



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise estática para detectar a evolução 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 Rodrigues — 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 evolução da linguagem java em projetos open source / Thiago Gomes Cavalcanti, Vinícius Correa de Almeida. Brasília : UnB, 2015.

87 p. : il. ; 29,5 cm.

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

1. análise estática, 2. evolução, 3. evolução de linguagens de programação linguagens, 4. language design, 5. software engineering, 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



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

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

Prof. Dr. Genaina Nunes Rodrigues Prof. Dr. Edson Alves da Costa Junior
CIC/UnB FE/UnB-Gama

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

Brasília, 31 de março de 2015

Dedicatória

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

Agradecimentos

Com imensa dificuldade de agradecer a tantas pessoas que de certo modo nos ajudaram nessa conquista, hora em momentos calmos hora apreensivos. Em especial a toda nossa família por dar todo suporte necessário para que pudéssemos concluir essa etapa em nossas vidas, também aluna Daniela Angellos pelo seu desdobramento e conhecimento para nos ajudar a criar essa ferramenta.

Em especial ao professor dr. Rodrigo Bonifácio que nos inseriu nesse imenso mundo da Engenharia de Software, hora apresentando um problemática hora ajudando a resolver barreiras as quais não conseguimos sozinhos.

E ainda a UnB por todo seu corpo docente que sem este essa jornada não seria concluída com excelência, em especial ao professor dr. Edson Alves da Costa Júnior por se deslocar da UnB-Gama para nos ajudar.

Resumo

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

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

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

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

Abstract

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

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

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

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

Sumário

Lista de Abreviaturas	vii
1 Introdução	1
1.1 Introdução	1
1.2 Objetivos	1
1.3 Metodologia	2
1.4 Problema a ser Atacado	2
2 Fundamentação	5
2.1 Evolução Linguagem Java	5
2.2 Engenharia de Linguagens de Software	7
2.3 Representação Intermediária Seção Revisada -Parser-	10
2.4 Refactoring	11
2.5 Análise estática	13
3 Suporte Ferramental	14
3.1 Entrada de dados	15
3.2 Análise da Representação Intermediária	16
3.2.1 Exportação dos Dados	19
4 Resultados	21
4.1 Adoção de Java Generics	23
4.2 Adoção de Java Lambda Expression	24
4.3 Análises adicionais	27
4.3.1 Oportunidades para uso da construção multi-catch	27
4.3.2 Try Resource	29
4.3.3 Switch String	30
5 Considerações Finais e Trabalhos Futuros	33
Referências	34

Lista de Figuras

2.1	Resumo cronológico da linguagem Java.	6
2.2	Portabilidade da linguagem Java.	6
2.3	Fases de aplicações com linguagens.	8
2.4	Fase do pipeline do FindBugs.	8
2.5	Ferramentas necessárias para construção do analisador estático.	9
2.6	Representação da uma frase.	10
3.1	Alto nível de funcionamento do analisador estático.	15
3.2	Input analisador estático.	15
3.3	Funcionamento analisador estático.	16
3.4	Funcionamento analisador estático.	17
4.1	Oportunidades de multi-catch nos projetos.	28
4.2	Adoção de Try-Resource nos projetos.	30
4.3	Oportunidades de <i>refactoring</i> em if-then-else por sistema.	32

Lista de Tabelas

3.1	Visitors com a respectiva descrição, complexidade ciclomática e LOC. . . .	18
4.1	Projetos.	22
4.2	Resumo dos tipos agrupados por idade e do tipo dos projetos.	23
4.3	Tipo declarado X Número de instância	23
4.4	Ocorrências de Expressões Lambda.	25
4.5	Classes concorrentes que <i>extends Thread</i> ou implementam <i>Runnable</i>	27
4.6	Oportunidades de multi-catch por tipo do sistema.	29
4.7	Adoção Try-Resource por tipo do sistema.	31
4.8	Adoção Switch String por tipo do sistema.	31
4.9	Oportunidade de aplicar switch por tipo de sistema.	32

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	Anonymous Inner Class
DI	Dependency Injection
IoC	Inversion of Control
CSV	Comma separated values
CC	Complexidade Ciclômática

Capítulo 1

Introdução

1.1 Introdução

Uma premissa na Engenharia de Software é a *natureza evolutiva* do software, e, com isso, custos significativos são relacionados com as atividades de manutenção. De forma semelhante, as linguagens de programação evoluem, com o intuito de se adaptarem as novas demandas e trazerem benefícios relacionados a produtividade e a melhoria da qualidade dos softwares construídos. Entretanto, um desafio inerente é a evolução de sistemas existentes em direção a adoção de novas construções disponibilizadas nas linguagens [11]. Conforme explicado por Jeffrey L. Overbey e Ralph E. Johnson. [16], tal evolução faz com que características obsoletas sejam mantidas e raramente são removidas de uma linguagem o que acarreta em um aumento da complexidade, aprendizagem e da manutenção do software. Isso naturalmente aumenta a dificuldade de desenvolvimento o que resulta em um aumento de dificuldade de aprendizagem de determinada versão já ultrapassada de uma linguagem e faz com que a equipe alterne entre propriedades atuais e antigas as quais passam a ser quase um dialeto da linguagem implicando no aumento de tempo para conceber um projeto e consequentemente gerindo aumento no custo final projeto.

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

1.2 Objetivos

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

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

1.3 Metodologia

A realização deste trabalho envolveu atividades de revisão da literatura, contemplando um estudo de artigos científicos que abordam a adoção de novas características da linguagem Java ao longo do lançamento das diferentes versões para a comunidade de desenvolvedores [7, 9, 11, 12, 16, 17, 21]. Com isso, foi possível compreender a limitação dos trabalhos existentes e, dessa forma, definir o escopo da investigação.

Posteriormente, foi necessário buscar uma compreensão sobre como implementar ferramentas de análise estática, e escolher uma plataforma de desenvolvimento apropriada (no caso, a plataforma Eclipse JDT [2]). Posteriormente, foi iniciada uma fase de implementação dos analisadores estáticos usando padrões de projetos típicos para essa finalidade: visitor, dependency injection, ...

Finalmente, foi seguida uma estratégia de Mineração em Repositórios de Software, onde foram feitas as análises da adoção de construções da linguagem Java em projetos *open-source*, de forma similar a outros artigos existentes [6, 8, 10, 16, 19, 21, 22, 23, 25].

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

Em seguida foi estudada a melhor arquitetura para a elaboração do analisador estático proposto de modo que esta no tivesse um fraco acoplamento entre os módulos necessários e facilitasse a pesquisa de outras características através da injeção do visitors [13] usando o *spring framework* [3]. Mais adiante a arquitetura escolhida será exibida com mais detalhes.

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 multi-plataforma 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 evolução de seus compiladores os quais não forneciam mais suporte a versões anteriores conforme relata Jeffrey L. Overbey e Ralph E. Johnson em [16], que como consequência forçou toda comunidade desenvolvedora a se atualizar. E ainda havia a possibilidade de certos trechos de código sofrer um *refactoring* 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 *refactoring* na grande parte das modernas IDEs mantém 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 *refactoring* ficar complexa e de difícil entendimento segundo Max Schäfer e Oege de Moor em [22]. Modernas IDEs como eclipse realizam complexos *refactoring* através da técnica de *microrefactoring* 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 *refactoring* ou manterá o código congelado expondo o mesmo a usuários maliciosos.

Capítulo 2

Fundamentação

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

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

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

2.1 Evolução Linguagem Java

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

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

Histórico Linguagem Java

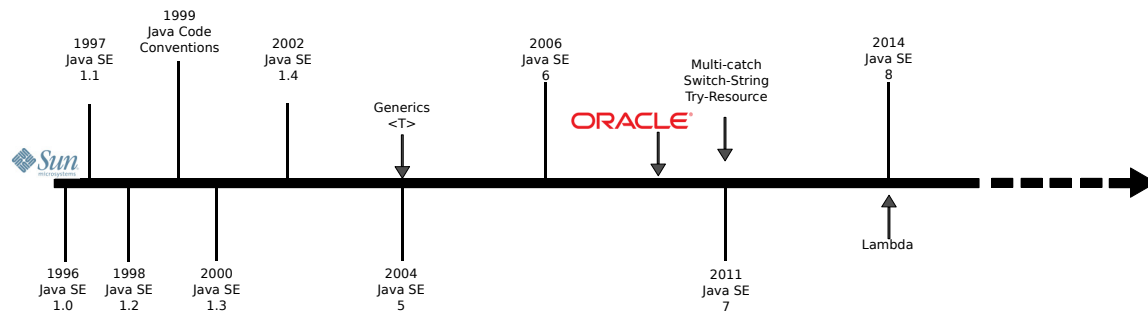


Figura 2.1: Resumo cronológico da linguagem Java.

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

Portabilidade Linguagem Java

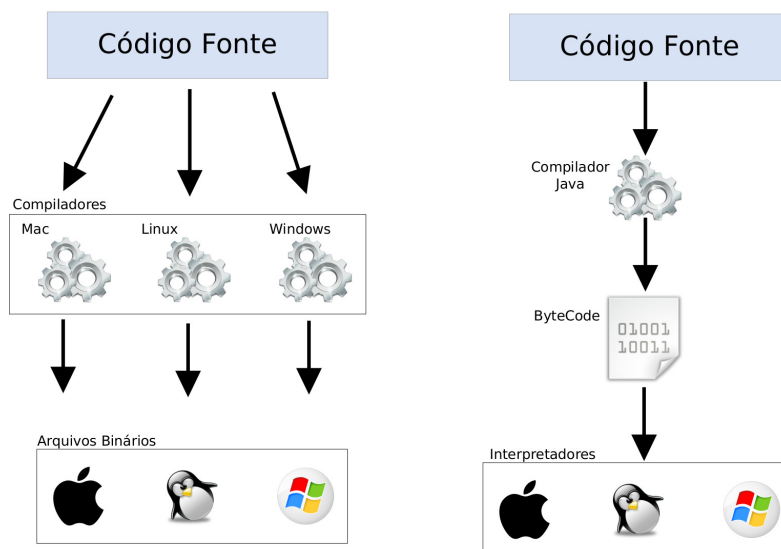


Figura 2.2: Portabilidade da linguagem Java.

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

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

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

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

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

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

2.2 Engenharia de Linguagens de Software

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

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

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

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

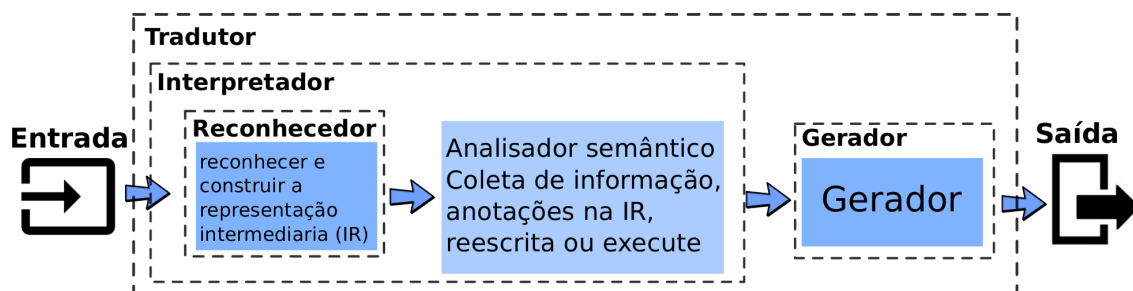


Figura 2.3: Fases de aplicações com linguagens.

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

Além dessas aplicações típicas, ferramentas para a identificação estática de *bugs*, por exemplo, também são comumente implementadas usando uma organização como a representada na Figura 2.3. A ferramenta *FindBugs* [1] serve como um exemplo de solução para identificação de possíveis erros em programas escritos na linguagem Java, a partir do *bytecode* resultante do processo de compilação. Note na Figura: 2.4 a semelhança arquitetural com as abordagens típicas para o processamento de artefatos de linguagens de programação.

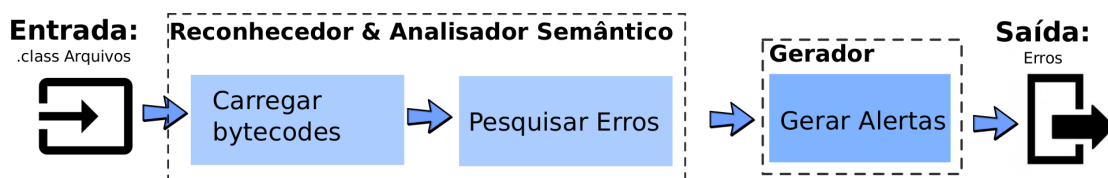


Figura 2.4: Fase do pipeline do FindBugs.

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

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

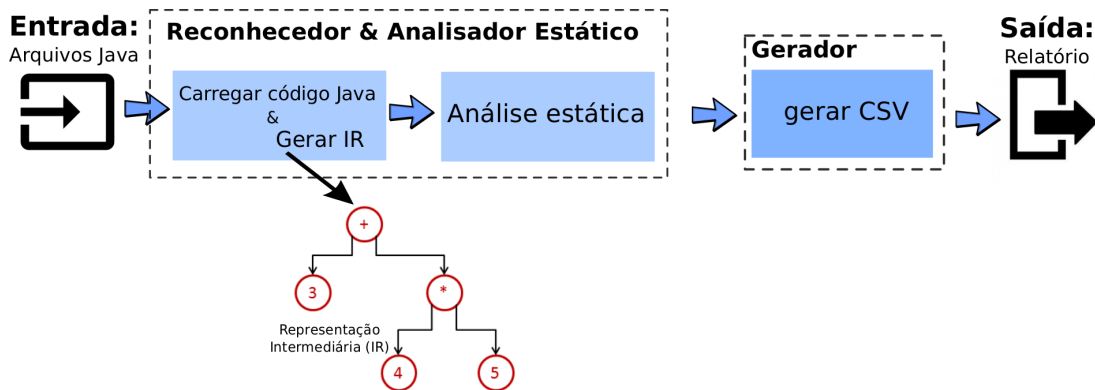


Figura 2.5: Ferramentas necessárias para construção do analisador estático.

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

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

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

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

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

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

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

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

2.3 Representação Intermediária **Seção Revisada -Parser-**

Conforme introduzido anteriormente na Figura: 2.5, é gerar uma representação intermediária da linguagem Java. Para efetuar essa tarefa é necessário antes de tudo reconhecer **frases** entretanto para criar um reconhecimento de **frases** faz-se necessário duas etapas. Distinguir as construções que compoem a linguagem e identificar os elementos e subestruturas das **frases**. Terrance Parr em [18], explica que o ato de reconhecer **frase** em computação é denominado *parser*.

A estrutura do *parser* é composta de maneira similar as sentenças da língua portuguesa, identificando verbos, nomes, substantivos, etc... De modo similar é necessário fazer a mesma identificação em um linguagem de programação, reconhecendo operadores, variáveis e outros símbolos. E a combinação dos símbolos é denominado **tokens**.

A representação do *parser* é feita através de uma árvore onde Tokens ficam nas folhas e estas compoem as frases. É utilizando uma árvore para representar porque com esta é possível extrair toda informação que seja necessária da linguagem. A Figura: 2.6 exhibe com mais detalhes o parser sendo representado por uma árvore.

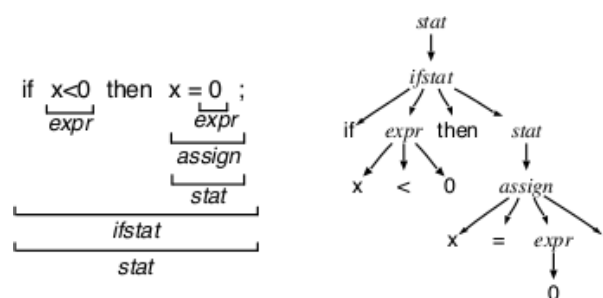


Figura 2.6: Representação da uma frase.

Existem muitos padrões de parses pois algumas linguagens são mais complexas que outras. Tendo em visata o desafio que é trabalhar e analisar uma linguagem de programação, alguns padrões adotados para este contexto facilitam esta tarefa e com isso este trabalho abordará os quatro conceitos mais importantes segundo Terence Parr em [18] para prover o parser de acordo com a necessidade.

- **Mapping Grammars to Recursive-Descent Recognizers**

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

- **LL(1) Recursive-Descent Lexer**

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

- **LL(1) Recursive-Descent Parser**

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

- **LL(k) Recursive-Descent Parser**

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

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

2.4 Refactoring

Por definição o refactoring é mudança interna do software sem alterar seu comportamento tornando assim seu entendimento mais claro. Visando evitar que sejam perdidas diversas horas para identificar possíveis oportunidades de classes onde possam ser evoluí-

das, este trabalho de conclusão identifica trechos de código dentro das classes que pode ser evoluídos.

Muitas modificações podem ser feitas em um software mas segundo M.Fowler at al [5] somente é considerado refactoring mudanças que facilitam o entendimento do software. Contrastando esta visão existem mudanças com objetivo de melhorar o desempenho do software onde somente são alterada as estruturas internas permanecendo inalterado o comportamento do software. Entretanto a melhoria na performance do software geralmente eleva o grau de dificuldade para sua compreensão, o que faz com que algumas dessas evoluções visando desempenho não sejam caracterizadas como refactoring dado a definição.

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

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

O Listing: 12, exemplifica de forma empírica um *filter* em um *collection* onde ocorre uma redução significativa de código além de um design mais atual por utilizar expressões lambda que foram adicionadas em Java 8. Vale destacandar que o Listing: 12 continua com mesmo comportamento após o refactoring, desta forma nenhum usuário ou desenvolvedor pode alegar que o software foi modificado.

```
1 // ...
2 for (T e: collection) {
3     if (e.pred(args)) {
4         otherCollection.add(e);
5     }
6 }
7
8 //might be replaced by:
9 collection.stream().
10    filter(e->pred(args)).
11    forEach(e -> otherCollection.add(e));
```

Listing 2.1: The EXMPLO DE FILTER PATTERN

Conforme explica M.Fowler at. al. [5] algumas vezes não se deve ser refatorar o código. Um desses casos é quando existir a necessidade de reescrever todo o código, um outro caso é a necessidade de manter um código de fácil entendimento para os programadores iniciantes. O que é uma decisão difícil, no caso do Listing: 12 é possível fazer uma evolução com a parte funcional de Java 8 entretanto alguns desenvolvedores podem não possuir conhecimento adequado da parte funcional e com isso é recomendado que não seja evoluído o código.

Tendo em vista que aplicar um *refactoring* demanda tempo isto torna uma tarefa custosa para empresas, este fator é determinante para que programadores não refatorem seu códigos em muitos casos. Com esse cenário é imprescindível o uso de ferramentas refatorem

ou auxiliem nesta tarefa. Auxiliando nesta tarefa este trabalho identifica possibilidades de refatoração, ao utilizarem estas ferramentas torna mais acessível ao programador e a empresa refatorar pois o trabalho é direcionado fazendo com que tempo seja poupado.

2.5 Análise estática

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

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

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

A linguagem Java proporciona duas maneiras de realizar análise estática, a primeira é através código fonte, *.java* e a segunda através do *bytecode*, *.class*. Este trabalho foca em realizar análise no código fonte, entre tanto nada impede que o trabalho seja realizado da segunda maneira. Existem programas renomados que realizam tal análise utilizando os *bytecodes* e um destes programas é o FindBugs [1].

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

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

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

Capítulo 3

Suporte Ferramental

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

Atualmente existem diversas tecnologias capazes de prover ferramentas para implementar um analisador estático conforme as nossas necessidades. Entretanto, devido a maior experiência dos participantes do projeto com uso da linguagem Java, foi feita a opção por se utilizar a infraestrutura da plataforma *Eclipse Java Development Tools* [2] (Eclipse JDT). O EclipseJDT [2] fornece um conjunto de ferramentas que auxiliam na construção de ferramentas que permitem processar código fonte escrito na linguagem de programação Java.

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

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

O Eclipse JDT [2] disponibiliza ainda um *Parser* para a linguagem Java que atende a especificação Java 8 da linguagem (a mais atual com lançamento público) e que produz a representação intermediária baseada no conjunto de classes Java mencionado anteriormente e que correspondem a uma *AST* do código fonte. A plataforma também oferece uma hierarquia de classes para travessia na *AST*, de acordo com o padrão de projeto *Visitors* [13], e que facilita a análise estática de código fonte.

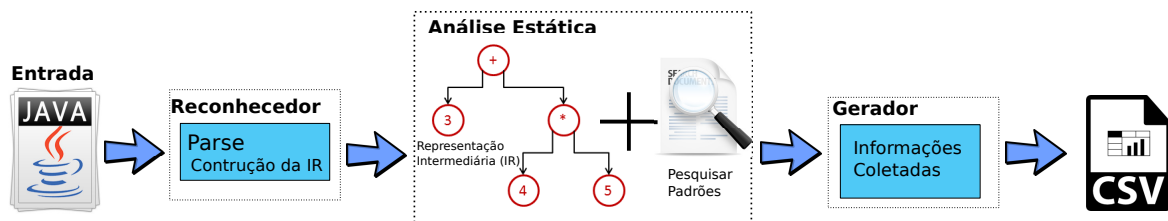


Figura 3.1: Alto nível de funcionamento do analisador estático.

O padrão de projeto *Visitor* [13] é um padrão de projeto de característica comportamental que representa uma operação a ser realizada sobre elementos de uma árvore de objetos. Neste caso a operação a ser realizadas é visitar nós de interesse da AST Java (como os nós que representam o uso de uma expressão Lambda em Java). Cada *visitor* permite que uma nova operação seja criada sem que a estrutura da árvore de objetos sofra alterações. Com isso é trivial adicionar novas funcionalidades em um *visitor* existente ou criar um novo.

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

3.1 Entrada de dados

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

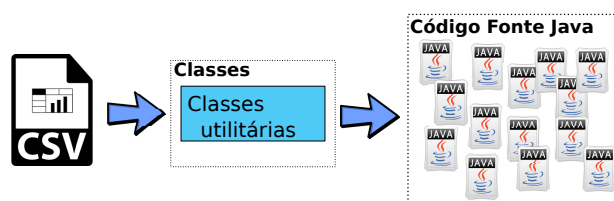


Figura 3.2: Input analisador estático.

3.2 Análise da Representação Intermediária

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

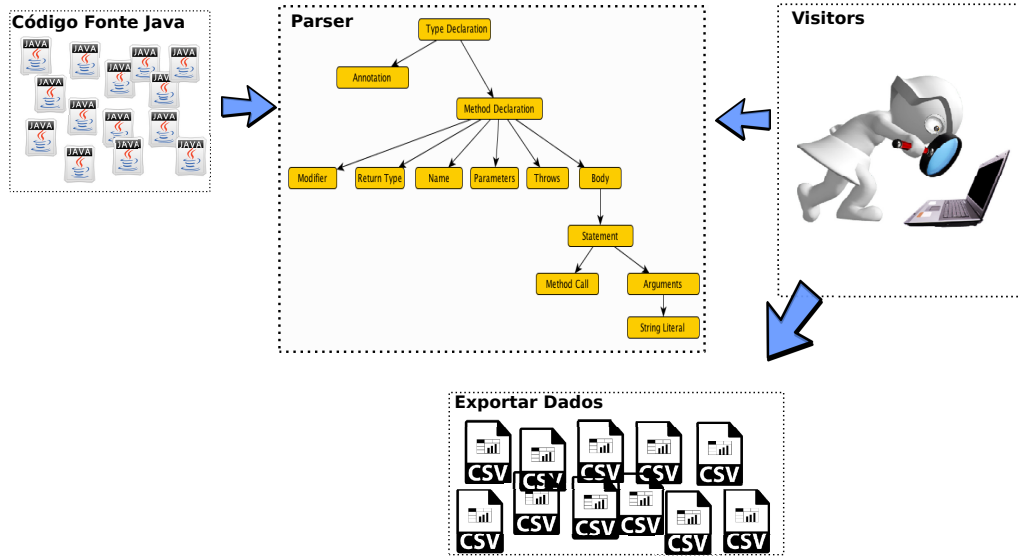


Figura 3.3: Funcionamento analisador estático.

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

Por exemplo, a Listagem 24 apresenta o código necessário para visitar e popular informações relacionadas a declaração de enumerações. A classe `public class Visitor<T> { ... }` possui uma coleção de objetos do tipo parametrizado, sendo possível adicionar instâncias desses objetos com a chamada `collectedData.addValue()`. Note que o exemplo apresentado corresponde a um dos mais simples *visitors* implementados. Outros *visitors* possuem uma lógica mais elaborada, como por exemplo os *visitors* que identificam oportunidades para usar construções como *multi-catch* ou *lambda expressions*.

```

1 public class EnumDeclaration {
2     private String file;
3     private int startLine;
4     private int endLine;
5
6     //constructos + getters and setters.
7 }
8
9 public class EnumDeclarationVisitor extends Visitor<EnumDeclaration> {
10
11     @Override
12     public boolean visit(org.eclipse.jdt.core.dom.EnumDeclaration node) {
13
14         EnumDeclaration dec = new EnumDeclaration(this.file, unit.getLineNumber
15         (node.getStartPosition()), unit.getLineNumber(node.getLength() - node.
16         getStartPosition()));
17
18         collectedData.addValue(dec);
19
20         return true;
21     }
22 }

```

Listing 3.1: Classes usadas para capturar declaração de enumerações.

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

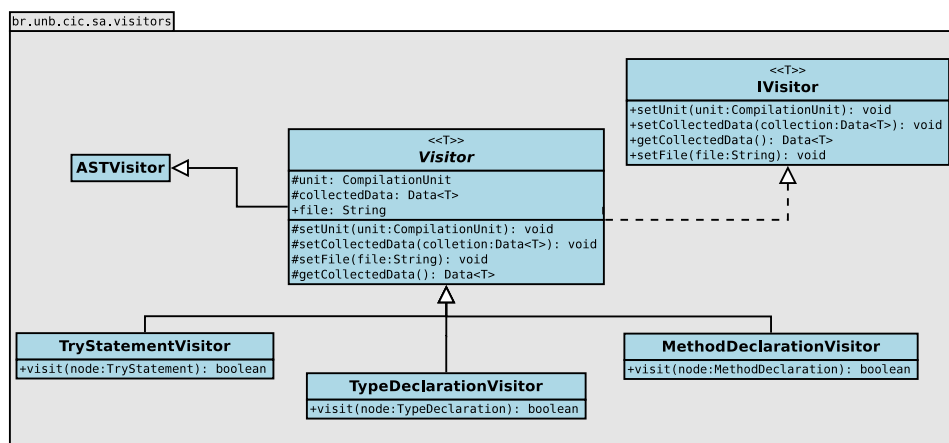


Figura 3.4: Funcionamento analisador estático.

A Tabela: 3.1 exibe todos os *Visitors* criados neste projeto com sua respectiva descrição. A complexidade ciclomática dos *Visitors* a qual é dada pela quantidade de caminhos independentes em um trecho de código, vale ressaltar que a complexidade dos *Visitors* varia de 1 a 8 sendo a média 2,5 e a quantidade média de linha de código total necessária para escrever cada *Visitor* é 47.

Tabela 3.1: Visitors com a respectiva descrição, complexidade ciclomática e LOC.

Visitor	Atribuição	CC	LOC
AIC	Pesquisar <i>Anonymous Inner Class</i> declaradas.	6	64
ExistPattern	Pesquisa <i>EnhancedFor</i> que iteram sobre uma coleção procurando qualquer ocorrência nessa coleção.	28	116
FieldAndVariableDeclaration	Lista todos as variáveis declaradas como os respectivos tipos.	8	81
FilterPattern	Lista todos os <i>EnhancedFor</i> que iteram uma coleção filtrando elementos desta mesma coleção.	43	168
ImportDeclaration	Lista todos os <i>imports</i> existentes em cada classe.	1	63
LambdaExpression	Pesquisa casos de utilização da expressões lambda.	1	33
Lock	Verifica se nos métodos declarados existe alguma variável chamada <i>Lock</i> , <i>ReentrantLock</i> , <i>ReadLock</i> ou <i>WriteLock</i> .	12	75
MapPattern	Pesquisa <i>EnhancedFor</i> que iteram sobre uma coleção onde seja aplicado algum método sobre os itens desta coleção.	27	103
MethodCall	Verifica onde esta sendo utilizado <i>reflection</i> no projeto.	2	22
MethodDeclaration	Coleta informações sobre os métodos declarados nos projetos.	5	45
ScriptingEngine	Verifica se o projeto faz chamada a algum <i>Scripting</i> .	5	39
SwitchStatement	Pesquisa <i>Switchs</i> que utilizam <i>String</i> como parâmetro.	3	35
SwitchStringOpportunities	Pesquisa <i>If-Else</i> aninhados onde no <i>If</i> contenha <i>String</i> , caracterizando uma possibilidade de adoção de <i>Switch</i> com <i>String</i> .	3	50
TryStatement	Pesquisa <i>trys</i> que utilizar <i>resource</i> , adoção de <i>multi-catch</i> e <i>trys</i> que possuem <i>catchs</i> aninhados.	11	80
TypeDeclaration	Pesquisa todos os tipos declarados.	2	41

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

O mecanismo de injeção de dependência foi incorporado na arquitetura com o uso do *framework* Spring [3], o que não causou nenhum impacto significativo na solução inicialmente proposta e que não fazia uso de tal mecanismo— os *visitors* eram instanciados de maneira *programática*. O uso do mecanismo de injeção de dependência serviu para flexibilizar não apenas a incorporação de novos *visitors*, mas também para definir, de forma mais flexível, a estratégia de exportação dos dados coletados. Graças ao mecanismo de injeção de dependência, o desenvolvedor pode concentrar seu esforço na criação de *visitors*, fazendo como que estes implementem a lógica necessária para extrair as informações. Para que novos *visitors* se conectem a plataforma, tornou-se necessário declarar o *visitor* no arquivo com a definição dos objetos gerenciados pelo Spring [3].

3.2.1 Exportação dos Dados

Na versão atual do suporte ferramental desenvolvido nessa monografia, os dados coletados pelo analisador estático são exportados exclusivamente no formato CSV. Esse formato facilita as análises estatísticas usando o ambiente e linguagem de programação R [20]. Também com foco na extensibilidade do sistema, os componentes envolvidos na geração de relatórios utilizam os mecanismos de injeção de dependência, mencionado na seção anterior, e introspecção de código, via a API *Reflection* da linguagem de programação Java. Tal mecanismo oferece aos programadores a capacidade de escreverem componentes que podem observar e até modificar a estrutura e o comportamento dos objetos em tempo de execução.

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

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

```
1 public class CSVData<T> implements Data<T>{
2     ...
3     @Override
4     public void export() {
5         try (FileWriter writer =
6             new FileWriter(this.makeCsv(head), true)){
7
8             StringBuffer str = new StringBuffer("");
9
10            if(data == null) { return; }
11
12            for(T value : data) {
13                //reflection code...
14                for(Field f: value.getClass().getDeclaredFields()){
15
16                    String fieldName = f.getName();
17                    String prefix = "get";
18
19                    if(f.getType().isPrimitive() &&
20                       f.getType().equals(Boolean.TYPE)) {
21                        prefix = "is";
22                    }
23
24                    String methodName = prefix +
25                        Character.toUpperCase(fieldName.charAt(0)) +
26                        fieldName.substring(1);
27
28                    try {
```

```

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

```

Listing 3.2: Reflection na geração de relatório.

Capítulo 4

Resultados

Neste capítulo foi investigado empiricamente a adoção de *features* Java e *standard libraries* por replicação de um estudo existente de Parnin et al [17]. Adicionalmente a contribuição deste trabalho é abordar outras quatro características da linguagem Java além de *Java Generics* que são *Java Lambda Expression*, *Multi-catch*, *Try-Resource* e *Switch-String* onde a questão que direcionou esta contribuição foi **RQ1: Qual o típico uso de Java Generics e Java Lambda Expressions?**

A replicação do estudo de *Java Generics* ocorreu através de projetos opensource. Onde adicionalmente além da replicação do trabalho estudo de Parnin et al [17] este adicionou a compreensão de como *Java Generics* se correlaciona com a *release* inicial dos projetos selecionados.

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

Para a realização a investigação os 46 projetos opensource escolhidos foram separados em 3 grupos **G1** projetos iniciados antes do lançamento de *Java Generics*, **G2** projetos iniciados após o lançamento de *Java Generics* e **G3** projetos com a última *release* em 2015. Alguns destes projetos são os mesmos utilizados em, [12, 17, 24], e também fora separados pela natureza da projeto, Aplicações, Bibliotecas e Servidores/Banco de dados conforme tabela: 4.1 o que totalizou mais de 9M de **LOC** detalhados na Tabela: 4.1.

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

Tabela 4.1: Projetos.

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

4.1 Adoção de Java Generics

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

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

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

Tipo de Projeto	Antes Java SE 5.0	Tipo	Tipo Genérico	Ratio(%)
Aplication	Yes	18168	177	0.97
Aplication	No	16148	744	4.61
Library	Yes	21537	1198	5.56
Library	No	22639	947	4.18
Server/Database	Yes	18038	552	3.10
Server/Database	No	11790	760	6.45

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

Tabela 4.3: Tipo declarado X Número de instância

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

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

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

```

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

```

Listing 4.1: Declaração não trivial de Generics.

Os resultados mostram que *Java Generics* é uma *feature* em que corresponde a 5% de todos os tipos declarados dos sistemas, portanto, uma grande quantidade de código repetido e tipo coerções (moldes) foram evitado usando tipos genéricos. Além disso, a partir desses tipos genéricos, quase 30% usam um recurso avançado (como *extends* e *super* envolvendo parâmetros de tipo). Também foi descoberto que quase 10% de todos os atributos e variáveis declaradas são tipos genéricos, embora a maior parte são instâncias de tipos genéricos da biblioteca *Java Collection*. Finalmente, embora Parnin et ai. [17] argumentam que uma classe como *StringList* pode cumprir 25% das necessidades de desenvolvedores entretanto, o uso de *Java Generics* não deve ser negligenciada devido aos benefícios que são incorporados ao sistema.

4.2 Adoção de Java Lambda Expression

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

Foi enviado mensagens para grupos do desenvolvedores sobre o assunto, e algumas respostas esclarecem a atual situação da adoção de *Java Lambda Expression*. Primeiro de tudo, para os sistemas estabelecidos, as equipes de desenvolvedores muitas vezes não

Tabela 4.4: Ocorrências de Expressões Lambda.

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

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

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

Também foi efetuada uma busca no *STACK OVERFLOW* tentando descobrir se *Java Lambda Expression* é um tema discutido atualmente ou não ¹, utilizando *tags* Java e Lambda. Foi encontrada mais de 1000 questões respondidas. Este número é bastante expressivo, quando considerou-se uma busca por questões marcadas com as *tag* de Java Generics levou-se a um número próximo de 10 000 perguntas, embora *Generics* tenha sido introduzido há mais de dez anos. Possivelmente, *Java Lambda Expression* está sendo usado principalmente em pequenos projetos e experimentais. Isso pode contrastar com os resultados de [12], que sugerem uma adoção antecipada de novos recursos da linguagem (mesmo antes de lançamentos oficiais). Com base nesses resultados, pode-se comprovar com este trabalho que a adoção antecipada de novos recursos da linguagem ocorre em projetos pequenos e experimentais.

Outra investigação foi se existia a oportunidade de adoção de *Java Lambda Expression* nos projetos estudados. Desta forma, foi complementado um testou maior [14], que investigou as mesmas questões porém eu um número de inferior de projetos. Existem dois cenários típicos para *refactoring* utilizando Expressões Lambda: *Anonymous Inner*

¹Última pesquisa realizada em Novembro 2015

Classes (**AIC**) e *Enhanced for Loops* (**EFL**). É importante notar que nem todas as **AICs** e **EFLs** podem ser reescritas utilizando *Java Lambda Expression*, e existem rígidas condições que são detalhadas em [14]. Neste trabalho foi utilizado uma abordagem mais conservadora para considerar se é possível refatorar *Enhanced for loop* para *Java Lambda Expression* o que evita falsos positivos. Entretanto, foi considerado somente oportunidades de refatorar **EFL** para *Java Lambda Expression* em 3 casos particular: **EXIST PATTERN**, **BASIC FILTER PATTERN** e **BASIC MAPPING PATTERN** de acordo com os Listing: 11, 12 e 15.

```

1 // ...
2 for(T e : collection){
3     if(e.pred(args)){
4         return true;
5     }
6 }
7 return false;
8
9 //pode ser refatorado para:
10 return collection.stream().anyMatch(e->pred(args));

```

Listing 4.2: **EXIST PATTERN**.

```

1 // ...
2 for(T e : collection){
3     if(e.pred(args)){
4         otherCollection.add(e);
5     }
6 }
7
8 //pode ser refatorado para:
9 collection.stream().filter(
10     e->pred(args).forEach(e->otherCollection.add(e)
11 );

```

Listing 4.3: **FILTER PATTERN**.

```

1 // ...
2 for(T e : collection){
3     e.foo();
4     e = blah();
5     otherCollection.add(e);
6 }
7
8 //pode ser refatorado para:
9
10 collection.stream().forEach(e->{
11     e.foo();
12     e = blah();
13     otherCollection.add(e);
14 });

```

Listing 4.4: **MAP PATTERN**.

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

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

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

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

4.3 Análises adicionais

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

Baseado nesta assertiva, foi feita uma investigação adicional que visa minerar o uso de construções obsoletas em código fonte existente e encontrar possíveis casos de trechos de código que poderiam ter evoluído ao longo das versões da linguagem Java. Essas situações caracterizam cenários potenciais de melhoria de código e preservam o comportamento do sistema (tipicamente um *refactoring*), com o objetivo único de usar construções introduzidas nas versões 7 e 8 da linguagem Java.

Dentre as evoluções da linguagem as características investigadas, esse trabalho foca em descobrir oportunidades de adoção das características `multi-catch`, `try resource` e `switch com string` e se as mesmas estão sendo adotadas.

4.3.1 Oportunidades para uso da construção `multi-catch`

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

Com as análises realizadas, foi possível identificar uma quantidade significativa de oportunidades de uso dessa construção, conforme exibido na Figura: 4.1. Ao todo 95%

dos projetos pesquisados possuem oportunidades reais para aplicação de `multi-catch` e foram encontrados 1474 blocos `try` que possuem `catchs` repetidos. Estas ocorrências estão distribuídas em 1028 arquivos e totalizando 30 936 LOC. Importante observar que o teste de similaridade entre os blocos `catch` foi realizado através de uma chamada a um método externo que verifica a igualdade da árvore sintática. Apesar dessa abordagem não fazer uso de uma estratégia de análise de similaridade de código mais robusta, a mesma pode ser facilmente alterada de acordo com algum algoritmo existente. A Tabela: 4.6 exibe a distribuição de oportunidades de `multi-catch` pela natureza do sistema.

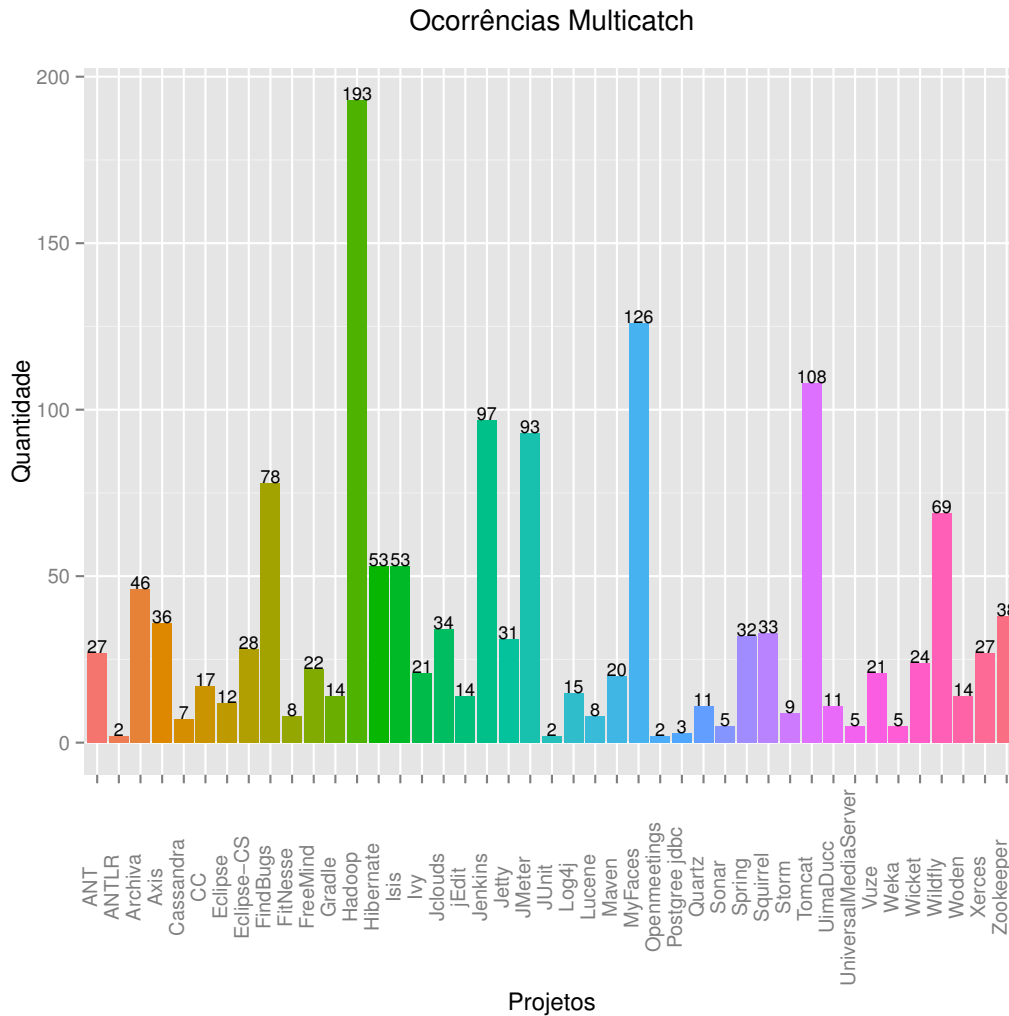


Figura 4.1: Oportunidades de `multi-catch` nos projetos.

A implantação de `multi-catch` em um projeto pode ser iniciada com substituição de blocos `catch` aninhados por este recurso tornando evidente a redução significativa das LOC duplicadas. A classe `AbstractNestablePropertyAccessor` do projeto *Spring 4.2.0.RC2* contém `catch` duplicado conforme a listagem: 16. Entretanto após um *refactoring* na listagem: 16 para implantação do recurso `multi-catch` pode-se verificar na listagem: 10 a redução significativa de código duplicado na ordem de 42% o que torna este recurso muito útil sem causar grandes mudanças no software.

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

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

```

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

```

Listing 4.5: Código sem adoção de multi-catch

```

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

```

Listing 4.6: Refactoring com uso de multi-catch

4.3.2 Try Resource

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

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


```

1 static String readFirstLineFromFileWithFinallyBlock(String path)
2                                     throws IOException {
3     BufferedReader br = new BufferedReader(new FileReader(path));
4     try {
5         return br.readLine();
6     } finally {
7         if (br != null) br.close();
8     }
9 }

```

Listing 4.7: Try sem adoção de resource.

```

1 static String readFirstLineFromFileWithFinallyBlock(String path)
2                                     throws IOException {
3     try (BufferedReader br = new BufferedReader(new FileReader(path));) {
4         return br.readLine();
5     }
6 }

```

Listing 4.8: Try adotando resource.

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

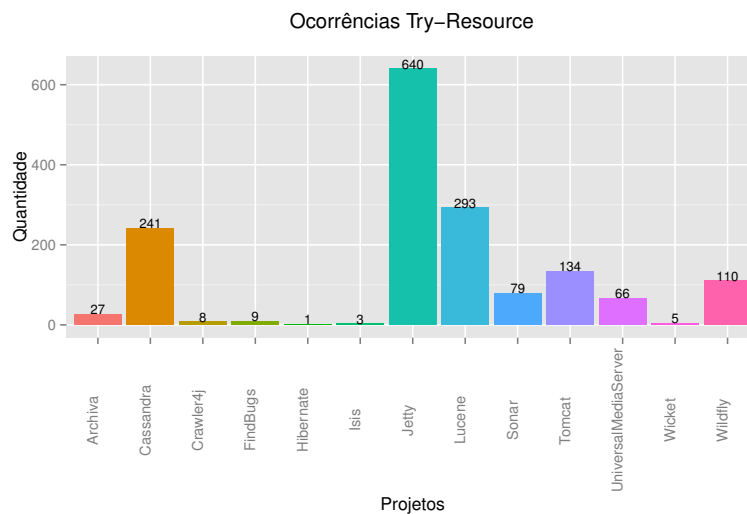


Figura 4.2: Adoção de Try-Resource nos projetos.

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

4.3.3 Switch String

O suporte de **String** em **Switch** foi permitido em Java 7 este recurso ajuda a preservar um padrão de codificação tendo em vista que atualmente é permitido operações de seleção

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

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

em um conjunto de constantes de `String`. Além de possuir desempenho superior ao `if-then-else` conforme a documentação [4] informa pois o gerado para `switch` que utilizam `string` é mais eficiente que o `if-then-else`.

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

Tabela 4.8: Adoção Switch String por tipo do sistema.

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

Com intuito de encontrar oportunidades de aplicar este recurso, foi verificado a existência de `if-then-else` que invoquem na sua expressão um método e verificando se este método utiliza o `equals()` comparando `String` conforme demonstrado na Listagem: 4 o qual é a oportunidade possível de efetuar um *refactoring* para o `switch`.

```

1 if (String.equals("...")) {
2   ...
3 }

```

Listing 4.9: Modelo para aplicação de Switch com String.

Direcionado pelo padrão da Listagem: 4 foram encontrados o total de 4940 oportunidades distribuídas em 45 dos 46 projetos, o que confirma que este recurso não está sendo adotado em sua plenitude. A Figura: 4.3 exibe a quantidade de oportunidades em cada projeto verificado, e a Tabela: 4.9.

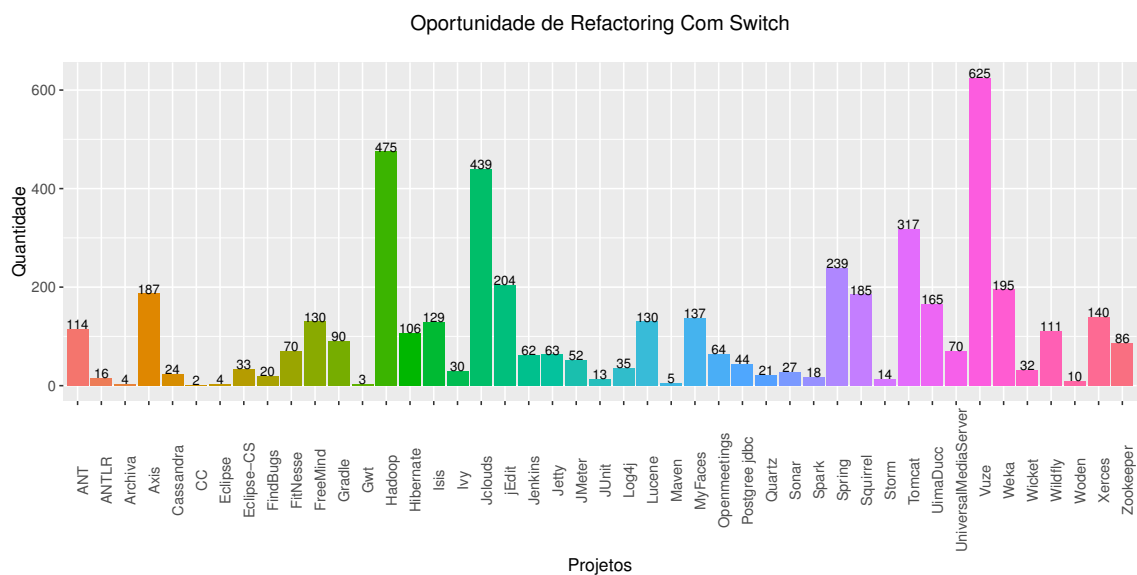


Figura 4.3: Oportunidades de *refactoring* em *if-then-else* por sistema.

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

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

Capítulo 5

Considerações Finais e Trabalhos Futuros

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

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

Referências

- [1]
- [2] Eclipse java development tools (jdt) @ONLINE. <http://www.eclipse.org/jdt/>. Accessed: 2015-07-06.
- [3] Spring framework reference documentation @ONLINE. <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/>. Accessed: 2015-06-06.
- [4] Strings in switch statements @ONLINE. <http://docs.oracle.com/javase/7/docs/technotes/guides/language/strings-switch.html>. Accessed: 2015-07-06.
- [5] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [6] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, September 2008.
- [7] Rodrigo Bonifácio, Tijs van der , and Jurgen Vinju. The use of c++ exception handling constructs: A comprehensive study.
- [8] Gilad Bracha, Martin Odersky, and David Stoutamire. Gj: Extending the javatm programming language with type parameters.
- [9] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, October 1998.
- [10] Alan Donovan, Adam Kiežun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. *SIGPLAN Not.*, 39(10):15–34, October 2004.
- [11] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. A large-scale empirical study of java language feature usage. 2013.
- [12] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

- [13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [14] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 543–553. ACM, 2013.
- [15] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.
- [16] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, October 2009.
- [17] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11*, pages 3–12, New York, NY, USA, 2011. ACM.
- [18] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.
- [19] Gustavo Pinto, Wesley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java’s concurrent programming constructs. *J. Syst. Softw.*, 106(C):59–81, August 2015.
- [20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.
- [21] Jeffrey L. Schaefer and Ralph E. Johnson. Regrowing a language: Refactoring tools allow programming languages to evolve. *SIGPLAN Not.*, 44(10):493–502, October 2009.
- [22] Max Schaefer and Oege de Moor. Specifying and implementing refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010.
- [23] Daniel von Dincklage and Amer Diwan. Converting java classes to use generics. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 1–14, 2004.
- [24] A Ward and D Deugo. Performance of lambda expressions in java 8. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 119. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.
- [25] Ba Wichmann, Aa. Canning, D. L. Clutterbuck, L A Winsborrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 1995.