

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
ESCOLA DE INFORMÁTICA APLICADA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Um Plug-In de Validação de Regras Arquiteturais para Inspeção Automatizada
de Código

Autores:

Carlos Magno Coutinho de Sena

Victor Fortunato Azevedo

Orientadores:

Leonardo Guerreiro Azevedo

Raphael de Almeida Rodrigues

Um Plug-In de Validação de Regras Arquiteturais para Inspeção Automatizada
de Código

Carlos Magno Coutinho de Sena

Victor Fortunato Azevedo

Projeto de Graduação apresentado à
Escola de Informática Aplicada da
Universidade Federal do Estado do Rio de
Janeiro (UNIRIO) para obtenção do título de
Bacharel em Sistemas de Informação

Rio de Janeiro - RJ

Julho/2015

Um Plug-In de Validação de Regras Arquiteturais para Inspeção Automatizada
de Código

Aprovado em ____/____/____

BANCA EXAMINADORA

Prof. Leonardo Guerreiro Azevedo, D.Sc. (UNIRIO; IBM Research Brasil)

Prof. Alexandre Luis Correa, D. Sc. (UNIRIO)

Sergio Luiz Ruivace Cerqueira, M.Sc. (LES PUC-RIO)

Raphael de Almeida Rodrigues, B. Sc. (UNIRIO)

Os autores deste Projeto autorizam a ESCOLA DE INFORMÁTICA APLICADA da UNIRIO a divulgá-lo, no todo ou em parte, resguardando os direitos autorais conforme legislação vigente.

Rio de Janeiro, ____ de _____ de _____

Carlos Magno Coutinho de Sena

Victor Fortunato Azevedo

Agradecimentos

Agradecemos primeiramente a Deus por todas as graças fornecidas durante esses anos. Não poderia deixar de agradecer de forma sincera aos meus familiares, em especial meus pais Roberto e Oneti, por terem me guiado no caminho do bem e me orientado durante todo o meu crescimento como pessoa, aluno e profissional. Jamais teria chegado até aqui sem eles.

Agradeço ao nosso orientador, Leonardo Azevedo, primeiramente por suas aulas que foram uma das melhores que tive na universidade e também pela dedicação e paciência que foram primordiais para a evolução deste trabalho. Agradeço ao nosso coorientador e amigo Raphael Rodrigues por toda a dedicação demonstrada.

Agradeço ao meu amigo e companheiro neste trabalho Victor Azevedo, por todas as horas de estudo, trabalhos e momentos difíceis que passamos durante os últimos anos. É um prazer enorme poder compartilhar este momento da minha vida com um amigo como você.

Agradeço o corpo docente da UNIRIO pelo conhecimento que nos foi transmitido, trilhando todo o caminho que seguimos para chegar até aqui. Uma menção especial aos professores Alexandre Correa, Marcio Barros, Sean Siqueira e Gleison Santos por terem proporcionado grandes aulas e pela dedicação ao seu trabalho que é feito de maneira exemplar.

Impossível deixar de agradecer aos profissionais com que tive o imenso prazer de trabalhar durante os últimos anos e que foram muito além de mentores e companheiros de trabalho foram grandes amigos que pude fazer. Em especial Sérgio Ruivace, João Pedro, João Manoel e Marcos Mele.

Por fim, agradeço aos meus colegas de graduação com os quais dividi momentos de felicidade, ansiedade e até de raiva durante esses anos. Em especial Daniel Machado, Sandro Silveira e William Brum. Obrigado por acreditarem em mim e pela ajuda essencial nos momentos em que precisei.

Carlos Magno Coutinho de Sena

Agradecimentos

Em primeiro lugar agradeço minha família por todo empenho e dedicação empregados na minha formação desde o início da minha vida.

Agradeço também o meu orientador, Leonardo Azevedo, por suas aulas que tanto me inspiraram, e por sua paciência e bom humor ao transmitir o conhecimento.

Agradeço o amigo e coorientador deste trabalho, Raphael Rodrigues, por nos acompanhar e instruir, e cuja dedicação foi imprescindível para a conclusão do mesmo.

Por fim agradeço a todos os amigos e colegas que me apoiaram durante esses anos de batalha na graduação, destacando Marcos Mele, Sérgio Cerqueira, Janaína Horácio, Daniel Machado, Alessandro Iglesias, Alessandro Campello, William Brum e em especial o grande amigo e companheiro de estudo Carlos Magno.

Victor Fortunato Azevedo

SUMÁRIO

CAPÍTULO 1: INTRODUÇÃO	13
1.1 CONTEXTUALIZAÇÃO E MOTIVAÇÃO	13
1.2 OBJETIVO DO TRABALHO	14
1.3 CONTRIBUIÇÕES.....	15
1.4 ESTRUTURA DO TRABALHO	15
CAPÍTULO 2: PRINCIPAIS CONCEITOS.....	16
2.1 ANÁLISE ESTÁTICA DE CÓDIGO	16
2.2 ANÁLISE LÉXICA E SINTÁTICA.....	17
2.2.1 Árvore Sintática Abstrata (AST).....	18
2.3 PADRÃO VISITOR	21
2.4 ARQUITETURA DE SISTEMAS.....	22
2.5 RESUMO DO CAPÍTULO	24
CAPÍTULO 3: INSPEÇÃO DE CÓDIGO	25
3.1 REVISÃO DE SOFTWARE.....	25
3.2 INSPEÇÃO.....	25
3.2.1 Checklist para inspeção de código.....	26
3.2.2 Processo de inspeção de código.....	26
3.3 AUTOMATIZAÇÃO DE INSPEÇÃO DE CÓDIGO.....	28
3.4 RESUMO DO CAPÍTULO	29
CAPÍTULO 4: PLUG-IN PARA ANÁLISE ESTÁTICA DE CÓDIGO	30
4.1 PROPOSTA DE AUTOMATIZAÇÃO.....	30
4.2 TECNOLOGIAS DE APOIO	32
4.2.1 SonarQube.....	32
4.2.2 Apache Maven	34
4.3 ESTRUTURA GENÉRICA DE ANÁLISE DE CÓDIGO UTILIZANDO UM PLUG-IN	35
4.4 PLUG-IN DE REGRAS PARA A FERRAMENTA SONARQUBE.....	37
4.5 ARQUITETURA DO PROJETO ANALISADO	41
4.6 REGRAS DO CHECKLIST	43
4.7 REGRAS DE NOMENCLATURA	50
4.7.1 Vocabulário da Língua Portuguesa.....	50
4.7.2 Algoritmo de análise de nomenclatura.....	52
4.7.3 Auxiliar de Nomenclatura	57

4.8	ALGORITMOS DE REGRAS GERAIS	58
4.8.1	<i>Nomenclatura de Pacotes</i>	58
4.8.2	<i>Nomenclatura de Classes</i>	60
4.8.3	<i>Nomenclatura de Métodos</i>	60
4.9	ALGORITMOS DE REGRAS DA CAMADA DE APRESENTAÇÃO	62
4.9.1	<i>Uso de Controlador Genérico</i>	62
4.9.2	<i>Localização e Sufixo de Controladores</i>	62
4.10	ALGORITMOS DE REGRAS DA CAMADA DE SERVIÇOS	63
4.10.1	<i>Nomenclatura de Serviços</i>	63
4.10.2	<i>Log interceptador na camada de Serviços</i>	64
4.11	ALGORITMOS DE REGRAS DA CAMADA DE DOMÍNIO	65
4.11.1	<i>Localização e Nomenclatura de Classes Value Objects</i>	65
4.11.2	<i>Mapeamento Relacional Lazy x Eager</i>	66
4.12	REGRAS DA CAMADA DE INFRAESTRUTURA	66
4.12.1	<i>Tecnologia de Acesso a Dados do Repositório Genérico</i>	67
4.12.2	<i>Localização de Classes de envio de e-mail</i>	67
4.13	RESUMO DO CAPÍTULO	68

CAPÍTULO 5: AVALIAÇÃO DO PLUG-IN PARA ANÁLISE ESTÁTICA DE CÓDIGO

69

5.1	PROJETO ANALISADO	69
5.2	DEMONSTRAÇÃO DOS RESULTADOS ENCONTRADOS.....	71
	• <i>Violação da Regra 1 - Nomenclatura de Pacotes</i>	71
	• <i>Violação da Regra 3 - Nomenclatura de Classes</i>	72
	• <i>Violação da Regra 5 - Nomenclatura de Métodos</i>	72
	• <i>Violação da Regra 8 – Uso de Controlador</i>	73
	• <i>Violação da Regra 11 – Sufixo de classe Controller</i>	73
	• <i>Violação da Regra 16 – Nomenclatura de Serviços</i>	73
	• <i>Violação da Regra 19 – Log Interceptador na camada de Serviços</i>	74
	• <i>Violação da Regra 22 – Localização de Classes com sufixo Vo</i>	75
	• <i>Violação da Regra 24 – Mapeamento Relacional Lazy x Eager</i>	75
	• <i>Violação da Regra 25 – Sufixo do Repositório Genérico</i>	76
	• <i>Violação da Regra 26 – Localização de classes de envio de e-mail</i>	76
5.2.1	<i>Falsos Positivos</i>	77

CAPÍTULO 6: CONCLUSÃO

79

CAPÍTULO 7: REFERÊNCIAS BIBLIOGRÁFICAS.....

81

APÊNDICE I – ALGORITMOS PARA VALIDAÇÃO DAS REGRAS

84

LISTA DE ABREVIATURAS

AST – Abstract Syntax Tree (Árvore Sintática Abstrata)

VO – Value Object (Objeto de valor)

LTS – Long-term support (Suporte em longo prazo).

DTO – Data Transfer Object (Objeto de transferência de dados)

LISTA DE FIGURAS

Figura 1 - Análise Léxica e Sintática de Código	18
Figura 2 - Árvore Sintática Abstrata.....	19
Figura 3 - Visualização de uma AST.....	21
Figura 4 - Processo de Inspeção (Adaptado de [Laitenberger, 2000])	27
Figura 5: Estrutura da Ferramenta <i>SonarQube</i>	33
Figura 6 - Arquivo pom.xml do plug-in desenvolvido.	35
Figura 7 - Atividades Genéricas para Análise	36
Figura 8 - Diagrama de sequência para criação das regras.....	39
Figura 9 - Diagrama de Sequência para execução das regras.....	40
Figura 10 - Estrutura de pacotes	40
Figura 11 – Principais camadas da arquitetura base	42
Figura 12 - Estrutura do vocabulário	51
Figura 13 - Exemplo de arquivo do vocabulário de adjetivos	52
Figura 14 - Exemplo de arquivo de adjetivos do vocabulário de exceção	52
Figura 15 - Grafo para nomenclatura de métodos	54
Figura 16 - Grafo para nomenclatura de classes.....	54
Figura 17 – Interface Algoritmo de análise de nomenclatura.....	57
Figura 18 - Implementação de fluxo de palavras para nomenclatura de métodos..	57
Figura 19 - Árvore de exemplo para classe <i>Jogo</i>	59
Figura 20 - Estrutura da Aplicação	70
Figura 21- Página Inicial do sistema analisado	70
Figura 22 - Visão geral da análise	71
Figura 23 - Discrepância de nomenclatura de pacotes	71
Figura 24 - Discrepância de nome de classe I	72
Figura 25 - Discrepância de nomenclatura de método	73
Figura 26 - Discrepância de não uso de controlador	73
Figura 27 - Discrepância de sufixo <i>Controller</i>	73
Figura 28 - Discrepância de nomenclatura de serviço.....	74
Figura 29 - Discrepância na localização do log interceptor.....	74
Figura 30 - Discrepância de localização de <i>value objects</i>	75
Figura 31 - Quantidade de ocorrências de <i>Eager</i>	75
Figura 32 - Discrepância de mapeamento <i>eager</i>	76

Figura 33 - Discrepância no sufixo do repositório genérico.....	76
Figura 34 - Discrepância na classe de envio de e-mail.....	77
Figura 35 - Falso positivo de nomenclatura de log.....	77
Figura 36 - Falso positivo para discrepância de nomenclatura de métodos	78
Figura 37 - Exemplo de discrepância a ser analisada	78

LISTA DE APÊNDICES

Apêndice I – Algoritmos para validação das regras	84
---	----

RESUMO

Todo software que possui uma arquitetura de referência está sujeito à degradação da mesma durante seu ciclo de desenvolvimento. Os motivos são diversos: Alta rotatividade e curva de aprendizado dos desenvolvedores, alteração de tecnologias utilizadas, evolução do software, dentre outros. Assim é comum que com o passar do tempo o código fonte do software perca a organização projetada anteriormente, o que pode acarretar em violações arquiteturais e dificultar a legibilidade do mesmo.

Uma maneira de combater a deterioração da arquitetura de um software é inspecionar seu código fonte com vistas a buscar falhas arquiteturais. Entretanto, com a crescente complexidade dos sistemas, é comum a existência de aplicações que contenham quantidade de linhas de código na casa dos milhões. Em softwares deste porte a inspeção manual se torna inviável, uma vez que para tal seria necessária uma grande equipe de inspetores e um tempo considerável para executar a atividade.

Para contornar este impasse, o presente trabalho apresenta uma proposta para a automatização de validação de regras arquiteturais através de análise estática de código. Para isso, foram elaborados algoritmos que validam um conjunto de regras arquiteturais de uma dada arquitetura em um código fonte. Estes algoritmos foram implementados através da criação de um de *plug-in* para a plataforma de análise estática *SonarQube*.

Para avaliar a solução projetada, o *plug-in* analisou o código fonte de um sistema real construído na linguagem Java e baseado na arquitetura de referência cujas regras o *plug-in* automatiza a validação.

Como resultado, a solução proposta se mostrou eficaz na medida em que identificou violações arquiteturais legítimas, demonstrando a viabilidade da execução da análise estática de código utilizando os algoritmos em outros sistemas que obedeçam uma determinada arquitetura.

Palavras-chave: Análise Estática de Código, Arquitetura, Regras Arquiteturais, SonarQube, Inspeção de código, Automatização de inspeção de código.

Capítulo 1: INTRODUÇÃO

1.1 Contextualização e Motivação

Com a crescente necessidade da informatização de sistemas, abrangendo até mesmo os de domínio mais simples, o desenvolvimento de software em ambientes corporativos se vê em alta. De modo a padronizar o desenvolvimento e viabilizar a manutenibilidade de seus códigos-fonte, diversas organizações definem arquiteturas de software. Nesse contexto, a aplicação da técnica de inspeção nos códigos produzidos possui como objetivo garantir que a arquitetura pré-definida pela organização e a legibilidade de seus softwares sejam mantidas durante o ciclo de desenvolvimento.

A arquitetura de software desempenha o papel de gerenciar a complexidade relativa ao software a ser desenvolvido. Segundo Garlan (2000), “Uma arquitetura de software envolve a descrição de elementos arquiteturais dos quais os sistemas serão construídos, interações entre esses elementos, padrões que guiam suas composições e restrições sobre estes padrões”.

A inspeção de código é a maneira mais adotada de revisão de artefatos encontrada em projetos de software [LAITENBERGER e DEBAUD, 2000] e traz muitos benefícios para o processo da garantia da qualidade do desenvolvimento de software. Apesar da análise estática de código exercer papel crítico dentro do ciclo de desenvolvimento de um software no que tange à garantia da qualidade, esta avalia o código fonte do sistema, e não o produto final (software). Essa qualidade é definida pelo quanto o código respeita a arquitetura definida, não tendo influência diretamente na qualidade funcional do produto.

O código implementado, através de uma linguagem de programação, é um componente facilmente manipulável. Portanto, garantir que um conjunto de regras estabelecidas sejam seguidas pelos desenvolvedores é uma tarefa de alta complexidade e demanda muito tempo, especialmente se considerarmos que as inspeções são feitas manualmente. Outro fator determinante, que contribui para essa complexidade, é a possibilidade de uma equipe de desenvolvimento sofrer diversas mudanças durante o ciclo de vida do projeto, dificultando o cumprimento das regras.

Devido ao grande volume de código que pode ser gerado em sistemas complexos, a inspeção de código feita de forma manual se torna uma tarefa inviável cujos resultados podem não ser confiáveis. Portanto é necessário pensar em soluções que atendam essas inspeções de forma automatizada.

A inspeção automatizada se propõe a realizar análises estáticas em códigos de diversas linguagens de forma a tornar esta tarefa viável e menos custosa. Esta análise consiste em varrer as linhas do artefato em busca de discrepâncias relacionadas a um conjunto de regras pré-estabelecidas na ferramenta responsável pela execução. Estas regras podem ser relativas à validação na nomenclatura do código, acesso entre camadas, quantidade de documentação e nível de cobertura de testes unitários, dentre outras métricas.

Ferramentas de análise estática de código possuem um conjunto de regras pré-estabelecido e, apesar da alta quantidade, dificilmente conseguem cobrir itens específicos da arquitetura de uma organização, e muito menos itens referentes à legibilidade, visto que esses são dependentes da língua em que o código do software é desenvolvido. Isto torna imprescindível que a inspeção automatizada do artefato de código possua uma maneira de realizar a validação nestes itens, visto que a inspeção manual poderia cobrir.

A solução para este problema é o ponto central deste trabalho. Para isso, será realizado um estudo de como implementar algoritmos de validação de regras arquiteturais e de nomenclatura de componentes através da análise estática de código e será codificado um *plug-in* direcionado à ferramenta *Sonarqube* contendo os algoritmos responsáveis por realizar esta atividade. A ferramenta foi escolhida por contemplar um conjunto de regras de boas práticas, além de fornecer uma forma de extensão para criação de regras customizadas.

1.2 Objetivo do trabalho

Este trabalho tem como objetivo solucionar o problema da automatização de código em itens específicos de uma arquitetura. A solução está na customização de uma ferramenta de análise de código através da elaboração de algoritmos que garantam a conformidade de código-fonte com a arquitetura do sistema.

O trabalho se limitou a um subconjunto de regras que validam a arquitetura de referência, além de esbarrar em algumas limitações próprias do desenvolvimento de soluções de análise estática de código, por exemplo, a análise do código é feita de arquivo em arquivo, e impossibilita que em tempo de execução se obtenha informações de um dado arquivo quando outro está sendo analisado.

1.3 Contribuições

Este trabalho possui como contribuição um projeto arquitetural para a inspeção de código considerando as seguintes características:

- *Plug-in* codificado na linguagem Java para análise de código-fonte em Java;
- Documentação de todas as regras implementadas;
- Algoritmo de análise de nomenclatura para a língua portuguesa em nomes de classes e métodos.

1.4 Estrutura do trabalho

Este trabalho está dividido em seis capítulos. O Capítulo 1 corresponde à presente introdução. O Capítulo 2 apresenta os principais conceitos relacionados a este trabalho. O Capítulo 3 apresenta a inspeção de código, base sobre a qual o resultado deste trabalho atua, mais especificamente sobre as técnicas de revisão de software e o processo de inspeção de código. O Capítulo 4 apresenta a proposta deste trabalho, demonstrando a extensão da ferramenta *SonarQube* para a inspeção de código em uma determinada arquitetura de software, exibindo os algoritmos que foram elaborados na construção do *plug-in*, enquanto o Capítulo 5 apresenta a avaliação da proposta. Finalmente, no Capítulo 6 são apresentadas as conclusões do trabalho e propostas de trabalhos futuros.

Capítulo 2: PRINCIPAIS CONCEITOS

Este capítulo define e apresenta os conceitos que estão diretamente relacionados a este trabalho.

2.1 Análise Estática de Código

Análise estática do código é a investigação do comportamento e semântica da aplicação de software feita unicamente com base na leitura do código. Neste tipo de varredura, o código não é executado no momento da análise. Esta prática tem como objetivo central o entendimento da estrutura da aplicação e de seu comportamento com base nas estruturas e comunicação entre os componentes da linguagem.

O resultado deste tipo de análise é um entendimento ou um “mapa mental” dos elementos de uma aplicação, seus atributos e suas dependências. Na maioria dos casos, dentro do processo de desenvolvimento, a análise estática é feita em um dado momento de release da aplicação [IBM, 2013].

Existem diversas ferramentas que fazem uso da técnica de análise estática de código para apoiar o processo de inspeção do artefato, funcionando da seguinte maneira: (i) O código fonte é lido, linha a linha, em busca de potenciais erros; (ii) A ferramenta posteriormente gera um relatório que contenha essas possíveis discrepâncias. Cabe ao desenvolvedor verificar a veracidade do que foi apontado e tomar as devidas providências.

A análise estática se diferencia de uma análise dinâmica da aplicação, na medida em que não requer casos de teste e, como citado anteriormente, pode ser executada sem a aplicação funcional. Para se analisar estaticamente uma aplicação é necessária uma ferramenta de apoio que forneça funcionalidades relativas a essa atividade, tais como:

- Leitura de código
- Armazenamento das discrepâncias detectadas
- Exibição das discrepâncias encontradas através de relatórios – em formatos como HTML ou PDF – e até marcações no próprio código em uma IDE (através de *plug-in*).

Visto que uma linguagem de programação é composta por diversos elementos de caráter léxico e sintático, o entendimento programático da estrutura de uma linguagem na qual um código está codificado, passa pela necessidade de conhecer quais os elementos que compõe a mesma e de que forma esses elementos podem estar dispostos dentro de um código. Por exemplo, se um determinado bloco de código corresponde a uma declaração de classe, interface ou assinatura de um método. Uma estrutura de dados capaz de representar esses elementos é a *Árvore de Sintaxe Abstrata (AST)* (Seção 2.2.1), que é gerada através da análise léxica e posteriormente análise sintática.

2.2 Análise Léxica e Sintática

O processo de análise passa pelo entendimento dos conceitos de alfabeto, linguagem e gramática. Menezes [2000] apresenta que um alfabeto é um conjunto finito de símbolos e que uma linguagem é um conjunto de palavras formadas sobre um alfabeto. Uma gramática pode ser definida de forma informal e simplificada como um sistema gerador de linguagens e uma maneira de representá-las.

A análise léxica tem por objetivo ler a entrada textual, caractere por caractere, procurando pela presença dos elementos léxicos da linguagem que a análise se propõe a fazer, ou seja, por membros do alfabeto dessa linguagem, como por exemplo, as palavras reservadas da linguagem Java: *class*, *interface*, *public*, *private*, etc.

Podem ser utilizados dois métodos para identificar os elementos do alfabeto, seja através da utilização de uma tabela de símbolos que contenha os identificadores e palavras-chave ou através de uma descrição dos símbolos feita por expressões regulares. Vale ressaltar que este tipo de análise não leva em consideração caracteres em branco, tabulações e quebras de linha.

O objetivo da análise léxica é, ao final da varredura do código, transformar a sequência textual lida em uma sequência de símbolos, chamados de *tokens*, que serão utilizados pela análise sintática.

O objetivo do analisador sintático é obter a cadeia de *tokens* provenientes das chamadas realizadas ao analisador léxico e validar se a cadeia pode ser gerada com base em uma gramática definida. Espera-se que o analisador sintático tenha uma

forma de reportar os erros e, em caso de sucesso, gere uma estrutura de dados que represente a sequência de *tokens*. Uma estrutura comumente utilizada é a *Árvore Sintática Abstrata* (Seção 2.2.1).

A análise sintática (do inglês, *parsing*) consiste, portanto, no processo de estruturar, de alguma forma, uma determinada representação linear de acordo com uma gramática conhecida.

Os passos para realizar as análises léxica e sintática são ilustrados pela Figura 1.

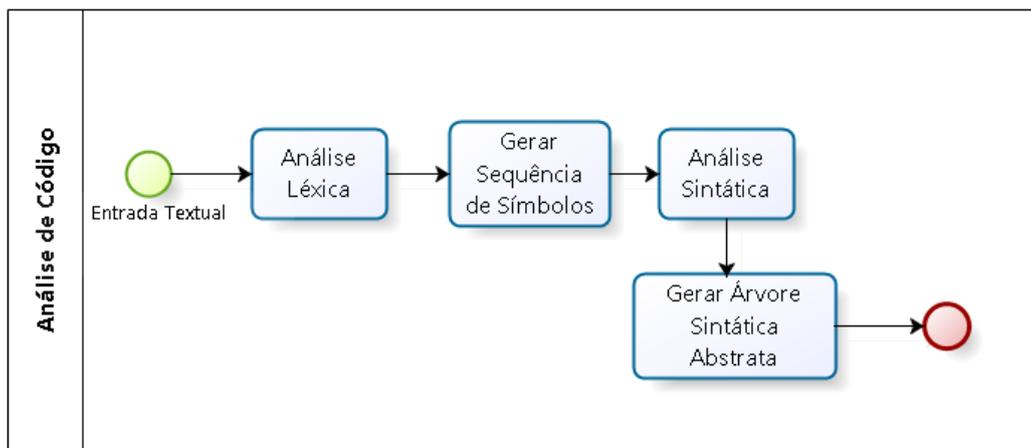


Figura 1 - Análise Léxica e Sintática de Código

2.2.1 *Árvore Sintática Abstrata* (AST)

A *Árvore Sintática Abstrata* (do inglês, *Abstract Syntax Tree*) é uma estrutura de dados, gerada após a análise dos símbolos (*tokens*), que se propõe a ser uma representação abstrata da estrutura de um código-fonte que foi escrito em uma determinada linguagem de programação. Através da mesma, é possível navegar entre os nós e obter o valor textual de cada *token*. Ou seja, é possível realizar uma análise na estrutura do código-fonte contido em um arquivo, sendo possível acessar a declaração de classes, variáveis, chamada de funções, dentre outras estruturas de uma linguagem.

A construção de uma *Árvore Sintática Abstrata* é feita após a análise léxica que coleta os símbolos da entrada textual e a consequente análise sintática, que verifica se os símbolos podem ser lidos de acordo com uma gramática. O Código 1 é um exemplo de código em *JavaScript* em que há somente uma declaração de variável e uma nova atribuição a mesma.

```
var teste = "TesteAST"
teste = "Novo Valor"
```

Código 1- Exemplo simples para geração de AST

Após a identificação dos elementos léxicos, como por exemplo, a palavra reservada *var*, é feita a análise sintática responsável por gerar uma estrutura de dados que possuirá diversos tipos de nós responsáveis por representar as diversas estruturas encontradas. No exemplo mencionado, o corpo da declaração de uma variável será um nó do tipo *VariableDeclaration* e a declaração da variável “*teste*” estará representada por um nó do tipo *VariableDeclarator* (Figura 2).

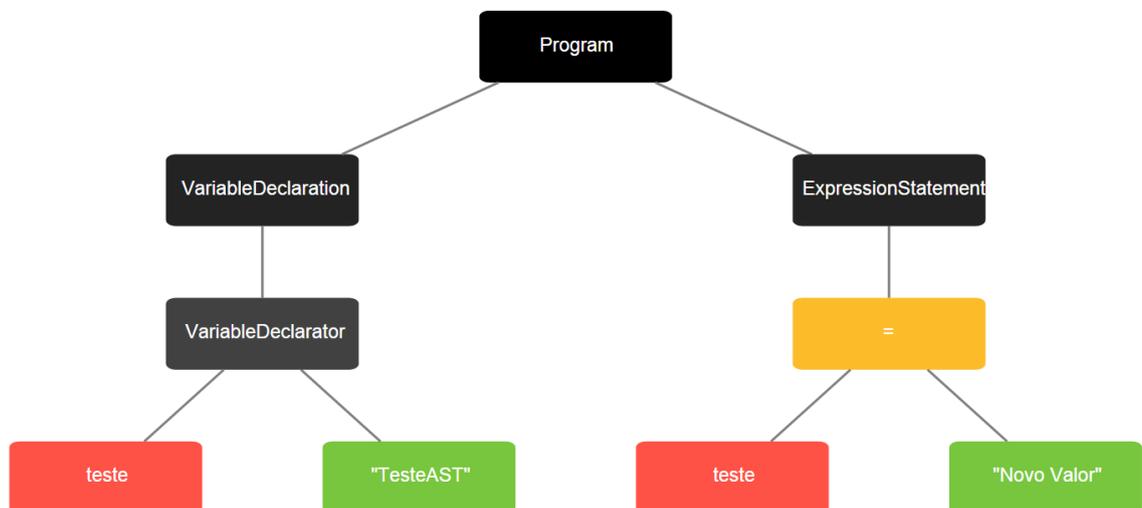


Figura 2 - Árvore Sintática Abstrata

Existem diversos *frameworks* disponíveis que realizam a formação de uma árvore sintática abstrata com base em um determinado código-fonte e que também fornecem uma API para acesso aos nós da árvore e navegação da mesma. Cada *framework* é baseado em uma determinada gramática geradora da linguagem a ser analisada, portanto só é possível gerar a árvore caso a análise sintática entenda que os elementos simbólicos podem ser lidos de acordo com a gramática.

Para um exemplo mais complexo, considere o Código 2 escrito na linguagem Java.

```
package br.unirio.tcc.astview;
/**
```

```

* Classe de exemplo para visualização da AST
* @author CarlosMagno
* @author Victor Azevedo
*
*/
public class GeradorAST {

    private String atributoDaClasse;

    public GeradorAST(String atributoDaClasse) {
        this.setAtributoDaClasse(atributoDaClasse);
    }

    public String getAtributoDaClasse() {
        return atributoDaClasse;
    }

    public void setAtributoDaClasse(String atributoDaClasse) {
        this.atributoDaClasse = atributoDaClasse;
    }

}

```

Código 2 - Exemplo para visualização da AST

A ferramenta Java SSLR Toolkit¹ possibilita a visualização de uma AST. Com base no código acima a ferramenta realiza uma análise léxica coletando os *tokens* e uma posterior análise sintática que verifica se o código-fonte está de acordo com a gramática geradora da linguagem Java. Com a árvore gerada é possível navegar pelos nós e assim obter informações sobre o código.

¹Download disponível em <http://mvnrepository.com/artifact/org.codehaus.sonar-plugins.java/sslr-java-toolkit/2.4>

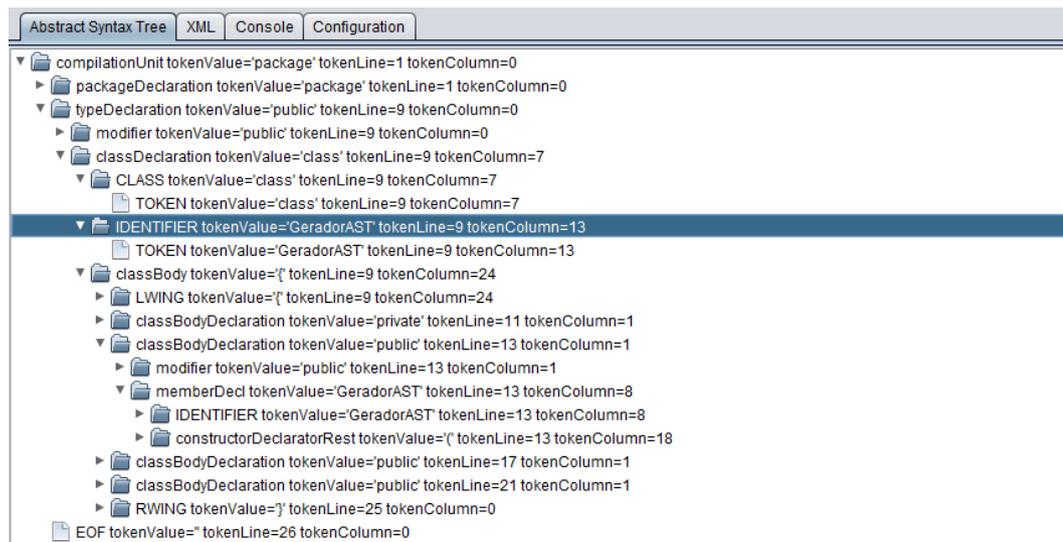


Figura 3 - Visualização de uma AST

Na Figura 3 é possível visualizar a árvore e o caminho percorrido até chegar ao nó que possui como *tokenValue* a informação textual do nome da classe (*GeradorAST*) que foi declarada. Para a finalidade deste trabalho a geração da árvore sintática abstrata traz diversas informações que serão úteis, por exemplo, nós que contém informações como o número da linha e coluna.

2.3 Padrão Visitor

O acesso aos elementos de uma árvore sintática abstrata pode ser feito de diversas formas, mas é comum que as *APIs* fornecidas tenham como base a utilização do padrão de projeto *Visitor*. Segundo Gamma [2000], padrões de projeto são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular.

Para estender a funcionalidade de uma classe que possui uma hierarquia de subtipos, basta adicionar métodos com o comportamento desejado. Porém, em alguns cenários a alteração do comportamento pode ser inconsistente com o modelo de objetos definido. Em outros casos, os desenvolvedores responsáveis pela criação da hierarquia não poderão alterar o código rapidamente. Em cenários mais simples, pode-se desejar percorrer toda a hierarquia e obter informações específicas, que variam de acordo com o subtipo específico de cada objeto. Nestes cenários, o padrão Visitor pode e deve ser utilizado [LASATER, 2010].

Este padrão permite que o desenvolvedor da hierarquia construa seu modelo levando em consideração que outros desenvolvedores poderão estender a funcionalidade da mesma, com comportamentos que não necessariamente correspondam ao modelo inicial ou do negócio. Em suma, o principal objetivo deste padrão é definir uma nova operação para uma hierarquia sem que seja necessária a alteração das classes presentes nesta hierarquia [LASATER, 2010].

Neste projeto, o padrão foi utilizado para percorrer todos os nós de uma árvore, cujo tipo genérico é um nó da AST que é representado pela interface *Tree*. Porém as informações relevantes ficam armazenadas nos subtipos da hierarquia como um nó do tipo *MethodDeclaration*, que herda de *Tree*. Através do *Visitor*, podemos “visitar” somente os nós que são relevantes para a automatização da inspeção do código.

2.4 Arquitetura de Sistemas

O termo Arquitetura de Software é uma das grandes subdisciplinas que compõem a Engenharia de Software. Ela surgiu assim que os primeiros softwares começaram a serem divididos em módulos, fazendo assim que os programadores se tornassem os responsáveis pelas interações entre os módulos e as propriedades globais do conjunto [SHAW e GARLAN, 1996].

De uma maneira geral, arquitetura é a divisão de um todo em suas partes, com relações específicas entre elas [BACHMANN *et al.*, 2010]. Para explicar o que é de fato a arquitetura de software, muitos autores recorrem a uma analogia com a arquitetura de um edifício, onde esta é caracterizada pela maneira que os vários componentes do edifício são integrados para formar um todo coeso, o modo com que ele se ajusta em seu ambiente e integra com outros edifícios da vizinhança e o grau com que ele atende seu propósito expresso e satisfaz às necessidades de seu proprietário [PRESSMAN 2011]. Assim, a arquitetura não é o software operacional, e sim uma representação de suas partes, como elas interagem entre si e como estão estruturadas.

Entretanto, apesar das definições citadas, não existe uma formal oficial em que todos os autores concordem. O site do Instituto de Engenharia de Software (SEI - *Software Engineering Institute*) coleta definições da literatura e de diversos

praticantes ao redor do mundo. Para este assunto, mais de 150 definições já foram coletadas [BACHMANN *et al.*, 2010].²

Fowler [2002], numa tentativa de justificar a subjetividade intrínseca do termo, cita dois elementos em comum entre as diversas definições existentes:

- Simplificação em alto nível das partes de um sistema;
- Representação de decisões que são difíceis de mudar e, portanto, gostariam de tomar o quanto antes no início do desenvolvimento do projeto.

O autor afirma que um sistema pode possuir diversas arquiteturas dentro de si e o julgamento do que é considerado arquiteturalmente significativo pode se modificar conforme o tempo, bem como o que é percebido como algo difícil de se alterar em um dado momento, pode se tornar facilmente alterável no futuro. Assim, arquitetura tem a ver com partes importantes em um sistema em um dado momento no tempo [FOWLER, 2002]. Essas partes importantes podem ser manifestadas de diversas formas em um sistema, por exemplo:

1. Estratégia padrão adotada para resolver um determinado tipo de problema, isto é, o uso de algum *design pattern*. Por exemplo, o uso do padrão de camadas [Fowler 2002] para atingir a inversão de controle, considerado imprescindível no desenvolvimento de qualquer sistema web moderno.
2. Frameworks adotados no projeto. A utilização de *frameworks* confere agilidade no desenvolvimento de sistemas, visto que executam algum tipo trabalho ou resolvem um problema genérico independente do domínio, isto é, são mais que um bloco reutilizável, mas subsistemas inteiros reutilizáveis, de maneira que possuem uma arquitetura própria [Pree 2006]. Apesar de ser independente do domínio, um *framework* impacta diretamente na arquitetura de um sistema, uma vez que define a maneira e o contrato de como ele deve ser utilizado (API's).

² Lista de definições disponível em

<http://www.sei.cmu.edu/architecture/start/glossary/community.cfm>

3. Tecnologias utilizadas. A adoção de uma determinada tecnologia influencia a estruturação interna de um sistema, visto que pode adicionar ou remover responsabilidades tratadas no sistema.

Com base nos três exemplos citados, é fácil inferir outra característica intrínseca sobre a arquitetura de software: Elas definem regras implícitas ou explícitas. Uma regra arquitetural pode ter sua origem justificada em inúmeras manifestações das partes importantes de um sistema. Assim, ela pode ditar a maneira como um determinado problema deve resolvido, como um *framework* deve ser utilizado (ou não utilizado), etc.

Muitas vezes uma regra arquitetural explícita, ou no pior dos casos, implícita, não é suficientemente clara, o que pode levar a interpretações ambíguas ou até mesmo ser completamente ignorada, e, conseqüentemente, podendo trazer resultados catastróficos para o desenvolvimento do sistema. A construção de um *checklist* arquitetural é boa uma prática muito importante que combate estas possíveis confusões, descrevendo de maneira clara todas as regras contempladas.

O conceito de regra arquitetural é o conceito relacionado à arquitetura de software mais importante deste trabalho, uma vez que são essas regras em específico que serão verificadas automaticamente pelo *plug-in* desenvolvido.

2.5 Resumo do capítulo

O presente capítulo apresentou os conceitos que fundamentam de que maneira é realizada a análise estática de um código-fonte que segue uma determinada arquitetura corporativa para desenvolvimento de sistemas, com o objetivo de identificar falhas arquiteturais para a garantia da conformidade entre código e arquitetura. O capítulo seguinte abordará a inspeção de código e a forma como ela atua na revisão de software enquanto atividade dentro do processo de desenvolvimento, que é o item proposto para automatização.

Capítulo 3: INSPEÇÃO DE CÓDIGO

O presente capítulo apresentará uma visão geral sobre os conceitos de revisão de software e inspeção de código que por sua vez é a principal conceituação que envolve o trabalho desenvolvido.

3.1 Revisão de Software

Presmann [2011] define que revisões de software são como um “filtro” para a gestão de qualidade e são aplicadas em várias etapas durante o processo de engenharia de software, servindo para revelar erros e defeitos que podem ser eliminados. A prática de revisão se trata de uma grande aliada na garantia da qualidade de software e traz como benefício a diminuição de custos no desenvolvimento do software se aplicada regularmente [PRESSMAN 2011; FAGAN, 1986].

As técnicas de revisão são de grande importância para o processo de desenvolvimento como um todo. Freedman e Weiberg citados por Pressman (2011, p.373) definem que “O trabalho técnico precisa de revisão pela mesma razão que o lápis precisa de borracha: Errar é humano”. Existem diversas formas de se realizar uma revisão de um software, cada qual com sua devida particularidade. Segundo Yourdon (2006) as técnicas de apoio à revisão mais conhecidas são o *Walkthrough*, uma forma de revisão por pares de qualquer produto técnico através de uma “caminhada” sobre o sistema e a inspeção.

3.2 Inspeção

Schach [2008] relata que as atividades das inspeções foram propostas inicialmente por Fagan [1976]. Fagan [1976] define que inspeções são métodos eficientes, formais e econômicos para encontrar erros no design e no código e ainda argumenta a necessidade de uma equipe de apoio para realização destas atividades, indicando um número ideal de quatro pessoas. Segundo Schach [2008], o padrão do IEEE prevê uma equipe formada de três a seis pessoas.

A inspeção pode ter diversos objetivos como, por exemplo, a nomenclatura dos componentes, o acesso entre os mesmos, a legibilidade do código em si, a documentação dos métodos e propriedades utilizadas, a organização e a codificação no que diz respeito às boas práticas da linguagem abordada, sendo inclusive possível encontrar alguns erros que podem preceder *bugs* no aspecto funcional da aplicação. Esta prática permite a uma equipe de desenvolvimento avaliar a sua maneira decodificar e tomar decisões de como alterar, refatorar ou até mesmo consertar a aplicação em aspectos arquiteturais ou funcionais.

3.2.1 Checklist para inspeção de código

Para apoiar a atividade de inspeção, é necessário que haja uma técnica de leitura que permita sistematizar o trabalho da equipe responsável e para que todos saibam quais os itens devem ser analisados e levados em consideração. As técnicas mais conhecidas e frequentemente utilizadas para leitura são a *ad-hoc* e o *checklist*. O método *ad-hoc* não oferece muito suporte ao especialista que está realizando a inspeção, visto que neste método de leitura, o inspecionador conta somente com suas habilidades e experiência, não tendo, portanto, nenhuma técnica sistemática bem definida de como a leitura deve proceder. Por sua vez, o *checklist* representa uma forma mais estruturada [Porter, *et al*].

Koskinen e Kollanus [2007] afirmam que o método de inspeção original, apresentado por Fagan [1976], incluía a ideia de utilização de um *checklist* com a finalidade de encontrar erros e auxiliar a equipe de inspeção onde todos utilizam do mesmo *checklist* para realizar a validação do código. A utilização desta técnica é de suma importância para o aprimoramento da atividade de inspeção na medida em que serve de guia para os inspecionadores e segundo Porter *et al*, a confecção dos *checklists* pode capturar importantes lições aprendidas através de inspeções anteriores.

3.2.2 Processo de inspeção de código

Para ilustrar um processo de inspeção de código genérico, podemos utilizar o mesmo proposto por Laitenberger [2000], que levantou seis fases características em diversos processos de inspeção, presente em diferentes metodologias (Figura 4).

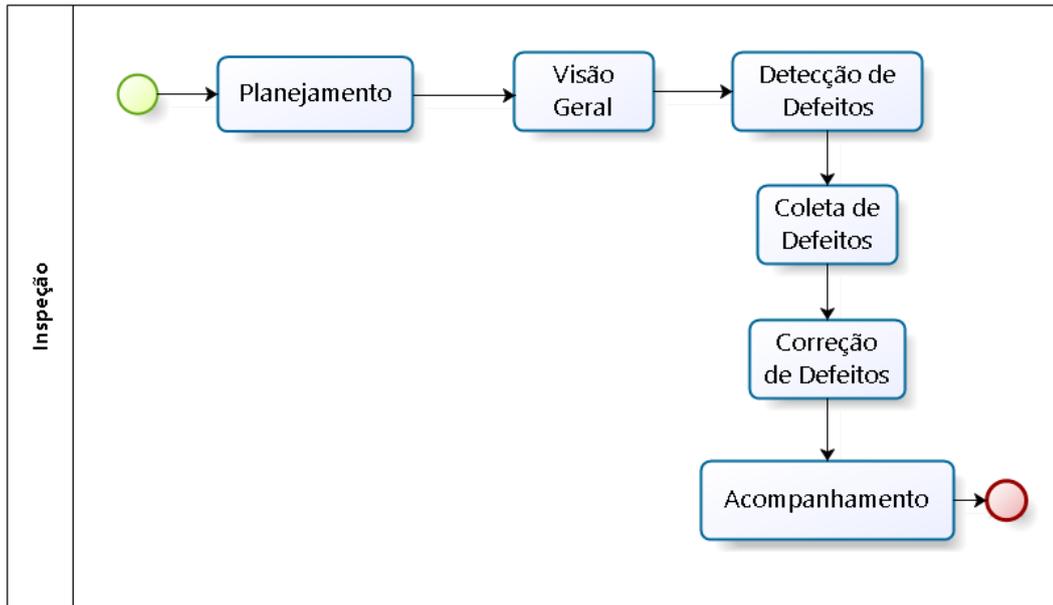


Figura 4 - Processo de Inspeção (Adaptado de [Laitenberger, 2000])

Uma breve descrição de cada etapa é apresentada abaixo:

1) Planejamento

O objetivo da fase de planejamento é organizar uma inspeção referente ao material a ser inspecionado. A fase inclui a seleção dos participantes da inspeção, seus papéis, agendamento de reuniões de inspeção e a distribuição do material.

2) Visão Geral

Essa fase consiste de uma reunião inicial onde o autor do material explica o produto sujeito a inspeção para os outros participantes da inspeção. O objetivo central é fazer com que a compreensão acerca do produto esteja bem clara. Entretanto, no caso da inspeção de códigos, Laitenberger menciona que essa fase não é obrigatória.

3) Detecção de Defeitos

Essa fase é o coração da inspeção. O principal objetivo da fase é dissecar o código para extrair defeitos.

4) Coleta de Defeitos

O objetivo central desta fase é documentar os defeitos levantados na fase anterior. Está incluso também tomar decisões sobre a validade de um defeito e se uma

inspeção deve ser feita novamente. Normalmente essa fase é realizada em reuniões do grupo de inspeção, visto que essas decisões normalmente são tomadas em conjunto.

5) Correção de Defeitos

Esta é a fase que o autor do material deve retrabalhar e corrigir o código e lidar com cada defeito reportado pelo time de inspeção.

6) Acompanhamento

A fase de acompanhamento tem como objetivo verificar se o autor corrigiu todos os defeitos levantados anteriormente.

3.3 Automatização de Inspeção de Código

Apesar de a inspeção, de forma manual, ser uma forma de revisão extremamente útil, a mesma sofre de um grande problema que é o tamanho em linhas de código dos sistemas. Scach [2008] traz à tona o seguinte questionamento: “O tempo e o esforço adicional de uma inspeção valem a pena?”.

A resposta para essa pergunta pode ser muito complexa ou subjetiva, uma vez que depende de vários outros fatores mais circunstanciais do que explícitos. Apesar de constituir um terreno amplo para discussões, o assunto que trata do custo-benefício entre os recursos gastos para a garantia de qualidade de um software e o valor que este software agrega ao negócio não é abordado nesse trabalho. Mas em especial, é nesta problemática que reside a motivação deste trabalho: Ao invés de avaliar se vale a pena ou não investir na atividade de inspeção, este trabalho foca em reduzir o tempo e o esforço necessário para executá-la através da automatização da mesma.

A diferença entre a inspeção manual e automática será demonstrada mais a frente, onde serão destacados seus prós e contras. Primeiramente, temos de definir o ponto de partida que viabiliza a análise de código automatizada.

Para analisar programaticamente um código de qualquer linguagem de programação, antes é necessário transformá-lo em alguma estrutura programaticamente analisável. Essa estrutura programaticamente acessível é a representação de sua Árvore Sintática Abstrata (*Abstract Syntax Tree*), que deve ser gerada para cada arquivo fonte contido em uma aplicação sujeita à análise.

Essa transformação consiste em capturar o texto contido no arquivo, isto é, o dado não estruturado, e transformar, através de sua análise léxica e sintática, em uma estrutura que permita navegabilidade entre seus elementos. Para que essa navegação seja viabilizada, é necessário o uso do padrão de projeto *Visitor*, transformando cada nó da *AST* em um elemento visitável. O processo de criação de uma *AST* é detalhado no Capítulo 2.

A partir dessa transformação do texto em uma *AST* é possível checar diversas características relativas à sua estrutura e, conseqüentemente, verificar regras arquiteturais. Os algoritmos e estratégias utilizados para essas verificações são abordados na seção seguinte.

3.4 Resumo do capítulo

O presente capítulo abordou a inspeção de código e de que maneira a mesma auxilia no processo de desenvolvimento de sistemas trazendo benefícios para o projeto. Além disso, foi ilustrado um processo de inspeção de código que pode ser apoiado por uma ferramenta. O próximo capítulo apresentará a solução de automatização de inspeção de código proposta neste trabalho utilizando um *plug-in* para uma ferramenta que apoia o processo de inspeção.

Capítulo 4: PLUG-IN PARA ANÁLISE ESTÁTICA DE CÓDIGO

Este capítulo tem como objetivo apresentar a proposta de automatização de inspeção de código. Para a realização da atividade será apresentada uma estrutura genérica para análise de código utilizando um *plug-in* e a instância que foi implementada neste trabalho. O capítulo ainda abordará as regras validadas e os respectivos algoritmos responsáveis pela validação das mesmas.

4.1 Proposta de Automatização

O presente trabalho tem como proposta elaborar algoritmos para automatização da inspeção de código com base em um *checklist* de regras utilizadas por uma organização no seu processo de inspeção manual.

O grande desafio da inspeção automatizada é inferir programaticamente e corretamente que funcionalidade ou qual papel um componente exerce em uma aplicação para, então, verificar as devidas regras associadas a eles.

Uma forma de melhorar o entendimento de um componente do código é através dos metadados, ou seja, dados que descrevem dados com objetivo de identificar ou categorizar o componente. Por exemplo, *tags* HTML que indicam a categoria da página (*e.g.*, blog, Jornal, entretenimento, etc.). Na análise estática de código, o uso de metadados associados aos componentes do código (*e.g.*, classes e interfaces) é uma estratégia que ajuda na identificação do papel de cada componente dentro do sistema. Por exemplo, uma determinada classe poderia ter uma anotação (metadado) que indicasse o papel arquitetural que ela desempenha. Mais especificamente, poderíamos ter uma classe *'ServicoDeRelatorio'* com a anotação *'servico'*. Desta forma, saberíamos que esta classe desempenha o papel arquitetural de um serviço no contexto do sistema.

Portanto, os metadados desempenham um papel crucial no processo de desenvolvimento e de manutenção do código. Além de identificar a função dos componentes aos quais estão associados, viabilizam a automatização da inspeção de

regras arquiteturais. O uso correto destes metadados nos componentes é uma premissa para que algumas regras arquiteturais específicas possam ser validadas.

Qualquer premissa deve se basear em uma regra ou padrão seguido pelos desenvolvedores. Consequentemente, uma premissa pode ser respeitada ou não, como o próprio nome sugere. Em função disso, é importante explicitar três implicações dessa estratégia:

1. Uma premissa pode se basear em outra regra arquitetural que também possui premissas.
2. Para verificar se uma regra arquitetural é violada ou não, suas premissas devem ser respeitadas.
3. As correções de violações de regras no código podem acarretar na identificação de novas violações que antes não eram percebidas.

Se em um dado momento o relatório de análise de um código identificou um conjunto de discrepâncias de regras arquiteturais que são premissas de outras regras, é possível que existam outras violações não detectadas. Assim, uma vez que o código é corrigido e analisado novamente, ele pode revelar novas violações.

A utilização de premissas abre espaço para discussão sobre as limitações da automatização da inspeção de código. Como foi explicado, podem existir cenários em que uma regra depende de uma premissa e essa premissa é verificada por outra regra, isto é, para conseguir validar a regra são necessárias duas regras: Uma para verificar se sua premissa é respeitada e outra para a própria regra.

Isso caracteriza uma estrutura hierárquica e, portanto, as regras-base dessa hierarquia ou não possuirão premissas ou possuirão premissas que não têm como serem validadas.

Uma das grandes limitações da inspeção automatizada reside neste último caso, pois pode haver a impossibilidade de validar algumas premissas através de outras regras, seja por estas não existirem ou por não serem automatizáveis. É o ponto em que existe abertura para falhas, caso não existam componentes humanos em qualquer parte no processo de inspeção.

Este último ponto é um dos fatores que fragiliza a automatização de inspeção de código, contribuindo para a ocorrência de falsos positivos ou falsos negativos. Por isso, a proposta deste trabalho não descarta completamente a interação humana, e

recomenda que o resultado da análise seja revisado por alguém capacitado que consiga distinguir o que é uma violação arquitetural de fato.

4.2 Tecnologias de apoio

Nesta seção serão descritas as principais tecnologias que apoiaram a implementação e empacotamento do *plug-in* gerado neste trabalho.

4.2.1 SonarQube

Para cumprir o objetivo deste trabalho, foi selecionada uma ferramenta para que fosse realizada a codificação de um *plug-in* que pudesse contemplar as regras definidas no *checklist* de inspeção de código. A ferramenta escolhida foi o *Sonarqube*³, e para os fins de sua escolha, os seguintes aspectos foram considerados:

- Código aberto: A ferramenta é *open source*, o que possibilita uma grande participação da comunidade na sua customização.
- Linguagens analisadas: Apesar de ser feita primeiramente para análise da linguagem Java, a ferramenta possui diversos *plug-ins* para análise de mais de 20 diferentes linguagens.
- API para criação de regras e acesso à AST: O ponto central do trabalho é a criação de regras customizadas de acordo com regras específicas. A ferramenta fornece um ponto de extensão que possibilita a criação de novas regras, assim como acesso à Árvore Sintática Abstrata gerada durante a análise de um código-fonte.
- Estabilidade da versão: A versão escolhida foi a 4.5.4 que é LTS (abreviatura para *Long Term Support*) que é um tipo de versão especial projetada para ter um suporte prolongado, que varia de acordo com a empresa responsável. No caso do Sonarqube, a empresa garante que manterá a compatibilidade dos *plug-ins* até o lançamento da próxima versão.⁴

³ Download em <http://www.sonarqube.org/downloads/>

⁴ Informação obtida em <http://www.sonarqube.org/walking-the-tightrope-balancing-agility-and-stability/>

O *plug-in* codificado conterá os algoritmos desenvolvidos para validar as regras do *checklist* com o apoio da Árvore Sintática Abstrata.

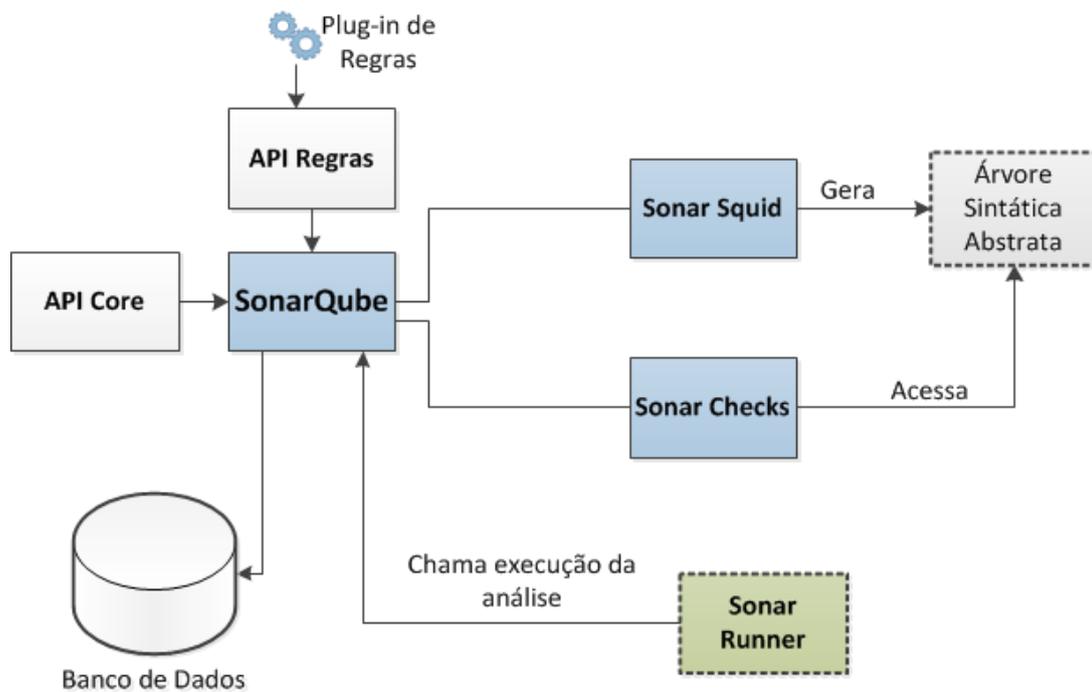


Figura 5: Estrutura da Ferramenta *SonarQube*

A Figura 5 ilustra uma visão conceitual do *SonarQube*. A ferramenta tem uma arquitetura que envolve a presença de um banco de dados, onde são armazenados metadados dos códigos e projetos analisados ao longo tempo. A ferramenta armazena um histórico de análise com o intuito de fornecer uma visão geral da evolução do mesmo. As informações são analisadas com o apoio de um cliente de análise de código. No exemplo utilizado neste trabalho, será utilizado o *Sonar Runner*, uma ferramenta auxiliar fornecida pelo *SonarQube* que roda dentro do diretório onde o código-fonte está presente.

Ao realizar uma análise, o *Sonar Runner* chama o core da aplicação *SonarQube* que possui um módulo responsável pela análise sintática do código, o *Sonar Squid*. Este módulo gera uma AST que fornece as informações dos códigos em uma API consultada pelo módulo de análise de código, denominado *Sonar Checks*. Dentro deste estão presentes as classes que contêm os algoritmos de validação do código de acordo com as regras desejadas. Cada análise possuirá um conjunto de discrepâncias encontradas que serão persistidas na base de dados. As discrepâncias podem ser

consultadas pelo usuário através de um sistema Web fornecido pela ferramenta e publicado em um *Web Server*. O sistema contém *dashboards* responsáveis pela exibição dos projetos analisados e dos resultados de cada análise.

O core da aplicação possui algumas funcionalidades, por exemplo, diversos *widjets* que podem ser usados para customizar os *dashboards*. Existem alguns *plug-ins* já desenvolvidos que realizam integrações do *SonarQube* com outras ferramentas. Estes são feitos através da API para desenvolvimento de *plug-ins*.

O *SonarQube*, por padrão, possui regras somente para a linguagem Java, entretanto é possível instalar *plug-ins* que trazem regras de outras linguagens como C# e C++. Isto só é possível graças à API para criação de novos repositórios de regras. Entretanto vale ressaltar que ainda assim é necessário ter uma forma de gerar a AST relativa à linguagem desejada.

A ferramenta possui um analisador léxico e sintático para a linguagem Java que gera uma interface para acesso à AST da linguagem. É através deste ponto de extensão que o *plug-in* desenvolvido neste trabalho irá atuar (Seção 4.4).

4.2.2 Apache Maven

O Apache Maven⁵[MAVEN,2015] é uma ferramenta de compreensão e gerenciamento de projetos de desenvolvimento de software . O *Maven* é baseado no conceito *Project Object Model* (POM) e pode gerenciar a construção (*build*) de um projeto, relatórios e documentação centralizando suas configurações em um arquivo denominado *pom.xml*.

O projeto de desenvolvimento de *plug-in* de inspeção de regras para a ferramenta *SonarQube* utiliza o *Maven* para o gerenciamento das dependências e também como empacotador do *plug-in*. Vale ressaltar que, no momento de empacotamento do *plug-in*, o *Maven* também executa os testes unitários para garantir o correto funcionamento das funcionalidades da aplicação.

Nas configurações do projeto, é apontada a versão do *SonarQube* para a qual o *plug-in* será empacotado e também a versão do módulo Java *Squid*, responsável pela geração da AST. É de suma importância que, ao gerar o *plug-in* para outra versão da ferramenta, essas configurações sejam alteradas.

⁵Download disponível em <https://maven.apache.org/download.cgi>

Por fim, é preciso apontar dentro do arquivo de configurações do *Maven* qual é a classe que estende o ponto de entrada da API do *SonarQube*. Esse processo é feito com o auxílio de um *plug-in* chamado *sonar-packaging-maven-plugin*. A estrutura citada pode ser vista na Figura 6 onde a classe *RegrasJavaPlugin* estende a API do *SonarQube*.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.sonar</groupId>
      <artifactId>sonar-packaging-maven-plugin</artifactId>
      <version>1.12.1</version>
      <extensions>true</extensions>
      <configuration>
        <pluginClass>br.unirio.tcc.sonar.plugin.RegrasJavaPlugin</pluginClass>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Figura 6 - Arquivo pom.xml do plug-in desenvolvido.

4.3 Estrutura genérica de análise de código utilizando um plug-in

No processo de automatização da inspeção de código, a alternativa para contemplar os itens do *checklist* que não são cobertos pelas ferramentas existentes é realizar uma customização de regras nas mesmas, com base numa estrutura genérica que, caso não exista nativamente, deverá ser implementada. As atividades a serem realizadas, de forma geral, podem ser vistas na Figura 7.

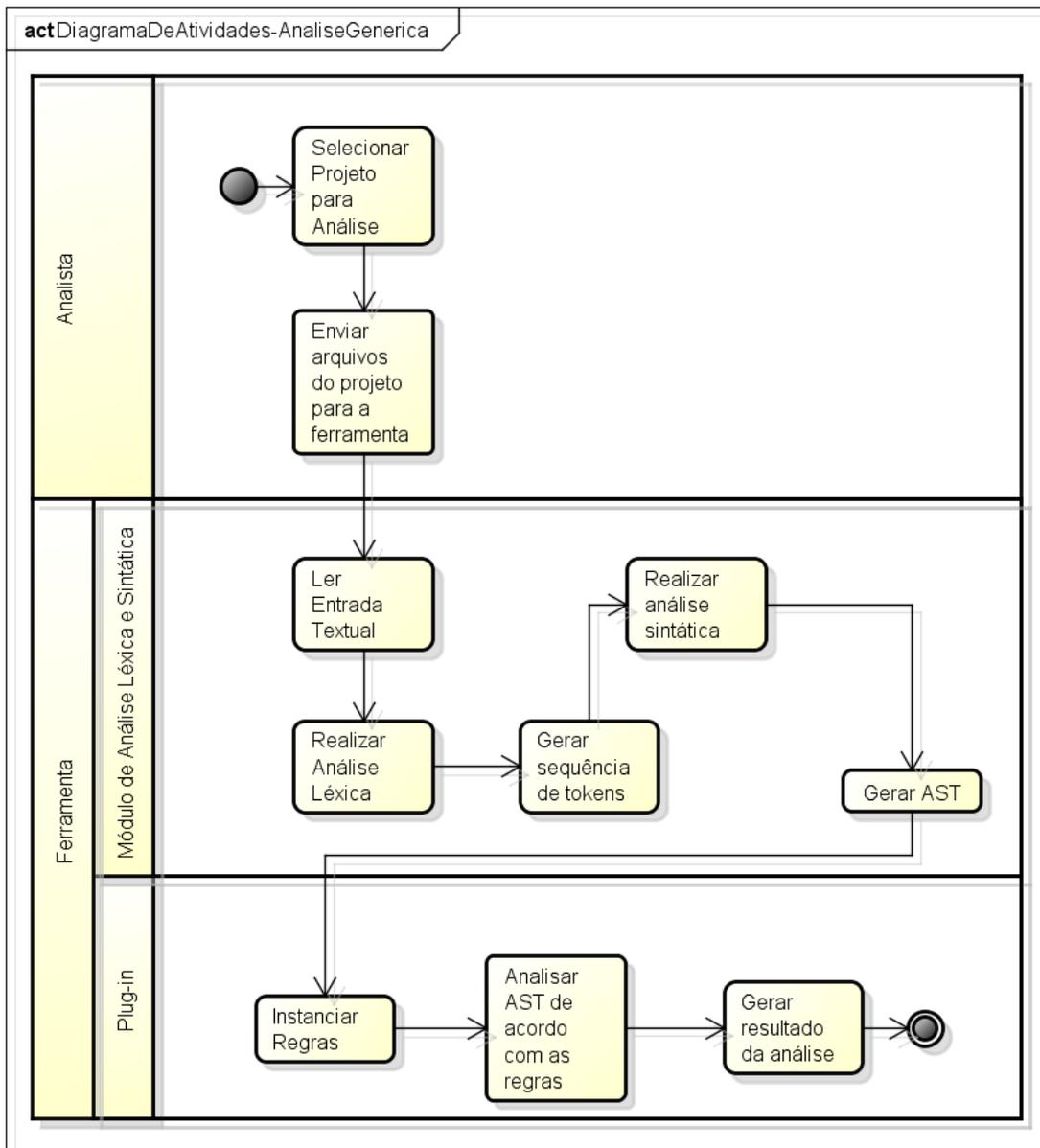


Figura 7 - Atividades Genéricas para Análise

Neste diagrama de atividades, o ponto de início é a escolha do projeto a ser analisado, que deverá ser enviado para uma ferramenta de inspeção automática de código que fará a leitura e passará a entrada para os módulos responsáveis pela análise léxica e sintática.

A análise será feita sempre arquivo por arquivo, que por sua vez estará relacionado a um projeto. Cada arquivo fonte possuirá uma cadeia textual que foi escrita numa determinada linguagem, que deverá ser conhecida antes da codificação do *plug-in*. A mesma passará por um módulo de análise léxica e sintática para que

seja gerada a Árvore Sintática Abstrata. A ferramenta para a qual o algoritmo está sendo projetado deverá, portanto, possuir um ponto de extensão no qual o *plug-in* atuará.

O *plug-in* possuirá um módulo analisador, no qual estarão implementados os algoritmos de validação das regras que serão customizadas. É importante ressaltar que a estrutura da ferramenta deve fornecer uma maneira de apontar os erros encontrados na análise para futura consulta por parte dos usuários envolvidos.

Caso a ferramenta não possua algum dos módulos necessários, a implementação do *plug-in* deverá contemplá-los. Um exemplo disso é a própria ferramenta *SonarQube* que possui uma *API* para criação de repositórios de regras, mas que não fornece a *API* para análise léxica e sintática das linguagens, ficando a cargo do usuário desenvolver este módulo. Vale frisar a necessidade de uma *API* para acesso aos nós da *AST*. Quanto melhor o projeto da estrutura, menor a curva de aprendizado para que os desenvolvedores possam criar novas regras com o tempo.

Um ponto negativo desta abordagem pode ser a questão do desempenho na análise, visto que a criação de diversas árvores sintáticas abstratas, por diferentes bibliotecas, acarreta numa análise mais demorada.

4.4 Plug-in de Regras para a ferramenta *SonarQube*

A estrutura do *plug-in* de regras para a ferramenta *SonarQube* deve se preocupar em utilizar o ponto de extensão da ferramenta indicando pelo menos uma classe de cada um dos seguintes tipos:

- *Server Extension*: os objetos definidos por essas classes são inicializados no momento que o servidor é iniciado e o *plug-in* é carregado. Neste caso são criados os repositórios e as regras na base de dados da ferramenta, caso elas já não existam.
- *Batch Extension*: os objetos definidos por essas classes são instanciados no momento da análise de um determinado projeto. Neste caso, as classes que implementam as regras que possuirão a lógica da validação automatizada dos itens propostos são instanciadas e executadas.

A Figura 8 ilustra a sequência para a criação de regras em seus respectivos repositórios na ferramenta *SonarQube* realizada para o *plug-in* codificado neste trabalho.

No momento em que o *plug-in* é carregado na inicialização do servidor, a classe que estende o ponto de extensão da ferramenta (*RegrasJavaPlugin*) é chamada e a mesma retorna as extensões declaradas (passo 1). A extensão neste método é a do tipo *server extension*, que é realizada pela classe *DefinicaoRegras*.

A classe *DefinicaoRegras* define, no passo 2, o repositório onde as regras serão carregadas e então aciona a ferramenta *SonarQube* para que ela crie o repositório de regras. Esse passo é importante, pois nenhuma regra pode existir se não estiver em um repositório pré-definido.

No passo 3, a classe *DefinicaoRegras* pede para a classe *InstanciadorDeRegras* retornar todas as classes definidas no *plug-in* que estendem da interface *JavaCheck*. Cada classe representa uma regra e deve possuir uma anotação responsável por identificar algumas informações da regra como nome, descrição e chave (que deve ser única para cada regra na ferramenta).

Por fim, são passadas para a ferramenta todas as classes com essas informações para que as regras sejam persistidas na base de dados. A criação e persistência da regra é feita pelo método *load(repo,checks)* somente para as regras encontradas que não tenham sido previamente persistidas no banco de dados.

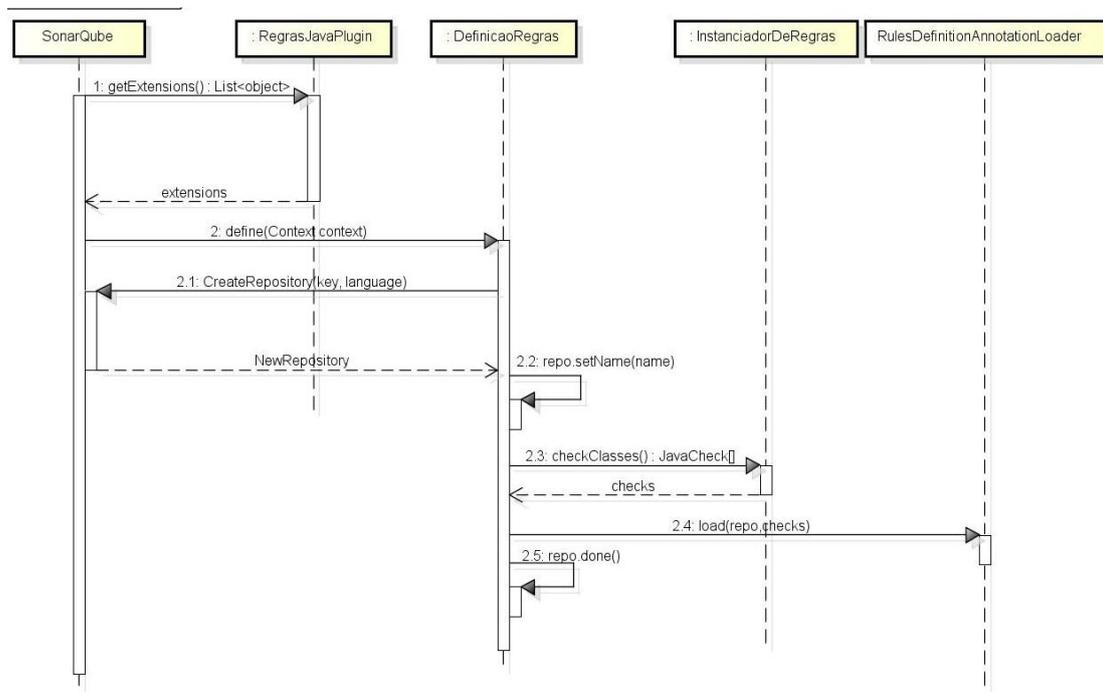
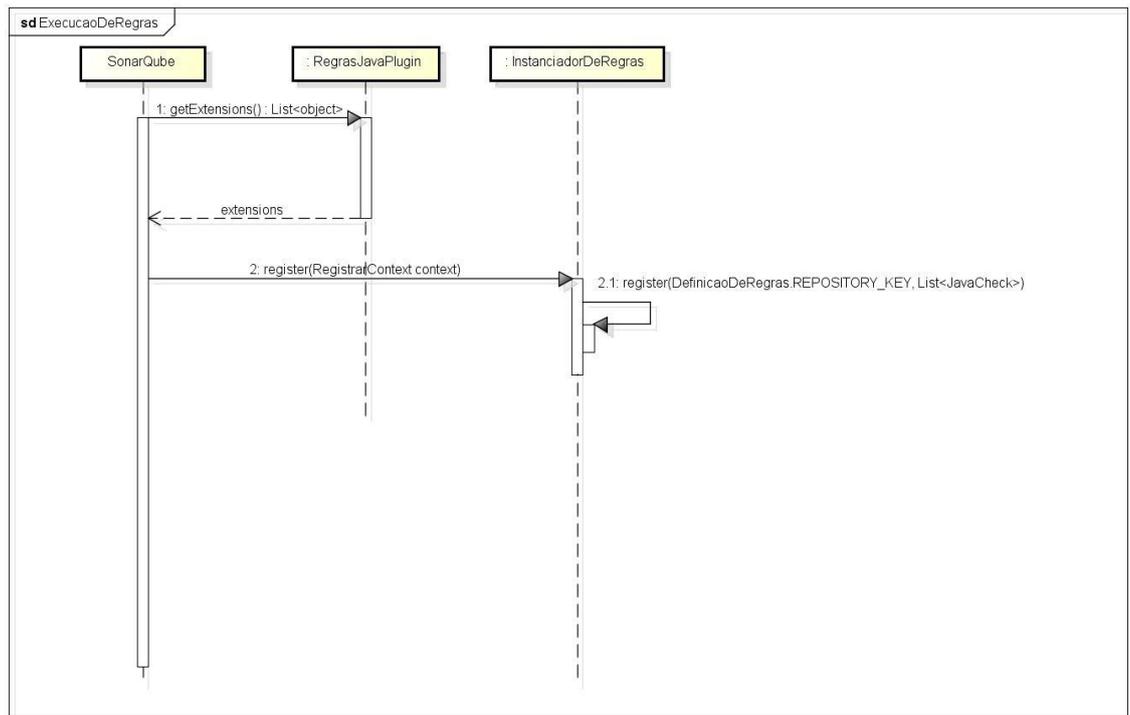


Figura 8 - Diagrama de sequência para criação das regras

Na Figura 9, é apresentado um diagrama de sequência da execução de uma análise de código onde as classes Java que possuem a lógica das regras de inspeção de código são instanciadas para realizar a análise. No momento em que é iniciada uma análise de código em um projeto Java, a ferramenta *SonarQube* chama o *plug-in* da linguagem e obtém a classe que realiza a extensão do tipo *batch* que, neste caso, é a classe *InstanciadorDeRegras*. Tal classe é responsável por registrar a lista de classes que serão instanciadas durante a execução do projeto.



powered by Astah

Figura 9 - Diagrama de Sequência para execução das regras

A organização dos pacotes do *plug-in* pode ser vista em maiores detalhes na Figura 10. Essa estrutura de pacotes, que definem a estrutura interna do *plug-in*, foi projetada pensando em modularizar o projeto da seguinte forma:

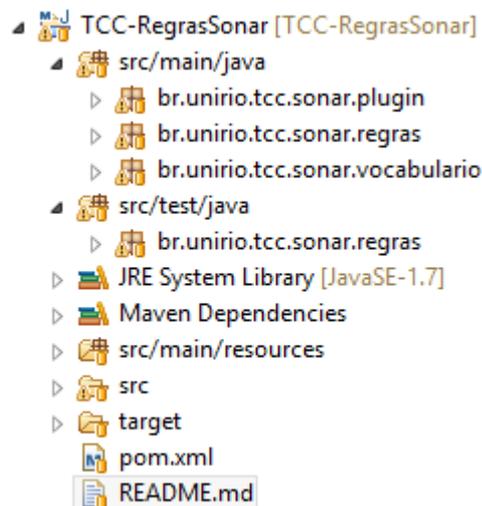


Figura 10 - Estrutura de pacotes

- *Plugin*: Pacote que contém as classes de acesso ao ponto de entrada de extensão da ferramenta *SonarQube*.
- *Regras*: Pacote que contém as classes que possuem a lógica de validação das regras automatizadas. Cada regra é transformada em uma classe que, para seguir o padrão da ferramenta, possui o sufixo “*Check*” em seu nome.
- *Vocabulário*: Pacote contendo classes responsáveis pela validação de itens relativos à nomenclatura, mais detalhados na Seção 4.7.1 deste documento.
- *Resources*: Localização dos arquivos no formato *.txt* utilizados como insumo para o módulo de vocabulário (Seção 4.7.1).
- *Test*: Pacote onde estão presentes as classes de testes unitários que utilizam a infraestrutura fornecida pela própria *API* da ferramenta.

4.5 Arquitetura do Projeto Analisado

Para implementar a solução de automatização de inspeção de código com base em um *checklist*, foi necessário selecionar uma arquitetura de software como referência para tal.

A arquitetura foi desenvolvida por uma empresa de mercado com foco em consultoria na área de Tecnologia da Informação que realiza projetos de implementação de arquitetura de referência.

Basicamente, a arquitetura é composta de cinco camadas, exibidas na Figura 11: Apresentação, Serviços, Domínio e Infraestrutura.

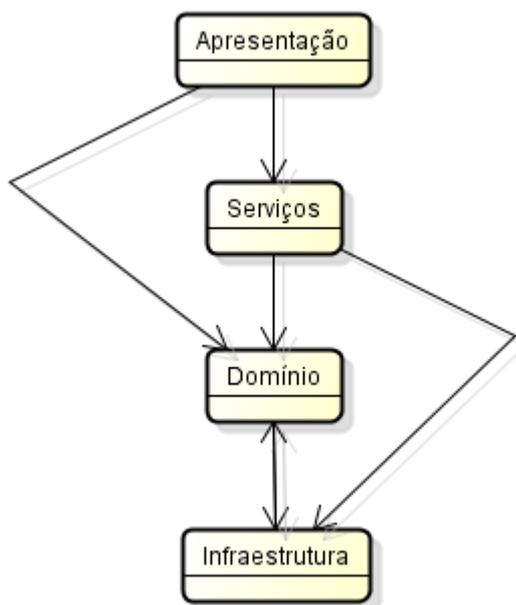


Figura 11 – Principais camadas da arquitetura base

Os seguintes frameworks foram utilizados: *Spring* para *MVC* e injeção de dependências, *Hibernate* como *Object-Relational Mapper*, e *AspectJ* para utilização de aspectos.

A camada de apresentação tem como objetivo exibir e controlar a interface com o usuário e implementa o padrão *MVC*. Esta camada só pode ter acesso aos itens da camada de serviços e domínio. É nesta camada que estão localizados os controladores e páginas *JSP (Java Server Pages)* da aplicação.

A camada de serviços é responsável por executar os passos necessários para atender à requisição do usuário e coordena as camadas inferiores. Esta camada só tem acesso às camadas de domínio e infraestrutura.

A camada de domínio representa o domínio do negócio, centraliza as regras de negócio do sistema e contém as classes que implementam os padrões *DTO (Data Transfer Object)*⁶ e *VO (Value Object)*⁷. A camada de domínio não possui acesso a qualquer outra camada.

⁶ O Data Transfer Object é um padrão de projeto que define um objeto que agrega dados com o intuito de os transportar. Os DTOs não possuem qualquer comportamento.

A camada de infraestrutura é responsável por prover acesso e persistência de dados e implementa o padrão *Repository*⁸ para isso. Só tem acesso à camada de domínio. É nesta camada que as bibliotecas do *Hibernate* são utilizadas para acesso ao banco, bem como é o local apropriado para consumo de *Web Services* e envio de e-mail.

Na arquitetura definida, existem anotações presentes nos *frameworks* que são utilizadas como premissas na inspeção automatizada de algumas regras. As seguintes anotações são utilizadas:

- *@Controller*: Anotação presente no pacote *org.springframework* que identifica uma classe que faz o papel de controlador.
- *@Service*: Anotação presente no pacote *org.springframework* que identifica uma classe que faz o papel de serviço.
- *@Repository*: Anotação presente no pacote *org.springframework* que identifica uma classe que faz o papel de repositório.
- *@Entity*: Anotação presente no pacote *javax.persistence* que identifica a classe como uma entidade do domínio.
- *@Aspect*: Anotação presente no pacote *org.aspectj* que identifica um aspecto, responsável por encapsular os interesses que atravessam os vários objetos de um sistema.
- *@Around*: Anotação presente no pacote *org.aspectj* que identifica um método que envolve uma execução de um *joint point* que é um ponto durante a execução do programa afetado pelo aspecto.

4.6 Regras do Checklist

O *plug-in* para automatização da análise de código foi desenvolvido com base em um *checklist* que auxilia a validação da arquitetura de referência descrita na seção 5.1.

⁷ O padrão Value Object representa um pequeno objeto cujo parâmetro de igualdade não é baseado na sua identidade e sim em seu valor, isto é, se dois VOs possuem os mesmos valores, eles são considerados iguais;

⁸ O padrão de projeto Repository é uma abstração de uma coleção de objetos com o objetivo de prover acesso aos dados, além de encapsular a lógica de mapeamento objeto-relacional dentro de si.

As seguintes regras têm sua validação feita de forma automatizada através do plug-in elaborado neste trabalho:

- Regra 1 - Nomenclatura de Pacotes
- Regra 2 - Nomenclatura de Classes – Pascal Case
- Regra 3 - Nomenclatura de Classes
- Regra 4 - Nomenclatura de Métodos – Camel Case
- Regra 5 - Nomenclatura de Métodos
- Regra 8 – Uso de Controlador Genérico
- Regra 10 – Localização de classe Controller
- Regra 11 – Sufixo de classe Controller
- Regra 16 – Nomenclatura de Serviços
- Regra 19 – Log Interceptador na camada de Serviços
- Regra 20 – Prefixo do Log Interceptador
- Regra 22 – Localização de Classes com sufixo Vo
- Regra 24 – Mapeamento Relacional Lazy x Eager
- Regra 25 – Sufixo do Repositório Genérico
- Regra 26 – Localização de classes de envio de e-mail

Todas as regras definidas no *checklist* são descritas abaixo:

Regras Gerais

Regra 1 - Nomenclatura de Pacotes

A nomenclatura dos pacotes deve seguir o padrão *br.unirio.<sistema>.<pacote>*, onde *<pacote>* deve assumir um dos seguintes valores: apresentação, serviços, domínio, infraestrutura, útil, exceções.

Propósito da regra: Estabelecer um padrão para os nomes dos pacotes é importante para organizar o código em hierarquia, fazendo com que os mesmos estejam agrupados de acordo com algum parâmetro. No caso da arquitetura em questão, procura-se separar os códigos de acordo ou com a camada em que estão inseridos, ou de acordo com a função que exercem: classes de exceção ou utilitárias.

Regra 2 - Nomenclatura de Classes – Pascal Case

O nome das classes deve estar no padrão *Pascal Case*.

Propósito da regra: Verificar se a convenção Java para nomenclatura de classes está sendo respeitada.

Regra 3 - Nomenclatura de Classes

O nome das classes deve seguir o padrão de nomenclatura previsto para as palavras que compõem o seu nome (Tabela 2).

Propósito da regra: Verificar se o nome padronizado proposto para as classes com o objetivo de melhorar a legibilidade do código está sendo respeitado.

Regra 4 - Nomenclatura de Métodos – Camel Case

Os métodos devem estar declarados em *Camel case*.

Propósito da regra: Verificar se a convenção Java para nomenclatura de métodos está sendo respeitada.

Regra 5 - Nomenclatura de Métodos

Os métodos das classes da camada de serviço e de domínio devem seguir o padrão de nomenclatura definido para o fluxo de palavras que compõe o nome, com exceção dos métodos *get* e *set*. (Tabela 1).

Propósito da regra: Verificar se o nome padronizado proposto para os métodos com o objetivo de melhorar a legibilidade do código está sendo respeitado.

Regra 6 - Nomenclatura de Interfaces

As interfaces devem seguir o padrão de nomenclatura de classes precedidas da letra I.

Propósito da regra: Verificar se o nome padronizado proposto para as interfaces com o objetivo de melhorar a legibilidade do código está sendo respeitado

Regra 7 - Nomenclatura de variáveis e atributos

Variáveis e atributos devem possuir uma nomenclatura que esteja contida no vocabulário proposto (seção 4.7.1).

Propósito da regra: Verificar se o nome das variáveis e atributos possuem algum sentido de maneira a melhorar a legibilidade do código.

Regras da Camada de Apresentação

Regra 8 – Uso de Controlador Genérico

Toda classe que representa um controlador (identificada pela anotação *@Controller*) deve estender o Controlador Genérico definido para a arquitetura.

Propósito da regra: Verificar se um controlador genérico que agrega métodos e funcionalidades úteis a todos os controladores foi criado.

Regra 9 – Utilização de anotação Controller

Toda classe que representa um controlador deve ter a anotação *@Controller*.

Propósito da regra: Verificar se o *Spring framework* está sendo utilizado corretamente nos controladores do projeto.

Regra 10 – Localização de classe Controller

Toda classe que representa um controlador deve estar no pacote apresentação.

Propósito da regra: A regra se propõe a verificar se as classes que implementam os controladores estão no pacote correto.

Regra 11 – Sufixo de classe Controller

Toda classe que representa um controlador deve ter nomenclatura com o sufixo *Controller*.

Propósito da regra: Verificar se o nome padronizado proposto para os controladores está sendo respeitado.

Regra 12 – Controladores e lógica de negócio

As classes que representam os controladores devem estar livres de lógicas de negócio.

Propósito da regra: Os controladores atuam como ponte entre a interface gráfica e o *back-end* da aplicação, invocando os componentes necessários para se atender a requisição. A regra verifica se nenhuma regra de negócio está sendo validada dentro dos controladores.

Regra 13 – Uso da interface Validator

Os validadores utilizados nos controladores devem implementar a interface *Validator*.

Propósito da regra: Verificar se *framework Hibernate* está sendo utilizado para executar a lógica de validação simples nos controladores.

Regra 14 – Localização dos validadores

Os validadores devem estar no pacote *apresentacao.validadores*

Propósito da regra: A regra se propõe a verificar se as classes que implementam validadores estão no pacote correto.

Regra 15 – Invocação de método de controladores

Controladores não devem invocar métodos de outros Controladores.

Propósito da regra: Cada controlador deve possuir toda lógica necessária para atender a uma requisição dentro de si sem compartilhar com outros. A regra verifica se isso está sendo respeitado.

Regras da Camada de Serviços

Regra 16 – Nomenclatura de Serviços

As classes da camada de serviço, exceto as com prefixo Log devem seguir o padrão de nomenclatura:

<ServicosDe><NomeDaEntidade/NomeDoServiço/CasoDeUso/Modulo>.

Propósito da regra: Verificar se o nome padronizado proposto para as classes de serviços com o objetivo de melhorar a legibilidade do código está sendo respeitado.

Regra 17 – Uso de anotação Service

As classes da camada de serviço, exceto as com prefixo Log, devem utilizar a anotação *@Service*.

Propósito da regra: Verificar se *Spring framework* está sendo utilizado corretamente nos serviços da aplicação.

Regra 18 – Log Interceptador e anotação Aspect

A classe que exerce o papel de interceptador para log da aplicação deve utilizar a anotação *@Aspect*.

Propósito da regra: Verificar se o *framework AspectJ* está sendo utilizado corretamente para interceptação via aspecto na classe de interceptador de log.

Regra 19 – Log Interceptador na camada de Serviços

A classe que exerce o papel de interceptador para log da aplicação deve estar na camada de serviços.

Propósito da regra: A regra se propõe a verificar se a classe que implementa o interceptador de log está no pacote correto.

Regra 20 – Prefixo do Log Interceptador

A classe que exerce o papel de interceptador para log da aplicação deve possuir a nomenclatura com prefixo Log.

Propósito da regra: O prefixo no nome da classe de interceptação de log auxilia o desenvolvedor pois explicita a função que a classe exerce sem que ele necessite a consultar.

Regra 21 – Serviços Stateless

Todas as classes de serviços devem ser *stateless*.

Propósito da regra: A regra se propõe a verificar se não existe nenhum tipo de informação sendo armazenada nos serviços da aplicação, uma vez que esses devem atender cada requisição de maneira independente. Assim, todos os serviços não podem possuir estado. A regra verifica se isso está sendo respeitado.

Regras da Camada de Domínio

Regra 22 – Localização de Classes com sufixo Vo

Todas as classes que possuem o sufixo “Vo” devem estar presentes no pacote domínio.vo.

Propósito da regra: A regra se propõe a verificar se as classes que implementam o padrão *value object* estão no pacote correto.

Regra 23 – Nomenclatura de Classes do pacote Vo

Todas as classes que estão no pacote domínio.vo devem possuir o sufixo Vo

Propósito da regra: O sufixo no nome dos *value objects* auxilia o desenvolvedor pois explicita o padrão que a classe implementa sem que ele necessite consultá-la.

Regra 24 – Mapeamento Relacional Lazy x Eager

Todo mapeamento objeto-relacional utilizando a *Java Persistence API* (JPA) deve utilizar a estratégia *lazy* ao invés de *eager*.

Propósito da regra: Em entidades muito relacionadas que usam a estratégia *eager* uma simples busca pode trazer todos os registros das tabelas, sobrecarregando o sistema e causando problemas de performance. A regra tem como objetivo identificar todas as ocorrências do uso dessa estratégia de maneira que os casos possam ser analisados.

Regras da Camada de Infraestrutura

Regra 25 – Sufixo do Repositório Genérico

A classe *RepositorioGenerico* deve possuir o sufixo que indique o tipo de tecnologia utilizada para acesso aos dados, sendo aceitos os nomes *RepositorioGenericoJpa* ou *RepositorioGenericoJdbc*.

Propósito da regra: O sufixo dos repositórios auxilia o desenvolvedor pois demonstra qual o tipo da tecnologia utilizada para acesso aos dados naquela classe sem necessitar que ele verifique isso por outros meios. A regra verifica se este sufixo está sendo utilizado.

Regra 26 – Localização de classes de envio de e-mail

As classes que possuem lógica de envio de e-mail, realizando a importação de bibliotecas específicas previstas para a arquitetura (*org.springframework.mail*), devem estar presentes no pacote de infraestrutura.

Propósito da regra: A regra se propõe a verificar se as classes que tratam de envio de e-mail estão no pacote correto.

Regra 27 – Utilização de anotação Repository

As classes que representam a implementação dos repositórios devem utilizar a anotação *@Repository*.

Propósito da regra: A regra se propõe a verificar se as classes que tratam de acesso aos dados estão no pacote correto.

Regra 28 – Localização de Repositórios no pacote de infraestrutura

As classes que representam a implementação dos repositórios devem estar no pacote infraestrutura.

Propósito da regra: A regra se propõe a verificar se as classes que tratam de acesso aos dados estão no pacote correto.

Regra 29 – Existência de Repositório Genérico

No pacote infraestrutura deve ser definido um Repositório Genérico.

Propósito da regra: Verificar que um repositório genérico foi criado, garantindo que não haja repetição de código para execução de *queries* cujo código é o mesmo, só alterando a entidade que as mesmas buscam no banco.

Regra 30 – Repositório Genérico como classe abstrata

O Repositório Genérico deve ser definido como uma classe abstrata.

Propósito da regra: A classe do repositório genérico não pode ser instanciada pois ela só contém métodos para execução de *queries* padrão de *CRUD*. A classe deve estendida pelos repositórios concretos. Assim, a classe deve ser abstrata.

Regra 31 – Contexto de persistência

O Repositório Genérico deve possuir o contexto de persistência a ser utilizado.

Propósito da regra: A regra visa verificar se o contexto é obtido a partir de injeção de dependência e não por outros meios.

4.7 Regras de Nomenclatura

Na proposta de automatização de regras do presente trabalho, existiam itens relativos à nomenclatura de elementos da linguagem com o objetivo de garantir a legibilidade do código, a garantia da conformidade entre os mesmos e um padrão definido por uma determinada organização. Por ser uma validação que pode acarretar em diversos ciclos de desenvolvimento para o refinamento das regras, foi analisada uma alternativa que trouxesse uma forma de realizar essa validação da nomenclatura.

4.7.1 Vocabulário da Língua Portuguesa

Para validar os itens de nomenclatura baseados na Língua Portuguesa é necessário, primeiramente, entender a classificação gramatical de uma determinada

palavra. O *plug-in* de regras possui um módulo *vocabulário* responsável por essa tarefa, o mesmo faz uso de uma estrutura de dados obtida do trabalho de Rodrigues (2013) que forneceu um conjunto de arquivos no formato “.txt” contendo uma lista de palavras, as quais assumem a classificação gramatical identificada no título do arquivo (Figura 12).

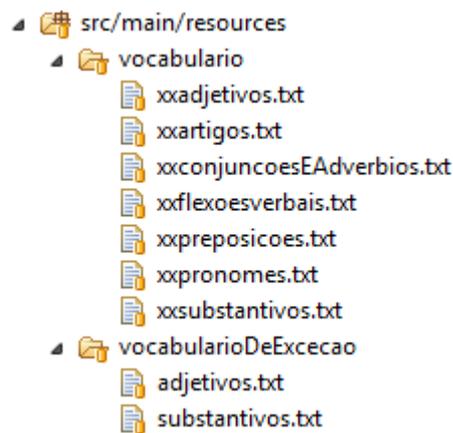
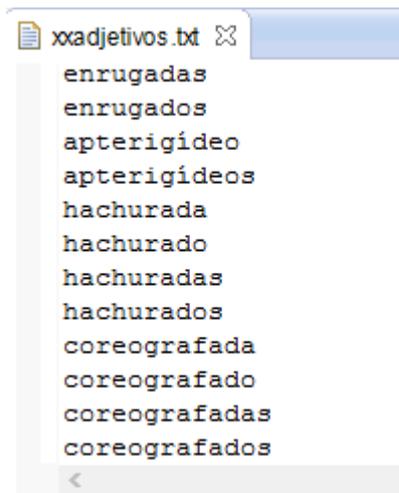


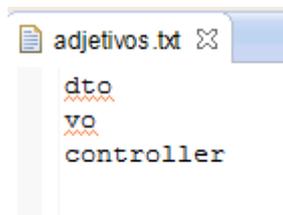
Figura 12 - Estrutura do vocabulário

Além da estrutura originalmente obtida, também foi criado um vocabulário de exceção com o intuito de refinar a execução do algoritmo de validação de uma determinada estrutura através da adição de vocábulos que são aceitos apesar de não existirem na língua portuguesa, por exemplo, palavras da língua inglesa que são comumente utilizadas durante o desenvolvimento de sistemas. Um exemplo da utilização do vocabulário de exceção foi a inclusão dos termos “*dto*” e “*vo*” (que se referem à *Data Transfer Object* e *Value Object*, respectivamente) como palavras reconhecidamente válidas e que exercem a função de adjetivos. A Figura 13 exibe um exemplo do conteúdo do arquivo “*xxadjetivos.txt*” e a exibe o conteúdo do arquivo “*adjetivos.txt*” do vocabulário de exceção.



```
xxadjetivos.txt
enrugadas
enrugados
apterigídeo
apterigídeos
hachurada
hachurado
hachuradas
hachurados
coreografada
coreografado
coreografadas
coreografados
```

Figura 13 - Exemplo de arquivo do vocabulário de adjetivos



```
adjetivos.txt
dto
vo
controller
```

Figura 14 - Exemplo de arquivo de adjetivos do vocabulário de exceção

4.7.2 Algoritmo de análise de nomenclatura

Com a finalidade de implementar um modelo reutilizável e de fácil manutenibilidade para validação da nomenclatura dos elementos de um código, foi projetado e implementado um algoritmo capaz de percorrer um fluxo de nós, representado através de uma matriz de adjacências, onde cada nó representa um elemento da língua portuguesa, e assim reconhecer as estruturas de nome estipuladas no *checklist* para as regras de nomenclatura, especialmente de classes e métodos. A escolha dessa abordagem se deve principalmente à complexidade da língua portuguesa e a maneira como suas expressões e elementos gramaticais pode estar estruturada.

Antes de explicar o algoritmo que percorre o fluxo de nós, é necessário entender como um nó é estruturado e como é representado em uma matriz de adjacência. Um nó representa uma classe gramatical da língua portuguesa e define uma regra de transição para outros nós, que também são representações de outras classes gramaticais da língua. Neste trabalho, foram separadas sete classes gramaticais

julgadas essenciais: *Substantivos, Pronomes, Preposições, Flexões Verbais, Conjunções e Advérbios, Artigos e Adjetivos*. Cada regra de nomenclatura deve definir para cada nó suas transições válidas, isto é, para quais classes a classe que o nó representa permite transição, de modo a caracterizar uma nomenclatura considerada válida. Além disso, as regras devem definir seu nó inicial e seus nós terminais.

Essa estrutura caracteriza um grafo direcionado sem pesos. Assim, uma maneira de representá-la é através de uma matriz de adjacências. A Tabela 1 e a Tabela 2 representam as matrizes de adjacências para a validação de nomenclatura de métodos e classes respectivamente.

Nó	Inicial	Final	Substantivos	Pronomes	Preposições	Flexões Verbais	Conjunções e Advérbios	Artigos	Adjetivos
Inicial	0	0	0	0	0	1	0	0	0
Final	0	0	0	0	0	0	0	0	0
Substantivos	0	1	0	0	1	0	0	0	1
Pronomes	0	0	0	0	0	0	0	0	0
Preposições	0	0	1	0	0	0	0	0	0
Flexões Verbais	0	0	1	0	0	0	0	0	0
Conjunções e Advérbios	0	0	0	0	0	0	0	0	0
Artigos	0	0	0	0	0	0	0	0	0
Adjetivos	0	0	0	0	0	0	0	0	0

Tabela 1 – Matriz de adjacências para nomenclatura de métodos

Nó	Inicial	Final	Substantivos	Pronomes	Preposições	Flexões Verbais	Conjunções e Advérbios	Artigos	Adjetivos
Inicial	0	0	1	1	0	0	0	0	0
Final	0	0	0	0	0	0	0	0	0
Substantivos	0	1	0	0	1	0	0	0	1
Pronomes	0	0	1	0	0	0	0	1	0
Preposições	0	0	1	0	0	0	0	0	0
Flexões Verbais	0	0	0	0	0	0	0	0	0
Conjunções e Advérbios	0	0	0	0	0	0	0	0	0
Artigos	0	0	1	0	0	0	0	0	0
Adjetivos	0	1	0	0	1	0	0	0	0

Tabela 2 - Matriz de adjacências para nomenclatura de classes

Como mencionado, as matrizes também podem ser representadas por grafos. As figuras Figura 15 e Figura 16 ilustram a representação em forma de grafo para as matrizes das tabelas Tabela 1 e Tabela 2 respectivamente.

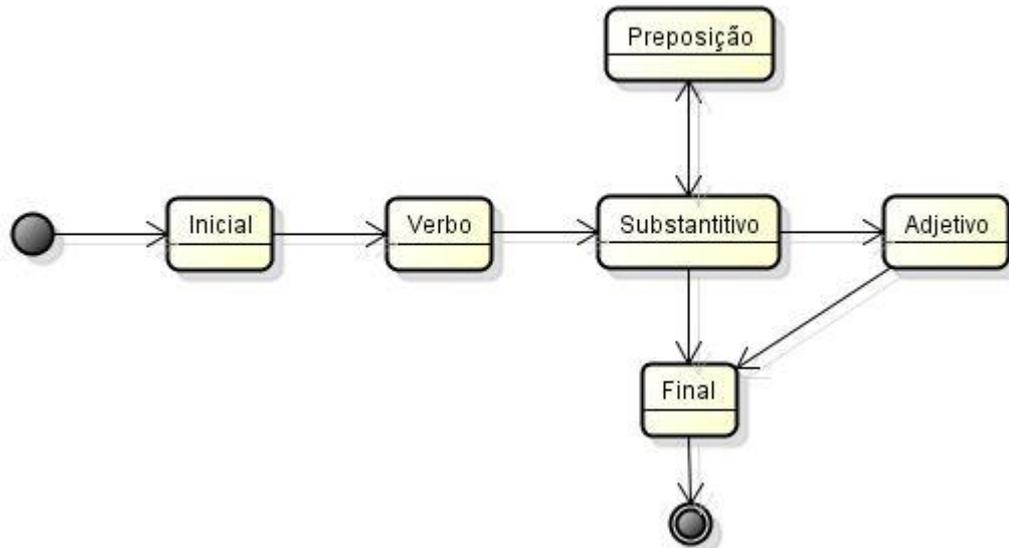


Figura 15 - Grafo para nomenclatura de métodos

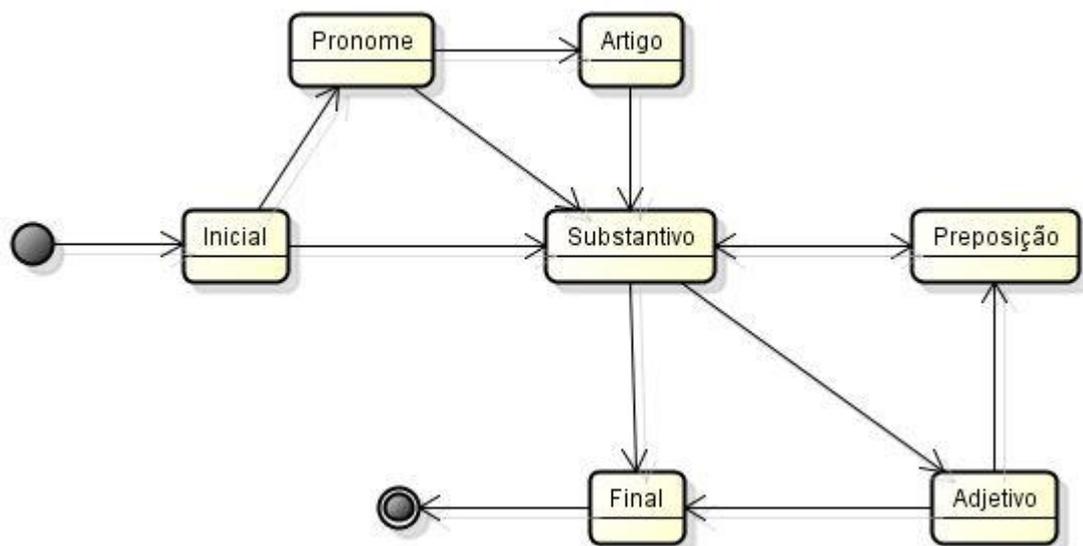


Figura 16 - Grafo para nomenclatura de classes

Para exemplificar e melhor ilustrar a estrutura, consideremos a nomenclatura de métodos (Tabela 1). A matriz é preenchida com 0 ou 1, onde 1 indica uma transição válida entre o Nó[linha] e Nó[coluna]. Por exemplo, do nó de *Flexões Verbais*, só existe uma transição válida, que é para o nó *Substantivos*.

Para interpretar quais palavras podem iniciar um nome válido, basta verificar quais transições o nó *Inicial* possui. No exemplo, um nome válido só pode começar

com uma palavra que seja um verbo, pois o nó *Inicial* só possui transição válida para o nó *Flexões Verbais*.

Para interpretar quais palavras podem ser as últimas em um nome válido, basta verificar quais nós possuem transição válida para o nó *Final*. No exemplo, um nome válido tem que terminar com uma palavra que seja um substantivo ou um adjetivo, pois somente os nós *Adjetivos* e *Substantivos* possuem transição válida para o nó *Final*.

No exemplo, a ocorrência de duas palavras da mesma classe não é válida, pois a diagonal principal da matriz é nula. Exemplo: substantivo+substantivo.

Aqui estão listados nomes considerados válidos pela matriz de adjacência de nomenclatura de métodos: “*processarInformacoesSubmetidas*” (verbo, substantivo, adjetivo), “*buscarNomesDeUsuarios*” (verbo, substantivo, preposição, substantivo), “*acessarBaseDeDados*” (verbo, substantivo, preposição, substantivo), “*removerUnidades*” (verbo, substantivo), “*atualizarRegistro*” (verbo, substantivo).

Exemplos de nomes considerados inválidos são: “*execucaoDeRotina*” (“execução” não é um verbo), “*processarPorLotes*” (“por” não é um substantivo), “*ajustarIndiceDeVendasAtraves*” (“através” não é terminal, ou seja, não é preposição ou adjetivo), “*processarDadosExecutarRotina*” (“Dados” é um substantivo, após um substantivo deve seguir uma preposição ou adjetivo e “executar” não é de nenhum dos dois tipos).

O funcionamento do algoritmo que executa o fluxograma está ilustrado no Algoritmo 1.

```
1 boolean verificarSeNomeEValido(string nome)
2     string [] palavras = obterPalavras(nome)
3     classeDaPrimeiraPalavra = obterClassePalavra(palavras[0])
4     no = obterNoInicial()
5     if(!no.permiteTransicao(classeDaPrimeiraPalavra))
6         return false
7     no = obterNoDaClasse(classeDaPrimeiraPalavra)
8     for(palavra in palavras) do
9         proximaPalavra = obterProximaPalavra()
10        classeDaProximaPalavra = obterClassePalavra(proximaPalavra)
11        if(!no.permiteTransicao(classeDaProximaPalavra))
12            return false
13        no = obterNoDaClasse(classeDaProximaPalavra)
14    if(no.ehFinal())
15        return true
16    else
17        return false
```

Algoritmo 1 - Algoritmo de Análise de Nomenclatura

Para validar se um nome é considerado válido ou não, em primeiro lugar é necessário separar este nome em suas partes. Esta separação é feita na linha 2 do algoritmo, onde um vetor de *strings* é gerado para representar as palavras do nome. O critério para a separação das *substrings* é a ocorrência de uma letra maiúscula (seção 4.7.3). Em seguida, verifica-se a classe da primeira palavra (linha 3). Se o nó inicial permite transição para a classe gramatical da primeira palavra, a verificação continua, caso contrário o nome é inválido. O resto do processamento se resume em um laço de repetição entre as palavras que compõem o nome analisado. Para cada palavra tenta-se identificar a classe gramatical da próxima palavra. Se o nó não permite transição para a classe gramatical da próxima palavra, o nome é considerado inválido e a iteração é finalizada. Caso contrário, a iteração continua e o nó é incrementado da seguinte maneira: Substitui-se o nó atual pelo nó associado à classe gramatical da próxima palavra. Ao final do laço de repetição (linha 14), é verificado se o nó é final, isto é, se de acordo com a regra de nomenclatura a classe gramatical associada ao nó pode ser a última no nome analisado. Se sim, o nome é válido. Caso contrário, é inválido.

O algoritmo é genérico e serve tanto para nomenclatura de classes quanto para nomenclatura de métodos. Esse desacoplamento é muito útil, pois as regras de transições entre os nós podem ser alteradas sem que haja impacto no algoritmo. Se por algum motivo deseja-se adicionar uma nova transição ou remover uma existente de um nó para outro nó em determinada regra, o algoritmo continua intacto.

Os fluxos foram desenvolvidos através de enumeradores que são as manifestações das matrizes de adjacência em nível de código. Cada enumerador possui um conjunto de itens enumerados, isto é, os nós. Assim, os enumeradores são responsáveis por definir as regras de nomenclatura compostas de nós que representam classes gramaticais (substantivo, adjetivos, verbos) levadas em consideração e obtidas através do vocabulário da língua portuguesa. Com base na regra que foi definida para a formação de uma expressão válida, cada item enumerado (nó) determina para quais outros nós é possível ir.

A interface *AlgoritmoDeAnaliseDeNomenclatura* deverá ser implementada por quem for realizar a validação do fluxo de palavras, para isso ela possui um método responsável por retornar o estado referente à palavra analisada (Figura 17).

```

public interface AlgoritmoDeAnáliseDeNomenclatura {
    /**
     * Método que obtém o estado referente à palavra analisada de acordo com a regra
     * que foi definida para a nomenclatura do componente.
     * @param palavra - vocábulo presente em uma expressão a ser validado
     * @return Estado referente à palavra analisada
     */
    public AlgoritmoDeAnáliseDeNomenclatura proximo(String palavra);

    public AlgoritmoDeAnáliseDeNomenclatura encerrar();
}

```

Figura 17 – Interface Algoritmo de análise de nomenclatura

Todos os enumeradores são iniciados no estado *inicial* e podem passar ao estado *falha* caso o nome analisado não seja reconhecido pelo fluxo ou pertença a uma classe gramatical que não tenha um caminho válido a partir do estado atual (ou seja, classe gramatical atual). Um exemplo é o primeiro estado da nomenclatura de métodos que deve ser um verbo, caso contrário o fluxo passa para o estado de falha, como verificado na implementação da enumeração *FluxoParaMetodos* (Figura 18).

```

public enum FluxoDePalavrasParaMetodo implements AlgoritmoDeAnáliseDeNomenclatura {
    inicial
    {
        @Override
        public AlgoritmoDeAnáliseDeNomenclatura proximo(String palavra) {
            if (Vocabulario.instancia().ehVerbo(palavra)) {
                return verbo;
            }
            return falha;
        }
    }
}

```

Figura 18 - Implementação de fluxo de palavras para nomenclatura de métodos

A utilização dos fluxos de palavras é orquestrada pelo Analisador Sintático, que itera pelas *substrings* que compõe o nome de um método/classe/variável e verifica se, ao final da iteração, o estado é válido.

4.7.3 Auxiliar de Nomenclatura

A classe *AuxiliarDeNomenclatura* é uma classe que possui métodos estáticos que servem de apoio para a validação dos itens de nomenclatura através do processamento de cadeias de caracteres. Abaixo estão listados seus principais métodos junto com uma breve descrição de sua implementação:

Método: Obter Palavras. Este método recebe como parâmetro uma cadeia de caracteres e retorna um vetor de cadeia de caracteres em caixa baixa que representa o argumento fragmentado, onde cada item deste vetor é uma *substring* dele. A cadeia é separada a cada ocorrência de um caractere em maiúsculo. Isto é feito através de uma expressão regular. Abaixo seguem exemplos de nomes e o retorno do método:

- "cadeiaDeCaracteres" - retorna o vetor = ["cadeia", "de", "caracteres"].
- "Cadeiadecaracteres" - retorna o vetor = ["cadeiadecaracteres"].

Método: Esta Em Camel Case. Este método recebe uma cadeia de caracteres e retorna um valor booleano indicando se a cadeia de caractere está em *Camel Case*. Essa verificação é feita através de uma expressão regular. Abaixo seguem exemplos de nomes e o retorno do método:

- "metodoDeTeste" - retorna *true*.
- "MetodoDeTeste" - retorna *false*.

Método: Esta Em Pascal Case. Este método recebe uma cadeia de caracteres e retorna um valor booleano indicando se a cadeia de caractere está em *Pascal Case*. Essa verificação é feita através de uma expressão regular. Abaixo seguem exemplos de nomes e o retorno do método:

- "classeDeTeste" - retorna *false*.
- "ClasseDeTeste" - retorna *true*.

4.8 Algoritmos de Regras Gerais

Nesta seção serão descritos, de maneira geral, os algoritmos que foram desenvolvidos para validar regras do *checklist* que se aplicam a todas as camadas. A ideia principal é demonstrar de que forma as informações acerca do código são obtidas com base na AST (Seção 2.2.1) e como são feitas as descobertas de erros.

4.8.1 Nomenclatura de Pacotes

Para validar a regra de nomenclatura de pacotes (Regra 1), primeiramente, o algoritmo deve indicar que o nó visitado será a raiz da estrutura gerada. No caso da implementação da *AST* utilizada neste trabalho, o nome deste tipo de nó é

CompilationUnit. Um exemplo de parte da AST gerada para um código referente a uma classe no pacote “*br.unirio.tcc.dominio*” pode ser visto na Figura 19.

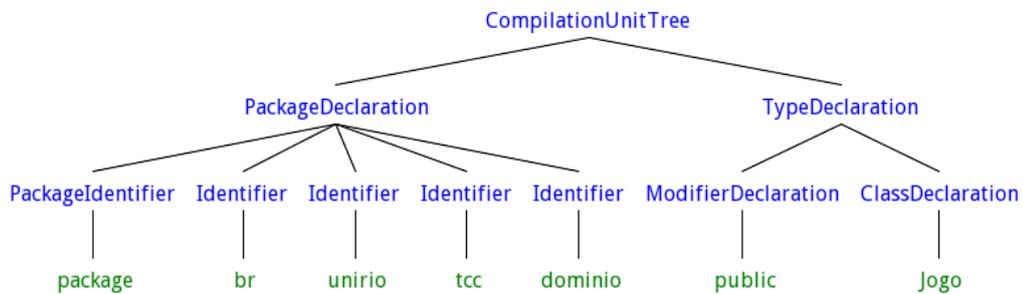


Figura 19 - Árvore de exemplo para classe Jogo

Depois de declarar ao *Visitor* qual será o tipo de nó visitado, as principais operações executadas são:

1. Obter a subárvore a partir do nó raiz e então obter a subárvore que detém o nome do pacote (Algoritmo 2);
2. Obter o nome do pacote analisado através da árvore de declaração de pacote. Essa subárvore deve ser percorrida a partir do segundo nó filho, pois o primeiro armazena o *token* que declara um pacote em Java (*package*). Os nós filhos - do tipo folha - armazenam o valor textual (*tokenValue*) contido na declaração do pacote (Algoritmo 3);
3. Por fim, armazena-se o nome completo do pacote retirando o “.” e inserindo os nomes em uma estrutura de dados. A partir desse ponto é possível validar se o nome do pacote corresponde ao padrão, verificando se o primeiro elemento da estrutura corresponde à nomenclatura “br”, o sucessor à nomenclatura “unirio” e se o quarto item possui uma das nomenclaturas permitidas para camadas. O terceiro item corresponde ao nome do projeto e não é validado. Além disso, também se verifica o número de itens inferior a quatro. Caso alguma dessas regras sejam infringidas o algoritmo deve indicar uma discrepância naquele pacote (Algoritmo 4). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore em que se encontra o erro. Assim é possível indicar a localização exata no código onde o problema está sendo apontado.

4.8.2 Nomenclatura de Classes

Para validar as regras de nomenclatura de classes (Regra 2 e Regra 3), o algoritmo desenvolvido deve indicar primeiramente ao *Visitor* que o nó a ser visitado é o de declaração da classe (*Class Declaration*). Após isso, os seguintes passos devem ser seguidos:

1. Obter o nome completo da classe a partir da subárvore de declaração e chamar o método para validar se a classe está em *Pascal Case* armazenando o valor em uma variável (Algoritmo 5);
2. Validar se a classe está em *Pascal Case* utilizando o método auxiliar *estaEmPascalCase()* (Seção 4.7.3);
3. Verificar se o nome da classe corresponde ao padrão aceito conforme a estrutura definida do fluxo de palavras para nomenclatura de classe (Tabela 2). No Algoritmo 6, primeiramente se obtém as palavras que compõem o nome da classe (linha 2) e o fluxo assume o estado inicial (linha 3). Cada palavra que compõe o nome da classe passa pelo fluxo que passa a assumir o estado correspondente à classificação gramatical (linha 5). Por fim, o fluxo é encerrado e se verifica se o mesmo está no estado “fim”, caso contrário indica-se que o nome da classe não passou pela validação (linha 8);
4. Indicar as discrepâncias encontradas que podem ser relativas à nomenclatura em *Pascal case* ou o sobre o padrão de nomenclatura aceito. Neste caso existe a exceção que é o *RepositorioGenerico* que possui uma regra de nomenclatura distinta (Algoritmo 7). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro.

4.8.3 Nomenclatura de Métodos

Para validar as regras de nomenclatura de métodos (Regra 4 e Regra 5), o algoritmo indica primeiramente ao *Visitor* que o nó a ser visitado é o de declaração de métodos (*Method Declaration*). Após isso, os seguintes passos devem ser seguidos:

1. Se o nó visitado for a raiz, obter a subárvore correspondente.

2. Obter o nome do pacote através da árvore de expressão de declaração do pacote (Algoritmo 8 – linha 4).
3. Verificar se o pacote pertence ao domínio ou serviços, que são as camadas em que a regra deve ser aplicada (Algoritmo 9).
4. Verificar se o método a ser analisado é do tipo *get* ou *set*, visto que o algoritmo não deve gerar discrepâncias que apontem erro na formação das palavras para esse caso que é uma exceção prevista na regra (Algoritmo 11).
5. Verificar se o método é a sobrescrita de algum método, através da verificação se o mesmo contém a anotação *@Override*, visto que o algoritmo não deve gerar discrepâncias que apontem erro na formação das palavras para esse caso (Algoritmo 12).
6. Verificar se o nome do método corresponde ao padrão aceito, conforme a matriz de adjacência (Tabela 1). No Algoritmo 10, primeiramente se obtém as palavras que compõem o nome do método (linha 2) e fluxo de palavras assume o estado inicial (linha 3). Cada palavra que compõe o nome do método passa pelo fluxograma de análise de nomenclatura que passa a assumir o estado correspondente à classificação gramatical (linha 5). Por fim, o fluxograma de análise de nomenclatura é encerrado e se verifica se a mesma está no estado “fim”, caso contrário indica-se que o nome do método não passou pela validação (linha 8).
7. Indicar as discrepâncias para o caso em que a nomenclatura do método não está em *camel case*, aplicando essa validação a todos os métodos (Algoritmo 13).
8. Indicar as discrepâncias para o caso em que o método pertence a um pacote válido, não é do tipo *get* ou *set*, não é a sobrescrita de um método e não passou pela validação feita com auxílio do fluxo de palavras para métodos (Algoritmo 13). Em ambos os casos de discrepância, o algoritmo passa uma mensagem sobre o erro que está sendo adicionado e qual a árvore da classe em que se encontra o mesmo.

4.9 Algoritmos de Regras da Camada de Apresentação

Nesta seção serão descritos, de maneira geral, os algoritmos que foram desenvolvidos para validar regras do *checklist* que se aplicam aos componentes que, segundo a arquitetura, devem estar presentes na camada de apresentação.

4.9.1 Uso de Controlador Genérico

Para realizar a validação da regra sobre uso de controlador genérico (Regra 8) é utilizada a premissa de que toda classe que usa a anotação *@Controller* representa um controlador. Primeiramente, o algoritmo precisa indicar que o nó visitado será o do tipo de declaração de classe. Depois de indicar ao *Visitor* qual será o nó visitado os seguintes passos devem ser executados

1. Obter a subárvore relativa à declaração da classe (Algoritmo 14).
2. Verificar se a classe possui a anotação *@Controller*, armazenando o resultado em uma variável (Algoritmo 15 e Algoritmo 16).
3. Verificar se a árvore de declaração da classe possui uma subárvore que identifique a classe estendida e validar se o nome da mesma é “*Controlador Generico*” armazenando o resultado em uma variável (Algoritmo 16).
4. Apontar as discrepâncias caso a classe tenha a anotação *@Controller*, não estenda da classe *ControladorGenerico* e excluir o caso em que o próprio *ControladorGenerico* está sendo analisado. O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro (Algoritmo 17).

4.9.2 Localização e Sufixo de Controladores

Para realizar a validação da regra de anotação *Controller* e sufixo da nomenclatura de controladores (Regra 10 e Regra 11), o algoritmo indica primeiramente ao *Visitor* que visitará dois nós: o correspondente à raiz da AST e o de declaração da classe. Após isso, os seguintes passos devem ser executados:

1. Se o nó visitado é a raiz, obter a subárvore de compilação e chamar função que verifica se o pacote é o de apresentação (Algoritmo 18 – linhas 3 e 4).
2. Verificar se o pacote da classe é o de apresentação (Algoritmo 19).

3. Se o nó visitado for o de declaração da classe, obter a subárvore da classe (Algoritmo 18 – linha 6) e chamar função que verifica se a classe possui uma determinada anotação (Algoritmo 15), passando “*Controller*” como parâmetro.
4. Verificar se a classe possui o sufixo *Controller* (Algoritmo 20). Entretanto é necessário verificar se a classe analisada não é “ControladorGenerico” que é a exceção para a regra.
5. Indicar as discrepâncias de acordo com as informações obtidas para o caso em que a classe possui a anotação e não está no pacote de apresentação e para o caso em que a classe possui a anotação, está no pacote correto, mas a sua nomenclatura não tem o sufixo *Controller* (Algoritmo 21). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro.

4.10 Algoritmos de Regras da Camada de Serviços

Nesta seção serão descritos, de maneira geral, os algoritmos que foram desenvolvidos para validar regras do *checklist* que se aplicam aos componentes que, segundo a arquitetura, devem estar presentes na camada de serviços.

4.10.1 Nomenclatura de Serviços

No caso da validação da regra da nomenclatura de serviços (Regra 16), a premissa utilizada é a de que toda classe que utiliza a anotação *@Service* representa um serviço. O algoritmo implementado não possui o conhecimento dos sufixos permitidos e valida somente o prefixo “*ServicosDe*”. Primeiramente, o mesmo deve indicar ao *Visitor* que visitará dois nós: o correspondente à raiz da *AST* e o de declaração da classe. Após isso, os seguintes passos devem ser executados:

1. Se o nó visitado é a raiz, obter a subárvore de compilação e chamar função que verifica se o pacote é o de serviços (Algoritmo 22 – linhas 3 e 4).
2. Verificar se o pacote da classe é o de serviços (Algoritmo 19).
3. Se o nó visitado for o de declaração da classe, obter a subárvore da classe (Algoritmo 22 – linha 6) e chamar função que verifica se a classe possui uma determinada anotação (Algoritmo 15) passando “*Service*” como parâmetro.

4. Verificar se a classe possui em seu nome os sufixos “*ServicosDe*” e “Log” que são aceitos pela regra como corretos (Algoritmo 23).
5. Indicar as discrepâncias encontradas de acordo com as informações obtidas para o caso em que a classe está no pacote de serviços e não possui a anotação *@Service*, levando-se em conta que a classe não possui a nomenclatura de Log, pois esta possuirá outra anotação. Além disso, deve ser indicada discrepância para o caso em que a classe está no pacote de serviço, possui a anotação correspondente mas a nomenclatura da mesma não possui o sufixo “*ServicosDe*” (Algoritmo 24). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro de nomenclatura.

4.10.2 Log interceptador na camada de Serviços

Para validar a regra da localização das classes que representaram interceptadores para log e suas respectivas nomenclaturas (Regra 19 e Regra 20) é utilizada a premissa de que as classes de log interceptadoras usam a anotação *@Aspect* e pelo menos um de seus métodos possui a anotação *@Around*. Para fazer a validação, o algoritmo deve indicar ao *Visitor* que visitará dois nós: o correspondente à raiz da *AST* e o de declaração da classe. Após isso, os seguintes passos devem ser executados:

1. Se o nó visitado é a raiz, obter a subárvore de compilação e chamar função que verifica se o pacote é o de serviços (Algoritmo 25– linhas 3 e 4).
2. Verificar se o pacote da classe é o de serviços (Algoritmo 19).
3. Se o nó visitado for o de declaração da classe, obter a subárvore da classe (Algoritmo 25 – linha 6) e chamar função que verifica se a classe possui uma determinada anotação (Algoritmo 15) passando “*Aspect*” como parâmetro.
4. Verificar se a classe possui a anotação *@Around* em algum de seus métodos (Algoritmo 26). Isso é feito percorrendo-se todas as subárvores de declaração de método e suas respectivas subárvores de declarações de modificadores. É importante verificar se o modificador corresponde a uma subárvore de anotação.

5. Indicar as discrepâncias encontradas de acordo com as informações obtidas para o caso em que a classe possui anotação *@Aspect* e um método com anotação *@Around* e não se encontra no pacote de serviços ou no caso de não possuir a nomenclatura com prefixo “*Log*” (Algoritmo 27). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro de nomenclatura.

4.11 Algoritmos de Regras da Camada de Domínio

Nesta seção serão descritos, de maneira geral, os algoritmos que foram desenvolvidos para validar regras do *checklist* que se aplicam aos componentes que, segundo a arquitetura, devem estar presentes na camada de domínio.

4.11.1 Localização e Nomenclatura de Classes Value Objects

Para validar as regras sobre a nomenclatura e localização de classes *Value Objects* (Regra 22 e Regra 23), o algoritmo deve indicar primeiramente que visitará tanto o nó raiz da *AST* para obter o nome do pacote quanto o nó de declaração da classe para obter o nome da mesma, no caso da implementação da *AST* utilizada neste trabalho, o nome do nó se chama *Class Declaration*. Após indicar ao *Visitor* esses dois tipos, os seguintes passos devem ser executados:

1. Verificar se o nó visitado é a raiz da árvore ou o nó de declaração da classe e obter a subárvore correspondente. (Algoritmo 28).
2. Se o nó visitado for do tipo raiz, obter o nome de pacote conforme abordado no Algoritmo 3 e validar se o mesmo corresponde ao pacote *domínio.vo* retornando o resultado a ser armazenado em uma variável. (Algoritmo 29).
3. Se o nó visitado for do tipo de declaração da classe, obter a árvore correspondente e, através dela, verificar se o nome da classe termina com o sufixo “*Vo*” utilizando um método de *obterPalavras()* para quebrar o nome completo em cada palavra e armazená-las em uma estrutura de dados. É importante ressaltar que neste caso a sigla “*Vo*” não pode estar escrita com ambas as letras em maiúsculo. (Algoritmo 30).
4. Por fim, realizar a verificação de discrepâncias analisando se a classe termina com o sufixo “*Vo*” e está no pacote *domínio.vo* e se existe alguma classe do

pacote *domínio.vo* sem o sufixo “Vo”. (Algoritmo 31). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro de nomenclatura ou que está localizada no pacote incorreto.

4.11.2 Mapeamento Relacional Lazy x Eager

Para validar a regra de mapeamento *eager* (Regra 24), o algoritmo deve indicar primeiramente que visitará todos os nós que representam a estrutura de anotação da linguagem Java. No caso da implementação da *AST* utilizada neste trabalho, o nome deste tipo de nó é *Annotation*. Depois de declarar ao *Visitor* o tipo de nó a ser visitado os seguintes passos devem ser executados:

1. Para cada anotação encontrada deve ser obtida a árvore correspondente, a mesma conterá diversos nós filhos correspondentes às outras estruturas de uma anotação como, por exemplo, uma propriedade e seu respectivo valor (Algoritmo 32).
2. Obter o nome da anotação a partir da árvore e verificar, através dele, se a anotação encontrada corresponde a uma indicação de relacionamento no *JPA* (Algoritmo 33).
3. Caso seja uma anotação de relacionamento, obter a subárvore correspondente aos argumentos da mesma e validar se existe algum com a propriedade “*fetch*” indicando o mapeamento *Eager* (Algoritmo 34). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da anotação em que foi encontrada a propriedade *Eager*.

4.12 Regras da Camada de Infraestrutura

Nesta seção serão descritos, de maneira geral, os algoritmos que foram desenvolvidos para validar regras do *checklist* que se aplicam aos componentes que, segundo a arquitetura, devem estar presentes na camada de infraestrutura.

4.12.1 Tecnologia de Acesso a Dados do Repositório Genérico

Para validar a regra de sufixo do Repositório Genérico (Regra 25) a premissa utilizada é a de que toda classe que usa a anotação *@Repository* representa um repositório. O algoritmo deve indicar ao *Visitor* que visitará o nó de declaração da classe. Após essa etapa, os seguintes passos devem ser realizados:

1. Obter a subárvore de declaração da classe (Algoritmo 35 – linha 2);
2. Verificar se a classe possui a anotação utilizando o Algoritmo 15 passando como parâmetro o valor “*Repository*” (Algoritmo 35 – linha 3).
3. Verificar se a classe possui o papel de repositório genérico. Neste caso verifica-se a nomenclatura da classe iniciando com “*RepositoryGenerico*” (Algoritmo 36).
4. Indicar as discrepâncias encontradas para classes que contenham a anotação que indique o repositório e possuam a nomenclatura de Repositório genérico e não possuam o sufixo que identifique qual é o tipo de tecnologia para acesso a dados, neste caso os valores aceitos são “*Jdbc*” e “*Jpa*” (Algoritmo 37). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore da classe em que se encontra o erro.

4.12.2 Localização de Classes de envio de e-mail

Para validar a regra de localização das classes que possuem algum método que envia e-mail (Regra 26), o algoritmo deve indicar ao *visitor* que visitará o nó da raiz da AST (Seção 2.2.1). Após essa etapa, os seguintes passos devem ser seguidos:

1. Obter a subárvore de compilação a partir do nó raiz (Algoritmo 38 – linha 2).
2. Obter uma lista de subárvores do tipo *import* que representam as estruturas de importação de bibliotecas em Java (Algoritmo 38 – linha 3).
3. Verificar a camada da classe analisada utilizando o Algoritmo 19 e passando como parâmetro “infraestrutura” (Algoritmo 38 – linha 4);
4. Verificar a partir das árvores de *imports* da classe se existe alguma que contenha a importação “*org.springframework.mail*” referente ao framework previsto na arquitetura (Algoritmo 39). Para isso, o algoritmo percorre cada árvore de *import* da classe e obtém a árvore que representa a expressão. Cada nó filho da subárvore de expressão corresponde a um valor da mesma, sendo o primeiro a própria palavra reservada da linguagem (*import*). É preciso,

portanto, obter os nós irmãos da mesma e armazenar o valor presente em cada nó do tipo folha. Por fim é feita a validação se algum dos *imports* é relativo à biblioteca de envio de e-mail.

5. Indicar a discrepância encontrada para o caso de classe que contém importação de biblioteca de e-mail e que não se encontra no pacote de infraestrutura (Algoritmo 40). O algoritmo passa uma mensagem sobre o tipo de discrepância que está sendo adicionada e qual a árvore de compilação em que se encontra a importação da biblioteca de e-mail na classe que está localizada no pacote incorreto.

4.13 Resumo do capítulo

O presente capítulo abordou os aspectos tecnológicos envolvidos neste trabalho. Primeiramente foi apresentada a proposta de automatização da análise de código e apresentadas as ferramentas que deram suporte ao desenvolvimento das regras responsáveis pela análise estática do código em Java. A estrutura do *plug-in* e os pontos de extensão da ferramenta foram tratados e, por fim, foram abordados os algoritmos que serviram como base para a lógica do código do *plug-in*, desenvolvido na linguagem Java, para contemplar a análise das regras do *checklist* que foram escolhidas para serem automatizadas. Além de apresentar os algoritmos, as regras foram brevemente explicadas.

Capítulo 5: AVALIAÇÃO DO PLUG-IN PARA ANÁLISE ESTÁTICA DE CÓDIGO

Este capítulo apresenta a avaliação da solução desenvolvida neste trabalho. A Seção 5.1 apresenta o projeto que foi utilizado como cenário da avaliação enquanto a Seção 5.2 exhibe os resultados encontrados, tanto aqueles que demonstram o funcionamento das regras quanto ocorrências de falsos positivos que ilustram algumas limitações dos algoritmos.

5.1 Projeto analisado

Para a elaboração desta avaliação foi escolhido como alvo um sistema Web na linguagem Java chamado “*Octopus*” que foi desenvolvido pelos autores do presente trabalho na disciplina de Desenvolvimento em Servidores Web (DSW) do Bacharelado em Sistemas de Informação (BSI) da Universidade Federal do Estado do Rio de Janeiro. O sistema corresponde a um bolão da Copa do Mundo de 2014 e foi desenvolvido com o intuito de aprender os conceitos de programação para a Web e aprendizado de frameworks que auxiliem o desenvolvimento.

A arquitetura original do projeto foi feita inicialmente pelos integrantes do grupo, mas foi adaptada para seguir a arquitetura mencionada na seção 4.5. A avaliação da solução consiste em verificar se as regras customizadas são eficazes na garantia da conformidade do código em relação à arquitetura.

A fim de demonstrar o funcionamento dos algoritmos em determinadas violações de regras, foram incluídos alguns exemplos de violações de determinadas regras, por exemplo, a presença do log interceptador na camada de serviços (Regra 19) e o uso de controlador genérico (Regra 8).

O Projeto Octopus possui:

- 3.751 linhas de código (de um total de 5.693 linhas);
- 57 arquivos;
- 13 pacotes (contabilizando sub-pacotes);
- 59 classes;
- 307 métodos;

A organização do projeto pode ser vista na Figura 20. A Figura 21 ilustra a página inicial do sistema. O *Octopus* disponibilizava a funcionalidade para os usuários apostarem nos resultados das partidas realizadas na Copa do Mundo de 2014, bem como criarem e participarem de grupos de apostas. O sistema realizava o cálculo da pontuação dos usuários de acordo com as apostas e atualizava um *ranking* geral e um no contexto de cada grupo em particular. O administrador do sistema era o responsável por atualizar os resultados e indicar o término das fases da Copa do Mundo. O sistema possuía a funcionalidade “Pergunte ao Povo Paul” que gerava para o usuário uma aposta com base no favoritismo das equipes que se enfrentariam em uma determinada partida.

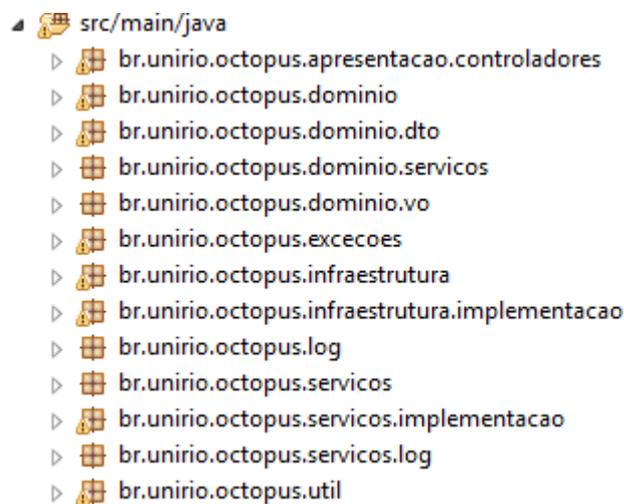


Figura 20 - Estrutura da Aplicação

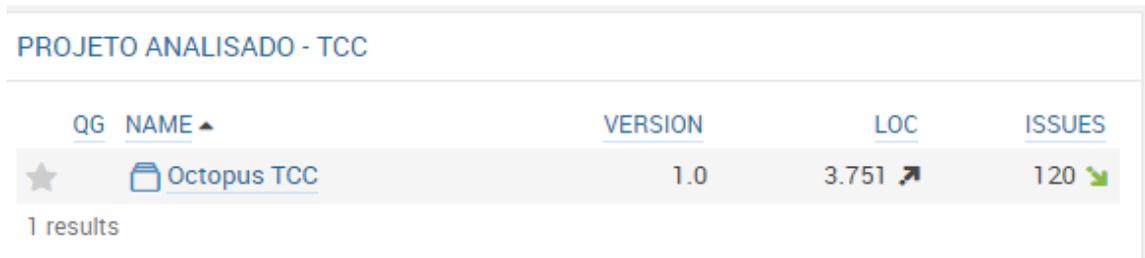


Figura 21- Página Inicial do sistema analisado

5.2 Demonstração dos resultados encontrados

O *checklist* definido neste trabalho para a inspeção de código contém 31 regras, sendo que destas, 16 regras foram automatizadas através da codificação do plug-in para a ferramenta *SonarQube*.

O total de erros encontrados na análise automática de código, incluindo aqueles que foram propositalmente adicionados no código como forma de exemplificar o funcionamento, foi igual a 120, conforme Figura 22. Desses 120 erros, foram adicionados 6 de forma proposital para ilustrar o funcionamento dos algoritmos.



The screenshot shows the SonarQube interface for a project named 'PROJETO ANALISADO - TCC'. It features a table with the following columns: 'QG', 'NAME', 'VERSION', 'LOC', and 'ISSUES'. A single entry is shown for 'Octopus TCC' with version '1.0', 3,751 lines of code (LOC), and 120 issues. A star icon is next to the project name, and a green arrow icon is next to the issue count. Below the table, it indicates '1 results'.

QG	NAME	VERSION	LOC	ISSUES
★	Octopus TCC	1.0	3.751 ↗	120 ↘

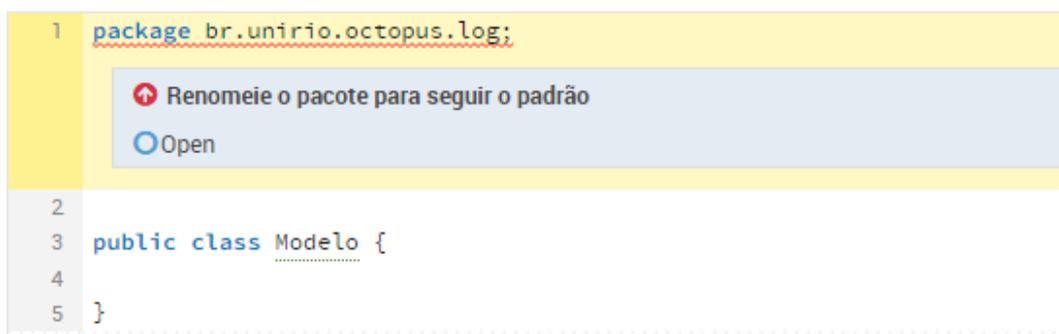
1 results

Figura 22 - Visão geral da análise

- **Violação da Regra 1 - Nomenclatura de Pacotes**

Ao executar a ferramenta *SonarQube* com o plug-in de regras codificado neste trabalho, foi encontrado um erro relativo à nomenclatura de um pacote (Regra 1), conforme pode ser visto na Figura 23. Neste caso, o nome do pacote está incorreto, pois o nome da camada não está entre os valores aceitos conforme a regra do *checklist*.

Uma solução para essa discrepância no código seria renomear o pacote ou mover o sub-pacote log para a camada de serviços.



The screenshot shows a code editor with the following code:

```
1 package br.unirio.octopus.log;
2
3 public class Modelo {
4
5 }
```

A tooltip is displayed over the package declaration, containing the text 'Renomeie o pacote para seguir o padrão' and an 'Open' button.

Figura 23 - Discrepância de nomenclatura de pacotes

- **Violação da Regra 3 - Nomenclatura de Classes**

A regra de nomenclatura de classes (Regra 3) foi avaliada e um dos erros encontrados pode ser visto na Figura 24. Neste caso o nome da classe não passou pela validação, pois a expressão é formada por:

- Previsão – Substantivo
- Palpite – Substantivo
- *Vo* – Adjetivo que foi adicionado ao vocabulário

Essa sequência não é válida segundo o fluxo de palavras para nomenclatura de classes. Uma possível solução para esse caso seria adicionar uma preposição tornando o nome da classe *PrevisaoDePalpiteVo*.

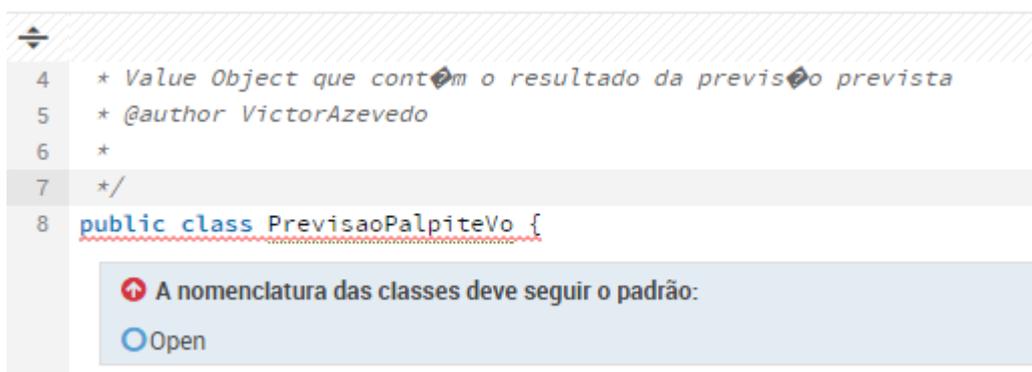


Figura 24 - Discrepância de nome de classe I

- **Violação da Regra 5 - Nomenclatura de Métodos**

A violação ilustrada pela Figura 25 demonstra uma nomenclatura incorreta de método. Neste caso as palavras que formam a expressão são:

- Palpite – Substantivo
- Valido – Adjetivo

Como a regra do *checklist* sobre nomenclatura de métodos (Regra 5) prevê que os métodos devem começar com um verbo, uma solução para este problema seria renomear o método para “*verificarPalpiteValido()*”.

```
38 private boolean palpiteValido(Palpite palpite) {  
    A nomenclatura dos métodos deve seguir o padrão:  
    Open  
39 Date dataDoJogo = palpite.getJogo().getDiaEHorario();
```

Figura 25 - Discrepância de nomenclatura de método

- **Violação da Regra 8 – Uso de Controlador**

Um exemplo de violação da regra que valida se as classes com anotação `@Controller` estendem de um Controlador Genérico pode ser visto na Figura 26.

```
8 *  
9 */  
10 @Controller  
11 public class ExibirImagemController {  
    A Classe faz o papel de Controlador e não estende do Controlador Genérico  
    Open
```

Figura 26 - Discrepância de não uso de controlador

- **Violação da Regra 11 – Sufixo de classe Controller**

Um exemplo de violação da nomenclatura do sufixo de controlador (Regra 11) pode ser visto na Figura 27. Neste caso a classe possui uma anotação `@Controller` e em sua nomenclatura não há o sufixo `Controller`.

```
28 @Controller  
29 public class UsuarioControlador extends ControladorGenerico {  
    A Classe que desempenha papel de Controlador ( utiliza annotation Controller) não possui nomenclatura de acordo com o padrão  
    Open  
30  
31
```

Figura 27 - Discrepância de sufixo Controller

- **Violação da Regra 16 – Nomenclatura de Serviços**

A violação da regra de nomenclatura de serviços (Regra 16) pode ser vista na Figura 28. Neste caso o problema é que o nome da classe não possui o sufixo

“*ServicosDe*”, tendo somente o nome do caso de uso associado. Uma solução para esse caso seria renomear o nome da classe para “*ServicosDeManterFase*” ou “*ServicosDeFase*”.

```
19 @Service
20 public class ManterFase implements IManterFase{
```

↑ A Classe que desempenha papel de serviço não possui nomenclatura de acordo com o padrão

○ Open

Figura 28 - Discrepância de nomenclatura de serviço

- **Violação da Regra 19 – Log Interceptador na camada de Serviços**

A Figura 29 ilustra a ocorrência de erro na localização da classe que faz o papel de interceptador para log utilizando a anotação `@Aspect`. Neste caso, a solução seria mover para o pacote “serviços” considerado como correto pelo *checklist*.

```
1 package br.unirio.octopus.log;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.Around;
5 import org.aspectj.lang.annotation.Aspect;
6
7 import br.unirio.octopus.servicos.log.Log;
8
9
10
11
12 @Aspect
13 public class LogInterceptador {
```

↑ A Classe que realiza o papel de interceptador de Log nao está na camada de serviços.

○ Open

```
14
15 @Around("execution(* br.unirio.octopus.servicos.implementacao.*.ServicosDe*.*(..))")
16 public Object criarLogServico(ProceedingJoinPoint joinPoint) throws Throwable{
17
```

Figura 29 - Discrepância na localização do log interceptador

- **Violação da Regra 22 – Localização de Classes com sufixo Vo**

A violação da regra relativa à localização de classes *Value Objects* (Regra 22) pode ser vista na Figura 30. Neste caso a classe possui a nomenclatura com sufixo “vo”, mas não está no sub-pacote “domínio.vo”, a solução para esta violação seria mover a classe para seu lugar correto segundo a arquitetura.

```
1 package br.unirio.octopus.dominio;
2
3 /**
4  * Value Object que contém o resultado da previsão prevista
5  * @author VictorAzevedo
6  *
7  */
8 public class PrevisaoPalpiteVo {
```

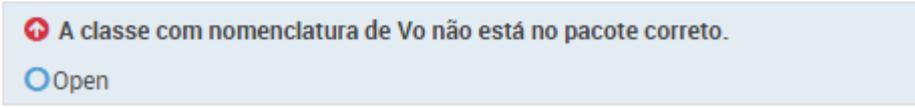


Figura 30 - Discrepância de localização de *value objects*

- **Violação da Regra 24 – Mapeamento Relacional Lazy x Eager**

Foram encontradas discrepâncias relativas à regra de mapeamento objeto-relacional utilizando *Eager* (Regra 24) em cinco ocorrências no código-fonte, conforme visto na Figura 31, sendo três delas na classe Jogo.



Figura 31 - Quantidade de ocorrências de Eager

Um dos erros encontrados é indicado na Figura 32. Neste caso, a entidade Jogo possui um relacionamento do tipo “Muitos para um” com a entidade Estádio e o relacionamento ocorre através da composição de Estádio na classe Jogo utilizando o *fetch* como Eager.

A solução para esse caso seria justificar a utilização desse mapeamento ou então modificar a estratégia para *lazy*. No caso de uma justificativa válida, a discrepância pode ser indicada como um falso positivo na própria ferramenta e a mesma passa a ignorar sua ocorrência.

```
53 @ManyToOne(fetch = FetchType.EAGER)
    Justifique a utilização do mapeamento Eager ou troque por Lazy
    Open
54 @JoinColumn(name = "IDEstadio")
55 private Estadio estadio;
```

Figura 32 - Discrepância de mapeamento eager

- **Violação da Regra 25 – Sufixo do Repositório Genérico**

Na Figura 33, é possível observar a violação da regra que valida o sufixo do Repositório Genérico definido para o projeto (Regra 25). Na classe em questão, o sufixo é “dados” quando, segundo a regra, deveria ser “jpa” ou “jdbc”.

```
23 import br.unirio.octopus.infraestrutura.IRepositorioGenericoDados;
24
25 @Repository
26 public abstract class RepositorioGenericoDados<T, TipoId> implements IRepositorioGenericoDados<T, TipoId> {
    A Classe de Repositorio Genérico não possui o sufixo com o Jpa ou Jdbc
    Open
27
```

Figura 33 - Discrepância no sufixo do repositório genérico

- **Violação da Regra 26 – Localização de classes de envio de e-mail**

A Figura 34 ilustra a violação da regra que ocorre, pois a classe analisada tem na sua estrutura de importações uma ou mais bibliotecas relativas ao envio de e-mail e a mesma não está localizada no pacote de infraestrutura conforme previsto na arquitetura de referência.

```
1 package br.unirio.octopus.servicos.implementacao;
2
3 import javax.mail.MessagingException;
4 import javax.mail.internet.MimeMessage;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.mail.MailException;
8 import org.springframework.mail.MailMessage;
9 import org.springframework.mail.SimpleMailMessage;
```

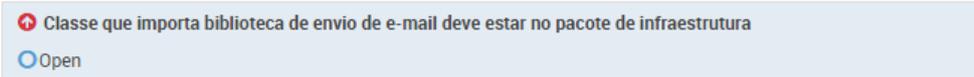


Figura 34 - Discrepância na classe de envio de e-mail

5.2.1 Falsos Positivos

Durante a execução de uma ferramenta de análise estática de código um dos grandes problemas é a ocorrência de falsos positivos que são indicações de violações das regras, mas que na verdade não o são. Uma forma de diminuir a incidência de falsos positivos é através do refinamento dos algoritmos utilizados para validar a regra e, no caso deste trabalho, o acréscimo de itens no vocabulário da Língua Portuguesa que pode assumir vocábulos de origem estrangeira, algo bastante corriqueiro no desenvolvimento de sistemas.

Um exemplo de falso positivo é a violação da nomenclatura do Log. Apesar de não fazer parte do vocabulário da língua portuguesa, a palavra “log” é comumente usada no meio da programação. A saída encontrada para eliminar este erro como falso positivo é adicionar o vocábulo “log” como um substantivo válido dentro do vocabulário de exceção utilizado pelo plug-in codificado neste trabalho (seção 4.7.1).

```
6
7 public class Log {
```

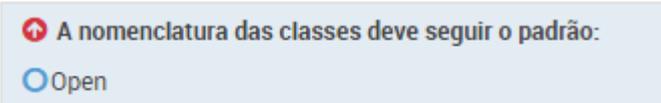


Figura 35 - Falso positivo de nomenclatura de log

Outro exemplo de discrepância é apresentado na Figura 36. Neste caso, o algoritmo que valida a nomenclatura de métodos adicionou uma discrepância para o método debug da classe *Log*. Apesar da regra do *checklist* indicar que a nomenclatura dos métodos deve ser iniciada com verbos, essa situação ilustra uma provável

exceção, não só no método `debug()` como em outros no contexto de *log* da aplicação como *error()* e *info()* que são comumente utilizados no desenvolvimento de sistemas. Esse é um exemplo de falso positivo que pode ser tratado no algoritmo utilizando o vocabulário de exceção (seção 4.7.1), mas para auxiliar a própria inspeção manual, seria uma boa prática descrever a regra de forma mais clara no *checklist*.

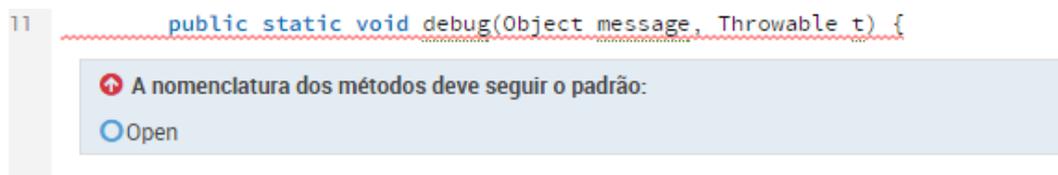


Figura 36 - Falso positivo para discrepância de nomenclatura de métodos

É importante ressaltar que as regras escolhidas para serem automatizadas geraram uma incidência de erros de nomenclatura de classes e métodos para casos em que a expressão pode fazer sentido para quem lê, contudo elas não estão de acordo com o padrão validado pelos algoritmos de nomenclatura. Caso seja detectado que alguma discrepância de nomenclatura possa ser considerada como um falso positivo, o plug-in pode ser melhorado com a adição de palavras ao vocabulário de exceção ou uma mudança no fluxo de palavras que serve como definição da regra. Um exemplo que ilustra essa situação é a discrepância apontada para o método “*EnviarEmailTexto()*”. Este caso é realmente uma discrepância pois o substantivo “Email” é procedido do substantivo “Texto” e essa transição não é aceita pela regra de nomenclatura de métodos (Regra 5), embora o nome do método possa ter um sentido semântico.

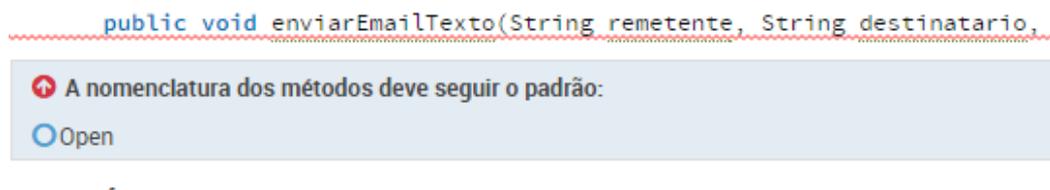


Figura 37 - Exemplo de discrepância a ser analisada

Capítulo 6: CONCLUSÃO

O presente trabalho teve como foco a elaboração de algoritmos para validação de regras arquiteturais, com o intuito de garantir através da análise estática a conformidade entre o código e a arquitetura. A inspeção do software produzido filtra falhas arquiteturais, mas é inviável de se executar manualmente em sistemas complexos devido à grande quantidade de linhas de código.

Os algoritmos foram implementados como resultado deste trabalho em forma de *plug-in* para a plataforma *SonarQube*, que provê meios para a análise estática de código. Para avaliar a solução desenvolvida, foi utilizado um sistema Web escrito na linguagem Java que seguia a arquitetura de referência com um conjunto de regras arquiteturais definidas em um *checklist*.

A análise do sistema *Octopus* com os algoritmos propostos identificou 120 discrepâncias. Dessas discrepâncias, 24 foram identificadas como falsos positivos, totalizando 96 discrepâncias reais. Assim, o valor da *precision*⁹ foi de 80%. Vale ressaltar que os falsos positivos identificados são relativos somente às discrepâncias de regras de nomenclatura. O principal motivo para a ocorrência destes falsos positivos decorre da necessidade de análise acerca do vocabulário utilizado (seção 4.7.1).

Para este trabalho não foi possível calcular o *recall*⁹, pois o conjunto universo de todas as violações de regra reais não é conhecido. Isso indica outra limitação, pois impossibilita o cálculo da sensibilidade dos algoritmos. Indicamos então como trabalho futuro a realização de uma inspeção manual minuciosa do código feita por diferentes desenvolvedores ou testadores para identificar regras que porventura não tenham sido encontradas pelos algoritmos de validação, encontrando desta forma o conjunto de todas as discrepâncias válidas. Além disso, este trabalho poderia ser comparado com o da inspeção automatizada no que diz respeito ao tempo gasto para a realização da atividade, com isso seria possível traçar um comparativo de tempo e discrepâncias encontradas entre as inspeções manuais e automatizadas para ilustrar qual seria a vantagem ao se automatizar essa atividade.

⁹ *Precision e Recall*. Mais informações em: https://en.wikipedia.org/wiki/Precision_and_recall

Tendo em vista o valor calculado da *precision*, a proposta obteve resultados positivos, pois a análise estática foi capaz de identificar falhas arquiteturais legítimas no código, muitas delas difíceis de serem identificadas, por exemplo, violações de nomenclatura de métodos e classes. Em uma inspeção manual algumas das falhas encontradas poderiam não ter sido identificadas devido ao conhecimento tácito na mente daqueles responsáveis pela inspeção. Em outras palavras, a visão subjetiva e o suposto entendimento do que o desenvolvedor quis fazer, porém sem fazê-lo de fato. Este tipo de subjetividade e parcialidade permite furos arquiteturais que podem se tornar constantes. Assim, fica claro uma importante característica da inspeção automática: a imparcialidade.

Outra limitação é a dependência de cada regra em relação as suas premissas e a complexidade envolvida em determinar se um elemento do código deve ser analisado ou não. Se uma premissa não é respeitada, a violação pode não ser identificada pelos algoritmos.

Como trabalhos futuros, sugerimos a criação de anotações para serem inseridas no código de maneira que desempenhem um papel descritivo acerca do objetivo daquele trecho de código, diminuindo a complexidade em determinar se um elemento do código deve ser analisado. Por exemplo, através do desenvolvimento de um meio de se consultar informações acerca de outros arquivos do projeto analisado em tempo de execução. Entretanto, isto é complexo de ser realizado por limitação imposta pela própria *API* da plataforma utilizada.

Outros possíveis trabalhos futuros são a criação de mecanismos utilizados para a validação de nomenclatura, como por exemplo, a melhoria das palavras do vocabulário com suas respectivas categorizações, o que aumentaria a cobertura das expressões que servem de entrada para o algoritmo. Outra sugestão seria um estudo acerca da utilização de premissas para a validação das regras que pode ser um limitador na inspeção automatizada de código. Além disso, seria interessante realizar um estudo do esforço necessário para a automatização de uma regra para verificar a viabilidade de automatizá-la.

Capítulo 7: REFERÊNCIAS BIBLIOGRÁFICAS

FAGAN, M. E.: **Design and Code Inspection to Reduce Errors in Program Development**”, IBM Systems Journal, 1976.

FAGAN, M. E.: **Advances in Software Inspection**, IEEE Transactions on Software Engineering, 1986.

GAMMA, Erich, HELM, Richard; JOHNSON, Ralph, VLISSIDES, John.: **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos**. 1. ed. Porto Alegre: Bookman, 2000.

GARLAN, D.: **Software Architecture: a Roadmap**. International Conference on Software Engineering: Future of SE Track, p. 91-101, Limerick, Ireland, 2000.

IBM (2013): **Documentação da ferramenta Rational Asset Analyzer**, disponível em <<http://www.ibm.com/developerworks/rational/library/manage-change-mainframe-applications-1/>>. Acessado em 01 de abril de 2015.

KOLLANUS, Sami; KOSKINEN, Jussi.: **Survey of Software Inspection Research: 1991-2005**. University of Jyväskylä, 2007. Artigo disponível em <http://users.jyu.fi/~kolli/research/Inspection_survey_WP.pdf>.

LAITENBERGER, O.; DEBAUD, J.: **An encompassing life cycle centric survey of software inspection**. Journal of Systems and Software, v. 50, n. 1, p. 5-31, 2000.

LASATER, C.G.: **Design Patterns**. Jones & Barlett Learning. Outubro de 2010.

MAVEN (2015): **Documentação da ferramenta Apache Maven**. Disponível em <<https://maven.apache.org/>>. Acessado em 11 de junho de 2015.

MENEZES, Paulo Blath.: **Linguagens Formais e Autômatos**. 3ª Edição, 2000.

PAUL, Clements; BACHMANN, Felix; BASS, Len; GARLAN, David; IVERS, James; LITTLE, Reed; MERSON, Paulo; NORD, Robert; STAFFORD, Judith .: **Documenting software architectures: Views and Beyond**. Second Edition, 2010.

PORTER, Adam A, VOTTA JR, Lawrence G., BASILI, Victor R.: **Comparing Detection Methods For Software Requirements Inspections: A replicated Experiment**.

PREE, Wolfgang.: **Meta Patterns – A Means For Capturing the Essentials of reusable object-oriented design**. 2006.

PRESSMAN, Roger S.: **“Engenharia de Software: uma Abordagem Profissional”**. 7ª edição. Porto Alegre: AMGH, 2011.

RODRIGUES, A.R.: **Um Framework Genérico para Geração de Texto em Linguagem Natural a partir de Modelos de Processo de Negócio**. UNIRIO 2013 – Projeto de Graduação, abril de 2013.

SHAW, M.; GARLAN, D.: **Software Architecture**. Prentice Hall, 1996.

Software Engineering Institute. Carnegie Mellon University.: **“Community Software Architecture Definitions”**. Disponível em <www.sei.cmu.edu/architecture/start/glossary/community.cfm>. Acessado em 21 de abril de 2015.

SONARQUBE (2015): **Extending Coding Rules**, disponível em <http://docs.sonarqube.org/display/SONAR/Extending+Coding+Rules>. Acessado em 01 de abril de 2015.

SCHACH, Stephen R.: **“Engenharia de Software.: Os Paradigmas Clássico e Orientado a Objetos”** . p.154. 7ª edição: MCGRAW-HILL BRASIL, 2008.

YOURDON, E.: **Just Enough Structured Analysis. Appendix D: Walkthroughs and Inspections**, Ed Yourdon, 2006. Disponível em <http://www.yourdon.com/PDF/oldJESA/JESA/JESAappD.pdf>

Apêndice I – Algoritmos para validação das regras

```
1 visitarNo (arvore)
2  arvoreCompilacao = obterArvoreDeCompilacao ()
3  if arvore != nulo then
4    arvoreDeExpressao = obterArvoreDeExpressao (arvore)
```

Algoritmo 2 – Visitar nó e obter subárvore correspondente

```
1 obterNomeDoPacoteAtravesDaArvore (arvoreExpressao)
2  arvoreDeclaracaoPackage = arvoreExpressao.obterPrimeiroFilho ()
3  arvore = arvoreDeclaracaoPackage.obterNoIrmao ()
4  while (arvore != nulo) do
5    nomeDoPacote += arvore.tokenValue ()
6    arvore = arvore.obterNoIrmao ()
7  end
8  return nomeDoPacote
```

Algoritmo 3 - Obter nome completo do pacote

```
1 validarNomeDoPacote (nomeDoPacote)
2  CAMADAS_CORRETAS = {
3    "dominio", "servicos", "infraestrutura", "apresentacao", "util", "excecoes"
4  }
5  elementos = nomeCompleto.split("\\.")
6  if (elementos.size () < 4) then
7    adicionarDiscrepancia (arvoreDeclaracaoPackage,
8      mensagemErroDePacote)
9  if (elementos[0] != "br" || elementos[1] != "unirio") then
10    adicionarDiscrepancia (arvoreDeclaracaoPackage,
11      mensagemErroDePacote)
12  if (CAMADAS_CORRETAS.contains (elementos.[3]) then
13    adicionarDiscrepancia (arvoreDeclaracaoPackage,
14      mensagemErroDePacote)
```

Algoritmo 4 - Validar nome do pacote

```

1  visitarNo (arvore)
2  arvoreDaClasse = obterArvoreDaClasse (arvore)
3  nomeDaClasse = arvoreDaClasse.tokenValue ()
4  estaEmPascalCase = estaEmPascalCase (nomeDaClasse)
5  ehNomeValido = validarNomeDeClasse (nomeDaClasse)

```

Algoritmo 5 - Obter nome da classe e obter informações

```

1  validarNomeDeClasse (nomeClasse)
2  palavras = obterPalavras (nomeClasse)
3  Estado estado = EstadosParaClasse.inicial
4  for (palavra in palavras) do
5      estado = estado.proximo (palavra)
6  end
7  estado = estado.encerrar ()
8  if (estado != EstadosParaClasse.fim) then
9      return false
10 return true

```

Algoritmo 6 - Validar nome de classe

```

1  verificarDiscrepancias ()
2  if (!estaEmPascalCase) then
3      adicionarDiscrepancia (arvoreDaClasse,
4                              mensagemErroPascalCase)
5  if (nomeDaClasse!startsWith("RepositorioGenerico") &&
6      !ehNomeValido) then
7      adicionarDiscrepancia (arvoreDaClasse,
8                              mensagemErroNomenclaturaDaClasse)

```

Algoritmo 7 - Verificar discrepâncias de nomenclatura de classe

```

1  visitarNo (arvore)
2  if (arvore.ehRaiz ()) then
3      arvoreDeExpressao = obterArvoreDeExpressao (arvore)
4      nomeDoPacote =
5          obterNomeDoPacoteAtravesDaArvore (arvoreDaExpressao)
6      ehPacoteValido = verificarPacote (nomeDoPacote)

```

```

7  if(arvore.ehMetodo()) then
8      arvoreDoMetodo = obterAvoreDoMetodo(arvore)
9      nomeDoMetodo = arvoreDoMetodo.tokenValue()
10     ehGetSet = verificarMetodoGetSet(arvoreDoMetodo)
11     ehSobrescrito = verificarMetodoSobrescrito(arvoreDoMetodo)
12     verificarDiscrepancia()

```

Algoritmo 8 - Visitar nó para regra de nomenclatura de métodos

```

1  validarPacote(pacote)
2  palavrasDoPacote = pacote.split("\\.")
3  if(palavrasDoPacote.length>4) then
4      if(palavrasDoPacote[3].equals("dominio") ||
5          palavrasDoPacote[3].equals("servicos")) then
6          return true
7  return false

```

Algoritmo 9 - Validar se é um pacote válido para a análise

```

1  validarNomeDeMetodo(nomeMetodo)
2  palavras = obterPalavras(nomeMetodo)
3  Estado estado = EstadosParaMetodo.inicial
4  for (palavra in palavras) do
5      estado = estado.proximo(palavra)
6  end
7  estado = estado.encerrar()
8  if (estado != EstadosParaMetodo.fim) then
9      return false
10 return true

```

Algoritmo 10 - Algoritmo de validação de nome de método

```

1  verificarMetodoGetSet(nomeDoMetodo)
2  palavras = Auxiliar.obterPalavras(nomeDoMetodo)
3  if(palavras.length>0) then
4      return (palavras[0] == "get" ||
5              palavras[0] == "set")
6  else
7  return false

```

Algoritmo 11 - Verificar se método é get ou set

```

1 verificarMetodoSobrescrito (arvoreDoMetodo)
2     arvoreDeModificadores = arvoreDoMetodo.obterModificadores ()
3     for (arvoreDeModificador in arvoreDeModificadores) do
4         if(arvoreDeModificador.ehAnotacao ()) then
5             arvoreAnotacao = obterArvoreAnotacao (arvoreDeModificador)
6             arvoreDoNome = arvoreAnotacao.obterArvoreDoNome ()
7             nomeDaAnotacao = arvoreDoNome.tokenValue ()
8             if (nomeDaAnotacao == "Override") then
9                 return true
10        end
11 return false

```

Algoritmo 12 - Verificar se o método é de sobrescrita

```

1 verificarDiscrepancia (arvoreDoMetodo)
2 if (!estaEmCamelCase (nomeDoMetodo)) then
3     adicionarDiscrepancia (arvoreDoMetodo,
4                             mensagemErroCamelCase)
5     if (ehPacoteValido && !verificarMetodoGetSet (nomeDoMetodo) &&
6         !verificarMetodoSobrescrito (arvoreDoMetodo)) then
7         ehNomeValido = validarNomeDeMetodo (nomeDoMetodo)
8         if (!ehNomeValido)
9             adicionarDiscrepancia (arvoreDoMetodo,
10                                    mensagemErroNomenclaturaDoMetodo)

```

Algoritmo 13 - Indicar discrepâncias para nomenclatura de métodos

```

1 visitarNo (arvore)
2 arvoreDaClasse = obterArvoreDaClasse (arvore) ;
3 possuiAnnotation =
4     verificarSeClassePossuiAnnotation (arvoreDaClasse, "Controller")

```

Algoritmo 14 - Obter subárvore da classe

```

1 verificarSeClassePossuiAnnotation (arvoreDaClasse, anotacao)
2 arvoreDeModificadores = arvoreDaClasse.obterModificadores ()
3 arvoresDeAnotacoes =
4     arvoreDeModificadores.obterArvoreDeAnotacoes ()
5 for (Arvore arvoreAnotacao in arvoresDeAnotacoes) do

```

```

5   arvoreDoNome = arvoreAnotacao.obterArvoreDoNome ()
6   nomeDaAnotacao = arvoreDoNome.tokenValue ()
7       if (nomeDaAnotacao == anotacao) then
8           return true
9   end
10  return false

```

Algoritmo 15 - Verificar se classe possui anotação

```

1  validarClasseEstendida (arvoreDaClasse)
2  arvoreDeExtends = arvoreDaClasse.obterArvoreExtends ()
3  if (arvoreDeExtends != nulo ) then
4      nomeDaSuperClasse = arvoreDeExtends.tokenValue ()
5      if (nomeDaSuperClasse == "ControladorGenerico" )
6          possuiSuperClasse = true

```

Algoritmo 16 - Validar se Classe estende Controlador Generico

```

1  verificaDiscrepancias (arvoreDaClasse)
2      nomeDaClasse = arvoreDaClasse.tokenValue ()
3      if (nomeDaClasse != "ControladorGenerico") then
4          if (possuiAnotacao && !possuiSuperClasse) then
5              adicionarDiscrepancia (arvoreDaClasse,
                                     mensagemErroControladorGenerico)

```

Algoritmo 17 - Verificar discrepância da regra

```

1  visitarNo (arvore)
2  if (arvore.ehRaiz ()) then
3      arvoreCompilacao = obterArvoreCompilacao ()
4      ehPacoteApresentacao = verificarCamada (arvoreCompilacao, "apresentacao")
5  else if (tree.ehClasse ()) then
6      arvoreDaClasse = obterArvoreDaClasse ()
7      possuiAnnotationController =
           verificarSeClassePossuiAnnotation (arvoreDaClasse, "Controller")
8      possuiSufixoController = possuiSufixoController (arvoreDaClasse)
9      verificarDiscrepancia (arvoreDaClasse)

```

Algoritmo 18 - Visitar nó para regra de anotação e sufixo de controladores

```

1  verificarCamada (arvoreCompilacao, nomeCamada)
2  arvoreDaExpressao = obterArvoreDeExpressao (arvoreCompilacao)
3  nomeDoPacote =

```

```

obterNomeDoPacoteAtravesDaArvore (arvoreDaExpressao)
4 elementos = nomeDoPacote.split("\\.")
5     if (elementos.size() >= 4) then
6         return (elementos[3] == nomeCamada)
7     else
8         return false

```

Algoritmo 19 - Verificar camada de uma classe a partir da árvore

```

1 possuiSufixoController (arvoreDaClasse)
2 nomeDaClasse = arvoreDaClasse.tokenValue ()
3 if (nomeDaClasse.equals ("ControladorGenerico")) then
4     return true
5 else
6     return nomeDaClasse.endsWith ("Controller")

```

Algoritmo 20 - Verificar se o sufixo da classe é Controller

```

1 verificarDiscrepancia (arvoreDaClasse)
2 if (possuiAnnotationController && !ehPacoteApresentacao) then
3     adicionarDiscrepancia (arvoreDaClasse,
4                             mensagemErroLocalizacaoPacote)
5 else if (possuiAnnotationController &&
6          ehPacoteApresentacao && !possuiSufixoController)
7     adicionarDiscrepancia (arvoreDaClasse,
8                             mensagemErroFaltaSufixoController)

```

Algoritmo 21 - Indicar discrepâncias de anotação e sufixo de controladores

```

1 visitarNo (arvore)
2 if (arvore.ehRaiz ()) then
3     arvoreCompilacao = obterArvoreDeCompilacao ()
4     ehPacoteServico = verificarCamada (arvoreCompilacao, "servicos")
5 else if (arvore.ehClasse ()) then
6     arvoreDaClasse = obterArvoreDaClasse ()
7     possuiAnnotationService =
8         verificarSeClassePossuiAnnotation (arvoreDaClasse, "Service")
9     possuiNomenclaturaDeServico =
10        verificaSePossuiNomeDeServico (arvoreDaClasse)
11    verificarDiscrepancia (arvoreDaClasse)

```

Algoritmo 22 - Visitar nó para regra de nomenclatura de serviços

```

1 verificarSePossuiNomeDeServico (arvoreDaClasse)

```

```

2 nomeDaClasse = arvoreDaClasse.simpleName().name()
3 return nomeDaClasse.startsWith("ServicosDe") ||
    nomeDaClasse.startsWith("Log")

```

Algoritmo 23 - Verificar se classe possui nome de serviço

```

1 verificarDiscrepancia(arvoreDaClasse)
2 possuiNomeLog = nomeDaClasse.startsWith("Log")
3 if(!possuiNomeLog) then
4     if (ehPacoteServico && !possuiAnnotationService) then
5         adicionarDiscrepancia(arvoreDaClasse,
            mensagemErroFaltaAnotacaoService)
6     if(ehPacoteServico && possuiAnnotationService &&
7         !possuiNomenclaturaDeServico) then
8         adicionarDiscrepancia(arvoreDaClasse,
            mensagemErroNomeDeServico)

```

Algoritmo 24 - Indicar discrepâncias de nomenclatura de serviços

```

1 visitarNo(arvore)
2 if (arvore.ehRaiz()) then
3     arvoreCompilacao = obterArvoreDeCompilacao()
4     ehPacoteServico =
verificarCamada(arvoreCompilacao,"servicos")
5     else if(arvore.ehClasse()) then
6         arvoreDaClasse = arvore.obterArvoreDaClasse()
7         possuiAnnotationAspect =
verificarSeClassePossuiAnnotation(arvoreDaClasse,"Aspect")
8         possuiMetodoComAround =
verificarAnnotationAround(arvoreDaClasse)
9         verificarDiscrepancia(arvoreDaClasse)

```

Algoritmo 25- Visitar nó para regra de log interceptador em serviços

```

1 verificarAnnotationAround(arvoreDaClasse)
2   for (arvore : arvoreDaClasse.obterMembros()) do
3     if (arvore.ehMetodo) then
4       arvoreDoMetodo = obterArvoreDoMetodo()
5       for (arvoreModificador : arvoreDoMetodo.obterModificadores()) do
6         if(arvoreModificador != null &&
arvoreModificador.ehAnotacao()) then
7           arvoreAnotacao =
obterArvoreDaAnotacao(arvoreModificador)
8           anotacoesEncontradas.add(arvoreAnotacao.tokenValue())
9         end
10      end
11 possuiMetodoComAround = anotacoesEncontradas.contains("Around")

```

Algoritmo 26 - Verifica se a classe possui algum método com anotação around

```

1 verificarDiscrepancia(arvoreDaClasse)
2   possuiPrefixoLog = arvoreDaClasse.tokenValue().startsWith("Log")
3   if(possuiAnnotationAspect && possuiMetodoComAround
4     && !ehPacoteServico) then
5     adicionarDiscrepancia(arvoreDaClasse,
        mensagemErroPacoteIncorreto)
6   if(possuiAnnotationAspect && possuiMetodoComAround
7     && !possuiPrefixoLog) then
8     adicionarDiscrepancia(arvoreDaClasse,
        mensagemErroPrefixoLog)

```

Algoritmo 27 - Indicar as discrepâncias encontradas para log interceptador na camada de serviços

```

1 visitarNo(arvore)
2   if (arvore.ehRaiz()) then
3     arvoreCompilacao = obterArvoreDeCompilacao(arvore)
4     arvoreDeExpressao =
obterArvoreDeExpressao(arvoreCompilacao)
5     ehPacoteVo = verificarPacoteVo(arvoreDeExpressao)
6   else if(tree.ehClasse()) then
7     arvoreDaClasse = obterArvoreDaClasse(tree)

```

```
8      ehClasseVo = verificarNomeDaClasse(arvoreDaClasse)
```

Algoritmo 28 - Obter subárvores correspondentes

```
1 verificarPacoteVo(arvoreDeExpressao)
2     nomeDoPacote =
obterNomeDoPacoteAtravesDaArvore(arvoreDeExpressao)
3     elementos = nomeDoPacote.split("\\.")
4     if(elementos.size()>=5) then
5         if(elementos[3] == ("dominio")) then
6             return elementos[4] == "vo"
7         else
8             return false
9     else
10        return false
```

Algoritmo 29 - Validar se é pacote vo

```
1 verificarNomeDeClasseVo(arvoreDaClasse)
2     nomeDaClasse = arvoreDaClasse.tokenValue()
3     palavrasDoNome = obterPalavras(nomeDaClasse)
4     return nomeCompletoDaClasse[nomeCompletoDaClasse.length-1] ==
"vo")
```

Algoritmo 30 - Verificar se a classe tem sufixo Vo

```
1 verificarDiscrepancia(arvoreDaClasse)
2     if (ehPacoteVo && !ehClasseVo) then
3         adicionarDiscrepancia(arvoreDaClasse,
                                mensagemNomenclaturaIncorreta)
4     else if(!ehPacoteVo && ehClasseVo) then
5         adicionarDiscrepancia(arvoreDaClasse,
                                mensagemLocalizacaoIncorreta)
```

Algoritmo 31 – Verificar discrepâncias da regra de nomenclatura e localização de classes Vo.

```
1 visitarNo(arvore)
2     arvoreDeAnotacao = obterArvoreDeAnotacao(arvore)
```

Algoritmo 32 - Obter árvore de anotação

```
1 verificarSeEhAnotacaoRelacionamento (arvoreDeAnotacao)
2   arvoreDoNome = arvoreAnotacao.obterArvoreDoNome ()
3   nomeDaAnotacao = arvoreDoNome.tokenValue ()
4   ANNOTATION_RELACIONAMENTOS = { "ManyToOne", "OneToMany",
                                   "ManyToMany", "Basic" }
5   if(ANNOTATION_RELACIONAMENTOS.contains(nomeDaAnotacao))
6     then
7     return true
8   else
9     return false
```

Algoritmo 33 - Verificar se é anotação de relacionamento

```
1 verificarSeEhEager (arvoreDeAnotacao)
2   arvoresDeExpressao = arvoreDeAnotacao.obterExpressoes ()
3   for(Arvore arvore in arvoresDeExpressao) do
4     arvoreDePropriedade = arvore.obterArvoreDePropriedade ()
5     nomeDaPropriedade =
6       arvoreDaPropriedade.arvoreNomeDaPropriedade.tokenValue ()
7     if(nomeDaPropriedade.tokenValue () == "fetch") then
8       arvoreDeValor = arvoreDePropriedade.arvoreDeValor ()
9       valorDaPropriedade = arvoreDeValor.tokenValue ()
10      if(valorDaPropriedade.contains("EAGER")) then
11        adicionarDiscrepancia (arvoreDeAnotacao,
12                               mensagemErroMapeamentoEager)
13    end
14  end
```

Algoritmo 34 - Verificar se existe propriedade *eager*

```
1 visitarNo (arvore)
2   arvoreDaClasse = obterArvoreDaClasse (tree)
3   possuiAnnotationRepository =
4     verificarSeClassePossuiAnnotation (arvoreDaClasse, "Repository")
5   temPapelDeRepositoryoGenerico =
6     verificarNomenclaturaDeRepositoryoGenerico (arvoreDaClasse)
7   verificarDiscrepancia ()
```

Algoritmo 35- Visitar regra para sufixo de Repositório Genérico

```

1 verificarNomenclaturaDeRepositorioGenerico (arvoreDaClasse)
2   nomeDaClasse = arvoreDaClasse.tokenValue ()
3   palavras = obterPalavras (nomeDaClasse)
4   if (nomeDaClasse.length >= 3) then
5     return (nomeDaClasse[0] == "repositorio"
6           && nomeDaClasse[1] == "generico")
7   else
8     return false

```

Algoritmo 36 - Verifica se a classe possui nomenclatura de repositório genérico

```

1 verificarDiscrepancia (arvoreDaClasse)
2   sufixosPermitidos = {"Jdbc", "Jpa"}
3   if (possuiAnnotationRepository && temPapelDeRepositorioGenerico)
4   then
5     nomeDaClasse = arvoreDaClasse.tokenValue ()
6     palavras = obterPalavras (nomeDaClasse)
7     if (!sufixosPermitidos.contains (
8         nomeDaClasse[nomeDaClasse.length-1])) then
9       adicionarDiscrepancia (arvoreDaClasse,
10          mensagemErroSufixoDoRepositorioGenerico)

```

Algoritmo 37 - Indica as discrepâncias encontradas para a regra de sufixo do repositório genérico

```

1 visitarNo (arvore)
2   arvoreCompilacao = obterArvoreCompilacao (arvore)
3   importsDaClasse = arvoreCompilacao.obterArvoreDeImports ()
4   ehPacoteInfraestrutura =
5     verificarCamada (arvoreCompilacao, "infraestrutura")
6   existeLibDeEmail = verificarSeExisteLibDeEmail (importsDaClasse)
7   verificarDiscrepancia (arvoreCompilacao)

```

Algoritmo 38 - Visitor nó para regra de localização de classes de envio de e-mail

```

1 verificarSeExisteLibDeEmail(importsDaClasse)
2   for (arvoreImport in importsDaClasse) do
3     arvoreDeExpressao = arvoreImport.obterExpressao()
4     arvoreDeclaracaoImport = arvoreDeExpressao.obterPrimeiroFilho()
5     arvore = arvoreDeclaracaoImport.obterNoIrmao()
6     while (arvore != nulo) do
7       nomeDoImport += arvore.tokenValue()
8       arvore = arvore.obterNoIrmao()
9     end
10    pacotesImportados.add(nomeDoImport)
11  end
12  for (pacote in pacotesImportados) do
13    if(pacote.startsWith(org.springframework.mail)) then
14      return true
15    end
16  return false

```

Algoritmo 39 - Verificar import de biblioteca de e-mail

```

1 verificarDiscrepancia(arvoreDeCompilacao)
2   if(existeLibDeEmail) then
3     if(!ehPacoteInfraestrutura) then
4       adicionarDiscrepancia(arvoreDeCompilacao,
                             mensagemErroImportEmailLocalizacaoIncorreta)

```

Algoritmo 40 - Verificar discrepância de classe de envio de e-mail fora da camada de infraestrutura

