como ponto vital no desenvolvimento que equipes não tem a evolução da linguagem como ponto relevante no desenvolvimento de seu produto. Conforme relatado pela equipe do Spring em seu forum a adoção de características como *Lambda* ou *Multicatch* ainda não foram adotadas pois seus projetos são desenvolvidos em Java7 e que ainda não faz total uso das *features* de Java7 o que leva a crer que somente farão adesão de features de Java8 quando utilizarem totalmente Java7.

6.1 Projeto Futuro

Existe também a possibilidade da extensão do anaslisador para indentificar *multicatch* por similaridade onde atualmente essa tarefa é realizada por igualdade. Com um algoritmo eficiente que detecte a similaridade com certeza a eficiência dessa *feature* irá crescer consideravelmente o que acarretará em um *refactoring* ainda mais significativo.Existe ainda a possibilidade de ampliar a pesquisa para encontrar objetos que contém método *close* dentro de blocos *Try's* o que acarretaria em um outro *refactoring* visando migrar estes objetos para os *resources* que estes pode receber.

Existem diversas caracterísitcas evolutivas da linguagem *Java* a serem atacados tais como *boilerplate* e *anonymousInnerClass* que podem ser tratados com *LambdaExpression*. Um projeto futuro de grande valia seria a evolução desta ferramenta para que obter um *refactoring* automático de códigos ultrapassados assim possibilitando a evolulção natural de *softwares* legados e *opensource*.

Como exemplo destes códigos temos o caso de *EnhancedFor* que iteram sobre uma *Collection* o que a evolução deste abre algumas possibilidade como a facilidade de aplicar concorrência na evolução com *Lambda* o que atualmente é muito complicado adicionar concorrência em um loop.

```
1 //   EnhacedFor
2   List<Transaction> groceryTransactions = new Arraylist<>();
3   for(Transaction t: transactions){
4     if(t.getType() == Transaction.GROCERY){
```

```
5         groceryTransactions.add(t);
6       }
7     }
8
9   //  Lambda Stream
10    List<Integer> transactionsIds =
11      transactions.stream().filter(t -> t.getType() == Transaction.GROCERY);
```

Referências

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

[2] Enhancements in java se 8 @ONLINE. http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html. accessed: 2015-05-27.

[3] Java se 7 advanced and java se 7 support @ONLINE. http://www.oracle.com/technetwork/java/javaseproducts/documentation/javase7supportreleasenotes-1601161.html. Acessed: 2015-05-20.

[4] Java se 7 features and enhancements @ONLINE. http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html. Acessed: 2015-05-20.

[5] The java tutorials @ONLINE. http://docs.oracle.com/javase/tutorial/extra/generics/index.html. Acessed: 2015-05-20.

[6] Jdk 1.1 new feature summary @ONLINE. http://www.public.iastate.edu/~java/docs/relnotes/features.html. Acessed: 2015-05-20.

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

[8] 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.

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

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

[11] 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.

[12] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[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.

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

[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.

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

[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.

[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.

[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.

[26] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009.