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.
67 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 Java Development Tools

ACDP Java Platform Debugger Architecture

JCP Java Community Process

EFL Enhanced for loop

AIC Annonymous Inner Class

DI Dependency Injection

IoC Inversion of Control

CSV Comma separated values

Capítulo 1

Introdução

1.1 Introdução

Uma premissa na Engenharia de Software é a *natureza evolutiva* do software, e, com isso, custos significativos são relacionados com as atividades de manutenção. De forma semelhante, as linguagens de programação evoluem, com o intuito de se adaptarem as novas demandas e trazerem benefícios relacionados a produtividade e a melhoria da qualidade dos softwares construídos. Entretanto, um desafio inerente é a evolução de sistemas existentes em direção a adoção de novas construções disponibilizadas nas linguagens [9]. Conforme explicado por Jeffrey L. Overbey e Ralph E. Johnson. [13], 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 [4, 6, 9, 10, 13, 14, 16]. 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 [3, 5, 8, 13, 15–17, 17, 18, 20].

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 [11] usando o *spring framework* [2]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

1.4 Problema a ser Atacado

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

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

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

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

Um bom exemplo a ser lembrado é FORTRAN quando adicionou orientação objetos em sua na sua versão do ano de 2003 forçando a evoulução de seus compiladores os quais não forneciam mais suporte a versões anteriores conforme relata Jeffrey L. Overbey e Ralph E. Johnson em [13], 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 [17]. Modernas IDEs como ecplise realizam complexos refectoring através da técnica de *microrefectoring* que nada mais é que a divisão de um bloco de código complexo em pequenas partes para tentar encontrar códigos mais intuitivos a serem modificados.

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

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

Capítulo 2

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 [3]. 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. [11] 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 [20] 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 [7] 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

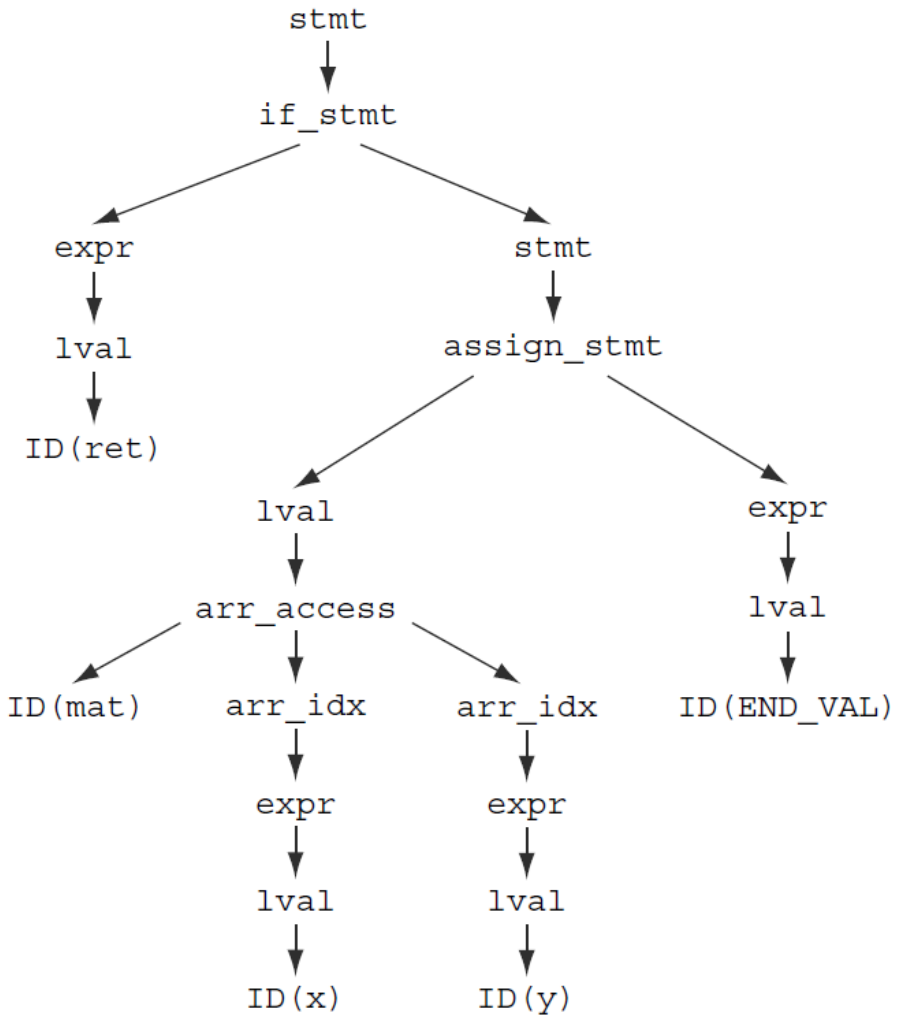

Figura 2.1: Árvore de parser.

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* [11].

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.

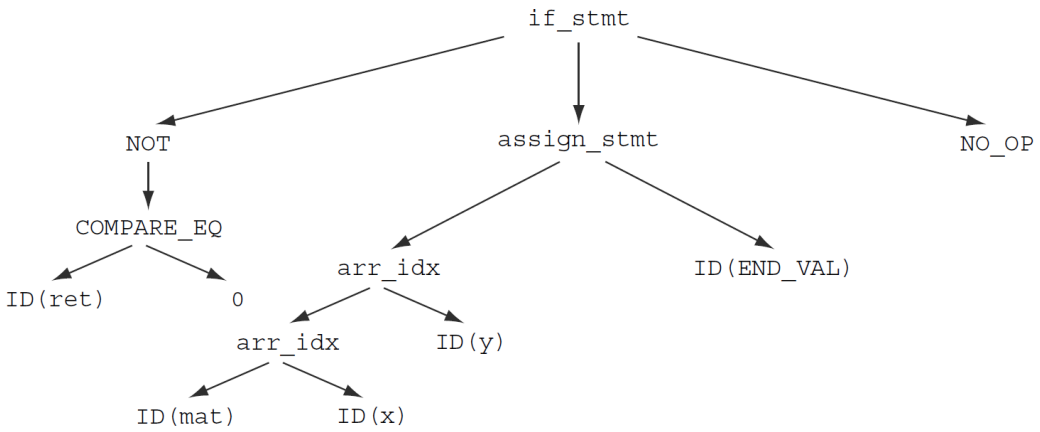

Figura 2.2: Árvore AST.

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

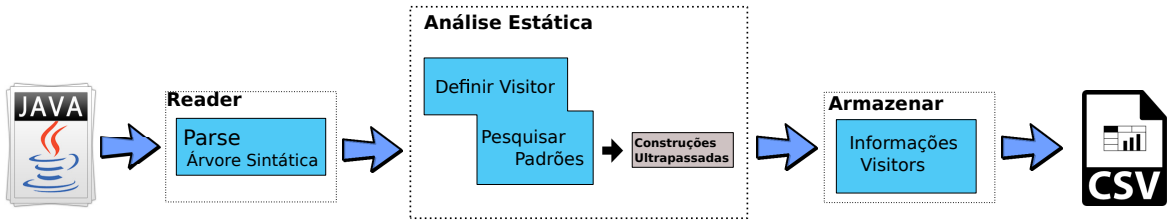

Figura 3.1: Alto nível de funcionamento do analisador estático.

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* [11] que possibilitam a análise estática de código fonte.

Um *visitor* é um padrão de projetos proposto por Eric Gamma [11], este padrão de projeto de característica comportamental que representa uma operação a ser realizada

sobre elementos de uma estrutura de um objetos. Neste caso operação a ser realizadas é visitar os nós de uma árvore sintática de um código fonte Java. Um *visitor* permite que uma nova operação seja criada sem que os elementos operados sofram alterações. Com isso é trivial adicionar novas funcionalidades em um *visitor* existente ou criar um novo.

A biblioteca Eclipse JDT não fornece mecanismo para extração de dados, entretanto foi acoplado um conjunto de classe visando obter a maior facilidade e flexibilidade na geração dos relatórios com os dados contidos na análise estática realizada. Essa flexibilidade foi alcançada com a utilização de introspecção de código que em Java é conhecido como *reflection*.

3.1 Entrada de dados

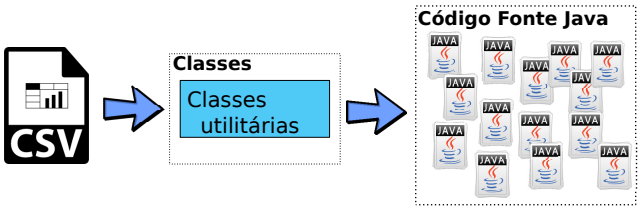

Figura 3.2: Input para funcionamento do analisador estático.

A entrada do analisador estático é iniciado com um arquivo CSV que contém nome dos projetos, caminho e quantidade de linhas de código conforme demonstrado na Figura: 3.2. As informações contidas no arquivo CSV são extraídas por um conjunto de classes utilitárias que varrem os diretórios de um determinado projeto pesquisando por todos os arquivos fonte Java. Os códigos fontes Java encontrados são a entrada descrita no funcionamento em alto nível da Figura: 3.1. Onde para cada projeto é feito a varredura de arquivos Java, gerado o *Parser* dos códigos Java e exportar os relatórios.

3.2 Análise Estática

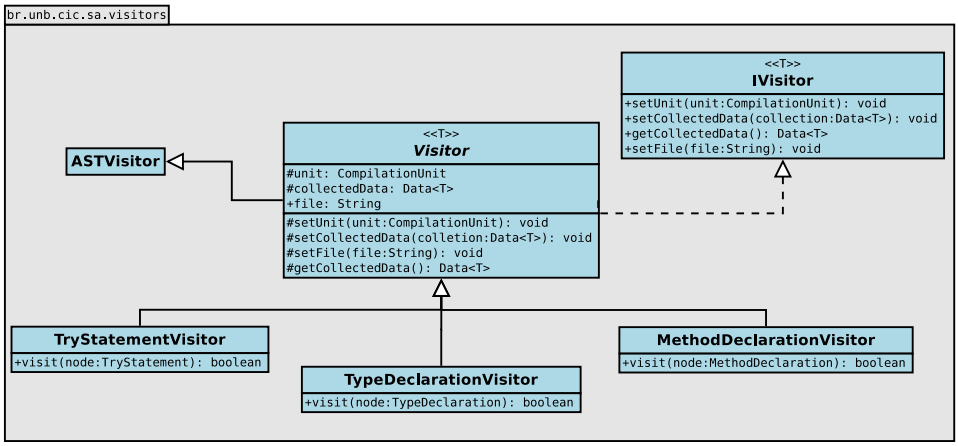

Figura 3.3: Diagrama Visitor.

Após o *Parser* é feita a coleta de dados utilizando uma infraestrutura de *Visitors*. Onde para este projeto uma classe *Visitor* é parametrizada em um tipo **T**, ***Visitor<T>***, onde este tipo **T** é a classe criada que vai armazenar as informações coletadas, conforme o diagrama da Figura: 3.3. O parâmetro <**T**> descrito faz referência a uma classe modelo composta por *getters* e *setters* para modelar os dados extraídos e que sejam armazenados corretamente.

Confome exemplificado na Figura: 3.3, com a necessidade de criar um *Visitor* que detecte e colete informações dos tipos declarados no sistema, basta criar uma classes modelo *TypeDeclaration.java* e setar o parâmetro <**T**> como <*TypeDeclaration*>, com isso os dados serão extraídas pelo *Visitor, TypeDeclarationVisitor.java*, que identifica as informações pertinentes.

A Tabela: 3.1 exibe todos os *Visitors* criados neste projeto com sua respectiva descrição.

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.

3.3 Extensibilidade da Plataforma

3.3.1 Injeçao de Dependência

Injeção de dependência DI, originalmente foi denominado inversão de controle IoC pois a sequência de criação dos objetos depende de como são solicitados pelo sistema.

Quando um sistema é iniciado todos os objetos que o compoe são criados mas utilizando. Entretanto isto não acontece quando é utilizado DI pois os objetos serão criados, injetados, a medida que são necessitados pelo sistema.

A extensibilidade da plataforma é alcançada devido o conceito de DI através do *framework* Spring [2] para flexibilixar a criação de *Visitors*. Com isso o acoplamento entre a classe que cria a árvore sintática de um código fonte e a classe que analisa esta árvore possuem um baixo acoplamento pois quando for solicitado um *visitor* para uma pesquisa o *framework* Spring [2] saberá o momento certo de instanciar e injetar o *visitor* na classe que o solicita.

Devido este conceito adotado no sistema o desenvolvedor deve concentrar seu esforço na criação de *visitors* fazendo como que estes extraim as informações mais precisas bastando somente declarar o *visitor* criado no arquivo *bean.xml* do Spring [2].

3.3.2 Reflection

Também focando na flexibilidade do sistema, a geração de relatórios é dada baseado no conceito de *Reflection* que é a introspecção de código que Java possui. Isto é a capacidade de um programa possui de se observar e até modificar sua estrutura ou comportamento em tempo de execução.

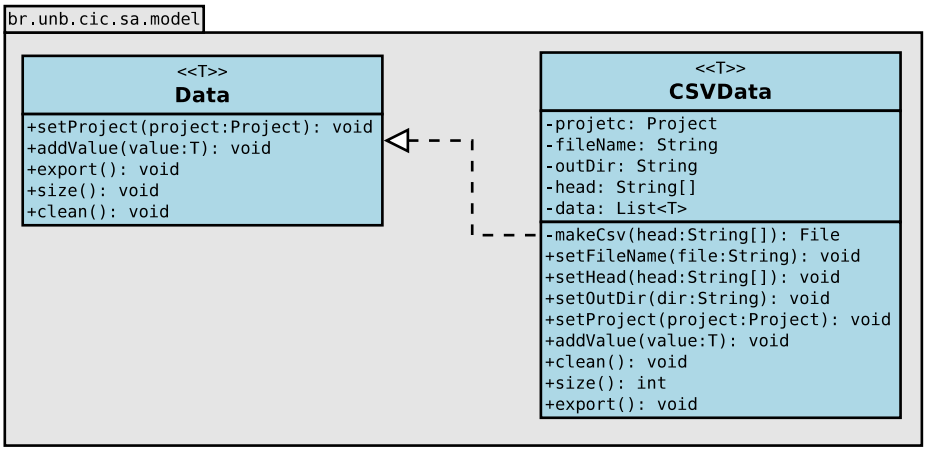

Figura 3.4: Diagrama geração de relatórios.

A geração dos relatórios ocorre utilizando a classe **CSVData<T>** onde o tipo **<T>** é o mesmo utilizado modelar os dados coletados pelos *Visitors*. Os dados são obtidos através dos métodos *getters* destas classes modelos e exportados para arquivos CSV, nesta classe também existe a injeção de dependência que é somente o cabeçalho do arquivo CSV o qual é declarado no *beans.xml* ao mesmo tempo em que o *Visitor* é declarado, a Figura: 3.4 exibe o diagrama de classe.

O cabeçalho de cada CSV é injetado pelo Spring no atributo **head:String[]**. Logo em seguida o método *export()* da classe **CSVData<T>** inicia seu trabalho realizando introspecção recuperando de uma coleção dos dados armazenados do projeto, lista **Data<T>**. Após a recuperação destes dados é realizada o *reflection* para recuperar cada classe modelo e com isso detectar todos os métodos *getters* destas classes e como isso imprimir-los

no relatório que é o arquivo CSV de saída. O Listing: 50 demonstra o uso de *reflection* para gerar os relatórios.

```
public class CSVData<T> implements Data<T>{
  ...
  @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) {
        //reflection code...
        for(Field f: value.getClass().getDeclaredFields()){

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

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

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

          try {
            Method m = value.getClass().getDeclaredMethod(methodName);
            str.append(m.invoke(value));
            str.append(";");
          }catch(NoSuchMethodException |  IllegalAccessException |
                IllegalArgumentException | InvocationTargetException e) {
            throw new RuntimeException("Type " +
                       value.getClass().getName() +
                       " must have a method named " + methodName);
          }
        }
        writer.append(str.toString());
        writer.append("\n");

      }
      writer.flush();

    }catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 3.1: Reflection na geração de relatório.

Capítulo 4

Resultados

Para a realização deste trabalho foram escolhidos 47 projetos open-source separados em 3 grupos **G1** projetos iniciados antes do lançamento de *Generics*, **G2** projetos iniciados após o lançamento de *Generics* e **G3** projetos com a última *release* em 2015. Alguns destes projetos são os mesmos utilizados em, [10, 14, 19], este também fora separados pela natureza da aplicação, Aplicações, Bibliotecas e Servidores/Banco de dados conforme tabela: 4.1 o que totalizou mais de 8.5M de LOC.

4.1 Generics

Relacionado com a adoção de *Generics*, foi descoberto que o maioria dos projetos apresentam um porção significante entre a quantidade de tipos genéricos e a quantidade total de tipos declarados em média(5.31% e 12.31%). Pode-se comprovar que em 16% dos sistemas não declaram nenhum tipo genérico e que o projeto *Commons Collections* é o sistema que com a relação mais expressiva de tipos parametrizados: 75% de todos os tipos declarados são genéricos.

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

Há também um número expressivo de campos e variáveis declaradas como instâncias de tipos genéricos. Isto é, a partir de 925925 variáveis e campos declarados em todos os projetos, 84.880 são instâncias de tipos genéricos, 10 % de todos os campos e variáveis declaradas. Além disso, a partir destes campos e as variáveis declaradas como instância de tipos genéricos, quase 17% são instâncias dos tipos presentes na Tabela: 4.3. Note que, em um trabalho anterior, Parning et al. [14] apresenta `List<String>` com quase 25% de todos os genéricos. Aqui pode ser confirmado que `List<String>` ainda é o tipo mais frequente na ocorrência de tipos genéricos, embora não tão difundido anteriormente. No entanto com 730720 métodos, apenas entre 6157, 0.84%, são *métodos parametrizados.*

Também fora investigado o uso mais avançado de Java *Generics*, incluindo construções que fazem polimorfismo parametrizado. Com este recurso é possível criar classes paramétricas que aceitam qualquer tipo **T** como argumento, uma vez que um tipo **T** satisfaça um determinada pré-condição isto é, o tipo **T** deve ser um qualquer um subtipo (usando o

modificador *extends*) ou um super-tipo (usando o modificador *super*) de um determinado tipo existente. Estes modificadores pode ser usado tanto na declaração de novos tipos, bem como na declaração de campos e variáveis em combinação com o wildcard **?**). A partir de 4355 tipos genéricos declarados em todos os sistemas, descobriu-se que 1.271, quase 30% usam alguns desses modificadores, extends, super, ou ?. Notavelmente, o modificador *extends* é o mais comum, e está presente em todos os tipos genéricos que usam os modificadores *?* e *super*. Em alguns casos de uso são combinações de modificadores, como no exemplo da Listing: 4, onde a classe IntervalTree (projeto CASSANDRA) é parametrizado de acordo com três parâmetros de tipo (C, D e I). Com relação aos campos e declarações de variáveis, quase 13% de todos os casos genéricos usam o *?* *wildcard* e 3,13% usam o *extends*.

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

Listing 4.1: Declaração não trivial de Generics.

Em suma, os resultados mostram que Java *Generics* é uma *feature* em que corresponde a 5% de todos os tipos declarados dos sistemas, portanto, um grande quantidade de código repetido e tipo coerções (moldes) foram evitado usando tipos genéricos. Além disso, a partir desses tipos genéricos, quase 30% usam um recurso avançado (como amplia e de super envolvendo parâmetros de tipo). Também foi descoberto que quase 10% de todos os atributos e variáveis declaradas são tipos genéricos, embora a maior parte são instâncias de tipos genéricos da biblioteca *Java Collection*. Finalmente, embora Parnin et ai. [14] argumentam que uma classe como *StringList* pode cumprir 25% das necessidades de desenvolvedores entretanto, o uso de Java *Generics* não deve ser negligenciada devido aos benefícios que são incorporados ao sistema.

4.2 Lambda Expression

Considerando os sistemas, foi encontrado um uso limitado de Expressões Lambda- independentemente das expectativas e reivindicações sobre os possíveis benefícios dessa construção. Na verdade, apenas cinco projetos faz a adoção deste recurso conforme a Tabela: 4.4, embora a maioria dos cenários de uso (quase 90%) estão relacionados com testes de unidade. Em um primeiro momento, este resultado nos levou a pensar que alguma nova versão de um quadro de teste de unidade poderia ter sido orientador develpers para testar o uso de expressões lambda. No entanto, depois de analisar manualmente o código-fonte, não encontramos qualquer orientação como essa ea adoção de expressões lambda para teste deve ocorrer de forma ad-hoc (como esforços individuais). Ou seja, a partir de milhares de casos de teste de unidade em Hibernate, a poucos testes para uma biblioteca específica (relacionados com cache) usar expressões lambda. Este pequeno uso de expressões lambda pode ser principalmente justfied por uma decisão estratégica de projetos estabelecidos para evitar a migração anterior do código-fonte para novas versões de um idioma.

Foi enviado mensagens para grupos do desenvolvedores sobre o assunto, e algumas respostas esclarecem a atual situação da adoção de Expressões Lamdba. Primeiro de

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

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

Também foi efetuada uma busca no STACK OVERFLOW tentando descobrir se expressões lambda é um tema discutido atualmente ou não [1], utilizando *tags* Java e Lambda. Foi encontrada mais de 1000 questões respondidas. Este número é bastante expressivo, quando considerou-se uma busca por questões marcadas com as *tag* de Java Generics levou-se a um número próximo de 10 000 perguntas, embora *Generics* tenha sido introduzido há mais de dez anos. Possivelmente, expressões lambda está sendo usado principalmente em pequenos projetos e projetos experimentais. Isso pode contrastar com os resultados de [10], que sugerem uma adoção antecipada de novos recursos da linguagem (mesmo antes de lançamentos oficiais da característica). Com base nesses resultados, pode-se comprovar com este trabalho que a adoção antecipada de novos recursos da linguagem ocorre em projetos pequenos e projetos experimentais.

Outra investigação foi se existia a oportunidade de adoção de Expressões Lambda nos projetos estudados. Desta forma, foi complementado um testou maior [12], que investigou as mesmas questões porém eu um número de inferior de projetos. Existem dois cenários típicos para *refactoring* utilizando Expressões Lambda: *Annonymous Inner Classes* (AIC) e *Enhanced for Loops* (EFL). É importante notar que nem todas as AICs e EFLs podem ser reescritas utilizando Expressões Lambda, e existem rígidas precondições que são detalhadas em [12]. Neste trabalho foi utilizado uma abordagem mais conservadora para considerar se é possível refatorar Enhanced for loop para Expressão Lambda para evitar falsos positivos. Entretanto, foi considerado somente oportunidades de refatorar EFL para Expressões Lambda em 3 particular casos: EXIST PATTERN, BASIC FILTER PATTERN e BASIC MAPPING PATTERN de acordo com os Listing: 11, 12 e 14.

[1] Última pesquisa realizada em Novembro 2015

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

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

Listing 4.2: EXIST PATTERN.

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

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

Listing 4.3: FILTER PATTERN.

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

//pode ser refatorado para:
collection.stream().forEach( e  >{
  e.foo();
  e = blah();
  otherCollection.add(e);
});
```

Listing 4.4: MAP PATTERN.

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

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

Tabela 4.1: Projetos.

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

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

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

Tabela 4.3: Tipo declarado X Número de instancia

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

Tabela 4.4: Ocorrências de Expressões Lambda.

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

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

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

Capítulo 5

Considerações Finais e Projeto Fututos

5.1 Projeto Futuro

Referências

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

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.