DOI:

JSNIFFER: A TOOL FOR AUTOMATIC IDENTIFICATION OF BAD SMELLS IN JAVA.

JSNIFFER: UMA FERRAMENTA PARA IDENTIFICAÇÃO AUTOMÁTICA DE BAD SMELLS EM PROJETOS NA LINGUAGEM JAVA

Julio Furtado
UNIVERSIDADE FEDERAL DO AMAPÁ - ORCID: https://orcid.org/0000-0002-1984-9587
Matheus Costa Silva
UNIVERSIDADE FEDERAL DO AMAPÁ - ORCID: https://orcid.org/0000-0002-9933-6377
Eduardo Luigi Ciuffi
UNIFAP - UNIVERSIDADE FEDERAL DO AMAPÁ - ORCID: https://orcid.org/0000-0003-1973-2263

Abstract
This work presents an automated instrument for identifying and verifying Bad Smells in source code, and shows their impact on software development. The tool presented aims to detect these Bad Smells in the source code and make it available in graphic form for the user.
In the literature, several studies suggest threshold values for Bad Smells extracted from the analysis of different software, but none of them can actually find all the problems, allowing the detection of several false positives. In addition to using complex methods to extract information, making future work difficult.
Conducting a study on the topic being researched. Conducting analysis of what was found in the surveys, to define the reason that leads to code anomalies, and through these studies, the metrics that led to the development of the tool were defined.
The tool obtained a good result in the analyzes performed, indicating that few false positives were generated, but it identified a considerable number of false negatives, which are perhaps more difficult for developers to identify, which indicates that the tool still needs more adjustments to increase your rates.
The methodology applied in this work aims to show in a simplified way how to solve a problem through a certain starting point (problem), passing through a means (tool) with the objective of reaching an end (results).
The main contributions are for, in general, a higher quality in the codes generated by the developers and consequently improvement in the systems and services provided that depend on various systems that are developed, as the source code will be made available for developers to use.

Key words: Bad Smell, Code Quality, Software Design and Architecture, Free Software Tools, Software Engineering

Resumo
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 disponibiliza-lo em forma de gráfico para o usuário.
Na literatura vários estudos sugerem valores limiares para Bad Smells extraídos da análise de diversos softwares, porém nenhum deles consegue de fato encontrar todos os problemas, permitindo a detecção de vários falsos positivos. Além de utilizarem métodos complexos para se extrair informações dificultando futuros trabalhos.
Realização de estudo sobre o tema que está sendo pesquisado Realização de análises sobre o que foi encontrado nas pesquisas, para que seja definido o motivo que leva a anomalias de código, e através desses estudos foram definidas as métricas que levaram ao desenvolvimento da ferramenta.
A ferramenta obteve um bom resultado na análises realizadas, 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.
A metodologia aplicada neste trabalho tem como objetivo mostrar de forma simplificado como realizar a resolução de um problema através de um determinado ponto inicial (problema), passando por um meio (ferramenta) com o objetivo de se chegar à um fim (resultados).
As principais contribuições estão para, de forma geral, uma maior qualidade nos códigos gerados pelos desenvolvedores e consequentemente melhora nos sistemas e serviços prestados que dependem de variados sistemas que são desenvolvidos, pois será disponibilizado o código fonte para que desenvolvedores possam fazer uso da ferramenta, e posteriormente melhorá-la.

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

JSniffer: A Tool for Automatic Identification of Bad Smells in JAVA.

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, Software Engineering.

JSniffer: Uma Ferramenta para Identificação Automática de *Bad Smells* em Projetos na linguagem JAVA

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, Engenharia de Software.

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.

O restante deste artigo está organizado da seguinte forma: na Seção 2 é apresentada a fundamentação teórica sobre os conceitos de Arquitetura de Software, Qualidade de Código e sobre alguns tipos de *Bad Smells*; na Seção 3 é apresentada a ferramenta desenvolvida para identificação de *Bad Smells* e suas principais funcionalidades; e, por fim, na Seção 4 são apresentadas as conclusões deste trabalho, a contribuição, como o mesmo foi avaliado e alguns trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

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).

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*.

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).

2.1 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 e as suas 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.

2.2 Bad Smells

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 trabalho, com o intuito de identificá-las e evitar problemas futuros.

2.2.1 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 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.

2.2.2 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).

2.2.3 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).

2.2.4 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.

3. A FERRAMENTA JSNIFFER

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.

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.

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/;

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.

3.1 Identificação das Estruturas de Código

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 1 apresenta a expressão regular que é utilizada para identificação de classes em códigos fontes.

```
/ \s*((public|protected|private)*\s*(static|abstract)*\s*
  (class|interface)+\s*(.*?)\s*)\{\s*((.\}?|\s?)*)\s*\}
```

Figura 1 – Expressão regular para identificação de classe.
Fonte: Elaborada pelo autor.

A FIGURA 2 apresenta a expressão regular que é utilizada para identificação de métodos e seus parâmetros em códigos fontes.

```
/ (public|protected|private)+\s*(static)*\s*([A-Za-
  záàâãéèêíïóôõöúçñÁÀÂÃÉÈÍÏÓÔÕÖÚÇ<>Ñ_]+)\s*([A-Za-z0-9_]+)\s*\((.*?)\)\s*
  ([A-Za-záàâãéèêíïóôõöúçñÁÀÂÃÉÈÍÏÓÔÕ;ÖÚÇ<>Ñ_ ]*)\s*\{*
```

Figura 2 – Expressão regular para identificação de métodos.
Fonte: Elaborada pelo autor.

A FIGURA 3 apresenta a expressão regular que é utilizada para identificação de métodos abstratos e seus parâmetros em códigos fontes.

```
/ (public|private|protected)+\s*(abstract)+\s*([a-zA-Z]+)\s*(.*?)\s*\
  ((.*?)\)\s*\;
```

Figura 3 – Expressão regular para identificação de métodos abstratos.
Fonte: Elaborada pelo autor.

A FIGURA 4 apresenta a expressão regular que é utilizada para identificação de métodos construtores e seus parâmetros em códigos fontes.

```
/ (public|private|protected)+\s*([A-Z]+[A-Za-z0-9]*)\s*\((.*?)\)\s*\{+
```

Figura 4 – Expressão regular para identificação de métodos construtores.
Fonte: Elaborada pelo autor.

A FIGURA 5 apresenta a expressão regular que é utilizada para identificação de corpo de métodos em códigos fontes.

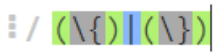

Figura 5 – Expressão regular para identificação de corpo de métodos.
Fonte: Elaborada pelo autor.

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.

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 analise 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 Funcionalidades da Ferramenta

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.

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 7) 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 13.

Ao acessar o sistema a primeira tela que aparece para o usuário é a tela Inicial, como mostra a FIGURA 6. Nesta tela podem-se visualizar dois botões superiores no lado esquerdo, possível de observar na FIGURA 7, 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.

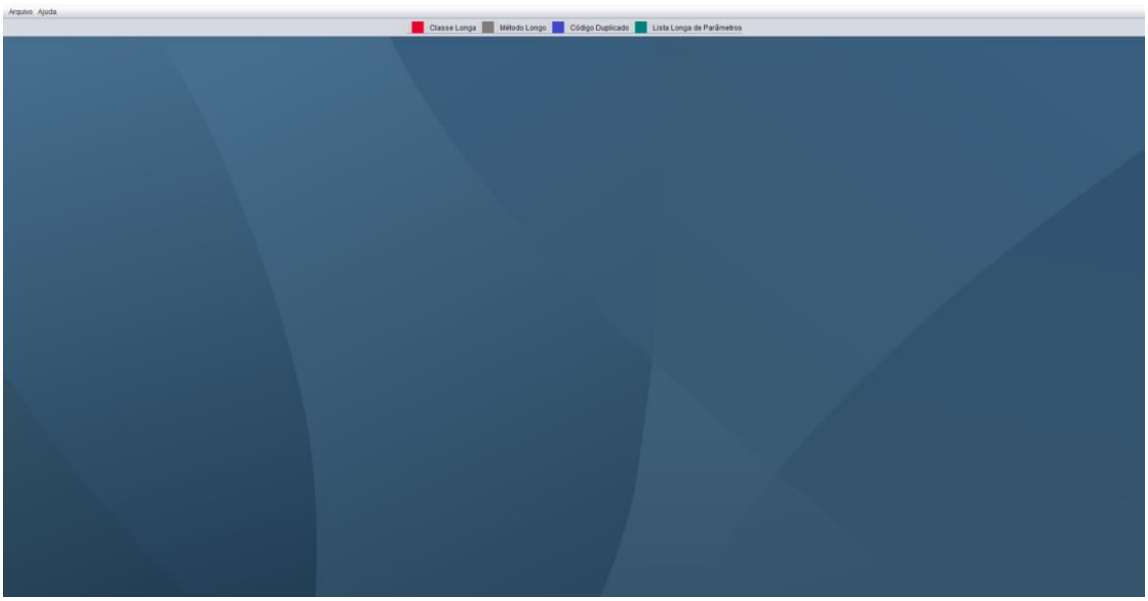

Figura 6 – Tela inicial da ferramenta.
Fonte: Elaborada pelo autor.

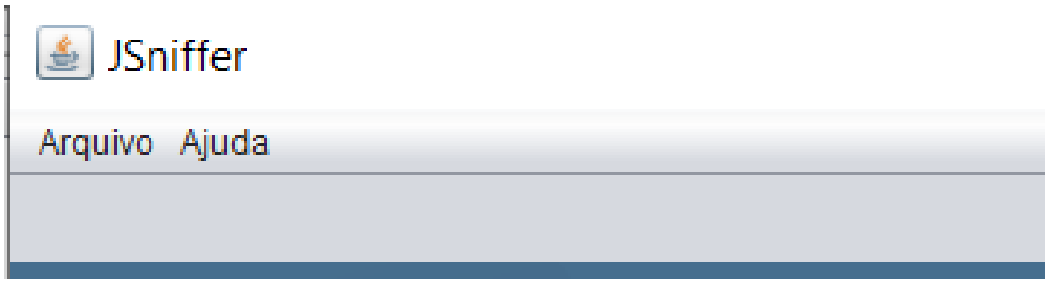

Figura 7 – Botões principais da tela inicial.
Fonte: Elaborada pelo autor.

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 8, e um submenu será exibido com as duas opções de obtenção de diretórios citadas anteriormente.

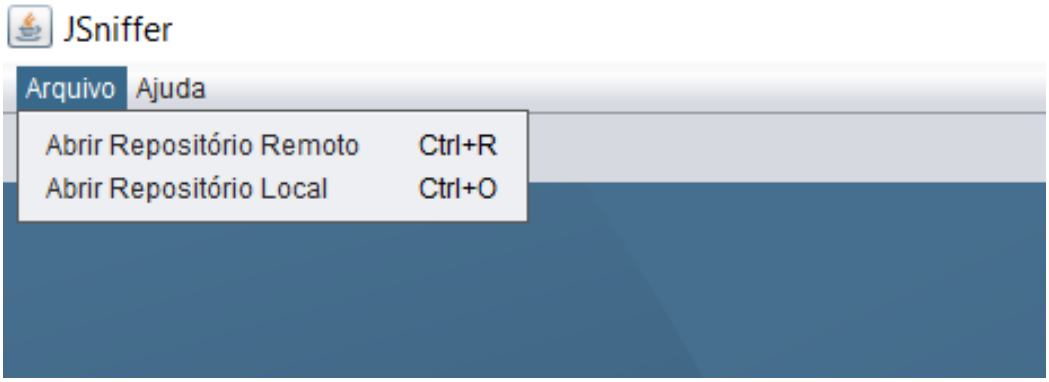

Figura 8 – Submenu para seleção de repositórios.
Fonte: Elaborada pelo autor.

Ao selecionar a primeira opção do submenu, a tela da FIGURA 9 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 credencias de autenticação para que o sistema possa ter acesso aos arquivos, como é apresentado na FIGURA 10 e 11, e logo após alguns segundos será possível visualizar o menu lateral com os diretórios do projeto na ferramenta, como é possível observar pela FIGURA 13. Ao selecionar a segunda opção do submenu, a tela da FIGURA 12 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”.

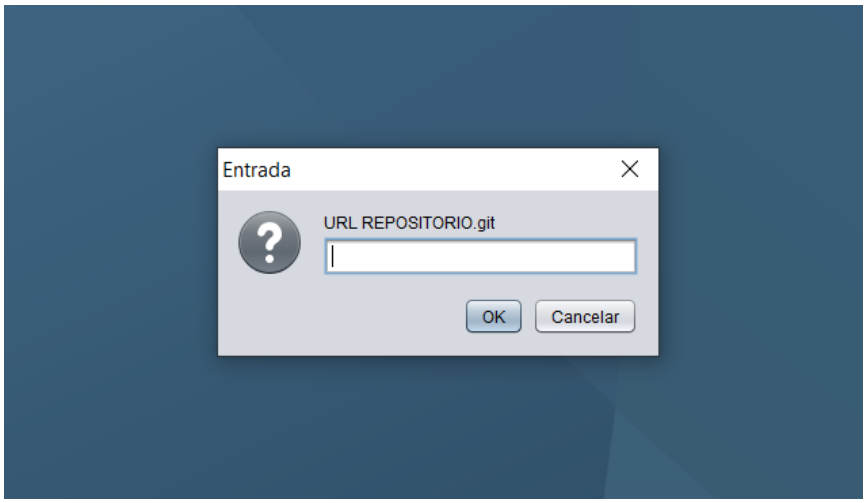

Figura 9 – Submenu para seleção de repositório remoto.
Fonte: Elaborada pelo autor.

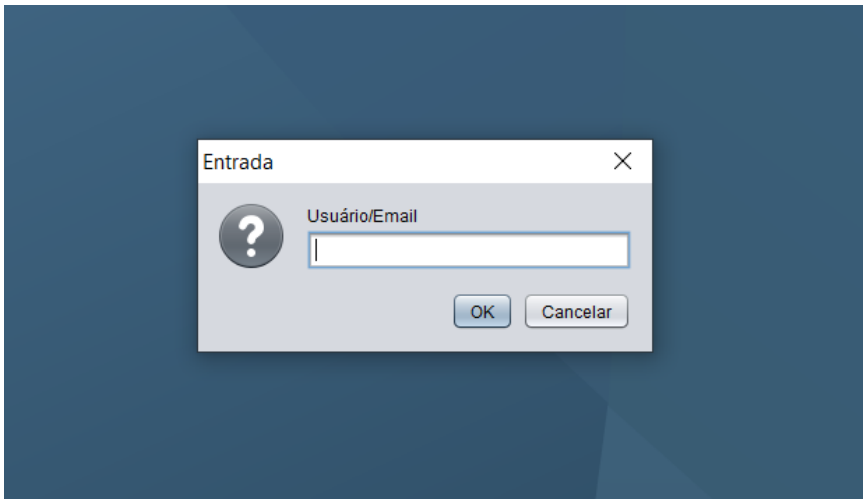

Figura 10 – Tela de inserção de usuário ou email para repositório privado.
Fonte: Elaborada pelo autor.

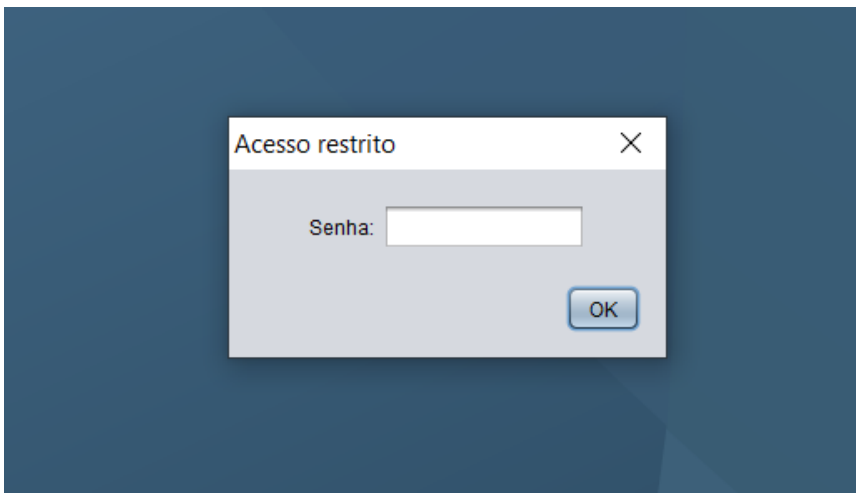

Figura 11 – Tela de inserção de senha para repositório remoto privado
Fonte: Elaborada pelo autor.

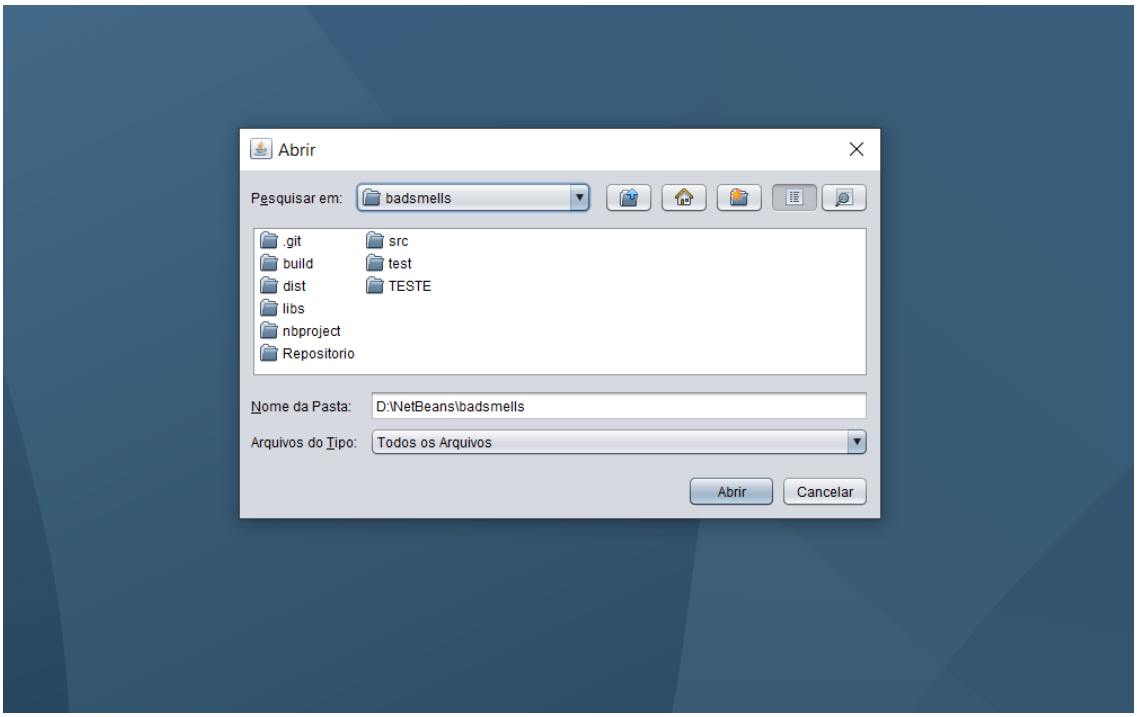

Figura 12 – Tela de Seleção de repositório local.
Fonte: Elaborada pelo autor.

A FIGURA 13 apresenta como a ferramenta está organizada após abrir um projeto. Na FIGURA 13A 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 13B 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.

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.

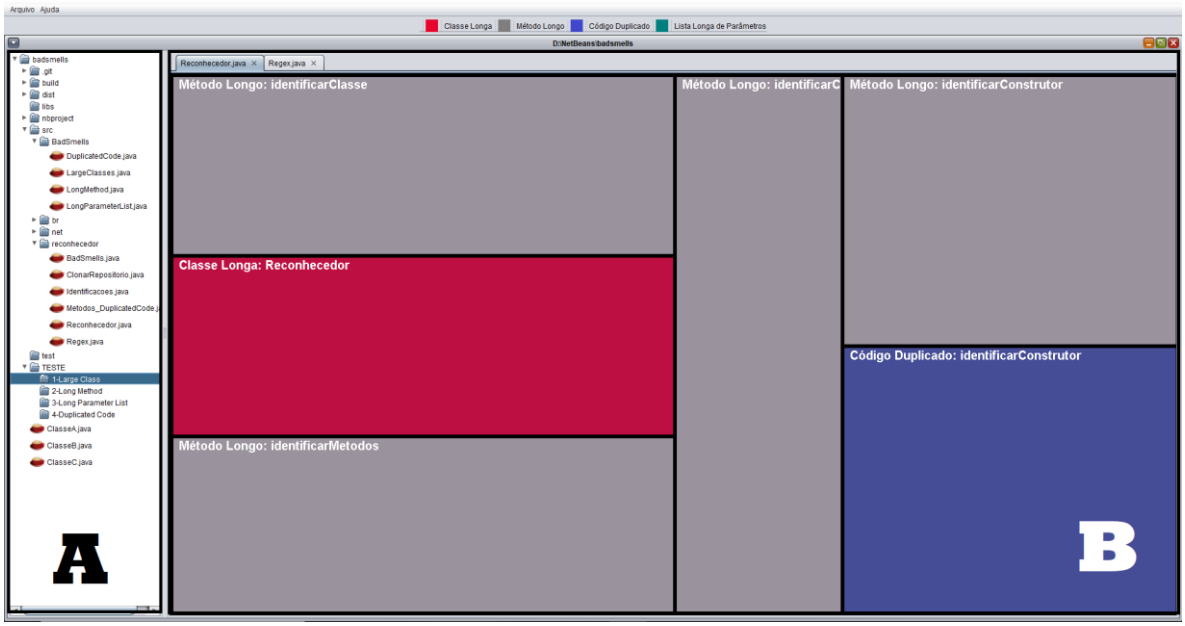

Figura 13 – Tela principal da ferramenta.
Fonte: Elaborada pelo autor.

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 14.

Figura 14 – Tela detalhes de *Bad Smell*.
Fonte: Elaborada pelo autor.

4. CONCLUSÕES

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 um especialista na área, que analisou todas as classes dos sistemas selecionados, e individualmente foi identificando *Bad Smells* para que seja colocado 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.

3.3 Análise no Sistema Compilador

Essa análise envolveu um sistema que funciona como uma espécie de compilador, passando pelos processos de analise 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.

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

Tabela 1 – Análise manual do sistema compilador.
Fonte: Elaborada pelo autor.

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

Tabela 2 – Análise automática do sistema compilador.
Fonte: Elaborada pelo autor.

Durante as análises 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.

3.4 Análise no Sistema JSniffer

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

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

Tabela 3 – Análise manual do sistema JSNIFFER.
Fonte: Elaborada pelo autor.

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

Tabela 4 – Análise automática do sistema JSNIFFER.
Fonte: Elaborada pelo autor.

Durante as análises 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.

3.5 Análise no Sistema Corretor

Essa análise 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.

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

Tabela 5 – Análise manual do sistema Corretor.
Fonte: Elaborada pelo autor.

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

Tabela 6 – Análise automática do sistema Corretor.
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.

3.6 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 15. Uma anomalia de código relevante é uma anomalia que também está presente na lista referência de anomalias de código (SANTOS, 2016).

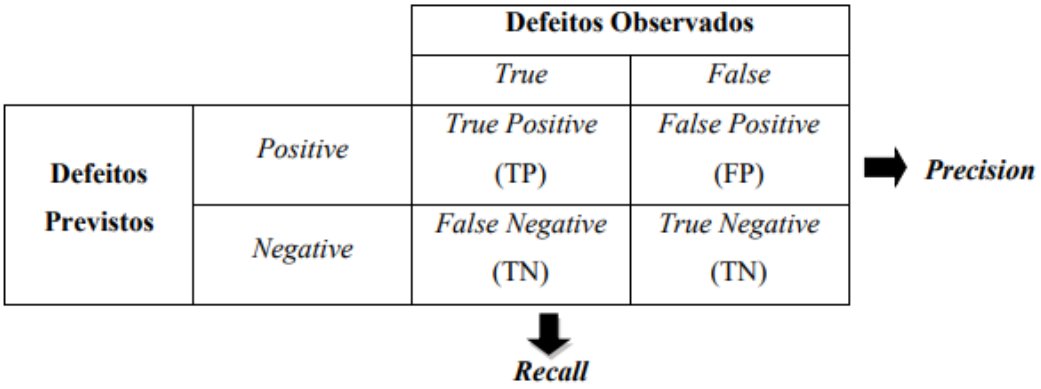

Figura 15 – *Recall* e *Precision* como modelo de avaliação.
Fonte: (ZIMMERMANN; PREMRAJ; ZELLER, 2007).

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:

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

O Resultado de *Recall* e *Precision* das análises pode ser visualizado na TABELA 7.

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

Tabela 7 – *Reccal* e *Precision* das análises.
Fonte: Elaborada pelo autor.

As análises realizadas obtiveram taxa média de *Recall* de 92,6%, 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 93,6% indicando que a ferramenta identificou menos anomalias do que deveria. De modo geral a ferramenta identificou valores próximos a 100% chegando à conclusão 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.

3.7 Conclusões

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, Classe longa, Método longo 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 BIBLIOGRÁFICAS

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.