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. Professor I — CIC/UnB
Prof. Dr. Professor II — CIC/UnB

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. Professor I
CIC/UnB

Prof. Dr. Professor II
CIC/UnB

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

Brasília, 31 de março de 2015

Dedicatória

Dedicamos a nossa família

Agradecimentos

Agradecemos a todos que ajudaram de alguma forma nessa jornada que se completa.

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

Capítulo 1

Introdução

1.1 Introdução

Com o passar do tempo as linguagens de programação evoluem, entretanto não sabe-se ao certo como os softwares projetados e implementados há alguns anos acompanharam tais atualizações [14]. Conforme explicado por Overbey [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.

A base para tal trabalho será o desenvolvimento de algumas ferramentas que auxiliem em mudanças em um código legado para reduzir suas estruturas obsoletas e com baixa performance. Essas ferramentas tem como base a sua construção em linguagem *Java* com o intuito de tornar o processo de construção mais ágil e posteriormente aberto para o acoplamento de novos módulos. A construção de uma árvore sintática é um passo onde é feito para cada arquivo de código *.Java* e posteriormente de todos os arquivos *.Java* contidos em qualquer projeto para posterior análise. Este parser implica em listar todos os arquivos Java e gerar uma Árvore Sintática Abstrata (AST) para que posteriormente haja um percorrimento usando um visitor [16] nos blocos de código contidos nos nós da

(AST) [15] afim de compará-los como estes com a versão atual.

O processo de refatoração [19] tem como motivação a reestruturação de código, de forma que o código considerado pelo processo, morto,duplicado ou com perca de desempenho não haja código morto, duplicado ou com perca de desempenho em um determinado trecho de código. Esse processo não tem como premissa a atualização do código para novas estruturas de versões da linguagem mais recentes. Essa nova abordagem de refatoração tem com motivação a retirada de código obsoleto, devido a novas abordagens e estruturas das novas versões, e com baixo desempenho de sistemas sem prejuízo na sua engenharia e funcionamento.

Devido a esse tipo de refatoração de código visa a evolução do código para a uma mais recente com estruturas onde não haja perda de desempenho devido a mudança e também a atualização do código legado para estruturas modernas. Tais estruturas antigas com a ação dessa proposta de refatoração, tendem com o tempo a sair das estruturas providas pela linguagem Java como aconteceu com Fortan 90 como visto em Overbey [17] e essa abordagem tem o intuito de diminuir a quantidade de estruturas antigas que não fazem mais sentido pois novas estruturas realizam as mesmas funções no interpretador da linguagem Java. As modificações propostas pela ferramenta de refatoração não deverão trazer com que perca de desempenho ou aumento da complexidade do código portanto deixando como uma sugestão a alteração do código ou segmento de código para o usuário a adesão das propostas de refatoração ou não.

A árvore de sintaxe abstrata (AST) proposta por Chomsky em 1956 [13], é uma estrutura de dados que representa estruturas de cadeias sintáticas representada por um esqueleto semântico da linguagem em questão. É constituída através de um framework do ambiente de desenvolvimento integrado chamado Eclipse onde o nome desse framework é chamado de EclispeJDT agir sobre traz métodos implementados pelo próprio framework para o percorrer e ações na árvore sintática. A ideia é transformar inicialmente qualquer código fonte java em uma árvore sintática e devido a isso, o mapeamento da arvore já construída que é muito conveniente para inspecionar o código fonte de um arquivo ou de um projeto com vários arquivos em diferentes diretórios. Com isso é possível realizar ou sugerir ao usuário modificações no código fonte através desta árvore construída e isto seria referenciado automaticamente no código fonte.

A proposta é criar ferramentas de análise estática para códigos fonte da linguagem Java para que possa-se apurar projetos pequenos e posteriormente em projetos *open-sources* para a verificação da existência de alguma defasagem [14] de estruturas entre qualquer versão da linguagem Java que fora concebido para a versão atual e estável na qual a linguagem se encontra. O desenvolvimento das ferramentas será com a versão mais atual da linguagem Java que neste momento é o Java versão 8 para verificar se os softwares desenvolvidos está versão nesta versão e como acompanharam a evolução de décadas de novas versões sem atualização do código por motivo de engenharia outro qualquer.

1.2 Objetivos Gerais

Este trabalho tem por objetivo geral analisar de forma estática projetos desenvolvidos em java para descobrir e entender se e como é a evolução da linguagem de programação o longo dos lançamentos das *releases* de um software, adotando ou ignorando as *features* mais recentes lançadas.

1.3 Objetivos Específicos

Criar um analisador estático de código para projetos java, com foco em projetos *open-source*, e através deste pesquisar através da infraestrutura de árvores sintáticas [13] e visitors [16] provida pela biblioteca *JDT* do eclipse e encontrar construções de código ultrapassadas e verificar se novas características veem sendo adotados pelo desenvolvedores ao longo do projeto.

1.4 Metodologia

Para a elaboração deste trabalho, fora realizado um estudo para saber como foi a adoção de novas características da linguagem java ao longo do lançamento destas para a comunidade de desenvolvedores.

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 *opensource*.

Em seguida foi estudada a melhor arquitetura para a elaboração do analisador estático proposto de modo que esta no tivesse um forte acoplamento entre os módulos necessários e facilitasse a pesquisa de outras características através da injeção do visitors [16] usando o *spring framework* [8]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

1.5 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 JDK 1.1 [7] 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 (APIs). 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* [6, 11], 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 APIs 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 [5] 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.

1.6 Aspectos evolutivos da liguagem Java

1.6.1 Java 2

A primeira versão do Java Security, disponível no JDK 1.1 [7], contém um subconjunto dessa funcionalidade, incluindo APIs 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:

```
import java.security.Signature;
import java.security.NoSuchAlgorithmException;

public class SignFile {
   Signature signature;

   private void init(String algorithm) throws NoSuchAlgorithmException{
      signature = Signature.getSignature(algorithm);
    }
}
```

- 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.

```
public class FixedStack {
   ...
    public java.util.Enumeration elements() {
        return new FixedStack$Enumerator(this);
    }
}

class FixedStack$Enumerator implements java.util.Enumeration {
   private FixedStack this$0;

   FixedStack$Enumerator(FixedStack this$0) {
      this.this$0 = this$0;
      this.count = this$0.top;
```

```
    }

   int count;
   public boolean hasMoreElements() {
      return count > 0;
   }

   public Object nextElement() {
      if (count == 0)
         throw new NoSuchElementException("FixedStack");

      return this$0.array[--count];
   }
}
```

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

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

- HelloImpl.java

```
package examples.hello;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello{
   private String name;

   public HelloImpl(String s) throws RemoteException {
      super();
      name = s;
   }

   public String sayHello() throws RemoteException {
      return "Hello World!";
   }

   public static void main(String args[]){

      System.setSecurityManager(new RMISecurityManager());

      try {
         HelloImpl obj = new HelloImpl("HelloServer");
         Naming.rebind("//myhost/HelloServer", obj);
         System.out.println("HelloServer bound in registry");
      } catch (Exception e) {
         System.out.println("HelloImpl err: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

1.6.2 Java 4

- *Assertion Facility* [3]. 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.

1.6.3 Java 5

- *Generics* [3, 6, 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:

```
static void expurgate(Collection c) {
   for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
         i.remove();
   }

static void expurgate(Collection<String> c) {
   for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
         i.remove();
}
```

- *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:

```
void cancelAll(Collection<TimerTask> c) {
   for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
      i.next().cancel();
}

void cancelAll(Collection<TimerTask> c) {
   for (TimerTask t : c)
      t.cancel();
}
```

- *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*

1.6.4 Java 6

Não ocorram mudanças ou introdução de novas estruturas na linguagem Java [3].

1.6.5 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 [5], 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 [4].

```
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* [6, 11, 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.

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
Map<String, List<String>> myMap = new HashMap<>();

List<String> list = new ArrayList<>();
list.add("A");

list.addAll(new ArrayList<>());

class MyClass<X> {
   <T> MyClass(T t) {
   ...
   }
}
```

1.6.6 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.

```
   List<String> stringList = new ArrayList<>();
   stringList.add("A");
   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. [5]

```
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));

   }
}
```

1.7 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 [19]. 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

Arquitetura

Como ambiente de desenvolvimento integrado (IDE) escolhido foi o eclipse devido ao favorecimento do desenvolvimento rápido com recursos que podem ser integrados através de *plugins*, além disso sua portabilidade para outras plataformas sem afetar os *plugins* adicionados favorecendo o desenvolvimento independente da plataforma escolhida pelo desenvolvedor.

O Eclipse Java *development tools* (JDT), fornece plugins que implementam a IDE eclipse servindo com apoio para o desenvolvimento de qualquer aplicativo Java, inclusive plugins para a própria IDE Eclipse.

São cinco os componentes que compõem o JDT, e cada componente pode operar com um projeto independente, que são:

- **APT** fornece plugins que adicionam e fornecem suporte ao processamento de anotações java.
- **Core** é o core da infraestrutura Java da IDE eclipse provendo compiladores, API's modelos definidas em árvores Java, documentação, assitente de código, suporte e formatação de código fonte.
- **Debug** componente de depuração da plataforma sendo definido independente da linguagem utilizada.
- **Text** fornece blocos básicos para editores de texto e textos dentro do eclipse e ainda contribui com o editor de texto padrão do eclipse.
- **UI** prove toda interface que necessite de iteração com o usuário final, e ainda fornece a manipulação e visualização do código Java na IDE.

2.1 Arquitetura

A arquitetura do analisador exibido na Figura: 2.1 foi implementado utilizando como base os componentes *JDT* porém de maneira independente da IDE eclipse o qual não é um *plugin* da IDE mas sim uma ferramente que pode ser utilizada como apoio em qualquer processo de desenvolvimento ou para levantamento de dados sobre a história evolutiva de um software.

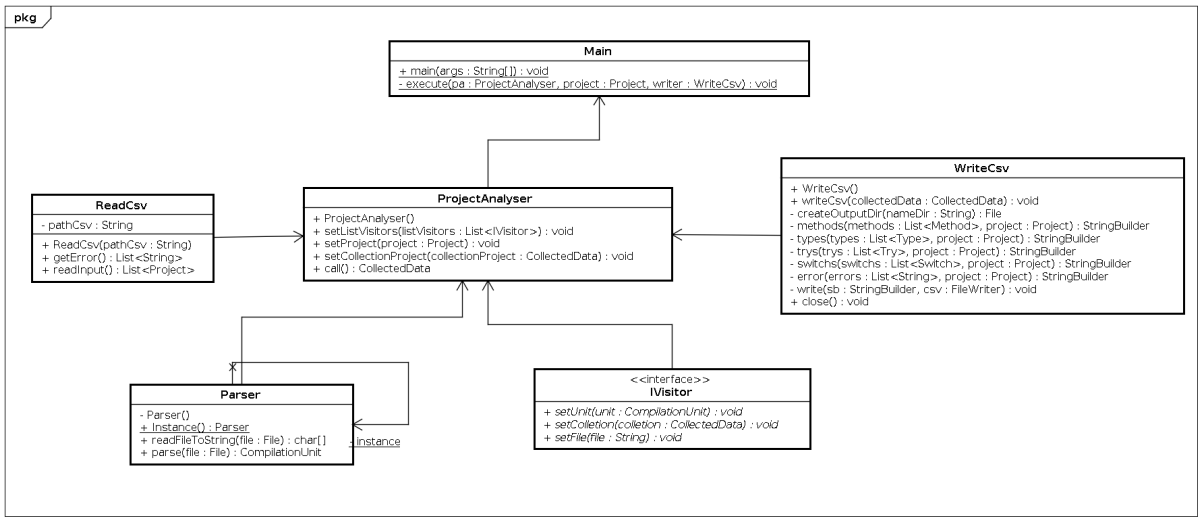

Figura 2.1: Arquitetura geral do software.

Este analisador tem com base para seu funcionamento o uso de *visitors* [16] exemplificado na Figura: 2.2 os quais possuem inteligência para verificar a adoção de códigos recentemente adicionados a novas *releases* da linguagem Java como *multicatchs* e até mesmo efetuar a pesquisa de código ultrapassados através de padrões pré-definidos para que possam vir a ser melhorados caso o desenvolvedor assim que julgue necessário conforme é o caso de vários *catchs* aninhados que pode ser um potencial caso de se tornar um bloco único de *multicatch* tornando o código mais elegante, atual e legível.

Tais analisador tem como entradas válidas um único arquivo arquivo separado por vírgula (CSV) que possui em seus campos o nome dos projeto, versões e caminho para que cada um seja analisado. Após o *input* é realizado uma pesquisa em seu diretório *src* o qual contém os códigos Java que compõem o projeto, após todos os arquivos listas é realizado um parse para a construção de árvores de sintáxe abstratas (AST) uma por arquivo e dai é lançando os *visitors* [16] com suas respectivas inteligências para pesquisar os padrões previamente definidos os quais são armazenados em uma única coleção de dados que gera um CSV para cada padrão designado e dentro informando onde e em qual versão do projeto fora encontrado.

2.1.1 Visitors

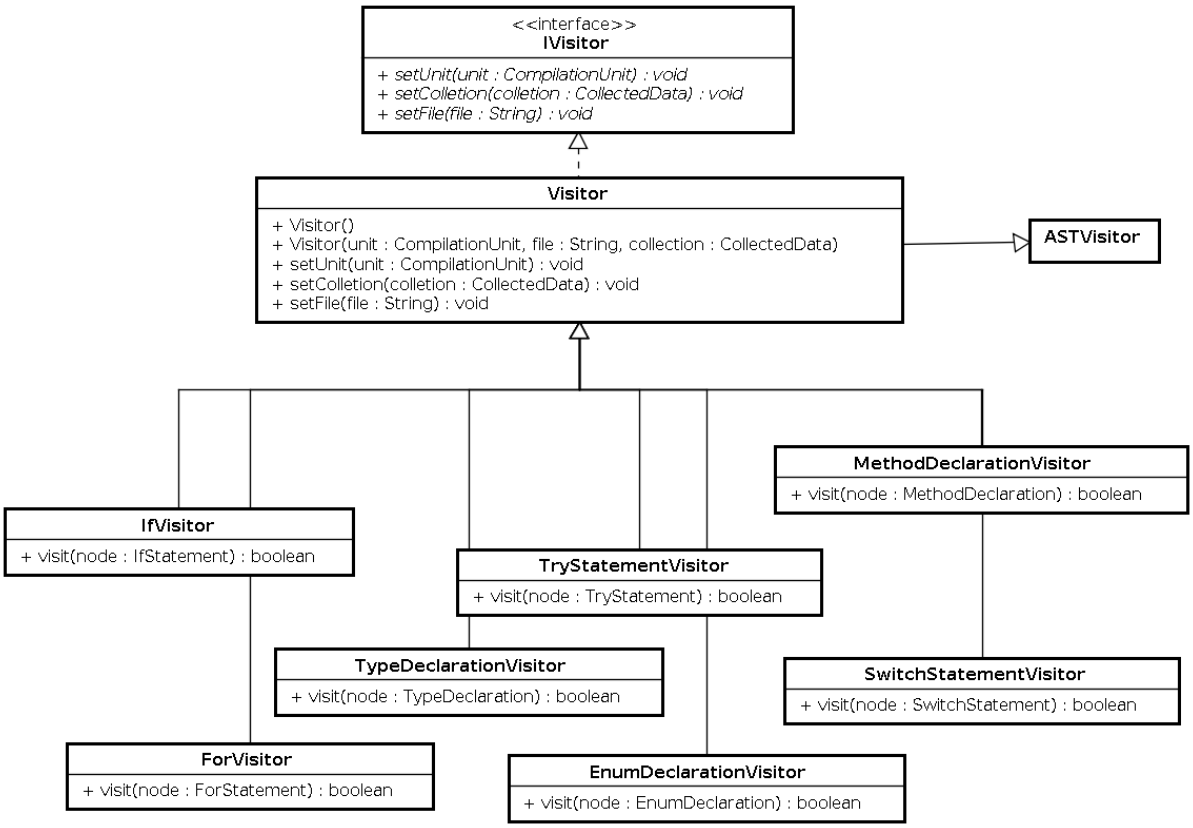

Figura 2.2: Organização dos Visitors.

Todos os visitors criados para o projeto extendem da classe *Visitor.java* a qual implementa a interface *IVisitor.java* conforme a Figura: 2.3, vale resaltar que a classe *Visitor.java* extende por sua vez de ASTVisitor a qual é fornecida pelo Core do JDT eclipse tendo como método principal utilizado neste projeto o *boolean visit(Statement node)* o qual recebe como parâmento qualquer *Statement* descrito na lib JDT, este método é sobrescrito com todo conteúdo necessário para recolher as métricas de acordo com os padrões previamente estabelecidos.

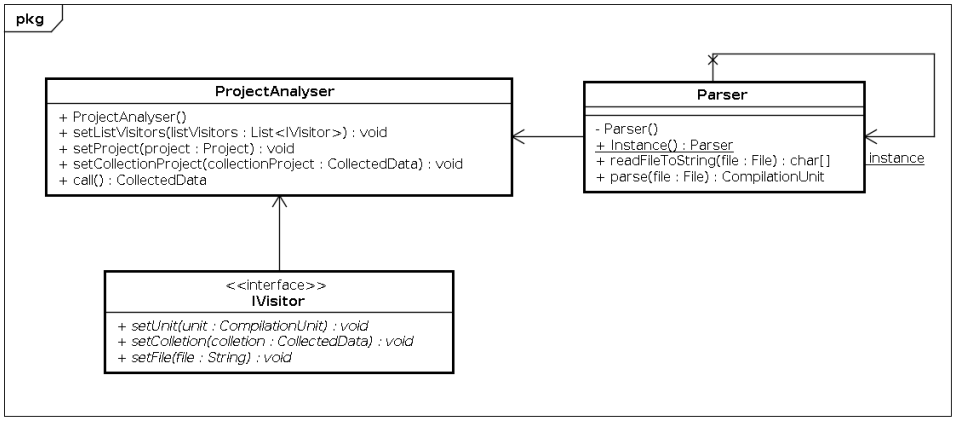

Figura 2.3: Project analyser.

Este diagrama exibe o coração da aplicação pois é aqui que ocorre a transformação de todo código Java que compõe o projete em árvore sintática para que os *visitors* [16] possam pesquisar em seus nós pelos padrões estabelecidos.

Um ponto interessante é que o *ProjectAnalyser.java* não tem a responsabilidade de instanciar a lista de IVisitor a qual possui referência pois estas são injetadas usando o padrão de projeto injeção de dependência**(DI)** o qual aqui faz-se presente através do *framework spring* que injeta uma lista de *bean* onde cada *bean* representa cada classe de extendida de *Visitor* descrita anteriormente.

2.1.2 Inversão de Controle e Injeção de Dependência

O padrão de projeto inversão de controle **(IoC)** e injeção de dependência **(DI)** é utilizado quando deseja-se obter um baixo nível de acoplamento entre módulos que compõem um sistema tornando mais suave ou até mesmo removendo o acoplamento entres módulos para que seja mais fácil evoluir e manter o software. Desta forma a injeção faz-se de maneira configurável através de um arquivo *XML* ou até mesmo uma classe *Java*. Para tal função usaremos o *framework Spring* devido sua consolidação e popularidade no assunto.

Para preparar o ambiente de acordo com as especificações do *Spring* é necessário definir o contexto da aplicação e para isso é faz-se necessário criar antes de injetar as dependências as configurações iniciais. Dentre as possíveis formas disponibilizadas pelo *framework* de conceder tal configuração para criação um contexto para aplicação *ApplicationContext* o projeto segue com a criação de um arquivo *XML* o que concentra as referências necessárias de quais objetos e onde estes serão injetados. O arquivo *XML* criado para tal finalidade é o *'Beans.xml'* exemplificado na figura 2.4 . Atualmente na versão 4.1.6 do *framework spring* é possível utilizar uma classe para fazer o mesmo trabalho do arquivo *XML*.

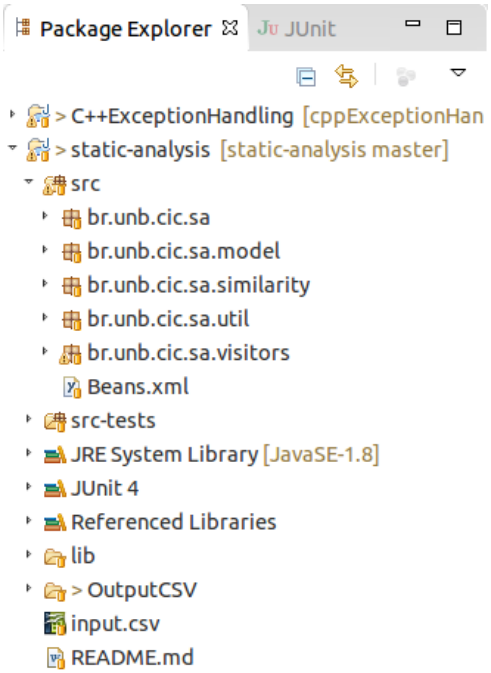

Figura 2.4: Localização Beans.xml.

Configuração necessária para o funcionamento da injeção de dependências com *Spring* conforme explicado na documentação do *framework* conforme [8].

```
<?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   </beans>
```

O *framework* trabalha com *beans* onde cada *bean* representa uma classe Java a ser injetada vale ressaltar que essas por simplicidade devem ter o construtor sem parâmetros para que o trabalho possa ocorrer da maneira mais simples, caso seja necessário parâmetros estes sejam passados vias os métodos Sets.

Identificação de cada bean no arquivo de configuração *XML*, é atributo **id** contido na *tag* **bean** onde cada deve ser único para que a gerência das dependências ocorra da forma preconizada pelo *framework*.

```
   <bean id="tsVisitor" class="br.unb.cic.sa.visitors.TryStatementVisitor"/>
   <bean id="mdVisitor"
       class="br.unb.cic.sa.visitors.MethodDeclarationVisitor"/>
   <bean id="typeVisitor"
       class="br.unb.cic.sa.visitors.TypeDeclarationVisitor"/>
   <bean id="swVisitor" class="br.unb.cic.sa.visitors.SwitchStatementVisitor"/>
   <bean id="enVisitor" class="br.unb.cic.sa.visitors.EnumDeclarationVisitor"/>
   <bean id="ifVisitor" class="br.unb.cic.sa.visitors.IfVisitor"/>
   <bean id="forVisitor" class="br.unb.cic.sa.visitors.ForVisitor"/>
   <bean id="collection" class="br.unb.cic.sa.model.CollectedData"/>
```

Todos os *Beans* são injetados na classe *ProjectAnalyser.java* através de uma lista de IVisitor. E desta forma indicamos no *XML* através da tag <**property name="listVisitors»** onde o atributo **name** deve conter o mesmo nome que o atributo na classe Java, neste caso o algo é **listVisitors** que existe uma lista de IVisitor com este mesmo nome dentro da classe *ProjectAnalyser.java*.

```
<bean id="pa" class="br.unb.cic.sa.ProjectAnalyser">
   <property name="listVisitors">
      <list>
         <ref bean="forVisitor"/>
         <ref bean="ifVisitor"/>
         <ref bean="tsVisitor"/>
         <ref bean="mdVisitor"/>
         <ref bean="typeVisitor"/>
         <ref bean="swVisitor"/>
         <ref bean="enVisitor"/>
      </list>
   </property>

</bean>
```

Para concluir com sucesso a injeção faz-se necessário somente indicar em qual classe e em qual local será injetada tal dependêcia, nesse caso a classe é a *ProjectAnalyser.java* e o local é definido pela anotação ***@Autowired*** logo acima do atributo o qual será injetado.

```
public class ProjectAnalyser... {

   @Autowired
   private List<IVisitor> listVisitors;
   .
   .
   .
}
```

Para finalizar todo ambiente de configuração de injeção de dependência é necessário somente criar uma classe que seja um *Singleton* [16] conforme cita Gamma e amigos e faz-se necessária conforme indica a documentação [8], para ter o controle de que exista somente um único ambiente de injeção.
Tal padrão que controla a quantidade de instâncias dos objetos se faz presente através da classe *CDI.java* qual

```
public class CDI {

   private static CDI instance;
   private ApplicationContext ctx;

   private CDI(){
      ctx = new ClassPathXmlApplicationContext("Beans.xml");
   }

   public static CDI Instance(){
      if(instance == null)
         instance = new CDI();

      return instance;
   }

   public ApplicationContext getContextCdi(){
      return ctx;
   }
}
```

Abaixo segue bibliotecas necessárias para que o ambiente de injeção de dependência utilizando o textitspring framework [8] ocorra de maneira correta na linguagem *Java*, estas devem ser adicionada como dependências do Maven conforme ilustas 2.5.

- spring-aop-4.1.6.RELEASE.jar
- spring-beans-4.1.6.RELEASE.jar
- spring-context-4.1.6.RELEASE.jar
- spring-core-4.1.6.RELEASE.jar
- spring-expression-4.1.6.RELEASE.jar

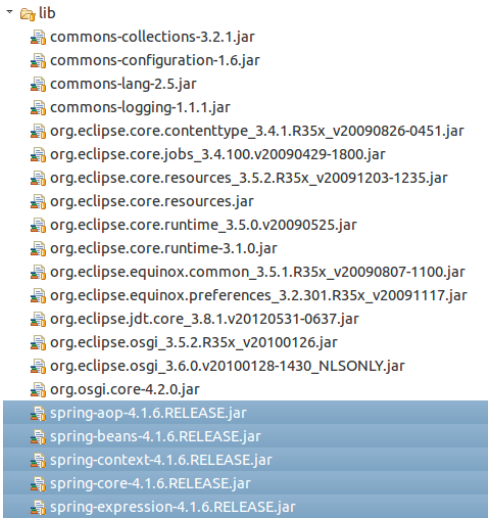

Figura 2.5: Libs Spring necessárias para CDI.

2.1.3 Apache Maven

Afim de gerenciar *builds* deste analisador estático, o Maven foi introduzido para tornar as configurações uniformes no ambiente de desenvolvimento deste seguindo as boas práticas, compilação, gerência de dependências e distribuição da aplicação.

O arquivo pom.xml segundo a documentação [2], *project object model* é a essência de um projeto Maven conforme Figura: 2.6, com poucas configurações é possível gerências as dependências, centralizando documentação do projeto e a compilação de *builds* para a distribuição da aplicação (por exemplo, *.jar, .war ou .ear*).

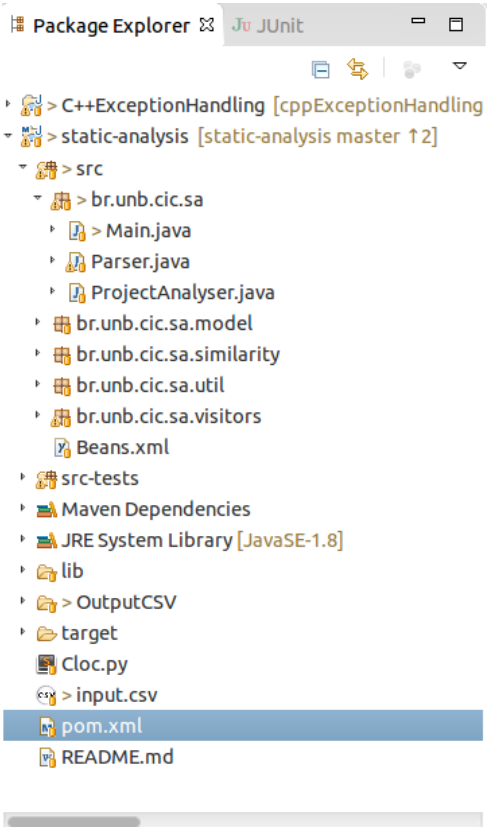

Figura 2.6: Pom.xml, arquivo de configuração do Maven.

Tabela 2.1: Dependências gerenciadas através do Maven

Dependência	**Versão**
junit	4.12
commons-collections	3.2.1
commons-configuration	1.6
commons-lang	2.5
ommons-logging	1.1.1
org.eclipse.core.contenttype	3.4.100.v20100505-1235
org.eclipse.core.jobs	3.5.100.v20110404
org.eclipse.core.resources	3.7.100.v20110510-0712
org.eclipse.core.runtime	3.1.200-v20070502
org.eclipse.equinox.common	3.6.0.v20100503
org.eclipse.equinox.preferences	3.4.0.v20110502
org.eclipse.jdt.core	3.10.0.v20140604-1726
org.eclipse.osgi	3.7.1
org.osgi.core	6.0.0
spring-aop	4.1.6.RELEASE
spring-beans	4.1.6.RELEASE
spring-context	4.1.6.RELEASE
spring-context	4.1.6.RELEASE
spring-expression	4.1.6.RELEASE

Capítulo 3

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 [9]. 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 e amigos em [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.

3.1 Análise léxica

Ferramentas que operam em código-fonte conforme [21] 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.

3.2 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.

```
stmt := if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval
assign_stmt := lval EQUAL expr SEMI
lval = ID | arr_access
arr_access := ID arr_index+
arr_idx := LBRACKET expr RBRACKET
```

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: 3.1 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.

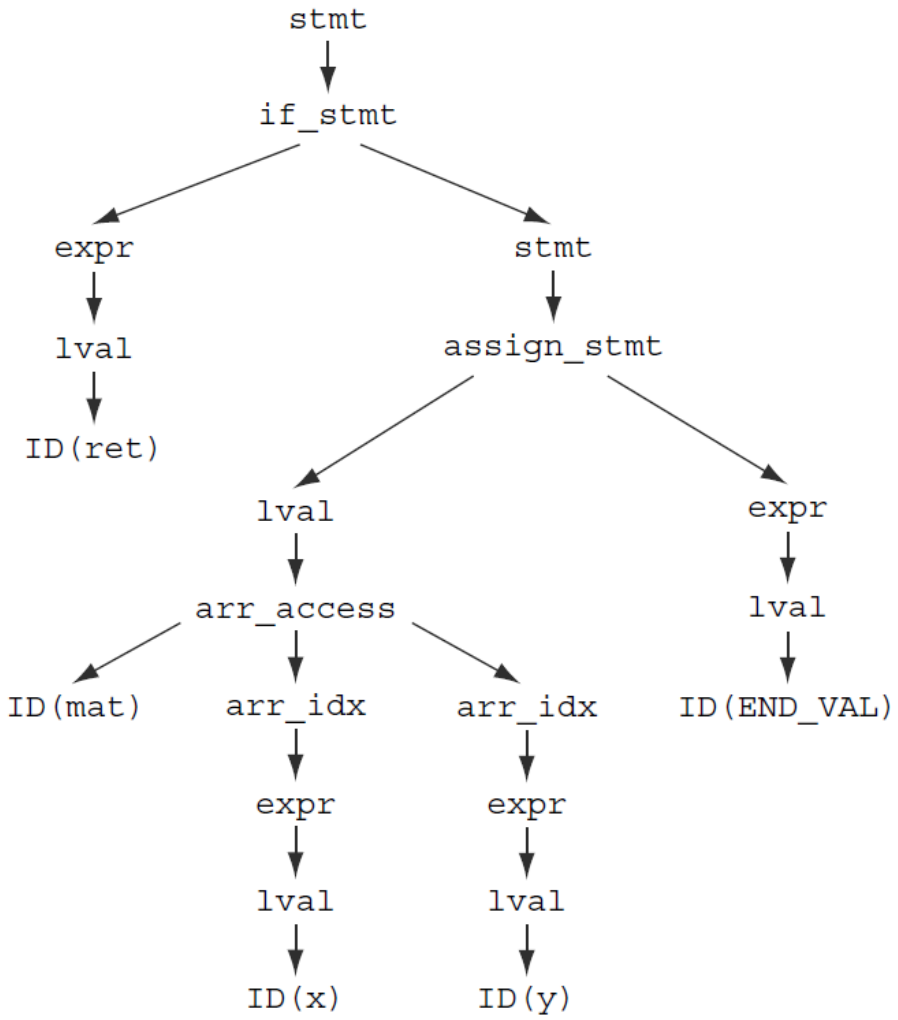

Figura 3.1: Árvore de parser.

3.2.1 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 árvore sintática de cada arquivo. O parse é gerado conforme as últimas definições da linguagem utilizando *AST.JLS8*.

```
ASTParser parser = ASTParser.newParser(AST.JLS8);

Map<String, String> options = JavaCore.getOptions();
options.put(JavaCore.COMPILER_COMPLIANCE, JavaCore.VERSION_1_8);
options.put(JavaCore.COMPILER_CODEGEN_TARGET_PLATFORM,
    JavaCore.VERSION_1_8);
options.put(JavaCore.COMPILER_SOURCE, JavaCore.VERSION_1_8);

parser.setKind(ASTParser.K_COMPILATION_UNIT);
parser.setCompilerOptions(options);
parser.setSource(contents);

final CompilationUnit cu = (CompilationUnit) parser.createAST(null);
return cu;
```

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].

3.3 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: 3.2 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.

3.4 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.

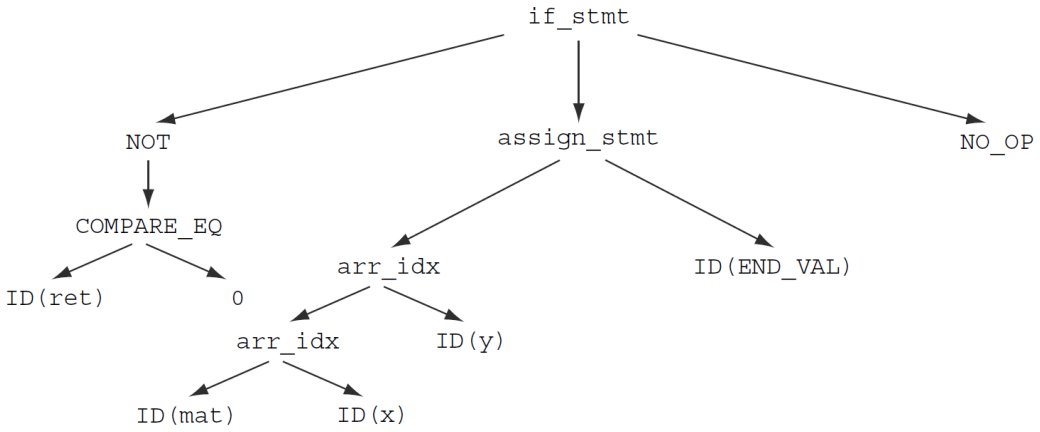

Figura 3.2: Árvore AST.

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.

3.5 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.

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

3.6 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.

3.7 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.

3.8 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.

3.9 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 4

Resultados

Foram analisados através de mais de 67.648.820 de **LOC** nos projetos Ant, Checkstyle, CommonsCollections, FindBugs, FreeMind, Hibernate, JBoss, Jetty, Log4j, Spring, SquirrelSql, Weka, Xerces.

4.1 LambdaExpression

A maior mudança ocorrida em Java 8 foi a introdução de expressões lambda, que tem por definição prover um bloco de código limpo e conciso para representar um interface usando uma simples expressão. Também melhoram a manipulação de *collections* tornando fácil a iteração através de filtros para extração de dados e adiciona novas características de concorrência que aumenta a performance em ambientes *multicores*.

Anonymous inner classes foram projetada para facilitar o desenvolvedor a tratar seus dados de forma fácil, mas isso não é tão simples como deveria ser, além de suas implementações serem usadas somente em local específico da aplicação, em boa parte de seu uso para tratar eventos decorrentes do usuário ou de algum processamento específico. Expressões Lambda podem vir a substituir *Anonymous inner classes* o que acarreta em uma redução na quantidade de linhas de código significativamente além de reduzir os tipos utilizados no sistema.

Dentre os projetos analisados foram encontrados ocorrências de tal característica, *Lambda Expression*, somente nos projetos *Checkstyle*, *Hibernate*, *Jetty Spring*, totalizando 774 ocorrências dentre um total de 64.179.440 **LOC**.

Tendo em vista que expressões lambdas podem vir a substituir *Anonymous inner classes* e *Types* dos projetos segundo [20]. Foram encontradas nos projetos *CheckStyle*, *Hibernate*, *Jetty* e *Spring*, 2201, 13654, 10016 e 113069 ocorrências respectivamente totalizando em 138940 ocorrências de *Anonymous inner classes*. Com uso de expressões lambda a quantidade de ocorrências de *Anonymous inner classes* e *Types* teria a tendência natural de diminuir seu uso.

Um uso corriqueiro de expressão lambda seria para substituir o seguinte bloco do projeto Checkstyle:

```
//Sem uso de Lambda Expression
button.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
      System.out.println("button clicked");
   }
});

//Usando Lambda Expression
button.addActionListener(event -> System.out.println("button clicked"));

//#############################################################################

TreeSet<String> paths = new TreeSet<String>(
      new Comparator<String>({
         public int compare(String o1, String o2){
            int paths1 = new StringTokenizer(o1,"/",false).countTokens();
            int paths2 = new StringTokenizer(o2,"/",false).countTokens();

            if (paths1 == paths2){
               return o1.compareTo(o2);
            }

            return paths2 - paths1;
         }
      }
);


ps = paths.stream().filter(p -> p.o1 != p.o2)
              .collect(Collectors.toSet());

return new TreeSet(ps);
```

Como este simples exemplo pode-se verificar do benefício das expressões lambda tendo em vista que redução do uso de *Anonymous inner classes* além de ser uma forma mais atual. Neste seria um redução de 80% sobre o código utilizado anteriormente. Além da redução de tipos 3 tipos(*ActionListener*, *ActionEvent* e *void*) para um único método anônimo que acarreta em 300% na redução de tipos, se fosse possível aplicar esta características em todas as 138940 ocorrências de *Anonymous inner classes* seria uma redução significativa de **LOC**.

Entretanto diante dos dados coletados é possível afirmar que Expressões Lambda não estão sendo adotadas com este intuito, o que pode leva a crer que esta característica por ser recente ainda pode não ter sido bem acolhida pela comunidade de desenvolvedores para tirar proveito desta característica para benefício de seus projetos.

4.2 MultiCatch

Conforme menciando anteriormente Java 7, introduziu suporte a *MultiCatch* que trouxe a possibilidade de trazer clareza e simplicidade no mecanismo de tratamento de exceção. O que aparentemente esta sendo um recurso pouco utilizado tendo em vista a grande quantidade de oportunidades de utilização desta elegante característica.

Foram encotrados um total de 5300 *trys* que possuiam mais de um bloco mais de um *catch*. O que acarretou em 46926 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.

Um simples *refactoring* unindo estes blocos semelhantes por igualdade acarretaria em redução de 32639 LOC o que gera uma redução de código duplicado na ordem de 75.95% onde todos os *catchs* após tal ação teriam um total 14287 LOC que torna essa *feature* muito relevante na rescrita de um software além de utilizar o novo estilo de programação proposto nesta versão da linguagem.

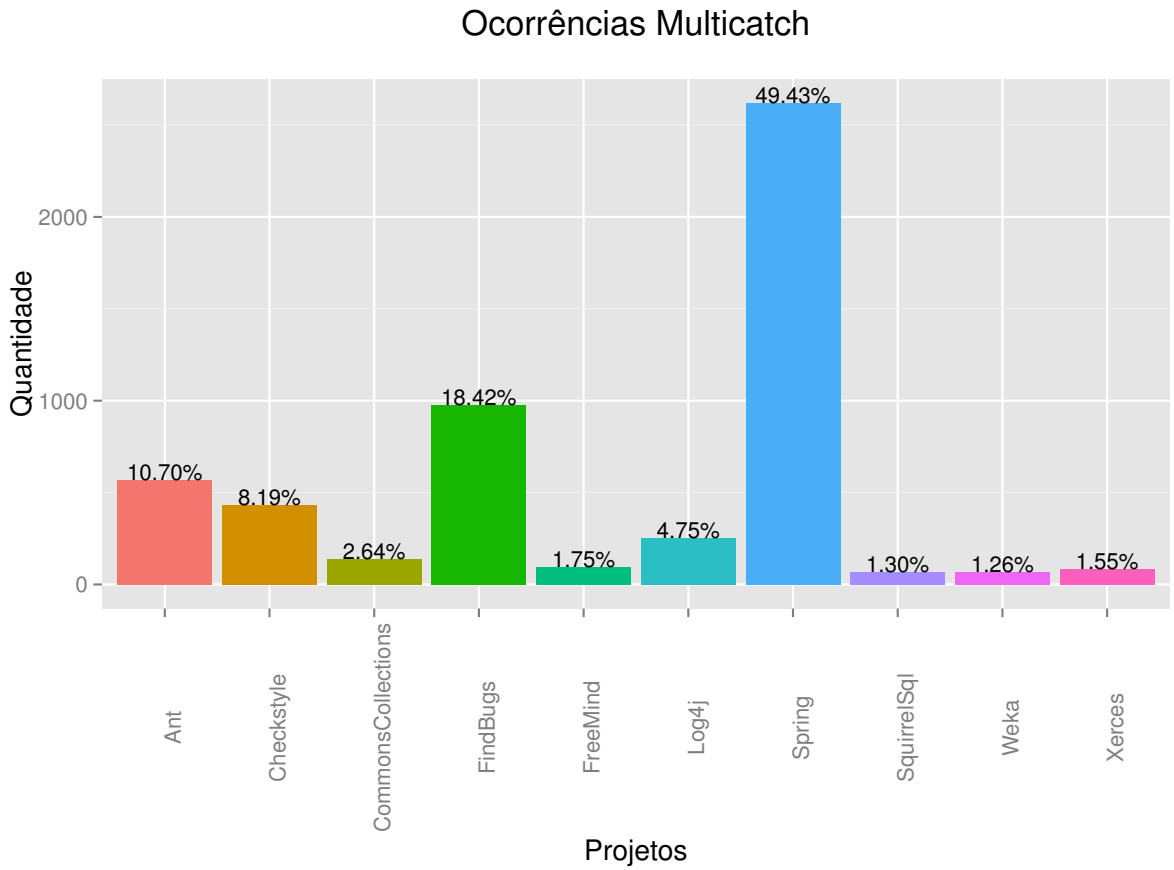

Figura 4.1: Oportunidades de *Multicatch* nos projetos.

De forma mais detalhada uma análise sobre uma classe do *Spring 4.2.0.RC2* como exemplo é *org.springframework.beans.AbstractNestablePropertyAccessor.java*, o qual é possível verificar que tal característica é simples e rápida de ser implantada o que impacta em uma redução significativa do trecho de código mencionado algo entorno de 58%.

```
   // Sem uso de Multicacth 17LOC
   try {
      return this.typeConverterDelegate.convertIfNecessary(propertyName,
          oldValue, newValue, requiredType, td);
   }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);
   }

// Com uso de Multicatch 10LOC
   try {
      return this.typeConverterDelegate.convertIfNecessary(propertyName,
          oldValue, newValue, requiredType, td);
   }catch (ConverterNotFoundException ex | IllegalStateException ex |
       ConversionException ex | IllegalArgumentException ex) {
      PropertyChangeEvent pce =
         new PropertyChangeEvent(this.rootObject, this.nestedPath +
             propertyName, oldValue, newValue);

      if(ex instanceof ConversionException || ex instanceof
          IllegalArgumentException){
         throw new TypeMismatchException(pce, requiredType, ex);
      }else{
         throw new ConversionNotSupportedException(pce, td.getType(), ex);
      }
   }
```

Uma análise mais detalhada sobre *Spring* com 2620 ocorrências o que resulta em afima que tal procedimento de tratar exceções estão presentes em todas as versões analisadas, iniciando na 3.0.0.M1 lançada em junho de 2009 até a mais recente até o momento 4.2.0.RC2. Para uma análise mais criteriosa será elaborada um detalhamento a partir da versão 4.0.0.M1 até a 4.2.0.RC2 totalizando 927 oportunidades de *multicatch* 35% das ocorrências no *Spring*.

Conforme exibido na figura: 4.2, obtem-se antes de um *refactoring* uma média de 371 LOC e após uma proposta concisa de *refactoring* é possível obter uma média de 106 LOC o que proporciona uma redução significativa de 71,46%.

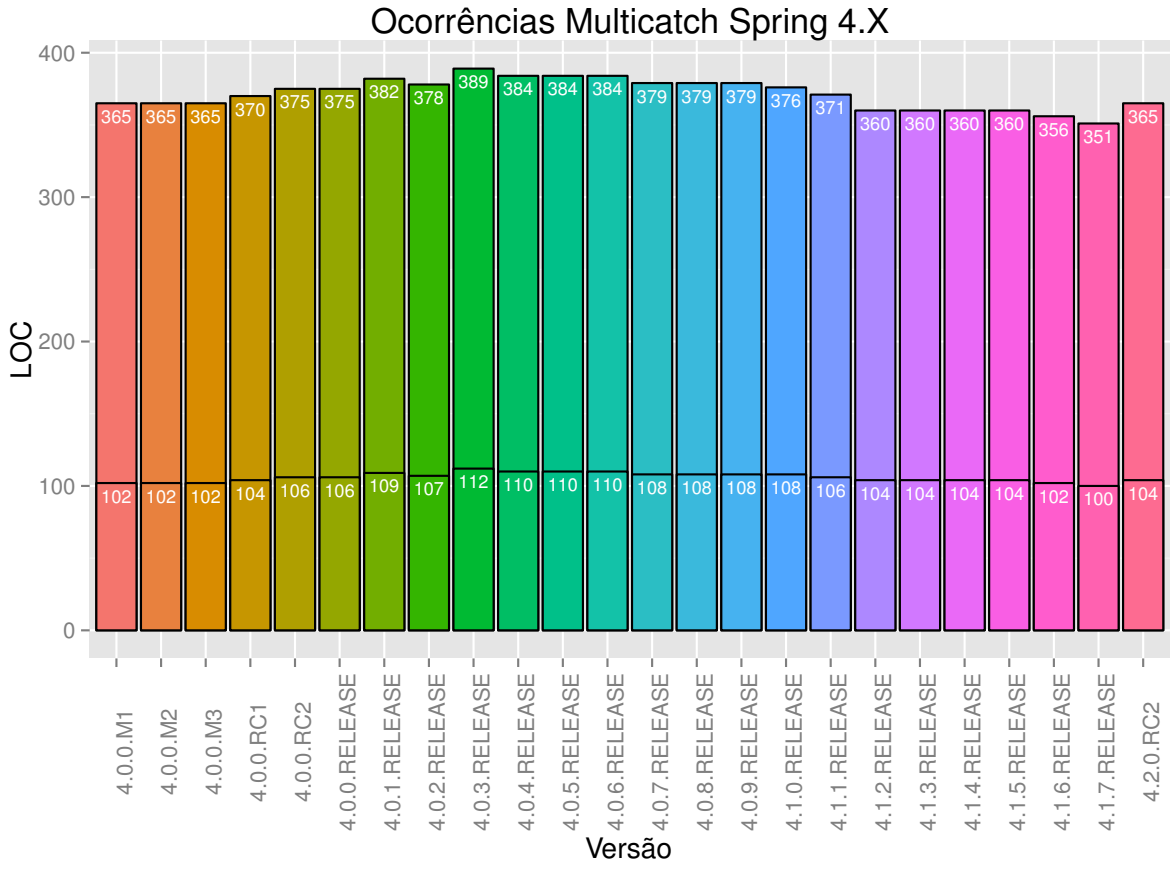

Figura 4.2: Ocorrências de *Multicatch* Spring 4.X.

Dada uma redução significativa 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.

4.3 ANT

Até a última versão deste projeto [1], 1.9.5, não foram encontradas utilização métodos com *vargs*, expressões lambdas, *switch* com *strings* e nem *try* com *resources*.

Este projeto faz um bom uso de tratamento de exceções sendo encontrado em toda história de desenvolvimento foram produzidas 28 versões deste e com um total de 34722 blocos *trys*, onde em média foram encontradas 1240 destes blocos por versão. E deste total pode-se verificar um total de 513 ocorrências de blocos *trys* com *catchs* iguais totalizando em 1,5% de código repetido neste quesito conforme ilustra Figura: 4.3.

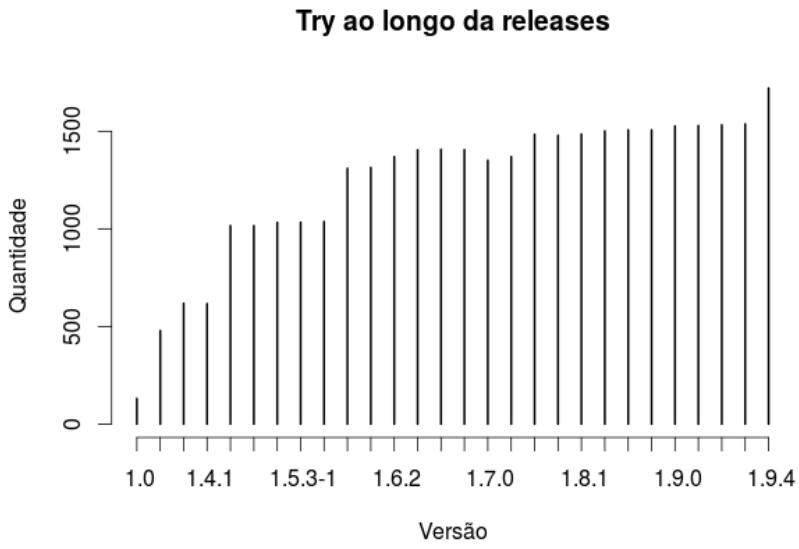

Figura 4.3: Tratamento de exceção ao longo das releases.

Entretanto pode-se constatar conforme ilustrado na Figura: 4.4 que em todas as versões do projeto *ANT* possui o tratamento de exceção como blocos *catchs* iguais sendo contabilizado um total de 513 ocorrências e dando atenção especial entre as versões 1.9.0 e 1.9.5. Entretanto a partir da versão 1.9.0 por volta de 2012, java possuía o mecanismo de *multicatch* que fora lançado por volta de 2011 em java 7. Entre as *releases* desta versão foram encontradas em cada um dos 5 lançamentos do *ANT* por volta de 27 ocorrências iguais de *catchs* e acarreta em um total de 135 blocos repetidos. Caso fosse adotado *multicatch* seria reduzido somente a 5 blocos a cada versão existente o que seria uma redução de código repetido em aproximadamente 18%, e isso acarretaria em um código mais atual e elegante.

Outro fato de bastente relevância é que o ANT faz uso de atributos parametrizados indicados na Figura: 4.5 e métodos parametrizados conforme Figura: 4.6 desde sua versão 1.7.0, dezembro de 2006, o que leva a crer que foi aderido juntamente com o lançamento de *Generics*, o que foi um marco na linguagem Java.

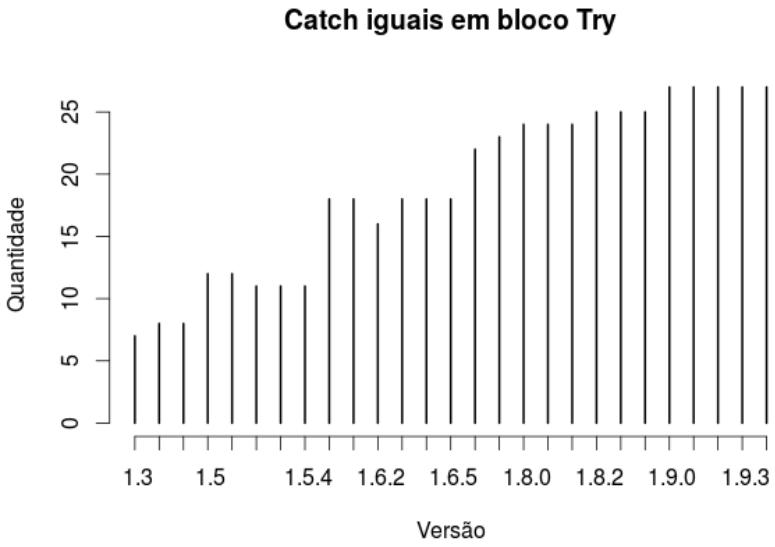

Figura 4.4: Bloco Try com catchs iguais ao longo das releases.

Vale ressaltar que de um total de 244137 atributos foi encontrado 1408 destes sendo paramentrizados o que acarreta em menos de 1% dos atributos são genéricos. E a respeito dos métodos foram encontrados um total de 282216 métodos sendo que deste somente 1080 são métodos parametrizados acarretando em menos de 1% são parametrizados.

O que leva a concluir que apesar do ANT fazer uso de tipos genéricos estes podem estar sendo subutilizados nesse projeto, ou esta característia não é de grande relevância para o projeto.

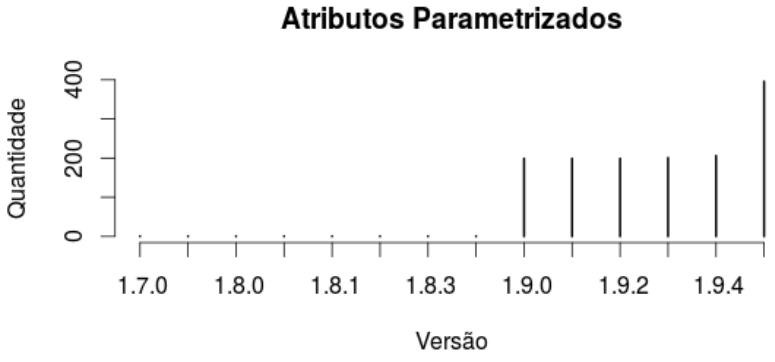

Figura 4.5: Atributos parametrizados ao longo das releases.

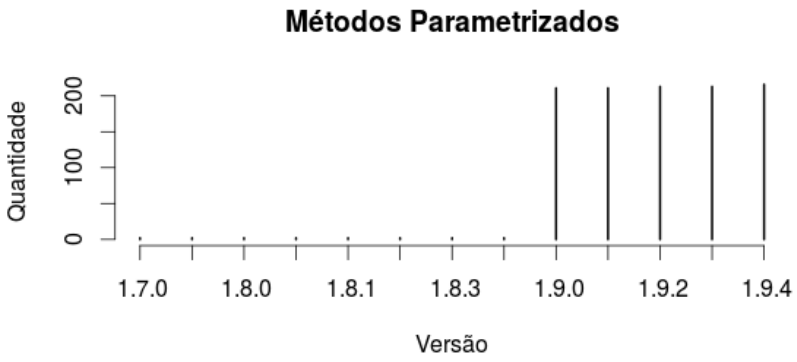

Figura 4.6: Métodos parametrizados ao longo das releases.

Referências

[1] Apache ant project @ONLINE. http://ant.apache.org/. Accessed: 2015-07-06. 43

[2] Documentation apache maven project @ONLINE. http://maven.apache.org/guides/. Accessed: 2015-07-06. 28

[3] Enhancements in java se 8 @ONLINE. http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html. accessed: 2015-05-27. 12, 13

[4] 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. 14

[5] Java se 7 features and enhancements @ONLINE. http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html. Acessed: 2015-05-20. 7, 13, 15

[6] The java tutorials @ONLINE. http://docs.oracle.com/javase/tutorial/extra/generics/index.html. Acessed: 2015-05-20. 6, 12, 15

[7] Jdk 1.1 new feature summary @ONLINE. http://www.public.iastate.edu/~java/docs/relnotes/features.html. Acessed: 2015-05-20. 4, 9

[8] Spring framework reference documentation @ONLINE. http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/. Accessed: 2015-06-06. 3, 24, 26

[9] 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. 30

[10] Gilad Bracha, Martin Odersky, and David Stoutamire. Gj: Extending the javatm programming language with type parameters. 9

[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. 6, 15

[12] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007. 31

[13] Noam Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956. 2, 3

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

[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. 2

[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. 1, 3, 20, 22, 26, 30, 33

[17] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: Refactoring tools allow programming languages to evolve. *SIGPLAN Not.*, 44(10):493–502, October 2009. 1, 2, 17

[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. 12, 15

[19] Max Schaefer and Oege de Moor. Specifying and implementing refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010. 2, 18

[20] RICHARD WARBURTON. *Java 8 Lambdas Functional Programming For The Masses*. O'Reilly Media; 1 edition (April 7, 2014), USA, 2014. 38

[21] 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. 31