UNIVERSIDADE FEDERAL DO AMAPÁ
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Matheus Costa Silva

UMA FERRAMENTA PARA IDENTIFICAÇÃO AUTOMÁTICA DE BAD SMELLS

Macapá
2021

UNIVERSIDADE FEDERAL DO AMAPÁ
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Matheus Costa Silva

UMA FERRAMENTA PARA IDENTIFICAÇÃO AUTOMÁTICA DE BAD SMELLS

Trabalho de Conclusão de Curso submetido à Banca Examinadora do Curso de Ciência da Computação da Universidade Federal do Amapá para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Julio Cezar Costa Furtado.

Macapá
2021

Dados Internacionais de Catalogação na Publicação (CIP)
Biblioteca Central da Universidade Federal do Amapá
Elaborado por Mário das G. Carvalho Lima Júnior – CRB-2/1451

Silva, Matheus Costa.
Uma ferramenta para identificação automática de Bad Smells / Matheus Costa Silva; Orientador, Júlio Cezar Costa Furtado. - Macapá, 2021.

78f.

Trabalho de Conclusão de Curso (Graduação) - Fundação Universidade Federal do Amapá, Coordenação do curso de Ciência da Computação,

1. Software – Desenvolvimento. 2. Arquitetura de software. 3. Ciência da computação. I. Furtado, Júlio Cezar Costa, orientador. II. Fundação Universidade Federal do Amapá. III. Título.

CDD - 005.1 / S586f

UNIVERSIDADE FEDERAL DO AMAPÁ
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
COORDENAÇÃO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

ATA DE DEFESA DE TCC

Realizou-se no dia 20 de agosto de 2021, às 14:30 horas, via videoconferência pelo Google Meet, a defesa do PROJETO DE TCC intitulado: **"UMA FERRAMENTA PARA IDENTIFICAÇÃO DE BAD SMELLS"**, do discente MATHEUS COSTA SILVA, matricula 201612200044. A Banca Examinadora foi composta pelo Prof. Dr. JULIO CEZAR COSTA FURTADO, presidente da banca e orientador; Prof. Me. ADOLFO FRANCESCO DE OLIVEIRA COLARES e Prof. Me. THIAGO PINHEIRO DO NASCIMENTO, examinadores. Concluída da defesa, foram realizadas as arguições e comentários. Em seguida, procedeu-se o julgamento pelos membros da Banca Examinadora, tendo o trabalho sido APROVADO com nota 9,8.

E, para constar, eu, Prof. Dr. JULIO CEZAR COSTA FURTADO, orientador e presidente da Banca Examinadora, lavrei a presente ata que, após lida e achada conforme, foi assinada por mim e demais membros da Banca Examinadora.

Macapá-Ap., 20 de agosto de 2021.

Julio Cezar Costa Furtado
Prof. Dr. JULIO CEZAR COSTA FURTADO
Orientador do Projeto de TCC

ADOLFO FRANCESCO DE OLIVEIRA COLARES:74382080282
Assinado de forma digital por ADOLFO FRANCESCO DE OLIVEIRA COLARES:74382080282

Prof. Me. ADOLFO FRANCESCO DE OLIVEIRA COLARES
Examinador (UNIFAP)

Thiago Pinheiro do Nascimento
Prof. Me. THIAGO PINHEIRO DO NASCIMENTO
Examinador (UNIFAP)

Declaro que as correções solicitadas pela banca foram realizadas pelo(a) discente.

Julio Cezar Costa Furtado
Prof. Dr. JULIO CEZAR COSTA FURTADO
Orientador do Projeto de TCC

Em: 17 / 09 / 2021

Dedico este trabalho a todos aqueles que contribuíram para sua realização, por meio de seu apoio ou incentivo.

AGRADECIMENTOS

Em primeiro lugar quero agradecer aos meus pais, por sempre estarem ao meu lado e me dado apoio nessa caminhada. Aos meus professores do curso que sempre estiveram presentes durante esse período, compartilhando conhecimento e aprendizado de vida que cada um possuía. Ao meu orientador, Prof. Dr. Julio Cezar Costa Furtado, pelos ensinamentos e pela dedicação e paciência que foi oferecida, ajudando na minha formação acadêmica. Enfim, agradeço a todos que acreditaram e contribuíram para a conclusão desse projeto.

“Melhor lutar por algo, do que viver para nada.”

(Winston Churchill)

RESUMO

O processo de desenvolvimento de software é uma etapa na Engenharia de Software que demanda planejamento e organização da equipe de desenvolvimento para produzir código fonte de qualidade. Esses, e demais atributos, são necessários para evitar possíveis problemas estruturais no código, conhecidos na literatura como *Bad Smells*, e/ou melhorar a legibilidade das linhas de código; além de que, sem elas, o processo de refatoração do código se torna mais dispendioso. Tendo em mente essa problemática, este trabalho apresenta um instrumento automatizado para identificação e verificação de *Bad Smells* em código-fonte, e mostra os impactos dos mesmos no desenvolvimento de software. A ferramenta apresentada tem objetivo de detectar esses *Bad Smells* no código fonte e disponibilizar, em forma de gráfico para o usuário, a localização e a categoria em que se enquadram. Este trabalho possui o intuito de detalhar seu funcionamento e contextualizar sua importância dentro da Engenharia de Software.

Palavras-chave: *Bad Smell*, Qualidade de Código, Design e Arquitetura de Software, Ferramentas de Software Livre.

ABSTRACT

The software development process is a step in Software Engineering that requires planning and organizing the development team to produce quality source code. These, and other attributes, are necessary to avoid possible structural problems in the code, known in the literature as Bad Smells, and / or to improve the readability of the lines of code; in addition, without them, the process of refactoring the code becomes more expensive. Bearing this problem in mind, this work presents an automated instrument for identification and verification of Bad Smells in source code and shows their impacts on software development. The tool presented aims to detect these Bad Smells in the source code and make available, in graphic form to the user, the location and the category in which they fall. This work aims to detail its operation and contextualize its importance within Software Engineering.

Keywords: Bad Smell, Code Quality, Software Design and Architecture, Free Software Tools.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ES	ENGENHARIA DE SOFTWARE
BS	*BAD SMELLS*
ISO	*INTERNATIONAL ORGANIZATION OF STANDARDIZATION*
IEC	*INTERNATIONAL ELECTROTECHNICAL COMMISSION*
CD	*CÓDIGO DUPLICADO*
ML	*MÉTODO LONGO*
LC	*LONGA CLASSE*
LLP	*LISTA LONGA DE PARÂMETROS*

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O processo de desenvolvimento de software, mais especificamente a programação, é o momento de construção da aplicação desejada utilizando-se da organização de ideias e designs previamente propostos. Essa preparação evita que o projeto apresente, posteriormente, estruturas que prejudiquem sua legibilidade. Em vista disso, é evidente o esforço para conseguir produzir código-fonte de qualidade.

Durante a vida útil de um sistema a sua evolução acaba sendo um processo natural, segundo Forno (2014) podem ocorrer mudanças de requisitos, pode ser necessária a correção de erros descobertos em fase de operação e podem surgir mudanças para adaptação a plataformas ou para melhorar o desempenho. As modificações nos sistemas de software resultam no aumento de seu tamanho e complexidade, o que, por sua vez, podem levar a degradações de suas estruturas (GUIMARAES; GARCIA; CAI, 2014). Um dos principais sintomas de degradação de um sistema é a manifestação progressiva de anomalias no código (SANTOS, 2016). A detecção e remoção dessas anomalias são tarefas importantes para prolongar a longevidade de um sistema (ARCOVERDE *et al*., 2012).

Para Fowler (1999), *Bad Smell* é um indicador de um possível problema estrutural em código fonte, que pode ser melhorado via refatoração. Alguns trabalhos têm sido desenvolvidos para identificar *Bad Smells* em software com o uso de métricas a partir de código fonte (LANZA; MARINESCU; DUCASSE, 2006). Esse termo será utilizado, neste trabalho, para designar-se às más práticas de programação e redundância de código.

Para auxiliar no desenvolvimento de código com qualidade, foi desenvolvida uma ferramenta, nomeada JSniffer, que tem como principal objetivo identificar e classificar quatro tipos de *Bad Smells* em códigos-fonte, sendo eles*:* Código Duplicado, Método Longo, Longa Classe e Lista Longa de Parâmetros*.* A ferramenta deverá ser executada em códigos com a sintaxe em Java.

1.1 MOTIVAÇÃO, JUSTIFICATIVA E CONTRIBUIÇÃO À ÁREA

Fowler (1999) propõe um catálogo com anomalias, denominadas *code smells*, que são problemas comuns em códigos de software. Tendo esse desafio de produzir código-fonte bem estruturado em mente, ferramentas de detecção de anomalias automatizam essas técnicas para auxiliar a identificação de anomalias (SANTOS, 2016).

Em consequência de alertar o desenvolvedor sobre *Bad Smells* em seu projeto, as ferramentas são importantes, também, pois busca melhorar a legibilidade do código fonte. Tornando a leitura mais agradável para outros desenvolvedores e facilitando o processo de correção futura de falhas e evolução de software. Em resumo, o procedimento realizado por ferramentas acaba cooperando para obter mais clareza e coerência nas linhas de código em desenvolvimentos individuais ou em equipe.

O trabalho proposto visa contribuir nesse aspecto, propondo uma ferramenta para identificação de *Bad Smells* automaticamente em software orientado a objetos na linguagem de programação Java, tendo em vista que ainda é uma linguagem de programação muito utilizada em ambientes acadêmicos e profissionais.

Detectar esses problemas nas fases iniciais do desenvolvimento de software contribui para que os problemas nas fases posteriores possam ser reduzidos. A maioria das abordagens de identificação de *Bad Smells* são executadas depois que o código-fonte é desenvolvido. O desenvolvedor nem sempre tem tempo ou motivação para executar todas as correções sugeridas pelas ferramentas após a implementação do código (SANTOS, 2016).

Na literatura vários estudos sugerem valores limiares para *Bad Smells* extraídos da análise de diversos softwares (SANTOS, 2016). Porém nenhum deles consegue de fato encontrar todos os problemas, permitindo a detecção de vários falsos positivos. As principais contribuições que esse projeto pode fornecer são:

1. Identificação de métricas que representam os aspectos relacionados à *Bad Smells*.

2. Uma ferramenta, para auxiliar desenvolvedores na identificação de *Bad Smells* durante o desenvolvimento.

3. Avaliação da ferramenta proposta, via experimentos com softwares.

4. Disponibilização do código fonte, para que desenvolvedores possam fazer uso da ferramenta, e posteriormente melhorá-la.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Desenvolver uma ferramenta para identificar quatro tipos de *Bad Smells*: Código Duplicado, Método Longo, Longa Classe e Lista Longa de Parâmetros para a linguagem Java.

1.2.2 Objetivos Específicos

1. Identificar as principais anomalias estruturais encontradas durante o desenvolvimento de código, e identificar possíveis refatorações que podem resolver os problemas detectados;
2. Elaborar uma técnica para identificação de *Bad Smells* baseada em métricas predefinidas, utilizando expressões regulares;
3. Desenvolver uma ferramenta que identifique e recomende a refatoração de *Bad Smells* com base na técnica proposta;
4. Avaliação da ferramenta, através de análises em sistemas de software.

1.3 METODOLOGIA

A metodologia empregada neste trabalho pode ser dividida em quatro etapas, descritas a seguir:

Etapa 1 – Na primeira etapa foi feito um estudo sobre o tema que está sendo pesquisado, tendo como objetivo conseguir resultados sobre técnicas para identificar e evitar *Bad Smells* em códigos, também foi analisado recomendações de refatoração para cada uma das anomalias identificadas. Foi realizado um estudo da literatura e o que ela diz sobre as anomalias em códigos fonte, seus principais motivos de ocorrência e as principais dificuldades em identificá-las, passando pela Arquitetura de software, abordando seus princípios e sua importância. Assim como Qualidade de software, passando por suas características e benefícios para o processo de desenvolvimento de código. Esta fase serviu como base para o restante do projeto com todo o referencial teórico necessário para a compreensão do problema.

Etapa 2 – A segunda etapa foi composta pelo tratamento dos problemas identificados, foi feita uma análise do que foi encontrado nas pesquisas, para que seja definido o motivo que leva a anomalias de código. Além disso, foi determinado os principais tipos de *Bad Smells* e as técnicas que foram utilizadas como base para o projeto, definindo as anomalias que deverão ser detectadas e as métricas que serão utilizadas na identificação delas, assim como seus parâmetros de avaliação, para que seja possível encontrar o que é necessário considerar ou corrigir, além de definir qual abordagem foi empregada para automatizar o processo de detecção de *Bad Smells*.

Etapa 3 – Nesta etapa foi iniciada a construção de uma ferramenta para identificar *Bad Smells* e recomendar refatorações em códigos, com base nos tipos de anomalias definidas na abordagem proposta.

Etapa 4 – Na quarta etapa foi realizada a avaliação da ferramenta desenvolvida. Foi feita uma análise da ferramenta, observando seus níveis de acertos e erros. A

ferramenta tem como objetivo testar diferentes cenários para avaliar sua eficiência, usabilidade e níveis de aceitação, tendo sua efetividade avaliada por um especialista.

1.4 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta a Contextualização e Fundamentação Teórica sobre os principais temas relacionados à pesquisa, nesse capítulo ocorre à explicação de conceitos de Arquitetura de Software e alguns de seus princípios, Qualidade de Código, *Bad Smells* em Código, além de exemplificar alguns problemas que estão relacionados diretamente aos tipos existentes de anomalias, também contém trabalhos relacionados que mostram ferramentas existentes para detecção de *Bad Smells* em código. Esse capítulo também é responsável por desenvolver a motivação para a realização dessa pesquisa, mostrando de que forma a mesma irá contribuir para a área e justificando sua utilidade. É feita a explanação dos objetivos dessa pesquisa e o que ela almeja alcançar, além de conter a metodologia de pesquisa utilizada nesse projeto.

O capítulo 3 é responsável por apresentar a ferramenta, passando pelas tecnologias utilizadas e conceitos arquiteturais que foram utilizados para a construção da mesma, além disso, é possível ter uma visão geral do processo utilizado para leitura do código fonte e o processo para identificação de estruturas defeituosas. Nesse capítulo também é possível ter uma visão geral da ferramenta através da apresentação de suas funcionalidades e do modo de uso da mesma.

O Capítulo 4 descreve a avaliação realizada na ferramenta como forma de verificar a eficácia da ferramenta em identificar os *Bad Smells* que estão sendo trabalhados nesse projeto. Além disso, são apresentadas a abordagem realizada e a análise dos resultados obtidos.

E, por final, o Capítulo 5 apresenta as considerações finais e as contribuições deste trabalho, indicando futuros trabalhos que podem ser realizados com base nesse.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo conceitua os tópicos utilizados para entender as principais abordagens relacionadas ao tema proposto. A Seção 2.1 explica a definição da arquitetura de software, contendo princípios e práticas para que haja uma melhor evolução do software. A Seção 2.2 discute conceitos relativos à qualidade de código, quando deve ser realizada e como deve ser avaliada, por meio de métricas, com o objetivo de corrigir e melhorar o tempo de desenvolvimento de uma solução. Na Seção 2.3 é abordado o problema em ter anomalias de código e a importância de haver uma técnica para detecção e refatoração, com destaque para os códigos duplicados, métodos longos, classes longas e lista longa de parâmetros. A Seção 2.4 aborda ferramentas para detecção de anomalias de código, apresentando algumas ferramentas e como elas são classificadas.

2.1 ARQUITETURA DE SOFWTARE

O termo Arquitetura de software não possui um consenso, tornando difícil sua definição (FOWLER, 1999). Apesar dessa falta de consenso a Arquitetura de software está preocupada com a compreensão de como um sistema deve ser organizado e com a estrutura geral desse sistema (SOMMERVILLE, 2011) outra interpretação é a de que arquitetura de um software consiste na estrutura dos componentes e nas suas regras de comunicação (VALIPOUR, 2009).

Sistemas grandes são sempre decompostos em subsistemas que fornecem um conjunto de serviços que estejam relacionados (SOMMERVILLE, 2011). Pressman (2011) define que a arquitetura não é o software operacional, mas sim, uma representação que permite:

- Analisar a efetividade do projeto no atendimento dos requisitos

declarados;

- Considerar alternativas de arquitetura em um momento em que realizar mudanças tenha um menor custo;
- Reduzir os riscos associados à construção do software.

Segundo Sommerville (2011), uma arquitetura quando adequadamente documentada, dentre os diversos fatores, facilita a compressão da estrutura de um sistema, evita a deterioração do código fonte e diminui as chances de possuir anomalias arquiteturais. Segundo Santos (2016) um sistema apropriado possui uma arquitetura que implementa todos os requisitos que atenda o objetivo do sistema. As modificações realizadas nos sistemas, muitas vezes, são feitas ao longo de um período de tempo, ocasionando danos à estrutura do sistema (SANTOS, 2016).

Para Santos (2016) é preciso considerar a estrutura arquitetural do código existente para aumentar a precisão da identificação e recomendação. Pois dessa forma há elementos a mais para evitar “falsas” identificações que são encontradas na maioria das técnicas automáticas existentes para detecção de *Bad Smells*.

2.1.1 Princípios e Práticas em Arquitetura de Software

Segundo Inspazo (2019) devemos observar algumas questões antes de desenvolver qualquer software:

1. Como é que a aplicação pode ser concebida para que seja flexível e sustentável ao longo do tempo?
2. Quais são as tendências arquiteturais que possam ter impacto sobre ela, no momento ou após a implementação?

Sendo esperada uma evolução contínua, o sistema deve ser construído para mudar, e não para durar (INSPAZO, 2019), situação essa que permite a redução de

riscos, já que sistemas devem ser construídos com a mentalidade de que pode haver mudanças e que dependendo da forma como o sistema será construído essas mudanças podem acabar sendo custosas.

Segundo Smith (2019) existe alguns princípios de design, como:

- Separação de interesses: Esse princípio declara que o software deve ser separado de acordo com os tipos de trabalho que ele executa (SMITH, 2019). Isso ajuda a garantir que o modelo de negócios seja fácil de ser testado e possa evoluir sem ter um acoplamento rígido com detalhes de implementação de nível inferior (SMITH, 2019).

- Única Responsabilidade: Cada componente ou módulo deve ser independente em si e responsável por uma característica específica ou funcionalidade (INSPAZO, 2019). Manter as responsabilidades o mais restritas possível significa que os usuários sabem da finalidade pretendida, o que leva a menos erros (JEFFERSON, 2019).

- Não-repetição: A implementação de qualquer funcionalidade deve ocorrer num único local e não deve ser repetida (INSPAZO, 2019). Um sistema deve evitar ao máximo especificar determinada funcionalidade em vários locais, pois sempre que for necessário alterar algo relacionado a essa funcionalidade, a mesma deverá ser modificada em todos os locais que ela se encontra, o que leva tempo e pode ocasionar erros, já que é provável que não seja feita alteração em alguns desses locais.

Falta de visão de longo prazo muitas vezes resultará em uma arquitetura inflexível que limita as possibilidades de futura adaptação e crescimento (ANDERS, 2019). No entanto, até mesmo um aplicativo independente que nunca deveria ser alterado pode, de repente, ser obrigado a integrar-se a sistemas externos ou evoluir com novas funcionalidades (ANDERS, 2019).

Para a maioria dos sistemas é necessário investir esforços para que um projeto tenha uma boa arquitetura para garantir que ele possa possibilitar mudanças e ter uma boa aparência em todos os estágios de seus ciclos de vida (ANDERS, 2019).

2.2 QUALIDADE DE CÓDIGO

Segundo Santos (2016) a qualidade de código aborda problemas relativos a aspectos de desenvolvimento e soluções de implementação ligadas à leitura, melhoria da escrita, documentação e reaproveitamento de código. Sendo assim, escrever código corretamente auxilia nos processos de mudanças e evolução de software, além de reduzir problemas com testes e validação de requisitos.

Técnicas de verificação e validação são fundamentais para identificar se um software possui defeitos ou está de acordo com o que foi especificado (SOMMERVILLE, 2011). A verificação é o processo de determinar se o software está sendo construído da maneira correta, de acordo com os requisitos descritos. A validação é o processo de confirmação de que o software é aquilo que os *stakeholders* querem.

Habitualmente, diz que um produto é confiável quando não falha. E em software, sabemos que a ocorrência de falhas é sempre uma possibilidade (KOSCIANSKI; DOS SANTOS SOARES, 2007). Boehm, Brown e Lipow (1976) definiram uma árvore de atributos de qualidade de software (FIGURA 1). As definições para funcionalidade, portabilidade, confiabilidade, eficiência, usabilidade e manutenibilidade serão definidas a seguir:

- **Funcionalidade:** A funcionalidade diz respeito àquilo que o software irá fazer para satisfazer aquilo que os usuários desejam. Pode-se dizer que é similar aos requisitos funcionais definidas por (SOMMERVILLE, 2011): “Os requisitos

funcionais para um sistema descrevem a funcionalidade ou os serviços que se espera que o sistema forneça".

- **Confiabilidade:** A capacidade de manter certo nível de desempenho quando operando em certo contexto de uso (ISO/IEC 9126-1). Sendo a tolerância e recuperação de falhas como principal característica de confiabilidade.

- **Usabilidade:** Usabilidade representa o quão fácil é utilizar um produto (ISO/IEC 9126). A definição de requisitos e seus estágios posteriores como a verificação e validação do produto acabam tornando, provavelmente, a característica mais difícil de tratar (KOSCIANSKI; DOS SANTOS SOARES, 2007).

- **Eficiência:** Eficiência está relacionada com o tempo de execução do software e a verificação se recursos envolvidos está conciliável com o nível de desempenho do software (ISO/IEC, 9126).

- **Portabilidade:** É a capacidade de um software ser transferido de um ambiente para outro (ISO/IEC 9126). Então um software deve ser elaborado para operar em ambientes com características distintas (KOSCIANSKI; DOS SANTOS SOARES, 2007).

- **Manutenibilidade**: Segundo a ISO/IEC 9126, manutenibilidade é a capacidade do produto de software ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais.

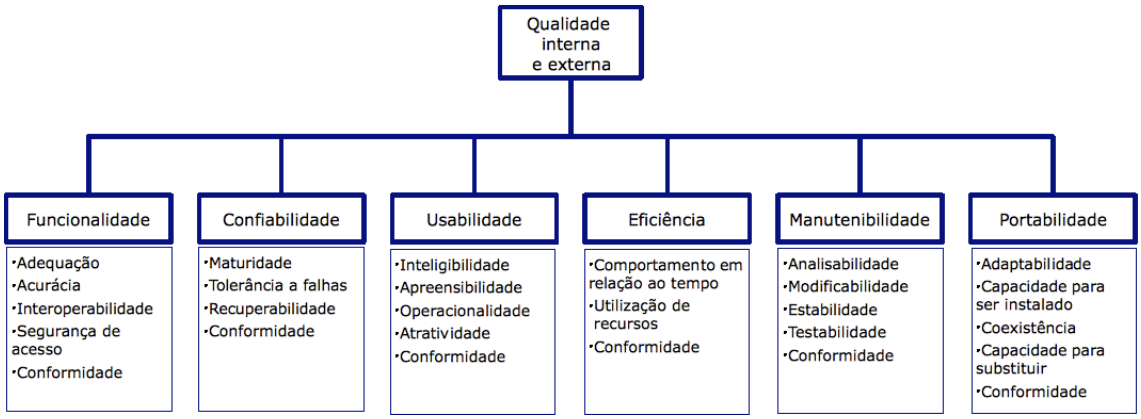
A característica de manutenibilidade está ligada à modificação de um produto de software de maneira fácil e eficiente (KOSCIANSKI; DOS SANTOS SOARES, 2007). As sub características da Manutenibilidade de acordo com (ISO/IEC 9126-1):

- **Analisabilidade:** Capacidade do produto de software de permitir o diagnóstico de deficiências ou causas de falhas no software, ou a identificação

de partes a serem modificadas.

- **Modificabilidade:** Capacidade do produto de software de permitir que uma modificação especificada seja implementada.
- **Estabilidade**: Capacidade do produto de software de evitar efeitos inesperados decorrentes de modificações no software.
- **Testabilidade:** Capacidade do produto de software de permitir que o software, quando modificado, seja validado.
- **Conformidade com a manutenibilidade:** Capacidade do produto de software de estar de acordo com normas ou convenções relacionadas à manutenibilidade.

FIGURA 1 - ÁRVORE DE CARACTERÍSTICAS DE QUALIDADE DE SOFTWARE



FONTE: (Boehm, Brown e Lipow, 1976).

Segundo Santos (2016) mesmo com todo trabalho de avaliação, se não for possível obter dados confiáveis, a avaliação é posta em risco. Métricas servem como forma de avaliação quantitativa na administração da qualidade, servindo como base de requisitos e definição de objetivos de qualidade de um produto (KOSCIANSKI; DOS SANTOS SOARES, 2007). Sendo assim, é preciso verificar se o resultado de uma medida usada para julgar um software é feito da maneira correta.

2.3 BAD SMELLS EM CÓDIGO

Bad Smells são problemas no código que podem dificultar a manutenibilidade do software. Fowler (1999) conceitua *Bad Smells* como problemas relacionados ao uso de más práticas que afetam a evolução do código.

Essas anomalias podem influenciar na inserção de erros responsáveis por futuras falhas, ou seja, são responsáveis pelas dificuldades encontrados na manutenção do sistema. Elas descrevem uma situação que pode indicar possíveis falhas de projeto do software (SANTOS, 2016).

É possível realizar o tratamento de *Bad Smells* enquanto ocorre o desenvolvimento do projeto, porém é necessário, de forma preventiva, identificar as partes do código que não estão compatíveis ou violam a estrutura do projeto.

Fowler (1999) e Monteiro e Fernandes (2006) apresentam diversas anomalias de código. Algumas delas são: Código duplicado (*Duplicated Code*), Classe longa (*Large Class*), Método longo (*Long Method*) e Lista longa de parâmetros (*Long Parameter List*). A seguir são apresentadas em detalhes algumas anomalias que serão utilizadas nesse estudo, com o intuito de identificá-las e evitar problemas futuros.

2.3.1 Código Duplicado

Bad Smell associado a fragmentos do código que se repetem em dois lugares ou mais. Causa redundância no código e aumento desnecessário de linhas.

Se for possível visualizar a mesma estrutura de código em mais de um local, pode-se ter certeza de que o programa será melhor se for capaz de encontrar uma maneira de unificá-los (FOWLER, 1999). O problema mais simples de código duplicado é quando você tem a mesma expressão em dois métodos da mesma classe. Então tudo

que precisa ser feito é usar a técnica "*Extract Method"* que consiste em chamar o mesmo trecho de código que aparece nos dois lugares em apenas um, centralizando a sua funcionalidade.

2.3.2 Classe Longa

Relacionado a classes demasiado longas que apresentam muitos elementos em seu corpo. Geralmente, a causa desse *Bad Smell* é a falta de organização e coerência dos componentes inseridos na classe, ou seja, os elementos que estão presentes nela poderiam estar em outra classe mais condizente com sua função, que é o princípio de uma técnica chamada "*Extract Class*", onde se deve criar uma nova classe e colocar nela somente os campos e métodos responsáveis pela funcionalidade relevante da mesma.

Quando uma classe está tentando fazer muito, muitas vezes aparece com muitas variáveis de instância. Quando uma classe tem muitas variáveis de instância, código duplicado não deve estar muito atrás (FOWLER, 1999).

Assim como uma classe com muitas variáveis de instância, uma classe com muito código é o principal terreno fértil para código duplicado. A solução mais simples é eliminar a redundância na própria classe. Se a classe tiver quinhentos métodos de linha com muito código em comum, você pode ser capaz de transformá-los em cinco métodos de dez linhas com outros dez métodos extraídos do original (FOWLER, 1999).

2.3.3 Método Longo

Métodos com corpos extensos que realizam muitas atividades, desnecessariamente agrupadas em um único local. Segundo Fowler (1999), nos

métodos longos:

- Quanto maior for o procedimento, mais difícil é entendê-lo;
- Toda vez que sentir a necessidade de comentar algo, deve-se escrever um método em vez disso;
- A chave não é o tamanho do método, mas a distância semântica entre o que o método faz e como ele o faz;
- Olhar os comentários é uma boa técnica para a identificar blocos que necessitam de refatoração;
- Condicionais e loops também dão sinais de extrações.

Os métodos que vivem mais são aqueles com curta duração. Programadores iniciantes em orientação a objetos geralmente têm a sensação de que os programas são infinitas sequências de delegação (FOWLER, 1999).

Uma boa técnica para encontrar blocos de códigos que devem ser substituídos é procurando comentários. Um bloco de código com um comentário que informa o que está fazendo pode ser substituído por um método cujo nome se baseia no comentário (FOWLER, 1999).

2.3.4 Lista Longa de Parâmetros

Este *Bad Smell* é relacionado ao número excessivo de parâmetros presentes na assinatura de métodos. Esses que, muitas vezes, são pouco utilizados ou poderiam ser substituídos por variáveis de classe ou globais.

Em nossos primeiros dias de programação, fomos ensinados a passar tudo o que for rotina como parâmetro. Isso foi compreensível porque a alternativa era utilizar dados globais, e dados globais são maus e geralmente dolorosos (FOWLER, 1999).

Listas longas de parâmetros são difíceis de entender, elas ficam inconsistentes e difíceis de usar, e porque você sempre irá alterá-los conforme você precisar de mais dados (FOWLER, 1999). Essa anomalia pode ser removida passando objetos porque é muito mais provável que você precise fazer algumas solicitações para obter um novo dado, e com objetos só será preciso passar um parâmetro quando algum dado for solicitado. Outra forma seria substituindo os parâmetros utilizados por uma técnica chamada "*Replace Parameter with Method Call*", que em vez de passar o valor por meio de um parâmetro, tentar adicionar uma chamada de método para tratar os valores, ou tratar eles dentro do próprio método, com o objetivo de diminuir a quantidade de dados passados para serem manipulados pelo método que está recebendo muitos parâmetros.

2.3.5 Importância da detecção de Bad Smells para refatoração de código

A investigação rápida das anomalias de código em um sistema é crucial para que sua remoção seja feita o mais cedo possível (SANTOS, 2016). Para manter a qualidade e a continuidade de um software é fundamental evitar anomalias, pois elas causam um grande impacto na estrutura e arquitetura do código.

2.4 TRABALHOS RELACIONADOS

Silva, Terra e Valente (2015) propõem a construção de um plugin para o Eclipse, o JExtract, que analisa e recomenda oportunidades de refatoração utilizando o *Extract Method,* que é necessário quando um método possui fragmentos de código que podem ser agrupados e movidos para um novo método e substituindo o antigo código por uma chamada de método. O JExtract implementa uma abordagem automatizando as

refatorações através de uma ferramenta que foi projetada para automaticamente identificar, classificar e aplicar a refatoração.

A abordagem é dividida em três fases: Geração de Candidatos, Pontuação e Classificação. Segundo possibilidades é criada uma classificação de acordo com uma função de pontuação baseada na semelhança entre conjuntos de dependências estabelecidas no código. Por fim, os candidatos com a melhor classificação são fornecidos como saída da ferramenta, permitindo que os desenvolvedores inspecionem as recomendações e apliquem as que desejarem.

Silva, Terra e Valente (2015) utilizaram de valores obtidos do cálculo de *precision* e *recall* para fazer avalição da ferramenta, indicando que a JExtract é mais eficaz para identificar códigos contíguos mal colocado em métodos do que a ferramenta JDeodorant (TSANTALIS; CHATZIGEORGIOU, 2011), uma outra ferramenta existente que foi utilizada como critério de comparação, o resultado pode ser observado na FIGURA 2, a análise foi obtida analisando vários sistemas distintos.

FIGURA 2 - RESULTADOS DE *RECALL* E *PRECISION* DA JEXTRACT

		JExtract						JDeodorant	
		Top-1		Top-2		Top-3			
System	#	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.
JHotDraw 5.2	56	19 (34%)	34%	26 (46%)	24%	32 (57%)	20%	2 (4%)	5%
JUnit 3.8	25	13 (52%)	52%	16 (64%)	33%	18 (72%)	25%	0 (0%)	0%
MyWebMarket	14	12 (86%)	86%	14 (100%)	50%	14 (100%)	33%	2 (14%)	33%
Total	***95***	***44 (46%)***	46%	***56 (59%)***	***30%***	***64 (67%)***	***23%***	***4 (4%)***	***6%***

FONTE: (Silva, Terra e Valente, 2015).

Sales *et al.* (2013) propõem uma ferramenta, a JMove, desenvolvida como plugin para IDE Eclipse, para sugerir e aplicar refatorações do tipo *Move Method* com base na igualde entre os métodos a partir do código fonte de um sistema Java.

A ferramenta identifica oportunidades de refatoração com base na similaridade entre os conjuntos de dependências, ele calcula a similaridade do conjunto de dependências estabelecido por um determinado método com os métodos de sua própria classe e os métodos de outras classes do sistema, então ele recomenda mover

um método para uma classe mais semelhante.

Segundo Sales *et al.* (2013) na avaliação realizada com treze sistemas de código aberto, a JMove se mostrou 41% mais eficaz em detectar métodos mal localizados do que a ferramenta JDeodorant (TSANTALIS; CHATZIGEORGIOU, 2009), que também realiza esse tipo de detecção e refatoração. Os resultados da análise podem ser observados na FIGURA 3.

FIGURA 3 - RESULTADOS DA ANÁLISE DA JMOVE

Sistema	Métodos Movidos	Sugestões Fornecidas		*Recall*	
		JDeodorant	JMove	JDeodorant	JMove
Ant 1.8.2	25	21 + 161*	21 + 136*	84%	84%
ArgoUml 0.34	37	23 + 30*	33 + 48*	62%	89%
Cayenne 3.0.1	47	18 + 105*	38 + 164*	38%	81%
DrJava r5387	18	13 + 293*	15 + 98*	72%	83%
FreeCol 0.10.3	17	10 + 281*	13 + 201*	59%	76%
FreeMind 0.9.0	12	7 + 60*	11 + 65*	58%	92%
JMeter 2.5.1	25	15 + 102*	21 + 64*	60%	84%
JRuby 1.7.3	41	24 + 399*	34 + 357*	59%	83%
JTOpen 7.8	39	23 + 427*	35 + 162*	59%	90%
Maven 3.0.5	24	13 + 56*	16 + 42*	54%	67%
Megamek 0.35.18	35	21 + 243*	29 + 292*	60%	83%
WCT 1.5.2	29	14 + 72*	25 + 44*	48%	86%
Weka 3.6.9	31	23 + 327*	27 + 313*	74%	87%
Total	**380**	**225 + 2.556***	**318 + 1.986***	**59,2%**	**83,7%**

FONTE: (Sales *et al.*, 2013).

Tsantalis e Chatzigeorgiou (2009) criaram uma ferramenta, a JDeodorant, que é um plugin para o Eclipse, de código aberto para Java, que detecta anomalias de código e resolve os problemas através da aplicação de refatorações. A ferramenta completa identifica cinco anomalias: *Feature Envy*, *Long Method*, *Type Checking*, *God Class* e *Duplicated Code* e suas técnicas de detecção baseiam-se em oportunidades de refatoração, o trabalho citado tem como base uma parte da ferramenta, parte que contem a identificação de Códigos Duplicados e Métodos Longos utilizando a técnica de *Extract Method* em códigos orientados a objetos e logo após apresentar como sugestões para o desenvolvedor. Assim que a análise for realizada as anomalias encontradas são adicionadas em uma tabela de visão e marcadores são adicionados no código sendo que cada um deles representa partes do código que devem, segundo a ferramenta, ser refatorados. A abordagem proposta pelo trabalho se baseia na união de pedaços de código que resultam da aplicação de um bloco baseado na técnica de

fatiar que também é proposta pelo mesmo. De acordo com Tsantalis e Chatzigeorgiou (2009), uma fatia consiste em todas as instruções em um programa que pode afetar o valor de uma variável x em um ponto específico de interesse p, onde o par (p, x) é referido como corte critério. Tsantalis e Chatzigeorgiou (2009) também citam que nessa técnica as fatias são calculadas encontrando conjuntos de declarações direta ou indiretamente relevantes com base nas dependências de controle e dados.

A avalição da ferramenta foi realizada por um desenvolvedor independente e o projeto de software que foi examinado é um emulador de um central telefônica, implementado em Java, que consiste em 61 classes, 144 métodos (sem métodos abstratos) e 4100 linhas de código.

Como resultado a quantidade total de sugestões foi igual a 47. Além disso, 8 dos 47 candidatos resultaram da união de duas fatias, enquanto os 39 restantes resultaram de uma única fatia. E segundo Tsantalis e Chatzigeorgiou (2009) se a refatoração não levasse em consideração a preservação do comportamento do código haveria problemas de duplicação que resultariam em 55 sugestões para o projeto examinado, então chegaram a conclusão que o desenvolvedor que foi auxiliado pela abordagem tem que inspecionar menos sugestões de refatoração e não tem que examinar cuidadosamente o código resultante após a aplicação de uma refatoração, o que afirma que a metodologia é capaz de identificar e criar novos métodos com funcionalidade útil e capaz de preservar o comportamento inicial do código, porém o próprio trabalho foi avaliado em apenas uma ferramenta e o mesmo afirma que é necessário utilizar a ferramenta em outros sistemas e utilizar outros desenvolvedores para realizar a avaliação a fim de confirmar os resultados obtidos.

Santos (2016) propõe uma nova técnica para identificar valores limiares para métodos longos que consideram o contexto arquitetural da classe e são extraídos de uma versão com a mesma arquitetura do software. Foi desenvolvida uma ferramenta que recomenda ações ao desenvolvedor durante o desenvolvimento do código.

O método proposto pelo trabalho de Santos (2016) é dividido em duas etapas. Na primeira etapa o objetivo é definir valores a partir de código-fonte para detecção de

métodos longos, definindo o principal interesse arquitetural de cada classe em um projeto exemplo, que também será utilizado para calcular valores limiares para o número de linhas de código em cada interesse arquitetural, que será utilizado para análise de classes que possuam o mesmo interesse arquitetural. Na etapa seguinte é realizada a análise das classes em desenvolvimento, inicialmente definindo qual o interesse arquitetural da classe e depois, utilizando dos valores identificados na primeira etapa para verificar se alguns dos métodos implementados ultrapassa o valor estipulado.

Para avaliação da ferramenta foi calculada a *precision* e *recall* em diferentes versões de um sistema de manipulação de mídias em dispositivos móveis. Segundo Santos (2016) na avaliação realizada ela obteve 100% de *recall*, significando que todos os métodos longos definidos previamente por especialistas foram identificados. A precisão do método alcançou 32%.

O trabalho de Santos (2016) tenta identificar anomalias através de falhas no design ao invés de exemplos de código fonte também, o que difere desse trabalho que utiliza o código fonte para identificar e recomendar alteração.

Todas essas ferramentas descritas anteriormente possuem bom desempenho a identificação de seus tipos de anomalias, mas elas focam apenas em um tipo de identificação, logo elas acabam sendo especificas para um tipo de *Bad Smell* cada, sendo que, dificilmente, um projeto de software irá possuir apenas um problema estrutural.

Algumas delas foram desenvolvidas como um plugin para o Eclipse, porém nem todos os desenvolvedores utilizam o Eclipse, muitos utilizam editores de texto modernos ou outras ferramentas como Netbeans.

Também é utilizada uma abordagem diferente nesse trabalho, usando uma forma de detecção mais simples como identificação através de blocos de expressões regulares e focada em uma linguagem especifica, além de ser possível aplicar ela em mais de um tipo de *Bad Smell*. Esse trabalho difere dos anteriores também por não ser necessário descrever regras da arquitetura que serão verificadas para encontrar

problemas, essas regras são extraídas automaticamente do código fonte existente através de medidas já definidas que podem ser alteradas de acordo com o projeto que está sendo desenvolvido.

3 A FERRAMENTA JSNIFFER

Nesse capítulo será apresentada a ferramenta desenvolvida para auxiliar a implementação do processo de identificação de *Bad Smells* por meio da utilização de processos sistematizados. Para isto, será apresentado o contexto em que a ferramenta está inserida, os detalhes de seu projeto técnico e suas principais telas, explicando seu funcionamento e operação.

3.1 OBJETIVO DA FERRAMENTA

A ferramenta foi desenvolvida com o intuito principal de automatizar a identificação de *Bad Smells* em códigos orientados a objetos que utilizam a linguagem de programação Java.

As métricas utilizadas para definir os *Bad Smells* na classe estão relacionadas com a estrutura utilizada na criação do código. Para obter os quatros tipos analisados, utiliza-se expressões regulares para definir uma “arvore de derivação” e assim montar uma forma de visualização de seus agrupamentos, separando assim, classes de seus atributos, de seus métodos e conteúdo desses métodos.

Utilizam-se valores fixos para definir o limite que uma classe e um método devem ter em quantidade de linhas para serem considerados longos, e o limite de parâmetros que um método deve possuir. Além disso, são utilizadas técnicas de comparação para obter padrões entre métodos e classificá-los em métodos duplicados ou não.

A ferramenta é gratuita, com licença GPL (General Public License), tornando-se importante para não acarretar maiores custos para os usuários pretendem adotá-las. A ferramenta pode ser utilizada em diferentes organizações, ou ambientes acadêmicos, e encontra-se disponível para download a partir do endereço https://github.com/jcfurtado86/badsmells.

3.2 PROJETO TÉCNICO DA FERRAMENTA

Nesta seção serão apresentados os requisitos técnicos utilizados na compreensão e desenvolvimento da ferramenta, como: arquitetura e tecnologias e as técnicas para leitura das estruturas de código e o modo como é realizada a identificação de *Bad Smells.*

3.2.1 Arquitetura da Ferramenta

Especificação dos diagramas de classe e demais especificações que demonstre a forma como estão organizadas as classes da ferramenta bem como está definida as estruturas para importações de projetos e análise dos mesmos.

Inicialmente a ferramenta precisou utilizar dois componentes (bibliotecas) externos, um para possibilitar a comunicação com a API do *Github* para poder recuperar repositórios hospedados na plataforma, e a outra para geração da interface gráfica no modelo *TreeMap* onde se encontram as informações dos *Bad Smells* identificados. A principal preocupação na definição da arquitetura utilizada pela ferramenta foi possibilitar a integração com essas ferramentas externas, de forma mais transparente possível para o usuário. Sendo assim, a ferramenta integra-se a ferramentas externas, também livres:

- É utilizada a biblioteca *JGit*, que é uma biblioteca Java pura licenciada por EDL (*new-style BSD*) que implementa o sistema de controle de versão Git como rotinas de acesso ao repositório, protocolos de rede e algoritmos de controle de versão principal. A biblioteca tem poucas dependências, o que o torna adequado para incorporação em qualquer aplicativo Java. Biblioteca disponível em https://www.eclipse.org/jgit/download/;

- É utilizado o *TreeMap Java Library*, uma biblioteca Java de visualização de mapa de árvore, para implementar facilmente o mapa de árvore de Shneiderman (http://www.cs.umd.edu/hcil/treemaps/). Mostra dados de árvore de forma eficiente como um mapa colorido de retângulo. Útil para monitorar milhares de arquivos em uma pequena janela. Biblioteca disponível em https://sourceforge.net/projects/treemap/files/treemap/;

Pode-se afirmar que esta integração é bem forte, de tal maneira que toda a gerência do que e quando executar é realizado pela ferramenta, de acordo com a interação do usuário com a mesma.

A ferramenta é flexível o suficiente para ser executada apenas localmente, de forma *stand-alone*, sem a necessidade de servidor para a comunicação com qualquer serviço, exceto quando houver necessidade de baixar algum projeto externo.

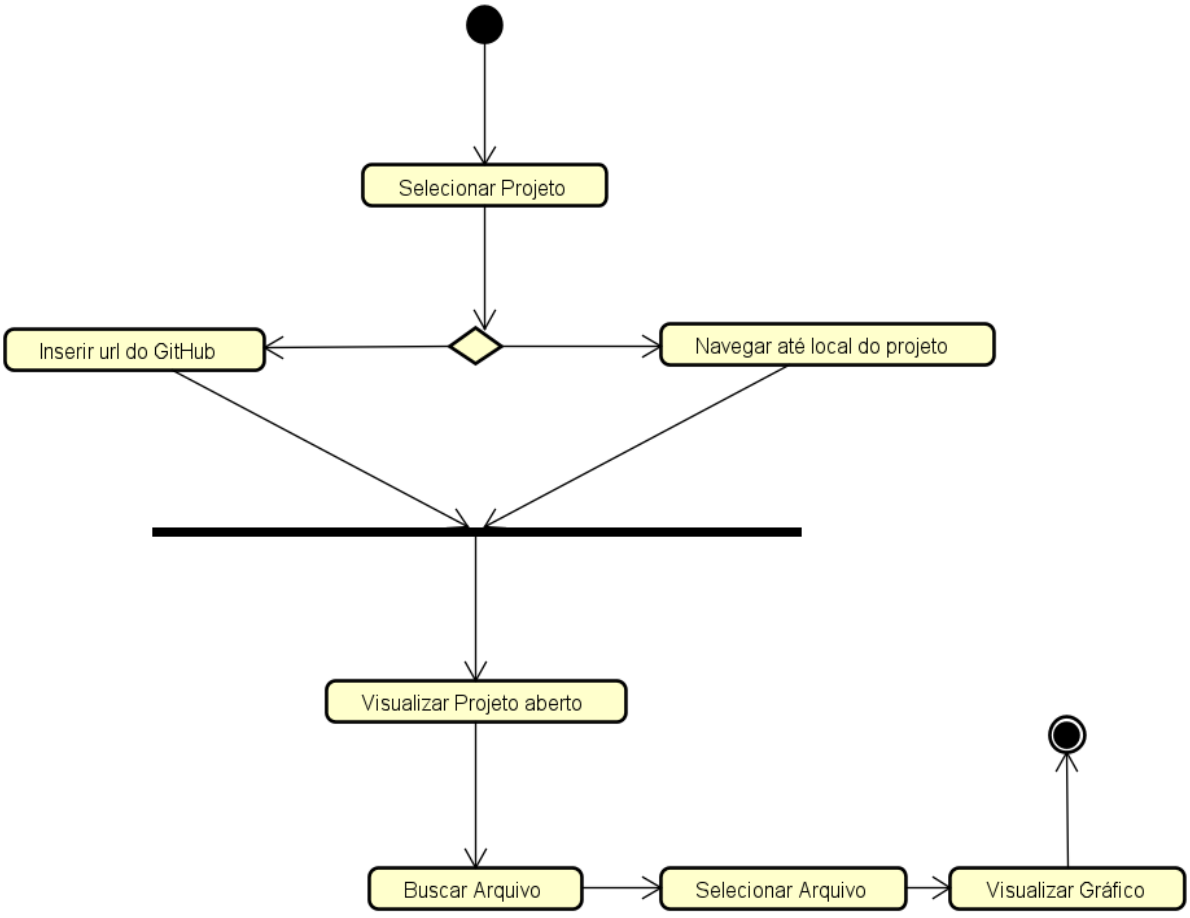
A FIGURA 4 representa o passo a passo que deverá ser feito pelo usuário para que ele possa executar por completo todas as funções que a ferramenta disponibiliza, incluindo caminhos alternativos para determinadas atividades, como selecionar repositório local ou remoto.

A FIGURA 5 representa o diagrama de classes ao qual o projeto está organizado, sendo que o projeto está organizado em quatro módulos, onde cada módulo representa um componente do sistema: componente de Interface gráfica no módulo “GUI”, componente de busca e download de repositórios remotos no módulo “Repositorio”, componente de análise e identificação de *Bad Smells* no módulo “Analise” e componente de especificações e definição das métricas de *Bad Smells* utilizadas para análise, no módulo “BadSmells”. Cada componente trabalhando em conjunto com o outro para com a finalidade de operarem como um só.

O módulo "Analise" e "BadSmells" são os principais, sendo assim é possível verificar que existe uma classe abstrata, a classe “Regex” no módulo “Analise”, que armazena todas as expressões regulares utilizadas para identificar as estruturas da classe que está sendo analisada, e existe a classe “Reconhecedor” do módulo

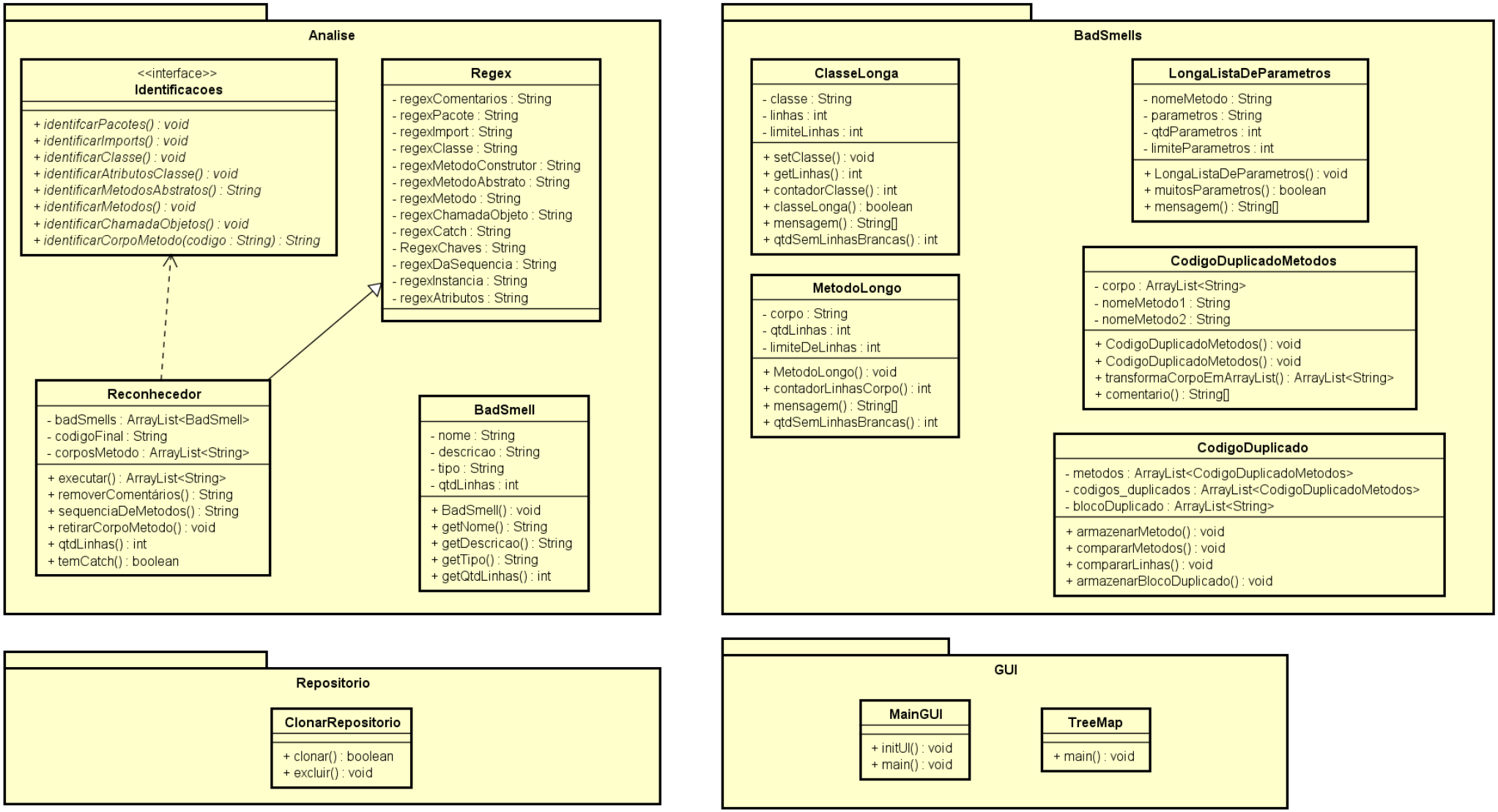
“Analise”, que terá como objetivo utilizar cada uma dessas expressões para que haja a identificação, por meio de uma sequência de métodos, onde cada método tem como objetivo identificar um possível *Bad Smell* que esteja presente no código analisado. Para auxiliar na análise do arquivo o módulo "BadSmells" entra com os métodos necessários para definir se a estrutura identificada pelo módulo "Análise" se encaixa em alguns dos *Bad Smells*, que possuem suas definições especificadas por cada uma das classes do mesmo módulo.

FIGURA 4 – DIAGRAMA DE ATIVIDADES DA FERRAMENTA



FONTE: Elaborada pelo autor.

FIGURA 5 – DIAGRAMA DE CLASSES DA FERRAMENTA



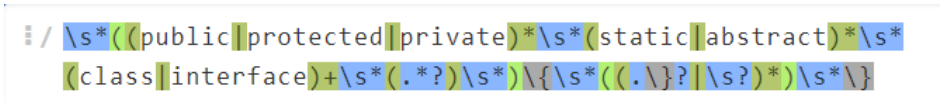
FONTE: Elaborada pelo autor.

3.2.2 Processo de Identificação das estruturas de código pela ferramenta

Para que ocorresse a definição das estruturas pertinentes em código fontes foram utilizadas expressões regulares, que são sequências de caracteres padronizados que associam sequências de caracteres em um determinado elemento textual. Cada expressão definida a seguir foi baseada na sintaxe da linguagem de programação Java.

A FIGURA 6 apresenta a expressão regular que é utilizada para identificação de classes em códigos fontes.

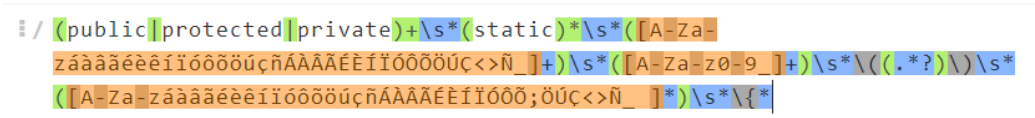
FIGURA 6 – EXPRESSÃO REGULAR PARA IDENTIFICAÇÃO DE CLASSE



FONTE: Elaborada pelo autor.

A FIGURA 7 apresenta a expressão regular que é utilizada para identificação de métodos e seus parâmetros em códigos fontes.

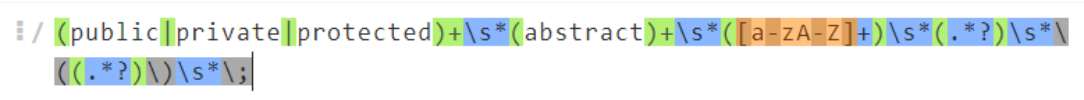
FIGURA 7 – EXPRESSÃO REGULAR PARA IDENTIFICAÇÃO DE MÉTODOS



FONTE: Elaborada pelo autor.

A FIGURA 8 apresenta a expressão regular que é utilizada para identificação de métodos abstratos e seus parâmetros em códigos fontes.

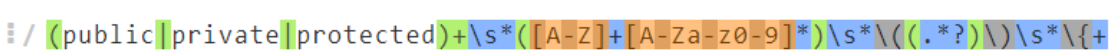
FIGURA 8 – EXPRESSÃO REGULAR PARA IDENTIFICAÇÃO DE MÉTODOS ABSTRATOS



FONTE: Elaborada pelo autor.

A FIGURA 9 apresenta a expressão regular que é utilizada para identificação de métodos construtores e seus parâmetros em códigos fontes.

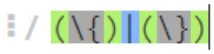
FIGURA 9 – EXPRESSÃO REGULAR PARA IDENTIFICAÇÃO DE MÉTODOS CONSTRUTORES



FONTE: Elaborada pelo autor.

A FIGURA 10 apresenta a expressão regular que é utilizada para identificação de corpo de métodos em códigos fontes.

FIGURA 10 – EXPRESSÃO REGULAR PARA IDENTIFICAÇÃO DE CORPO DE MÉTODOS



FONTE: Elaborada pelo autor.

Aplicando as expressões regulares acima é possível identificar as principais estruturas de um código, como suas classes, métodos, assinatura desses métodos e os conteúdos presentes neles.

Após a aplicação e identificação das estruturas do código é feita uma separação de acordo com o tipo de estrutura identificada, e logo após cada uma delas é separada e armazenada de acordo com o tipo, classificando cada estrutura identificada em Classes, Métodos, Parâmetros e Corpo de Métodos.

Cada tipo identificado é percorrido individualmente por uma análise em cada tipo de *Bad Smell* que a ferramenta identifica, a fim de analisar se o trecho de código possui alguma anomalia, cada estrutura passa pela sua respectiva inspeção, ou seja, estruturas definidas como Classe passa pela identificação de classes longas, estruturas como Métodos passam pela identificação de métodos longos, longa lista de parâmetros e pela verificação de código duplicado.

Por fim é feita uma separação e classificação dos blocos de código defeituosos e o trecho que possui o problema. Em todos os casos o trecho identificado como

defeituoso é separado e armazenado de acordo com tipo detectado para posteriormente ser utilizado para geração do resultado gráfico.

3.2.3 Processo para identificação de *Bad Smells*

Marinescu (2004) define Estratégia de Detecção como "uma expressão quantificável de uma regra, que permite avaliar se fragmentos de código estão de acordo com essa regra". Então para analisar se cada trecho de código é um possível *Bad Smell* é verificado se cada elemento infringe ou satisfaz as métricas definidas para cada tipo, e cada análise tende a seguir rigorosamente cada uma dessas regras predefinidas.

A revisão final para identificação de cada *Bad Smell* passa pelos seguintes passos de acordo com o seu tipo:

1. Código duplicado: Realizando-se laços de repetição em todos os métodos, e fazendo a comparação de seu conteúdo com o conteúdo de outros métodos para verificar igualdade de trechos de código, e caso sejam semelhantes e a quantidade seja superior ao definido então será classificado como anomalia. Para esse *Bad Smell* foi considerado um valor de 5 ou mais linhas duplicadas para que o mesmo se enquadre nesse tipo.
2. Classe longa: Analisando o código da classe e sua quantidade de linhas, e verificando se a quantidade é superior ao que foi definido. Para esse *Bad Smell* foi considerado um valor maior que 150 linhas para que a classe se enquadre nesse tipo.
3. Lista longa de Parâmetros: Analisando a lista de parâmetros de cada método e obtendo a quantidade de parâmetros que o método possui, e verificando se a quantidade é superior ao que foi definido. Para esse *Bad Smell*

foi considerado um valor maior que três parâmetros para que o método se enquadre nesse tipo.

4. Método longo: Analisando o corpo do método e obtendo a quantidade de linhas que o método possui, e verificando se a quantidade é superior ao que foi definido. Para esse *Bad Smell* foi considerado um valor maior que 30 linhas para que o método se enquadre nesse tipo.

3.2.4 Tecnologias utilizadas na Ferramenta

Para desenvolver a ferramenta foi escolhida a plataforma Java, *Standard Edition* (Java SE), por possibilitar um rápido e eficiente desenvolvimento para desktop pela capacidade de desenvolvimento que a plataforma disponibiliza. Foi levado em consideração o fato de ser uma tecnologia livre e ainda muito utilizada em ambiente acadêmico. Disponível em: www.oracle.com/technetwork/java/javase.

3.3 AS FUNCIONALIDADES DA FERRAMENTA

A ferramenta é gratuita e de código aberto, desenvolvida para Desktop. Encontra-se disponível para download no site: https://github.com/jcfurtado86/ badsmells.

A ferramenta recebe como entrada um projeto Java, no qual, através de expressões regulares, é realizada uma análise sintática com o objetivo de detectar classes, atributos, métodos, conteúdos de métodos e parâmetros. Com a posse desses dados é possível definir a quantidade de atributos e métodos que a classe possui, a quantidade de parâmetros de cada método e se há duplicidade entre os conteúdos

desses métodos. A partir dessas informações, a ferramenta é capaz de aplicar os parâmetros definidos para que um código analisado possa ser definido como contendo um *Bad Smell*. A ferramenta então gera um gráfico para que o usuário possa visualizar em qual parte do código há esses tipos de problemas.

Ao fazer a análise no código-fonte, a ferramenta identifica se o mesmo se enquadra em uma das quatro categorias de *Bad Smells* propostas por Fowler (1999): Código Duplicado (*Duplicated Code*), Classe Longa (*Large Class*), Método Longo (*Long Method*) e Lista Longa de Parâmetros (*Long Parameter List*).

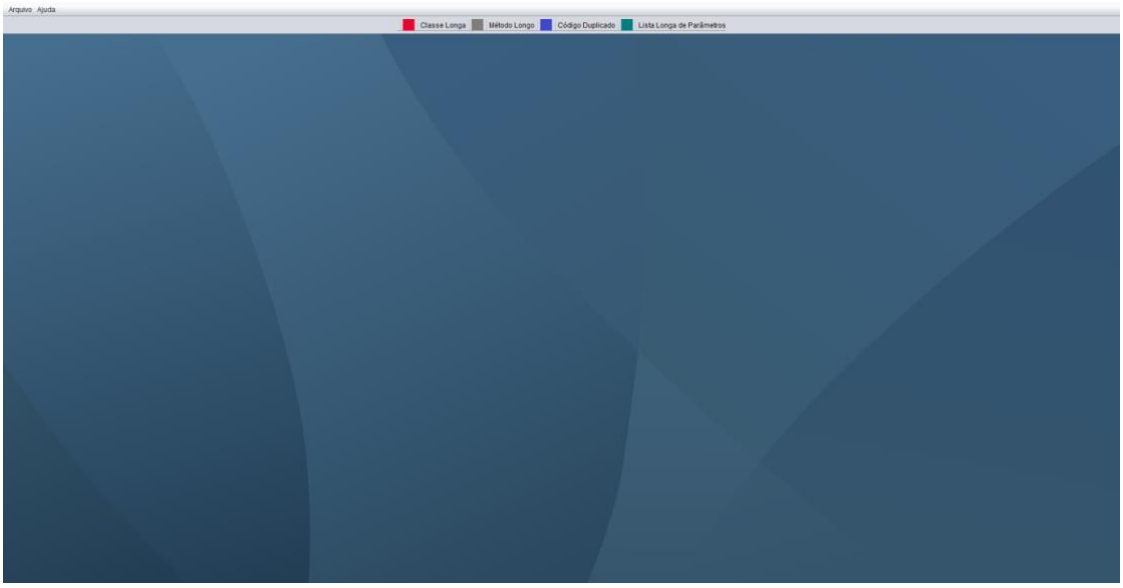
3.3.1 Uma Visão Geral da Ferramenta

Para utilizar a ferramenta, o usuário necessita apenas selecionar qual projeto será trabalhado. A tela inicial da ferramenta é um painel vazio, para selecionar um projeto basta apertar o botão “Arquivo” (FIGURA 12) no menu superior, então a ferramenta irá possibilitar que seja selecionado um repositório. Sendo assim, a ferramenta montará a árvore de menu lateral esquerdo com as pastas e arquivos que o usuário poderá operar. A ferramenta também executa restrições de arquivos, onde só poderá ser visualizados arquivos com a extensão Java, impedindo que outros tipos de arquivos sejam utilizados indevidamente. A tela inicial da ferramenta, além de possuir o menu na lateral esquerda, tem um painel na lateral direita onde será gerado um gráfico assim que um arquivo for selecionado para análise, nesse painel aparece as informações dos *Bad Smells* identificados, como é possível observar pela FIGURA 18.

Ao acessar o sistema a primeira tela que aparece para o usuário é a tela Inicial, como mostra a FIGURA 11. Nesta tela podem-se visualizar dois botões superiores no lado esquerdo, possível de observar na FIGURA 12, onde o primeiro tem como função possibilitar a seleção de algum projeto externo ou local, e o segundo para exibir

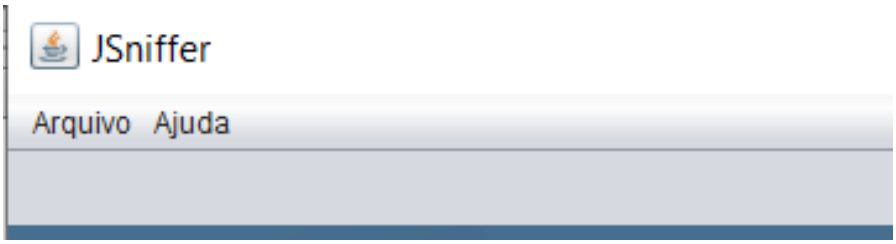
algumas informações do sistema, como equipe de desenvolvimento e um breve texto de apresentação.

FIGURA 11 - TELA INICIAL DA FERRAMENTA



FONTE: Elaborada pelo autor.

FIGURA 12 – BOTÕES PRINCIPAIS DA TELA INICIAL



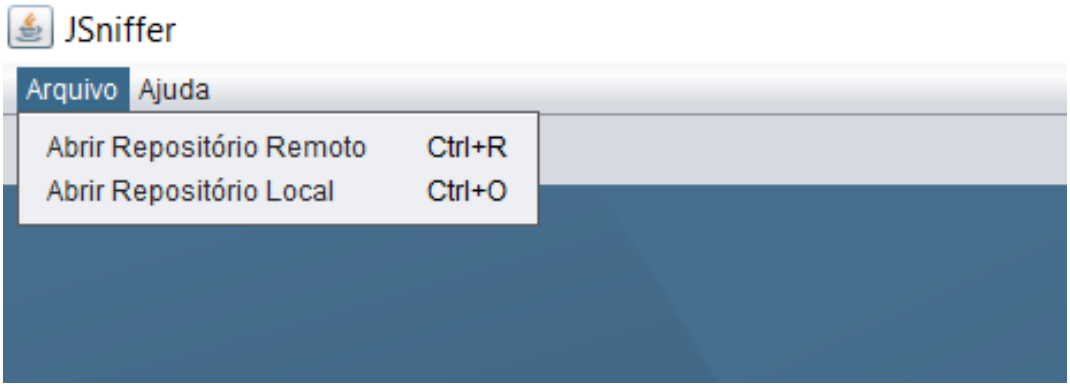
FONTE: Elaborada pelo autor.

3.3.2 Selecionar um Repositório

Para que a ferramenta possa realizar as análises de *Bad Smells*, primeiro é necessário abrir um projeto Java. É possível de se inicializar esse projeto para análise de duas formas: (i) é possível obter diretórios de forma local navegando pelos

repositórios do próprio sistema; ou (ii) remotamente através da lista de repositórios públicos ou privados do *Github*, inserindo o endereço URL do repositório e as devidas credenciais de acesso. É possível trabalhar em vários projetos concorrentemente. Para iniciar um projeto é necessário selecionar a opção “Arquivo” no menu superior, como é possível visualizar pela FIGURA 13, e um submenu será exibido com as duas opções de obtenção de diretórios citadas anteriormente.

FIGURA 13 – SUBMENU PARA SELEÇÃO DE REPOSITÓRIO

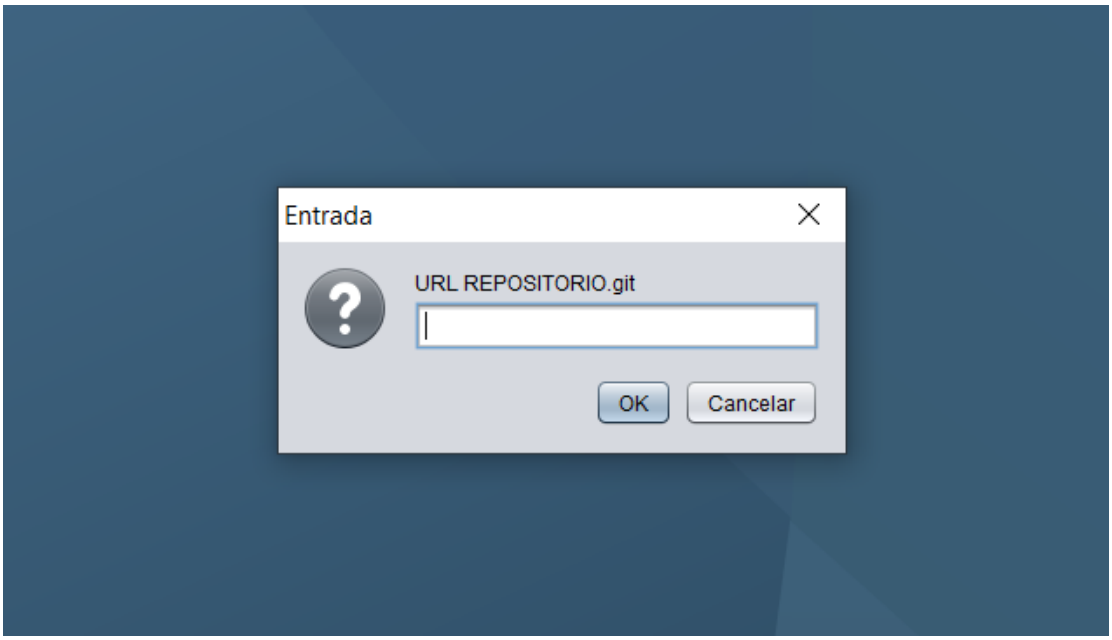


FONTE: Elaborada pelo autor.

Ao selecionar a primeira opção do submenu, a tela da FIGURA 14 será aberta, apresentando esta possibilidade para selecionar repositório remoto, projetos em Java que se encontram hospedados no *Github*. Para isso se faz necessário possuir e digitar a URL do repositório que queira utilizar na caixa de texto e selecionar a opção “OK”, ou “Cancelar” para finalizar o processo. Caso o repositório seja público uma arvore de diretórios será aberta após download dos arquivos, caso seja privado será necessário informar as credenciais de autenticação para que o sistema possa ter acesso aos arquivos, como é apresentado na FIGURA 15 e 16, e logo após alguns segundos será possível visualizar o menu lateral com os diretórios do projeto na ferramenta, como é possível observar pela FIGURA 18.

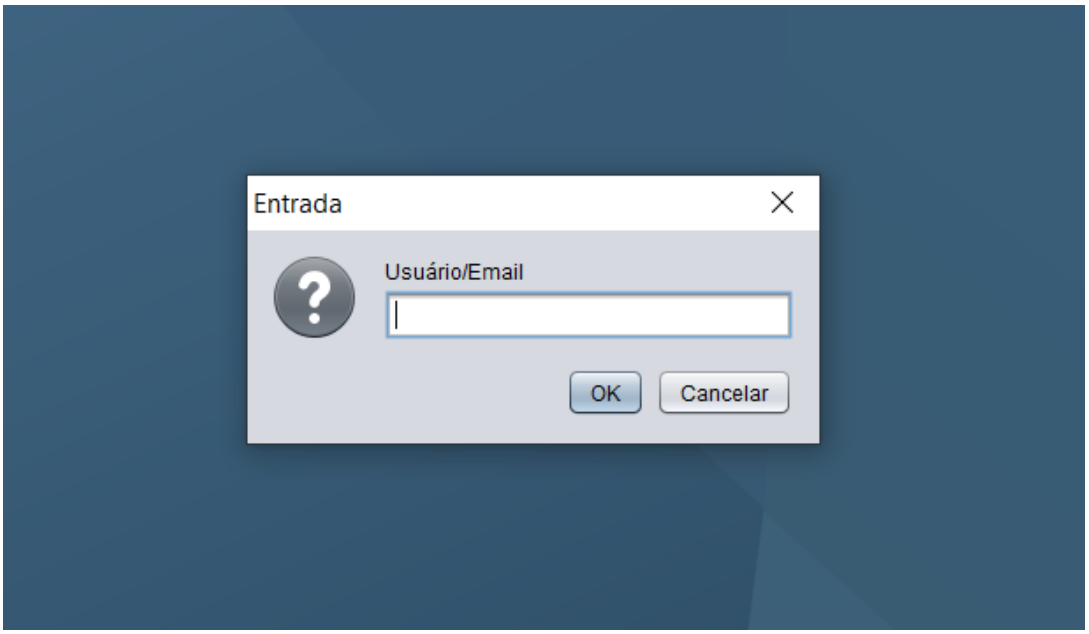
Ao selecionar a segunda opção do submenu, a tela da FIGURA 17 será aberta, apresentando esta possibilidade para selecionar repositório disponível no próprio sistema do usuário, será necessário apenas buscar o projeto e selecionar a opção “Abrir”.

FIGURA 14 - TELA DE SELEÇÃO DE REPOSITÓRIO REMOTO



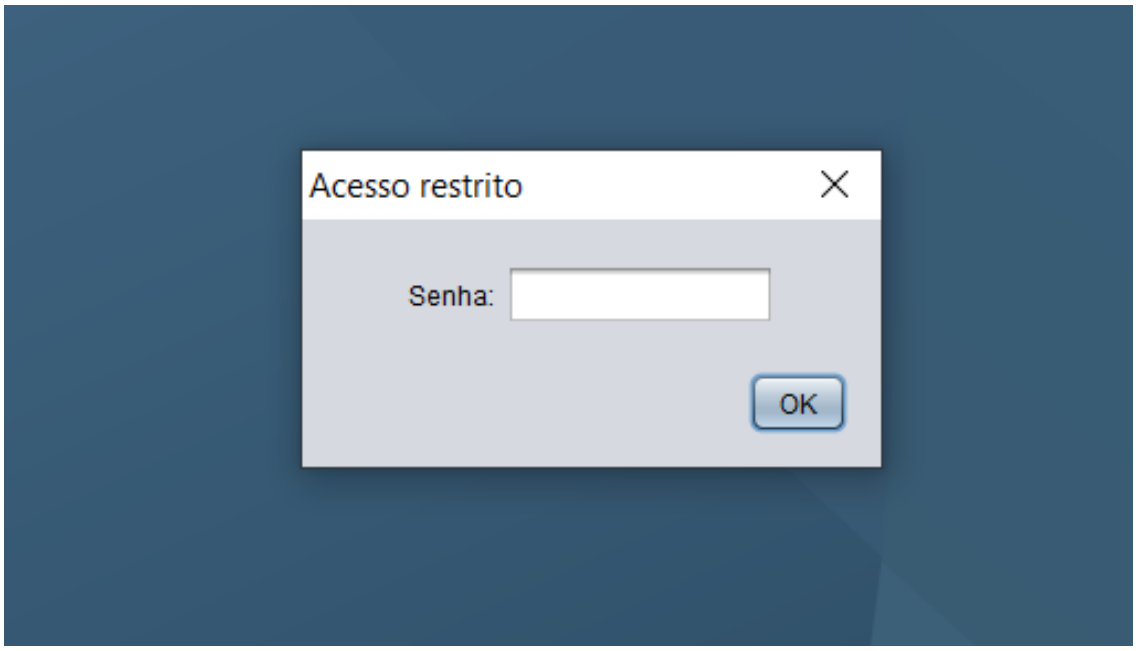
FONTE: Elaborada pelo autor.

FIGURA 15 - TELA DE INSERÇÃO DE USUÁRIO OU EMAIL PARA REPOSITÓRIO REMOTO PRIVADO



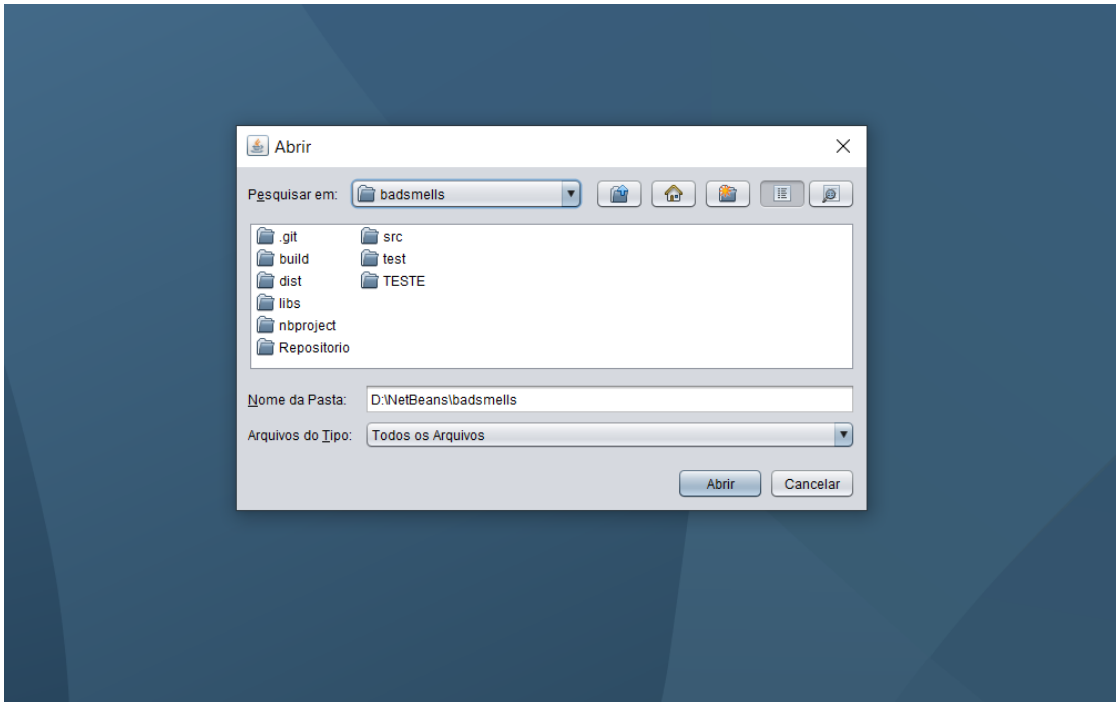
FONTE: Elaborada pelo autor.

FIGURA 16 - TELA DE INSERÇÃO DE SENHA PARA REPOSITÓRIO REMOTO PRIVADO



FONTE: Elaborada pelo autor.

FIGURA 17 - TELA DE SELEÇÃO DE REPOSITÓRIO LOCAL



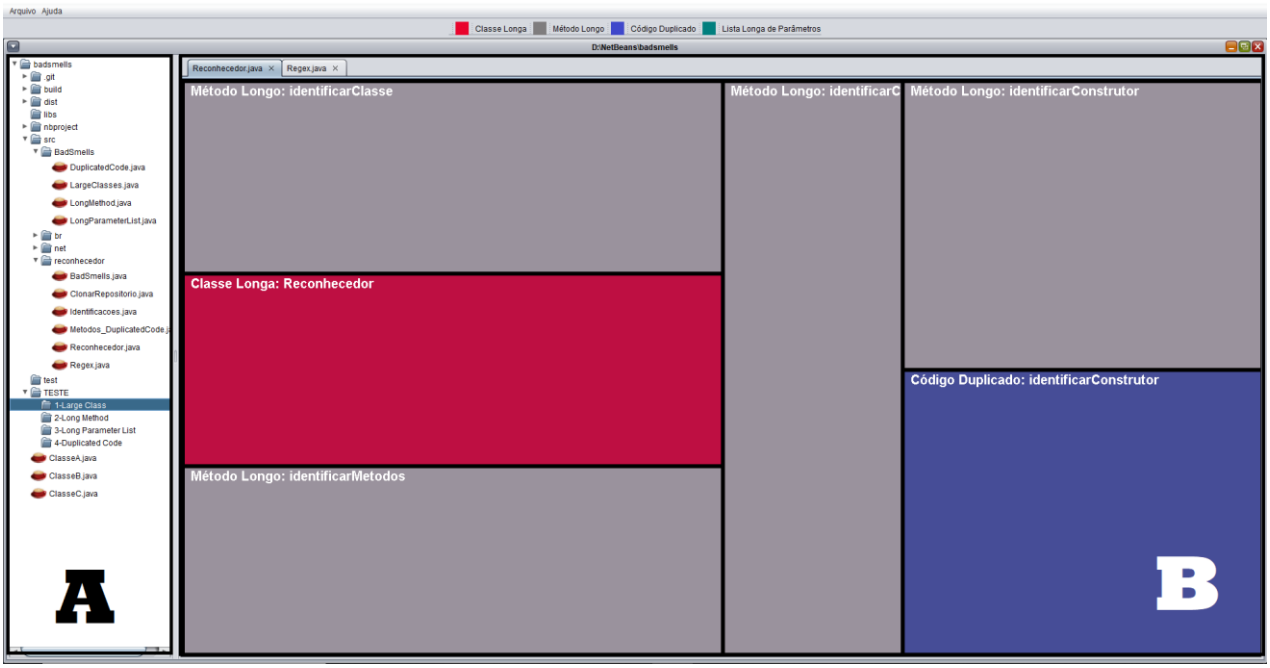
FONTE: Elaborada pelo autor.

3.3.3 Tela Principal

A FIGURA 18 apresenta como a ferramenta está organizada após abrir um projeto. Na FIGURA 18A é a árvore de arquivos. Esta árvore representa o projeto aberto, tendo como possibilidade navegar pelas pastas e arquivos do projeto.

Para efetivamente se realizar a análise é necessário abrir um arquivo de código fonte Java, navegando até ele e clicando duas vezes sobre o arquivo. Após, a ferramenta irá abrir uma nova aba na janela, apresentado os resultados da análise, a FIGURA 18B apresenta uma destas abas abertas, com o resultado dos *Bad Smells* identificados em um arquivo. É possível analisar vários arquivos ao mesmo tempo, onde para cada arquivo analisado uma nova aba será criada.

FIGURA 18 - TELA PRINCIPAL DA FERRAMENTA



FONTE: Elaborada pelo autor.

A ferramenta, além de gerar o gráfico com as estruturas identificadas, exibe um pequeno texto contendo uma explicação sobre a anomalia encontrada no código e uma

possível solução que pode ser tomada para corrigir o problema. Os trechos de código problemáticos identificados e as métricas não atendidas também são exibidos para informar ao desenvolvedor onde está o problema e o que ele fez para não atingir as expectativas.

No gráfico gerado, cada *Bad Smell* é identificado por blocos de cores distintas, e ao selecionar um bloco o mesmo é expandido, onde será possível observar detalhes sobre o *Bad Smell* identificado, como o método e trecho de código com o problema estrutural, quantidade de linhas, texto informativo sobre o problema identificado e uma sugestão de correção que pode ser realizada pelo desenvolvedor para resolver o problema, como é possível observar pela FIGURA 19.

FIGURA 19 - TELA DETALHES DE BAD SMELL

FONTE: Elaborada pelo autor.

4 AVALIAÇÃO DA FERRAMENTA

Este capítulo descreve o uso prático e o nível de eficácia da ferramenta JSNIFFER. Há um número crescente de ferramentas de análise de software disponíveis para a detecção de anomalias de código (PAIVA *et al*., 2006). O problema é como avaliar uma ferramenta para definir o quão eficaz ela pode ser para detecção de *Bad Smells*.

A dificuldade não está somente nas diferentes interpretações que as anomalias de código podem ter, mas também em sua identificação manual (SANTOS, 2016). Portanto, é difícil encontrar sistemas de código aberto com listas validadas de anomalias de código para gerar uma análise mais aprofundada (PAIVA *et al.*, 2006).

Foi nesse contexto que foi optado por utilizar como valores de referência o resultado de avaliações realizadas de forma manual, por 2 especialistas na área, que analisaram todas as classes dos sistemas selecionados, e individualmente identificaram os *Bad Smells* de acordo com suas experiências e conhecimentos com desenvolvimento de software*,* para que fossem colocados em comparação com os resultados obtidos pela ferramenta.

A ferramenta proposta nesse trabalho utiliza apenas métricas com valores limiares definidos. Considerando que os especialistas definiram os *Bad Smells* manualmente e com base na sua experiência isso pode levar a erros e influenciar nos resultados da ferramenta, pois pode haver, às vezes, imprecisão de suas definições, pois existem diferentes interpretações para cada anomalia de código.

A avaliação do sistema envolveu 3 sistemas distintos: um sistema Compilador, o próprio sistema da ferramenta descrita nesse trabalho e um sistema Corretor de cartões respostas.

4.1 AVALIAÇÃO NO SISTEMA COMPILADOR

Essa avaliação envolveu um sistema que funciona como uma espécie de compilador, passando pelos processos de análise léxica, sintática e gerador de código intermediário, contendo somente estruturas aritméticas e de repetição. Encontra-se disponível para download no site: https://github.com/matheuscslv/Compilador.

TABELA 1 – ANÁLISE AUTOMÁTICA DO SISTEMA COMPILADOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	5	98	0	0	0	0	0
B	7	128	0	0	0	0	0
C	3	20	0	0	0	0	0
D	4	252	0	3	1	0	4
E	2	87	0	1	0	0	1
F	2	53	0	0	0	0	0
G	13	185	3	0	1	0	4
H	18	245	3	0	1	0	4
I	13	284	0	3	1	0	4
J	4	31	0	0	0	0	0
K	9	238	0	1	1	0	2
L	7	183	0	2	1	0	3
M	5	28	0	0	0	0	0
N	7	125	0	0	0	0	0
TOTAL	99	1957	6	10	6	0	22

FONTE: Elaborada pelo autor.

4.1.1 Análise manual realizada pelo 1º especialista no sistema Compilador

TABELA 2 – ANÁLISE MANUAL DO 1º ESPECIALISTA NO SISTEMA COMPILADOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	5	98	0	0	0	0	0
B	7	128	0	0	0	0	0
C	3	20	0	0	0	0	0
D	4	252	0	3	1	0	4
E	2	87	0	1	0	0	1
F	2	53	0	0	0	0	0
G	13	185	1	0	1	0	2
H	18	245	2	0	1	0	3
I	13	284	0	3	1	0	4
J	4	31	0	0	0	0	0
K	9	238	0	1	1	0	2
L	7	183	0	2	1	0	3
M	5	28	0	0	0	0	0
N	7	125	0	0	0	0	0
TOTAL	99	1957	3	10	6	0	19

FONTE: Elaborada pelo autor.

Durante as análises realizadas pelo 1º avaliador foram encontrados 19 *Bad Smells* de forma manual observando as classes do sistema. A ferramenta, de forma automática, identificou 22 *Bad Smells,* distribuídos entre as classes do sistema*.* Sendo a maioria das anomalias identificadas iguais nas duas formas de detecção.

A análise manual identificou 3 *Bad Smells* do tipo Código Duplicado e a ferramenta identificou 6 do mesmo tipo. Entre as que divergiram está o código da FIGURA 20 que foi identificado pela ferramenta por aparecer em 2 métodos distintos, porém como a métrica definida pela ferramenta foi de 5 linhas, então a análise observando esse valor foi considerado, mas a primeira e a última linha foram desconsideradas durante a análise manual por considerarem como fechamento de uma estrutura de condição e por isso não foi considerado como *Bad Smell* na forma manual.

FIGURA 20 – PRIMEIRO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA COMPILADOR

```
}else{
    if(!pilha.isEmpty()){
        System.out.println("ERRO NA OPERAÇÃO ARITMÉTICA");
    }
}
```

FONTE: Elaborada pelo autor.

O código da FIGURA 21 que foi identificado pela ferramenta também se encaixa pelo mesmo motivo apresentado anteriormente, desconsiderando a primeira e as três últimas linhas.

FIGURA 21 – SEGUNDO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA COMPILADOR

```
}else if(token.getCodigo() == CodigosToken.ABRE_PARENTESES){
    pilha.addFirst(token.getCodigo());
    scan();
    estado1();
}else{
    System.out.println("ERRO OPERAÇÃO ARITMÉTICA");
}
```

FONTE: Elaborada pelo autor.

O código da FIGURA 22 identificado pela ferramenta é considerado correto considerando as métricas da ferramenta, porém durante a análise manual esse trecho de código foi desconsiderado como *Bad Smell* pois o conteúdo interno da condição presente nesse trecho possui uma quebra e o restante do conteúdo não é igual ao método ao qual ele foi comparado.

FIGURA 22 – TERCEIRO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA COMPILADOR

```
while(token.getCodigo() == CodigosToken.ABRE_PARENTESES){
    pilha.addFirst(token.getCodigo());
    scan();
}
if(token.getCodigo() == CodigosToken.DIFERENTE_ALONE){
    scan();
    estado12();
}else if(token.getCodigo() == CodigosToken.ID
        || token.getCodigo() == CodigosToken.VALOR_INTEIRO
        || token.getCodigo() == CodigosToken.VALOR_REAL){
    scan();
```

FONTE: Elaborada pelo autor.

4.1.2 Análise manual realizada pelo 2º especialista no sistema Compilador

TABELA 3 – ANÁLISE MANUAL DO 2º ESPECIALISTA NO SISTEMA COMPILADOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	5	98	0	0	0	0	0
B	7	128	0	0	0	0	0
C	3	20	0	0	0	0	0
D	4	252	0	3	1	0	4
E	2	87	0	1	0	0	1
F	2	53	0	0	0	0	0
G	13	185	1	0	1	0	2
H	18	245	2	0	1	0	3
I	13	284	0	3	1	0	4
J	4	31	0	0	0	0	0
K	9	238	0	1	1	0	2

L	7	183	0	2	1	0	3
M	5	28	0	0	0	0	0
N	7	125	0	0	0	0	0
TOTAL	99	1957	3	10	6	0	19

FONTE: Elaborada pelo autor.

Durante as análises realizadas pelo 2º avaliador foram encontrados os mesmos 19 *Bad Smells* identificados pelo 1º avaliador, de forma manual, observando as classes do sistema, o que garante que o resultado final das análises seja igual, para as semelhanças e diferenças, as identificadas durante a análise feita pelo 1º avaliador.

4.2 AVALIAÇÃO SO SISTEMA JSNIFFER

Essa avaliação envolveu o próprio sistema que está sendo descrito nesse trabalho. Encontra-se disponível para download no site: https://github.com/jcfurtado86/badsmells.

TABELA 4 – ANÁLISE AUTOMÁTICA DO SISTEMA JSNIFFER

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	4	87	0	1	0	0	1
B	6	72	0	0	0	0	0
C	5	49	0	0	0	0	0
D	3	44	0	0	0	0	0
E	7	162	0	2	0	0	2
F	3	37	0	0	0	0	0
G	7	283	0	2	1	0	3

H	3	106	0	1	0	0	1
I	2	41	0	0	0	0	0
J	2	159	0	1	1	0	2
K	1	118	1	0	0	0	1
L	4	71	0	0	0	0	0
M	1	44	0	0	0	1	1
N	6	40	0	0	0	1	1
O	2	71	0	1	0	0	1
P	0	18	0	0	0	0	0
Q	4	61	0	0	0	0	0
R	16	454	1	4	1	0	6
S	0	27	0	0	0	0	0
TOTAL	76	1944	2	12	3	2	19

FONTE: Elaborada pelo autor.

4.2.1 Análise manual realizada pelo 1º especialista no sistema JSNIFFER

TABELA 5 – ANÁLISE MANUAL DO 1º ESPECIALISTA NO SISTEMA JSNIFFER

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	4	87	0	1	0	0	1
B	6	72	0	0	0	0	0
C	5	49	0	0	0	0	0
D	3	44	0	0	0	0	0
E	7	162	1	2	1	0	4

F	3	37	0	0	0	0	0
G	7	283	0	2	1	0	3
H	3	106	0	1	0	0	1
I	2	41	0	0	0	0	0
J	2	159	0	1	1	0	2
K	1	118	0	0	0	0	0
L	4	71	0	0	0	0	0
M	1	44	0	0	0	1	1
N	6	40	0	0	0	1	1
O	2	71	0	1	0	0	1
P	0	18	0	0	0	0	0
Q	4	61	0	0	0	0	0
R	16	454	1	4	1	0	6
S	0	27	0	0	0	0	0
TOTAL	76	1944	2	12	4	2	20

FONTE: Elaborada pelo autor.

Durante as análises realizadas pelo 1º avaliador foram encontrados 20 *Bad Smells* de forma manual observando as classes do sistema. A ferramenta, de forma automática, identificou 19 *Bad Smells,* distribuídos entre as classes do sistema. Sendo a maioria das anomalias identificadas iguais nas duas formas de detecção.

A análise manual identificou 2 *Bad Smells* do tipo Código Duplicado e a ferramenta também identificou 2 do mesmo tipo, sendo que 1 (classe R) foi igual em ambas as análises, e outra identificação ocorreu em diferentes classes do código. Entre as que divergiram está o código da FIGURA 23 que foi identificado de forma manual e não foi pela ferramenta. O trecho de código se apresenta igualmente em dois métodos diferentes na mesma classe e a ferramenta não conseguiu identificar.

FIGURA 23 – PRIMEIRO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA JSNIFFER

```
if (node != null) {
    FileNode info = (FileNode) node.getUserObject();
    if (info.getFile().isFile()) {
        int opened = jTabbedPane.indexOfTab(node.toString());
        if (opened == -1) {

            try {
                JPanel newTab = TabbedPaneGUI.createTab();

                BufferedReader input = new
                    BufferedReader(new InputStreamReader(new FileInputStream(info.getFile())));
                JScrollPane tempJSCP = (JScrollPane) newTab.getComponent(0);
                JTextArea tempJTA = (JTextArea) tempJSCP.getViewport().getComponent(0);
                tempJTA.read(input, info.getFile().getName());

                Reconhecedor find = new Reconhecedor();
                ArrayList<String> resultado = find.executar(tempJTA.getText());
                String resultadoSaida = "";
                for (int i = 0; i < resultado.size(); i++) {
                    resultadoSaida = resultadoSaida + resultado.get(i);
                }

                tempJTA.setText(resultadoSaida);

                try {
                    newTab = DemoModel.main();
                    jTabbedPane.add(node.toString(), newTab);
                    jTabbedPane.setTabComponentAt(jTabbedPane.getTabCount() - 1,
                            new ButtonTabComponent(jTabbedPane));
                    jTabbedPane.setSelectedComponent(newTab);
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }

            } catch (IOException ex) {
                Logger.getLogger(OpenProjectGUI.class.getName()).log(Level.SEVERE, null, ex);
            }
        } else {
            jTabbedPane.setSelectedIndex(opened);
        }
    }
}
```

FONTE: Elaborada pelo autor.

O código da FIGURA 24 foi identificado pela ferramenta e não foi de forma manual, porém o mesmo é considerado incorreto pois durante a análise manual esse trecho de código não foi identificado em dois métodos diferentes na mesma classe.

FIGURA 24 – SEGUNDO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA JSNIFFER

```
Código Duplicado: ButtonTabComponent

#Métodos comparados:
    ButtonTabComponent => getText

#Linhas duplicadas:
  >             int i=pane.indexOfTabComponent(ButtonTabComponent.this);
  >             if (i !=-1) {
  >                 return pane.getTitleAt(i);
  >             }
  >             return null;

#Total de linha(s) duplicada(s): 5


O badsmell identificado mostra que o método 'ButtonTabComponent' possui 5 linha(s) duplicada(s) com
o método 'getText'. Linhas duplicadas na codificação de um programa são redundantes
e desnecessárias, o que resulta em um código maior do que indicado e reduz sua organização.
Neste caso, é aconselhável que o código seja refatorado, pois apresenta mais de 5 linhas
duplicadas.
```

FONTE: Elaborada pelo autor.

A análise manual também identificou 4 *Bad Smells* do tipo Classe Longa e a ferramenta identificou 3 do mesmo tipo, sendo que 3 são exatamente iguais nas duas análises. Entre a que divergiu está no código da Classe E, que foi identificado durante a análise manual por possuir mais de 150 linhas e a ferramenta não conseguiu identificar.

4.2.2 Análise manual realizada pelo 2º especialista no sistema JSNIFFER

TABELA 6 – ANÁLISE MANUAL DO 2º ESPECIALISTA NO SISTEMA JSNIFFER

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	4	87	0	1	0	0	1

B	6	72	0	0	0	0	0
C	5	49	0	0	0	0	0
D	3	44	0	0	0	0	0
E	7	162	1	2	1	0	4
F	3	37	0	0	0	0	0
G	7	283	1	1	1	0	3
H	3	106	0	1	0	0	1
I	2	41	0	0	0	0	0
J	2	159	0	1	1	0	2
K	1	118	1	0	1	0	2
L	4	71	2	0	0	0	2
M	1	44	0	0	0	1	1
N	6	40	0	0	0	1	1
O	2	71	0	1	0	0	1
P	0	18	0	0	0	0	0
Q	4	61	2	0	0	0	2
R	16	454	5	3	1	0	9
S	0	27	0	0	0	0	0
TOTAL	76	1944	12	10	5	2	29

FONTE: Elaborada pelo autor.

Durante as análises realizadas pelo 2º avaliador foram encontrados 29 *Bad Smells* de forma manual observando as classes do sistema. A ferramenta, de forma automática, identificou 19 *Bad Smells,* distribuídos entre as classes do sistema*.* Sendo um número considerável de anomalias identificadas iguais nas duas formas de detecção.

A análise manual identificou 12 *Bad Smells* do tipo Código Duplicado e a ferramenta identificou 2 do mesmo tipo. Entre as que divergiram está o código da FIGURA 23 que foi identificado de forma manual e não foi pela ferramenta, o trecho de código se apresenta igualmente em dois métodos diferentes na mesma classe e a ferramenta não conseguiu identificar. Outro trecho que divergiu está o código da classe G que foi identificado, de forma manual, um *Bad Smell* de Código Duplicado, pois, há um trecho contendo 1 linha duplicada. Na classe K foi identificado um *Bad Smell* de Código Duplicado, de forma automática pela ferramenta, porém deve ser considerado incorreto, pois esse trecho de código, representado pela FIGURA 24, não foi identificado em dois métodos diferentes na mesma classe, também em relação à Código Duplicado, a análise manual identificou um *Bad Smell* de Código Duplicado, pois há um trecho contendo 1 linha duplicada. Na classe L, durante a análise manual, foram identificados 2 trechos de *Bad Smell* relacionados à Código Duplicado. Na classe Q, na análise manual, foram identificados 2 trechos de *Bad Smell* relacionados à Código Duplicado. Na classe R foi identificada, de forma automática pela ferramenta, um trecho de código duplicado que não foi identificada de forma manual. Também na classe R foram identificados 5 trechos de *Bad Smells* relacionados à Código Duplicado, pela análise de forma manual. Cada trecho identificado pela forma manual contém apenas 1 linha duplicada, sendo assim, os trechos identificados anteriormente não foram identificados pela ferramenta pelo fato dela considerar 5 ou mais linhas duplicadas consecutivas para ser enquadrado como esse tipo de *Bad Smell*.

A análise manual identificou 10 *Bad Smells* do tipo Método Longo e a ferramenta identificou 12 do mesmo tipo. Entre as que divergiram está um método da classe G que a ferramenta acabou identificando como *Bad Smell* de Método Longo sendo que a avaliação manual não identificou como um *Bad Smell*. Na classe R foi identificado pela ferramenta um *Bad Smell* de Método Longo que não foi encontrado durante a análise manual.

A análise manual identificou 5 *Bad Smells* do tipo Classe Longa e a ferramenta identificou 3 do mesmo tipo. Entre as que divergiram está o código da classe K que foi identificado um *Bad Smell* de Classe Longa, mas a ferramenta acabou não

identificando, pelo fato da métrica utilizada considerar somente classes com 150 ou mais linhas, o que não está contido na classe analisada. Outra que divergiu está no código da Classe E, que foi identificado durante a análise manual por possuir mais de 150 linhas e a ferramenta não conseguiu identificar.

4.3 AVALIAÇÃO DA FERRAMENTA CORRETOR

Essa avaliação envolveu um sistema que funciona como uma espécie de Corretor de Cartões Resposta, utilizando apenas o seu módulo de correção. Encontra-se disponível para download no site: https://github.com/matheuscslv/corretor.

TABELA 7 – ANÁLISE AUTOMÁTICA DO SISTEMA CORRETOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	1	44	0	0	0	0	0
B	1	31	0	0	0	0	0
C	6	469	0	3	1	3	7
D	3	123	0	2	0	0	2
E	5	279	1	0	1	0	2
F	6	234	1	3	1	0	5
G	1	42	0	0	0	0	0
H	1	30	0	0	0	0	0
I	4	26	0	0	0	0	0
J	6	462	0	5	1	0	6
K	3	66	0	0	0	1	1
L	8	545	0	7	1	0	8

M	6	237	0	3	1	0	4
N	5	252	0	2	1	0	3
O	3	173	0	2	0	0	2
P	6	398	1	6	1	0	8
Q	3	147	0	1	0	0	1
R	4	311	1	3	1	0	5
S	3	156	0	1	0	0	1
TOTAL	75	4025	4	38	9	4	55

FONTE: Elaborada pelo autor.

4.3.1 Análise manual realizada pelo 1º especialista no sistema Corretor

TABELA 8 – ANÁLISE MANUAL DO 1º ESPECIALISTA NO SISTEMA CORRETOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	1	44	0	0	0	0	0
B	1	31	0	0	0	0	0
C	6	469	0	4	1	3	8
D	3	123	0	2	0	0	2
E	5	279	1	2	1	0	4
F	6	234	1	3	1	0	5
G	1	42	0	0	0	0	0
H	1	30	0	0	0	0	0
I	4	26	0	0	0	0	0
J	6	462	0	5	1	0	6

K	3	66	0	0	0	1	1
L	8	545	0	7	1	0	8
M	6	237	0	3	1	0	4
N	5	252	0	2	1	0	3
O	3	173	0	2	1	0	3
P	6	398	1	6	1	0	8
Q	3	147	0	1	0	0	1
R	4	311	0	3	1	0	4
S	3	156	0	1	1	0	2
TOTAL	75	4025	3	41	11	4	59

FONTE: Elaborada pelo autor.

Durante as análises foram encontrados 59 *Bad Smells* de forma manual observando as classes do sistema. A ferramenta, de forma automática, identificou 55 *Bad Smells,* distribuídos entre as classes do sistema*.* Sendo a maioria das anomalias identificadas iguais nas duas formas de detecção.

A análise manual identificou 3 *Bad Smells* do tipo Código Duplicado e a ferramenta identificou 4 do mesmo tipo. Entre o que divergiu está o código da FIGURA 25 que foi identificado pela ferramenta e não pela forma manual, porém o mesmo está incorreto pois o trecho não se apresenta em mais de um método diferente.

A análise manual também identificou 41 *Bad Smells* do tipo Método Longo e a ferramenta identificou 38 do mesmo tipo. Entre as que divergiram está 1 método da Classe C e 2 métodos da Classe E que apresentam mais de 30 linhas e não foram identificados pela ferramenta.

A análise manual também identificou 11 *Bad Smells* do tipo Classe Longa e a ferramenta identificou 9 do mesmo tipo. Entre as que divergiram estão as Classes O e S que foram identificadas de forma manual por terem mais de 150 linhas e não foram identificadas pela ferramenta.

FIGURA 25 – PRIMEIRO TRECHO DE CÓDIGO DUPLICADO DO SISTEMA CORRETOR

```
Código Duplicado: TabbedPane

#Métodos comparados:
    TabbedPane => executar

#Linhas duplicadas:
 >          setTitle("EDEF");
 >          setIconImage(new ImageIcon(getClass().getResource("/resources/icon.png")).getImage());
 >          setSize(500,800);
 >          setResizable(false);
 >          JTabbedPane jtp=new JTabbedPane();
 >          getContentPane().add(jtp);
 >          JPanel corretor=corrigirprovas();
 >          Janela cadastro=new Janela();
 >          jtp.addTab("Corretor",corretor);
 >          jtp.addTab("Alunos e Disciplinas Cadastradas",cadastro.AbrirJanela(jtp,this));
 >           try{
 >              Connection c=DriverManager.getConnection(url,username,password);
 >               Deleta todas as notas dos alunos
 >              String frase="insert into disciplinas values(?,?,?)";
 >              ps=c.prepareStatement(frase);
 >              ps.setString(1,"CCCGS");
 > ...

#Total de linha(s) duplicada(s): 24


O badsmell identificado mostra que o método 'TabbedPane' possui 24 linha(s) duplicada(s) com
o método 'executar'. Linhas duplicadas na codificação de um programa são redundantes
e desnecessárias, o que resulta em um código maior do que indicado e reduz sua organização.
Neste caso, é aconselhável que o código seja refatorado, pois apresenta mais de 5 linhas
duplicadas.
```

FONTE: Elaborada pelo autor.

4.3.2 Análise manual realizada pelo 2º especialista no sistema Corretor

TABELA 9 – ANÁLISE MANUAL DO 2º ESPECIALISTA NO SISTEMA CORRETOR

Classe	Métodos	Linhas	CD	ML	LC	LLP	Total
A	1	44	0	0	0	0	0
B	1	31	0	0	0	0	0
C	6	469	3	4	1	2	10
D	3	123	0	1	0	0	1
E	5	279	3	2	1	0	6

F	6	234	3	2	1	0	6
G	1	42	0	0	0	0	0
H	1	30	0	0	0	0	0
I	4	26	0	0	0	0	0
J	6	462	1	5	1	0	7
K	3	66	1	0	0	0	1
L	8	545	3	6	1	0	10
M	6	237	0	2	1	0	3
N	5	252	4	2	1	0	7
O	3	173	1	2	1	0	4
P	6	398	7	5	1	0	13
Q	3	147	0	1	1	0	2
R	4	311	1	2	1	0	4
S	3	156	0	1	1	0	2
TOTAL	75	4025	27	35	12	2	76

FONTE: Elaborada pelo autor.

Durante as análises foram encontrados 76 *Bad Smells* de forma manual observando as classes do sistema. A ferramenta, de forma automática, identificou 55 *Bad Smells,* distribuídos entre as classes do sistema. Sendo que, aproximadamente, metade das anomalias identificadas iguais nas duas formas de detecção.

A análise manual identificou 27 *Bad Smells* do tipo Código Duplicado e a ferramenta identificou 4 do mesmo tipo. Entre as que divergiram está o código da Classe C onde a análise manual identificou 3 *Bad Smells* de Código Duplicado. Na Classe E a análise manual identificou 3 *Bad Smell* de Código Duplicado e a ferramenta não identificou. Na Classe E a ferramenta de forma automática identificou um *Bad Smell* de código duplicado que não foi identificado de forma manual e segundo a métrica da

ferramenta deve ser considerado um *Bad Smell* por ter mais de 5 linhas consecutivas duplicadas. Na Classe F a análise manual identificou 3 trechos de Códigos Duplicados e a ferramenta não identificou. Na Classe F a ferramenta, de forma automática, identificou um *Bad Smell* de Código Duplicado que não foi identificado de forma manual e segundo a métrica da ferramenta deve ser considerado um *Bad Smell* por ter mais de 5 linhas consecutivas duplicadas. Na Classe J foi identificado um *Bad Smell* de Código Duplicado de forma manual que não foi identificado pela forma automática. Na Classe K foi identificado um *Bad Smell* de Código Duplicado de forma manual que não foi identificado pela forma automática. Na Classe L foi identificado 3 *Bad Smells* de Código Duplicado, de forma manual, que não foi identificado pela forma automática. Na Classe N foi identificado 4 *Bad Smells* de Código Duplicado, de forma manual, que não foi identificado pela forma automática. Na Classe O foi identificado 1 *Bad Smell* de Código Duplicado, de forma manual, que não foi identificado pela forma automática. Na classe P foi identificado 7 *Bad Smells* de Código Duplicado, de forma manual, que não foi identificado pela forma automática. Na Classe P a ferramenta, de forma automática, identificou um *Bad Smell* de Código Duplicado que não foi identificado de forma manual, porém estava correto de acordo com a métrica definida para esse *Bad Smell* de 5 ou mais linhas duplicadas consecutivas. Na Classe R foi identificado 1 *Bad Smell* de Código Duplicado, de forma manual, que não foi identificado pela forma automática. Na classe R a ferramenta de forma automática identificou um *Bad Smell* de Código Duplicado que não foi identificado de forma manual, porém estava incorreto pois o trecho de código não é localizado em outros lugares no código. Cada trecho identificado pela forma manual contém apenas 1 linha duplicada, sendo assim, os trechos identificados anteriormente não foram identificados pela ferramenta pelo fato dela considerar 5 ou mais linhas duplicadas consecutivas para ser enquadrado como esse tipo de *Bad Smell*.

A análise manual identificou 35 *Bad Smells* do tipo Método Longo e a ferramenta identificou 38 do mesmo tipo. Na Classe D foi identificado, de forma automática, um *Bad Smell* de Método Longo que a análise manual não considerou um *Bad Smell*, pois a métrica para método longo da ferramenta considera métodos com mais 30 linhas.

Outro que divergiu está o código da Classe C que foi identificado um *Bad Smell* de Método Longo, de forma manual, que não foi identificado pela ferramenta. Na Classe E foi identificado 2 *Bad Smells* relacionado ao tipo Método Longo, pela análise manual, que não foi identificado pela ferramenta. Na Classe F foi identificado 3 *Bad Smells* do tipo Método Longo, de forma automática, sendo que um deles não foi identificado de forma manual, a ferramenta identificou pela métrica de método longo que foi definido para esse tipo. Na Classe L foi identificado um *Bad Smell* de Método Longo que foi identificado pela ferramenta por causa da métrica que foi definida para esse tipo, mas não foi identificada pela análise manual. Na Classe M foi identificado um *Bad Smell* de Método Longo de forma automática por causa da métrica definida, o que não foi identificado pela forma manual. Na Classe P foi identificado 1 *Bad Smell*, de forma automática, para Método Longo considerando a métrica definida para a ferramenta, porém a análise manual não identificou como *Bad Smell*. Na Classe R foi identificado um *Bad Smell* de Método Longo, de forma automática, por causa da métrica definida, que não foi identificado pela análise manual.

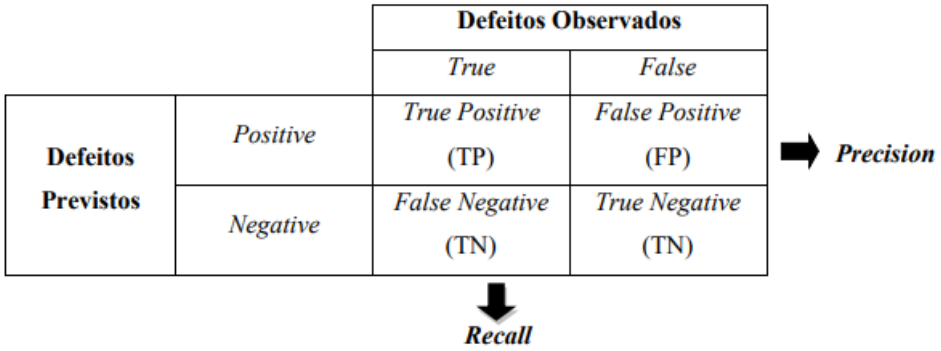
A análise manual identificou 12 *Bad Smells* do tipo Classe Longa e a ferramenta identificou 9 do mesmo tipo. Entre os que divergiram está a Classe O onde a análise manual identificou um *Bad Smell* de Classe Longa que não foi identificado pela ferramenta, a ferramenta errou, pois, o trecho deveria ser considerado um *Bad Smell* de Classe Longa pela métrica definida. Na Classe Q foi identificado, de forma manual, um *Bad Smell* de Classe Longa, porém a ferramenta não identificou por causa da métrica definida que foi de 150 linhas ou mais para ser considerado um *Bad Smell*. A Classe S também foi identificada de forma manual por ter mais de 150 linhas e não foi identificada pela ferramenta.

A análise manual identificou 2 *Bad Smells* do tipo Lista Longa de Parâmetros e a ferramenta identificou 4 do mesmo tipo. Entre as que divergiram está a Classe C que foi identificado um *Bad Smell*, de forma automática, de Lista Longa de Parâmetros que não foi identificado de forma manual. Na lasse K foi identificado, pela análise automática, um *Bad Smell* de Lista Longa de Parâmetros por ter 4 parâmetros, o que não foi considerado pela forma manual.

4.4 RESULTADOS

Como forma de quantificar a eficácia da ferramenta em detectar *Bad Smells* relevantes calculou-se duas medidas: *Precision* e *Recall*. O cálculo dos valores de *Precision* e *Recall* foram definidos por Zimmermann, Premraj e Zeller (2007), e tem como base a FIGURA 26. Uma anomalia de código relevante é uma anomalia que também está presente na lista referência de anomalias de código (SANTOS, 2016).

FIGURA 26 – *RECALL* E *PRECISION* COMO MODELO DE AVALIAÇÃO



Fonte: (ZIMMERMANN; PREMRAJ; ZELLER, 2007)

Santos (2016) define *Recall* como a razão entre o número de registos relevantes recuperados e o número total de registos relevantes analisados pela técnica. A fórmula desse cálculo é:

$$Recall = \frac{TP}{(TP + FP)}$$

Santos (2016) define *Precision* como a razão entre o número de registos relevantes recuperados e o número total de registos relevantes e irrelevantes recuperados. A fórmula desse cálculo é:

$$Precision = \frac{TP}{(TP + FN)}$$

Os resultados de *Recall* e *Precision* das análises podem ser visualizados na TABELA 7 e TABELA 8.

TABELA 10 – *RECALL* E *PRECISION* DAS ANÁLISES CONSIDERANDO O 1º AVALIADOR

SISTEMA	TP	FP	FN	RECALL	PRECISION
COMPILADOR	19	3	0	86%	100%
JSNIFFER	18	1	2	94%	90%
CORRETOR	54	1	5	98%	91%

FONTE: Elaborada pelo autor.

TABELA 11 – *RECALL* E *PRECISION* DAS ANÁLISES CONSIDERANDO O 2º AVALIADOR

SISTEMA	TP	FP	FN	RECALL	PRECISION
COMPILADOR	19	3	0	86%	100%
JSNIFFER	15	4	13	78%	51%
CORRETOR	44	11	33	78%	56%

FONTE: Elaborada pelo autor.

TABELA 12 – MÉDIA DE *RECALL* E *PRECISION* DAS ANÁLISES CONSIDERANDO OS DOIS AVALIADORES

SISTEMA	TP	FP	FN	RECALL	PRECISION
COMPILADOR	19	3	0	86%	100%
JSNIFFER	15	4	13	86%	70,5%
CORRETOR	44	11	33	88%	73,5%
TOTAL	78	18	46	86,6	81,3%

FONTE: Elaborada pelo autor.

Pode-se observar que os valores de *Recall* do 1º avaliador para o 2º avaliador caíram em aproximadamente 20%, demonstrando que o 2° avaliador, através de suas experiências, considerou menos trechos de código como *Bad Smell,* mas em sua maioria foram semelhantes nas duas análises. Em relação a *Precision* os valores caíram cerca de 50% entre os dois avaliadores, mostrando uma rigidez maior do 2º avaliador em relação ao que deve ser considerado um *Bad Smell*, principalmente em

relação a Código Duplicado que foi o tipo que teve o maior número de casos durante a análise manual, o 2º avaliador considerou casos em que 1 ou 2 linhas duplicadas fossem considerados nesse tipo, o 1º avaliador foi mais próximo do que a ferramenta considera como *Bad Smell* de Código Duplicado que é de 5 ou mais linhas, o que explica o motivo das taxas entre o 1º e 2º avaliador serem consideravelmente diferentes.

As análises realizadas obtiveram taxa média de *Recall* de 86,6%, indicando que, de modo geral, todas as principais anomalias identificadas pelos especialistas foram encontradas pela técnica proposta, porém o sistema acabou identificado algumas anomalias a mais que não estavam na lista de referência e logo foram consideradas incorretas.

A média de *Precision* foi de 81,3% indicando que a ferramenta identificou menos anomalias do que deveria. De modo geral a ferramenta identificou valores próximos a 90% chegando à conclusão de que a ferramenta conseguiu obter valores próximos aos identificados pelo especialista, porém ela ainda não conseguiu identificar algumas anomalias simples e obteve mais falsos negativos do que positivos e mostra que ela ainda precisa ser melhorada e pode em trabalhos futuros.

Também foi possível observar que além da quantidade de problemas encontrados ter sido semelhante em ambas as análises, os problemas detectados pela ferramenta também foram encontrados em mesmas classes que a avaliação manual, o que fortalece os valores de *Precision* e *Recall* obtidos.

Vale observar que a lista referência das ferramentas foi construída manualmente, utilizando a interpretação de um especialista. Isso pode aumentar a possibilidade de falhas na definição da lista referência. Nesses casos um número maior de especialistas que definem essa listagem poderia ser maior para reduzir esse tipo de problema.

5 CONSIDERAÇÕES FINAIS

Neste trabalho foi apresentada uma abordagem para identificar e recomendar refatorações de *Bad Smells* em códigos fonte. Muitos trabalhos abordam questões relacionadas a identificação, recomendação e utilização do contexto para apresentar anomalias para o desenvolvedor, porém não foram encontrados trabalhos que tenham o mesmo resultado proposto neste trabalho.

Para atingir o objetivo o método utiliza expressões regulares para identificar e separar as estruturas de código fonte em Java e após isso é identificado, através de valores já definidos, *Bad Smells* no código com o objetivo de alertar o desenvolvedor sobre esse problema e sugerir a modificação do mesmo. Diferente de outras abordagens, essa tem o objetivo de identificar 4 tipos de *Bad Smells*: Código duplicado (*Duplicated Code*), Classe longa (*Large Class*), Método longo (*Long Method*) e Lista longa de parâmetros. Além de exibir ao usuário, da melhor forma possível, um gráfico informativo sobre o problema identificado.

Foi desenvolvida uma ferramenta para recomendar refatorações. Dessa forma, a ferramenta permite utilizar a técnica proposta para apresentar e recomendar aos desenvolvedores quais tipos de *Bad Smells* estão presentes no código, tudo isso enquanto trabalham no desenvolvimento do software.

A avaliação da ferramenta foi realizada através da análise de três sistemas de software. Comparando os resultados obtidos com uma lista de referência criada por um especialista e calculando, para cada software, o valor de *Recall* e *Precision*. A ferramenta obteve boa taxa de *Recall* e *Precision*, indicando que foi gerado poucos falsos positivos, porém identificou um número considerável de falsos negativos, que talvez, sejam mais difíceis de identificar pelos desenvolvedores, o que aponta que a ferramenta ainda precisa de mais ajustes para aumentar suas taxas.

Como trabalhos futuros sugerimos: (i) utilizar outras métricas para identificação e calibração das anomalias; (ii) avaliar a técnica proposta em outros sistemas de

software; e (iii) realizar estudos para verificar se remover anomalias durante o desenvolvimento diminui a quantidade de anomalias detectadas no software.

REFERÊNCIAS

ANDERS, Karlsen. **Flexibility — a Software Architecture Principle**. **Medium**, 20 de jan. de 2019. Disponível em: <https://medium.com/faun/flexibility-a-software-architecture-principle-6eafe045a1d4>. Acesso em: 29 de dez. de 2019.

ARCOVERDE, R. *et al*. **Automatically detecting architecturally-relevant code anomalies**. 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE). Anais...IEEE, jun. 2012. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6233419>. Acesso em: 28 mar. 2015.

BOEHM, B.W.; BROWN, J.R.; LIPOW, M. **Quantitative Evaluation of Software Quality.** Proceedings of the 2nd International Conference on Software Engineering, Los Alamitos, 13-15 October 1976.

FOKAEFS, Marios; TSANTALIS, Nikolaos; STROULIA, Eleni; CHATZIGEORGIOU, Alexander. (2011). **JDeodorant: Identification and Application of Extract Class Refactorings**. Proceedings - International Conference on Software Engineering. 1037-1039. 10.1145/1985793.1985989.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology, v. 11, n. 2, p. 1–38, 2012.

FORNO, Mateus Henrique Dal. **EVOLUÇÃO DE SOFTWARE ATRAVÉS DE REENGENHARIA: UM PROCESSO DIDÁTICO**. TCC (TCC em Engenharia de Software) – Unipampa. Rio Grande do Sul. 2014.

FOWLER, M. (1999). **Refactoring: improving the design of existing code**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GUIMARAES, E.; GARCIA, A.; CAI, Y. **Exploring Blueprints on the Prioritization of Architecturally Relevant Code Anomalies -- A Controlled Experiment**. 2014 IEEE 38th Annual Computer Software and Applications Conference, p. 344–353, 2014.

INSPAZO, **Arquitetura de Software e Design – Objetivos, Princípios e Algumas Considerações-Chave**. 2019. Disponível em: <http://inspazo.pt/arquitetura-de-software-e-design-objetivos-principios-e-algumas-consideracoes-chave> Acesso em: 29 de dez. de 2019.

JEFFERSON, Henrique. 5 princípios fundamentais da arquitetura de software. **Uebile**, 17 de set. de 2019. Disponível em: < http://blog.uebile.com/5-principios-fundamentais-da-arquitetura-de-software>. Acesso em: 29 de dez. de 2019.

KOSCIANSKI, A.; DOS SANTOS SOARES, M. **Qualidade de Sotware**. 2ª Edição ed. [s.l.] Editora Novatec, 2007.

LANZA, M.; MARINESCU, R.; DUCASSE, S. (2006). **Object-Oriented Metrics in Practice**. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

MARINESCU, R. (2004). **Detection strategies: metrics-based rules for detecting design flaws**. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, pp. 350 – 359.

MONTEIRO, M.P.; FERNANDES, J.M.; **Towards a catalogue of refactorings and code smells for AspectJ**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), v. 3880 LNCS, p. 214–258, 2006.

PAIVA, T. et al. Experimental Evaluation of Code Smell Detection Tools, 2006.

PRESSMAN, Roger. **Engenharia de Software: Uma abordagem profissional.** 7. ed. Brasil. AMGH Editora Ltda, 2011.

SALES, V.; TERRA, R.; MIRANDA L. F.; VALENTE, M. T.**. JMove: Seus métodos em classes apropriadas**. In IVBrazilian Conference on Software: Theory and Practice (CBSoft), Tools Session, pages 1–6, 2013.

SANTOS, Cleverton. **UMA TÉCNICA PARA RECOMENDAR REFATORAÇÕES DE MÉTODOS LONGOS BASEADO NO CONTEXTO ARQUITETURAL DA CLASSE**. TCC (TCC em ciência da computação) – UFS. Sergipe. 2016.

SCALET *et al*., 2000: **ISO/IEC 9126 and 14598** integration aspects: A Brazilian viewpoint. The Second World Congress on Software Quality, Yokohama, Japan, 2000.

SILVA, D.; TERRA, R.; VALENTE, M. T. **JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings**. 19 jun. 2015.

SMITH, Steve. **Princípios de arquitetura.** Microsoft Corporation, 23 de dez. de 2019. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/architecture/modern-web-apps-azure/architectural-principles>. Acesso em: 29 de dez. de 2019.

SOLIMAN, T.; El-Swesy, A. & Ahmed, S. (2010). **Utilizing ck metrics suite to uml models: A case study of microarray midas software.** Em Informatics and Systems (INFOS), 2010 The 7th International Conference on, pp. 1 –6.

SOMMERVILLE, I. **Engenharia de Software**. 9ª edição ed. [s.l: s.n.].

THIOLLENT, M. **Metodologia da Pesquisa-ação**. 7º edição ed. [s.l.] Editora São Paulo: Cortez, 1996.

Tsantalis, N. Chatzigeorgiou, A. (2009). **Identification of move method refactoring opportunities**. IEEE Transactions on Software Engineering.

Tsantalis, N. Chatzigeorgiou, A. (2011). **Identification of extract method refactoring opportunities for the decomposition of methods**. Journal of Systems and Software, 84(10):1757–1782.

VALIPOUR, M. *et al*. **A brief survey of software architecture concepts and service oriented architecture**. In: Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on. [S.l.: s.n.], 2009. p. 34–38.

ZIMMERMANN, T.; PREMRAJ, R.; ZELLER, A. Predicting defects for eclipse. Proceedings - ICSE 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, PROMISE'07, 2007.