Lucas Gonçalves Antunes Paiva

Linter Bad Smells

Belo Horizonte, Minas Gerais
2021

Lucas Gonçalves Antunes Paiva

Linter Bad Smells

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Mariza A. S. Bigonha
Coorientador: Bruno L. Sousa

Belo Horizonte, Minas Gerais
2021

Sumário

1 Introdução

No contexto de engenharia de software existem algumas ferramentas como: *Checker* [Checker 2021], *ESLint* [ESLint 2021] e *Prettier* [Prettier 2021], que auxiliam os desenvolvedores a aumentar a qualidade dos produtos implementados. No contexto *Javascript*, o *ESLint* tem-se destacado como sendo uma das aplicações mais utilizadas, uma vez que possui mais de 100 milhões de *downloads* mensais de acordo com a sua página [ESLint 2022]. O *ESLint* é uma ferramenta de análise de código estática para identificar erros de programação ou na estilização do código [Linter 2021].

As regras avaliadas pelo *linter* podem ser definidas pelo próprio engenheiro de software. Entretanto, existem vários *plugins* que predefinem os princípios que serão avaliados por essa ferramenta. Alguns exemplos são: *Babel* [Babel 2021], *React* [React 2021], *Hooks* [Hooks 2021], entre outros. Atualmente, grande parte dos *linters* existentes possuem um foco maior na parte da estilização e formatação do código. Contudo, eles ainda não possuem suporte para a análise de *code smells*. *Code smells*, também conhecidos como *bad smells*, são sintomas presentes no código fonte de um software que podem indicar a existência de um problema mais profundo e necessidade de refatoração do código. [Brown et al. 1998, Fowler e Beck 1999, Lanza e Marinescu 2006].

Sendo assim, este Trabalho Final do Curso, POC I e II, propôs e desenvolveu uma ferramenta, denominada *Linter Bad Smells*. Esta ferramenta é um *linter* para a linguagem *Javascript* cujos propósitos são: auxiliar desenvolvedores de software na análise do código fonte; facilitar a identificação de possíveis *bad smells* em aplicações desenvolvidas nessa linguagem; difundir os conceitos de *code smells* para a comunidade *Javascript*; e auxiliar os engenheiros de *software* a refatorar *code smells* identificados pelo *Linter Bad Smells* no código fonte analisado via a adição de um módulo de refatoração associado à documentação da ferramenta. Tais objetivos visam aumentar a qualidade do código implementado.

1.1 Objetivos

Os objetivos principais da ferramenta proposta e desenvolvida, *Linter Bad Smells* são:

1. Auxiliar os desenvolvedores de software a encontrar possíveis *code smells* de uma forma fácil e prática. Por meio desta ferramenta, é possível identificar e eliminar estas estruturas indesejadas de uma forma rápida, melhoranado a qualidade do código desenvolvido.

2. Difundir os conceitos de *code smells* para a comunidade *Javascript* por meio de *Linter Bad Smells*.

3. Apoiar a refatoração dos *code smells* identificados pela ferramenta. Diferentes formas de refatoração para um dado *bad smell* foram inclusas na documentação para que sirvam como uma espécie de *guideline* para o usuário e o guie durante o processo de remoção dos *bad smells*.

1.2 Objetivos Específicos

1. Documentar as métricas e *code smells* utilizados na ferramenta.
2. Documentar os métodos de refatoração dos *code smells* que podem ser encontrados pela ferramenta *Linter Bad Smells*
3. Disponibilizar a documentação sobre refatoração do *plugin* para que a comunidade tenha acesso à informações de como solucionar os *bad smells* suportados pela ferramenta
4. Gerar um relatório sobre a utilização do *Linter Bad Smells*
5. Escrever o relatório final do POC I e POC II
6. Produção de vídeos para mostrar apresentação da ferramenta tanto para o POC-I quanto para o POC-II

1.3 Contribuições

A partir do desenvolvimento desse trabalho de pesquisa, este documento de monografia enumera todas as contribuições geradas neste POC.

1. Documentação das métricas e *code smells* utilizados na ferramenta.
2. Disponibilização do *Linter Bad Smells* como um *plugin* para a comunidade científica.
3. Disponibilização da documentação sobre a refatoração da ferramenta *Linter Bad Smells* na página do *plugin* para a comunidade científica.
4. Geração de um relatório final sobre o tema da monografia.
5. Atualização do *Linter Bad Smells* para suportar *links* dentro das mensagens de erro para levar o usuário para a documentação de refatoração.
6. Geração de vídeos explicando o que é a ferramenta e como utilzá-la.
7. Escrita e submissão de um artigo sobre a arquitetura e utilização do *Linter Bad Smells*.

1.4 Organização do Texto

O restante desta monografia foi organizada da seguinte forma.

Capítulo 2 apresenta o referencial teórico utilizado como base para a realização do trabalho. Descreve todos os conceitos necessários para o entendimento completo da ferramenta implementada, como *code smells*, métricas de *software* e, também, as estratégias de detecção de *code smells*.

Capítulo 3 descreve alguns trabalhos que também buscam analisar programas implementados em *JavaScript* para identificar possíveis padrões que fazem com que a qualidade do código desenvolvido seja reduzida. Além disso, foi mostrado as principais diferenças entre cada um dos trabalhos relacionados e o atual trabalho.

Capítulo 4 apresenta o projeto de desenvolvimento do *Linter Bad Smells*. Inicialmente foi realizado uma análise dos códigos fontes de alguns *linters* com a finalidade de entender como é feito esse processo de criação da ferramenta. Após este estudo inicial, efetuou-se o desenvolvimento da ferramenta proposta.

Capítulo 5 descreve a ferramenta implementada, mostrando seus requisitos, sua estrutura. Além disso, é apresentado um exemplo de funcionamento da ferramenta.

Por fim, o Capítulo 6 deste relatório conclui a documentação do Trabalho de Fim de Curso, POC, realizado.

2 Referencial Teórico

Este capítulo aborda os principais conceitos empregados neste trabalho. São eles: *code smells* (Seção 2.1), refatoração (Seção 2.2), métricas de software (Seção 2.3) e estratégias de detecção (Seção 2.4).

2.1 *Code Smells*

A ferramenta proposta neste trabalho, *Linter Bad Smells*, baseia-se no conceito de *code smells*, isto é, características presente no código fonte dos sistemas que podem indicar possíveis problemas de projeto.

Fowler e Beck [Fowler e Beck 1999] descrevem em seu livro uma série de *code smells* que podem impactar o código fonte de um projeto, bem como as características principais que auxiliam na sua identificação destes sintomas.

Alguns do principais *code smells* apresentados por Fowler e Beck [Fowler e Beck 1999] são:

- *Long method*: refere-se à métodos complexos, extensos e com várias responsabilidades. Métodos com este sintoma implementam uma grande quantidade de funcionalidade, tornando-se complexos e difíceis de serem entendidos.
- *Long parameter list*: refere-se à métodos com uma longa lista de parâmetros. O ideal é que as listas de parâmetros sejam curtas, pois quanto mais extensas se tornam, mais difíceis de ler e de usar.
- *Duplicated Code*: ocorre quando existem fragmentos de códigos idênticos presentes em mais um local no código fonte de um software.
- *Feature Envy*: ocorre quando métodos de uma classe estão mais interessados em métodos de outras classes e os utilizam em excesso. Quando esse sintoma ocorre é sinal de que existem métodos que deveriam ser removidos da classe.
- *Switch statements*: consiste em um uso excessivo de comandos *switches* no código fonte que acaba se propagando e gerando duplicações.
- *Shotgun Surgery*: são classes que ao serem alteradas impactam diversas outras classes do sistema, propagando mudanças para as demais.
- *Comments*: ocorre quando comentários são usados excessivamente em um código fonte com o intuito de tentar melhorar os nomes dos atributos e métodos que não estão suficientemente expressivos.

Além dos *smells* descritos por Fowler e Beck [Fowler e Beck 1999], Fard e Mesbah [Fard e Mesbah 2013] definem algumas anomalias mais específicas para o contexto de sistemas desenvolvidos na linguagem *JavaScript*. Dentre eles, as principais são:

- *Nested callbacks*: um *callback* consiste em uma função que é passada como argumento para uma outra função. Quando esta ação ocorre de forma aninhada e excessiva, é definido como *Nested callbacks*.
- *Empty catch*: refere-se à uma chamada assíncrona que não possui tratamento de erro.

2.2 Refatoração

Refatoração é o processo de alterar um sistema de software de tal forma que não altere a comportamento do código e ainda melhora sua estrutura interna [Fowler]. Tal ação é uma maneira de reestruturar o código visando minimizar as chances de introduzir *bugs*. Dessa forma, quando refatora-se algum código, o objetivo é melhorar a sua implementação.

O termo refatoração está diretamente ligado à *code smells*, uma vez que ao identificar algum desses *smells* deve-se encontrar alguma forma de remover essa estrutura do código. Assim, pode-se utilizar algumas das técnicas de refatoração para atingir esse objetivo. Alguns exemplos de técnicas de refatoração mais populares na literatura são:

- *Extract Method*: esta técnica é utilizada quando parte do código pode ser agrupado. Ou seja, o desenvolvedor pode mover o código para um novo método ou função e substituir o código antigo pela chamada do método.
- *Inline Method*: esta técnica é utilizada quando o corpo de um método é muito óbvio e simples. Neste caso, é possível copiar o código do método para onde ele é chamado, e em seguida remover o método.
- *Extract Variable*: esta técnica é utilizada quando uma expressão é muito complexa, portanto difícil de entender. Uma solução pode ser a criação de uma variável para definir o significado do resultado da expressão.
- *Move Method*: esta técnica é utilizada quando um método da classe A é usado mais na classe B. Neste caso, remove-se o método da classe A e adiciona o na classe B, uma vez que a classe A utiliza menos esse método.
- *Move Field*: esta técnica é utilizada quando um atributo da classe A é usado mais na classe B. Neste caso pode-se mover o atributo para a classe B, uma vez que a classe A utiliza menos esse atributo.

Grande parte dos métodos de refatoração apresentados podem ser utilizados independentemente da linguagem em que o código está implementado. Isto ocorre porque eles são apenas estruturas gerais que os desenvolvedores podem aplicar em seus projetos.

2.3 Métricas de Software

De acordo com [Pressman 2006], métrica consiste em um padrão de medição que é utilizado para avaliar uma dada propriedade interna de um sistema de software. Por meio de métricas é possível identificar desvios de código que tornam o software mais complexo e que podem ocasionar falhas. Para auxiliar na detecção dessas estruturas complexas, as métricas de software são utilizadas na composição de expressões quantificáveis que são definidas na literatura como estratégias de detecção [Marinescu 2002, Filó, Bigonha e Ferreira 2015, Souza 2016].

Diversos estudos têm definido métricas de software na literatura. Dentre as diversas métricas existentes, destacam-se o conjunto CK [Chidamber e Kemerer 1994] e as métricas definidas por [Lorenz e Kidd 1994]. Estes dois conjuntos de métricas são definidos de forma geral a seguir.

2.3.1 Métricas CK

Analisam o software orientado por objetos no nível de classe [Chidamber e Kemerer 1994]. Os aspectos avaliados por essas métricas são: acoplamento, herança e coesão. Este conjunto é composto por um total de seis métricas: métodos ponderados por classe (WMC), profundidade da árvore de herança (DIT), número de filhos (NOC), acoplamento entre classes de objetos (CBO), resposta de classe (RFC) e ausência de coesão em métodos (LCOM).

2.3.2 Métricas de Lorenz e Kidd

São definidas para classes e são divididas em quatro categorias principais: tamanho, baseada na contagem de atributos e operações da classe individualmente e na média para um valor do sistema como um todo; herança avalia como as operações são reutilizadas via hierarquia de classe; atributos internos avalia coesão; e, externos - o acoplamento e a reutilização - [Chidamber e Kemerer 1994, Pressman e Maxim 2016]. As principais métricas são: número de conexões aferentes (NCA), número de métodos públicos (NMP), número de atributos públicos (NAP), número de atributos (NOF), número de métodos (NOM), número de métodos sobrescritos (NORM), número de filhos (NSC), número de atributos estáticos (NSF), número de métodos estáticos (NSM) e número de parâmetros (PAR).

Dessa forma, *Linter Bad Smells* será capaz de analisar vários *code smells* na base de código do usuário e sugerir possíveis formas de refatorar a implementação, com a finalidade de aumentar a qualidade do produto desenvolvido.

2.4 Estratégias de Detecção para *Code Smells*

.

Estratégias de detecção podem ser entendidas como "uma expressão quantificável de uma regra", definida para "verificar se fragmentos do código-fonte estão em conformidade com essa regra definida"[Marinescu 2002] [Bertrán 2009]. Dessa forma, os principais componentes utilizados para a definir as estratégias de detecção de *code smells* são: mecanismos de filtragem de métricas e mecanismos de composição. O mecanismo de filtragem possibilita a redução do conjunto inical de dados, de mo que possa verificar uma característica especial de fragmentos que têm suas métricas capturadas. Os flitros permitem definir limites de valores para determinados aspectos. Por sua vez, os mecanismos de composição permitem combinar diferentes filtros, baseados em métricas de software distintas, por meio de conectivos lógicos, possibilitando, assim, a mescla de métricas para obter informações relevantes [Marinescu 2002] [Bertrán 2009] [Souza 2016].

2.5 Considerações Finais

Neste capítulo foram apresentados os principais conceitos utilizados neste trabalho de pesquisa, como *code smells*, métricas de software e estratégias de detecção que são necessários para um melhor entendimento do projeto desenvolvido.

O Capítulo 3 apresenta alguns estudos relacionados, descrevendo-os e destacando as principais características que os diferem do presente trabalho de pesquisa.

3 Trabalhos Relacionados

Existem alguns trabalhos que investigam como os *code smells* ou más práticas de programação impactam no contexto de desenvolvimento de software que são implementados em *JavaScript*. Dentre eles, destacam-se dois trabalhos: Nguyen et al. [Nguyen et al. 2012] e Gong et al. [Gong et al. 2015]. Tais trabalhos consistem, respectivamente, em um estudo empírico sobre *code smells* em projetos *JavaScript*, e uma ferramenta que permite análise de más práticas de desenvolvimento.

Nguyen et al. [Nguyen et al. 2012] buscam entender como os *code smels* impactam software que são desenvolvidos em *JavaScript*. Este estudo conclui que arquivos que não possuem *code smels* tem uma chance menor de não possuir falhas. Isso mostra que um código livre de *bad smells* reduz significamente a quantidade de erros. Embora Nguyen et al. [Nguyen et al. 2012] mostrem o impacto de code smells na qualidade da estrutura de software, eles não criam nenhuma ferramenta que permita desenvolvedores identificarem essas estruturas no código fonte. Desta forma, a principal diferença do presente trabalho para o trabalho conduzido por Nguyen et al. [Nguyen et al. 2012] é o fato de uma ferramenta de detecção de *code smells* ser proposta e construída.

Gong et al. [Gong et al. 2015] propõem uma ferramenta, *DLint*, para verificar se existem más práticas no código dinamicamente. Os autores identificaram alguns erros, cujo os quais ferramentas que realizam análise estática de código não conseguem detectar. *DLint* é capaz de encontrar uma série de más práticas de programação que, por exemplo, o *ESlint* não consegue. Essa ferramenta difere de *Linter Bad Smells* por ter uma abordagem com foco na análise dinâmica do código e por não buscar necessariamente por *code smells* durante sua análise do programa implementado.

3.1 Considerações Finais

Neste capítulo foram apresentados os principais trabalhos relacionados ao tema do Trabalho de Fim de Curso (POC). Foram descritos como esses trabalhos o abordam e suas principais diferenças com o trabalho de pesquisa desenvolvido.

No próximo capítulo será descrito como foi projetada e realizada a implementação da ferramenta *Linter Bad Smells*, a metodologia usada, suas principais características, bem como os resultados obtidos.

4 *Linter Bad Smells*

O *Linter Bad Smells* é um *linter* que realiza a análise estática de projetos implementados em *JavaScript* procurando por *code smells*, com a finalidade de aumentar a qualidade do código desenvolvido pelos engenheiros de software.

4.1 Metodologia

A metodologia utilizada para o desenvolvimento deste Trabalho de Fim de Curso, POC, foi dividida em nove etapas, as quatro primeiras estão relacionadas com o trabalho desenvolvido para o POC-I. Enquanto as cinco etapas restantes fazem parte do trabalho desenvolvido para o POC-II.

Etapa 1. *Definição dos code smells e métricas abordadas no trabalho.* Para isso foi realizada uma busca na literatura visando identificar *bad smells* que são analisados em aplicações JavaScript, bem como, estratégias de detecção capazes de identificar a presença destas anomalias neste contexto específico.

Etapa 2. *Estudo sobre a criação de linters.* Realizou-se uma análise do código-fonte de vários *linters* para entender quais conceitos seriam necessários dominar para realizar a implementação proposta.

Etapa 3. *Implementação da ferramenta Linter Bad Smells.* Nessa etapa, foi realizada a construção da ferramenta a partir dos conceitos estudados e definidos nas etapas anteriores.

Etapa 4. *Avaliação do funcionamento do Linter Bad Smells.* Nessa etapa, foi realizado a análise do funcionamento da ferramenta para verificar se o que foi proposto está de acordo com o resultado obtido.

Etapa 5. *Estudo e identificação de técnicas de refatoração existentes.* A partir dos *code smells* que a ferramenta suporta foi feito um estudo de quais técnicas de refatoração poderiam ser utilizadas para remover os *bad smells*.

Etapa 6. *Definição das refatorações a serem incluídas neste trabalho.* Após o estudo e identificação dos tipos de refatoração, definiu-se as técnicas de refatoração mais adequadas para solucionar os *code smells* suportados pela ferramenta.

Etapa 7. *Planejamento do funcionamento da recomendação de refatoração.* A partir da definição das técnicas de refatoração foi escolhido a maneira como a ferramenta do *Linter Bad Smells* iria disponibilizar essa solução para seus usuários. A forma

escolhida foi via uma documentação na página da ferramenta no `npm` e no `GitHub`. Além disso, decidiu-se que a ferramenta iria disponibilizar um *link* de acesso por meio do erro mostrado na tela do usuário do *eslint-plugin-bad-smells*.

Etapa 8. *Implementação da recomendação de refatoração.* A partir do planejamento do funcionamento da recomendação de refatoração foi realizado um estudo da viabilidade de adicionar o *link* para a refatoração dos *code smells* dentro do erro, ou, um aviso mostrado dentro do próprio *Eslint*. Em seguida, como a ferramenta do *Eslint* possui esse suporte para *links* foi realizada a implementação para todos os *code smells* que são suportados pela ferramenta desenvolvida.

Etapa 9. *Escrita da documentação com as refatorações definidas para cada bad smell.* Por fim, dentro da documentação da ferramenta foi realizada uma descrição de descreveu-se como refatorar cada um dos *code smells* suportados pela ferramenta. A documentação descreve passo a passo como realizar a sua implementação com exemplos práticos, bem como as vantagens e desvangens de realizar a refatoração, uma vez que não existe solução perfeita nesses casos.

4.2 Resultados

O resultado do trabalho desenvolvido é uma ferramenta, *Linter Bad Smells*, que está disponível no gerenciador de pacotes *npm* chamada de *eslint-pluguin-bad-smells* [1] [2]. A página do pacote possui documentação de como realizar a instalação da ferramenta, bem como possíveis refatorações para cada um dos *code smells* que *Linter Bad Smells* suporta.

A partir da instalação da ferramenta o usuário será capaz de habilitar cinco *code smells* para a realização da análise estática do código, que são:

- *Long method*: refere-se à métodos complexos, extensos e com várias responsabilidades. Métodos com este sintoma implementam uma grande quantidade de funcionalidade, tornando-se complexos e difíceis de serem entendidos.

- *Long parameter list*: refere-se à métodos com uma longa lista de parâmetros. O ideal é que as listas de parâmetros sejam curtas, pois quanto mais extensas se tornam, mais difíceis de ler e de usar.

- *Long message chain*: um *callback* consiste em uma função que é passada como argumento para uma outra função. Quando esta ação ocorre de forma aninhada e excessiva, é definido como *Nested callback*, um tipo de anomalia.

[1] `https://github.com/lucas-paiva98/eslint-plugin-bad-smells`
[2] `https://www.npmjs.com/package/eslint-plugin-bad-smells`

- *Nested callbacks*: refere-se a um método que define dentro do seu escopo um novo método. E nesse novo método criado é implementado mais outro método e, assim, sucessivamente aninhando diversas funções.

- *Empty catch*: refere-se à uma chamada assíncrona que não possui tratamento de erro.

Ademais, a documentação aborda para cada um dos *bad smells* uma forma de refatoração para solucionar os problemas encontrados, usando exemplos práticos para mostrar como é possível realizar mudança no código.

4.3 Considerações Finais

Este capítulo discutiu o projeto, o desenvolvimento de *Linter Bad Smells* desde o seu início no POC 1 e as principais ações para a conclusão dessa segunda etapa do Projeto de Final de Curso (POC 2). Mostrou-se também os resultados obtidos com o desenvolvimento desse trabalho.

O Capítulo 5 a seguir, descreve a arquitetura de *Linter Bad Smells* incluindo os requisitos e a estrutura da ferramenta implementada. Ademais, mostra que seu funcionamento está de acordo com o planejado, via exemplos.

5 Arquitetura de *Linter Bad Smells*

Neste capítulo é descrita a arquitetura de *Linter Bad Smells*. Para descrevê-la, inicialmente foram apresentados seus requisitos (Seção 5.1). Seção 5.2 mostra sua estrutura. Seção 5.3 exibe um exemplo de uso da ferramenta proposta e implementada. Seção 5.4 finaliza esse capítulo.

5.1 Requisitos

A ferramenta *Linter Bad Smells* possui os seguintes requisitos:

Importação do código fonte.
O código implementado pelos desenvolvedores é passado para a biblioteca do *ESLint* que transforma esse código em uma Árvore de Sintaxe Abstrata (AST) anotada. A AST consiste na representação abstrata das estruturas sintática e semântica de um código fonte.

Extração da árvore de sintaxe abstrata do código fonte.
A partir da extração da árvore sintática pela biblioteca do *ESLint*, é possivel identificar todas as propriedades que foram implementadas como número de linhas do código, número de parâmetros, nome da função, entre outras características do código.

Extração de métricas.
Após a geração da árvore de sintaxe abstrata anotada é possível identificar todas as características que definem determinado trecho de código.

Identificação dos *Bad Smells*.
Como a árvore de sintaxe abstrata anotada permite acessar às características do código, é possível identificar se existem *bad smells* nesse trecho de código analisando suas características. Por exemplo, um *Long Method* terá mais de 50 linhas de código e essa informação da quantidade de linhas do código é retornada a partir da árvore de sintaxe abstrata anotada, assim, sendo possìvel identificar os *code smells*.

Detecção de *Bad Smells* com retorno para o usuário.
Após a identificação dos *code smells* a ferramenta *Linter Bad Smells* produz um retorno em forma de erro ou *warning* dentro do editor de texto para o usuário, indicando se existe ou não um *code smell* no trecho de código implementado.

Recomendação de refatoração para remoção dos *Bad Smells* identificados.
A ferramenta também produz como resultado uma documentação com sugestões de

refatoração para cada *code smell* para que o usuário consiga solucioná-los.

Disponibilização de *links* dentro da ferramenta para direcionar para a documentação

A ferramenta introduz *links* dentro das mensagens erros ou avisos que aparecem no editor de texto. Dessa forma, os engenheiros de *software* conseguem acessar a documentação de maneira mais rápida e fácil para a parte de refatoração do *code smell* identificado pela *Linter Bad Smells*.

5.2 Estrutura da Ferramenta

A estrutura de pastas da ferramenta, ilustrada na Figura 1, está definida da seguinte maneira.

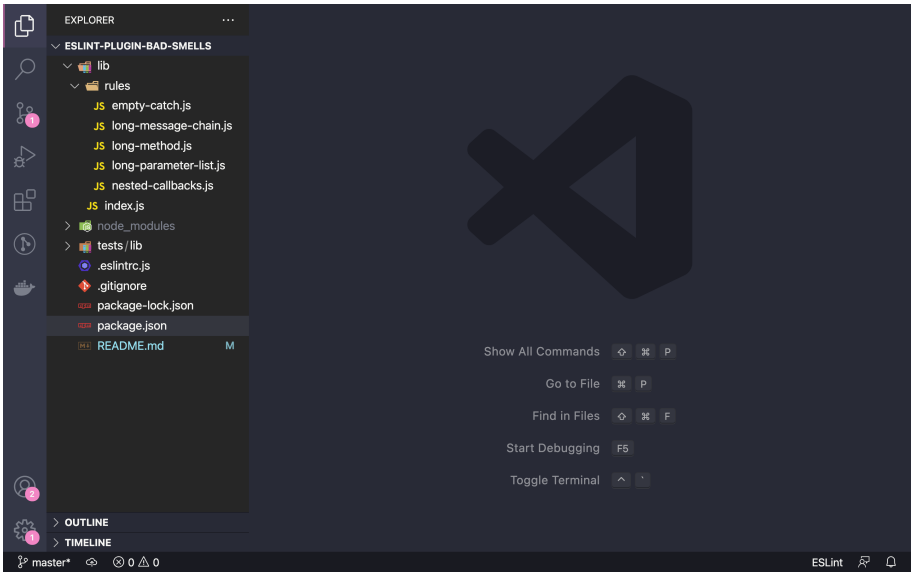

Figura 1 – Estrutura de pastas da *Linter Bad Smells*.

- Na pasta *lib/rules* são definidas as regras para encontrar cada *bad smell*.
- Na pasta *node modules* ficam todas as dependências instaladas para o projeto funcionar, como o *ESLint*.
- Na pasta *tests/lib* são escritos os testes das regras criadas para identificar os *smells*.
- No arquivo *README.md* encontra-se a documentação de *Linter Bad Smells*, enquanto os outros arquivos presentes na raiz do projeto apresentam as configurações do *ESLint*, do *GitHub* e dos pacotes que são instalados na ferramenta.

5.3 Exemplos de uso

A ferramenta funciona da seguinte maneira, o desenvolvedor de software ao implementar qualquer trecho de código recebe como retorno se existe ou não um *bad smell* no seu editor de texto. Figura 2 e Figura 3 ilustram esse comportamento facilitando a sua compreensão.

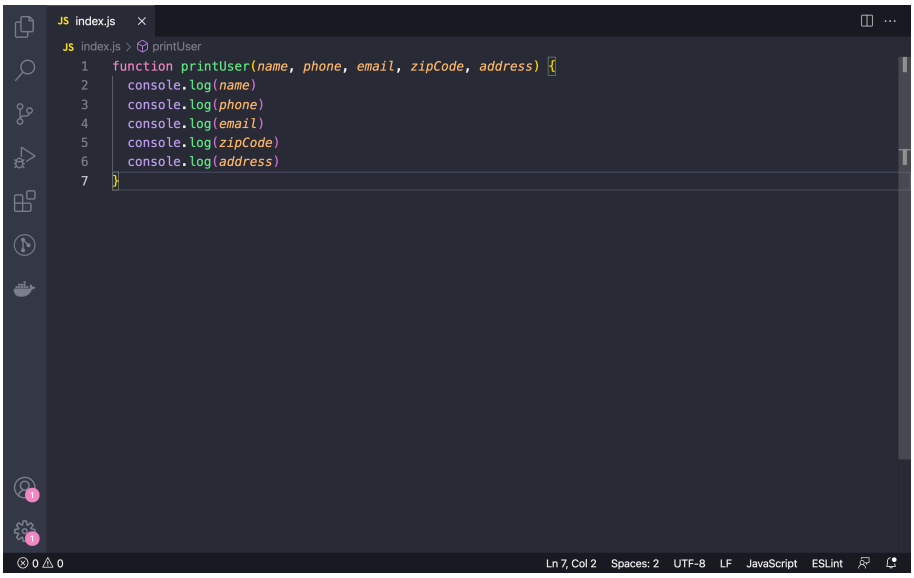

Figura 2 – Exemplo de código sem *code smell*

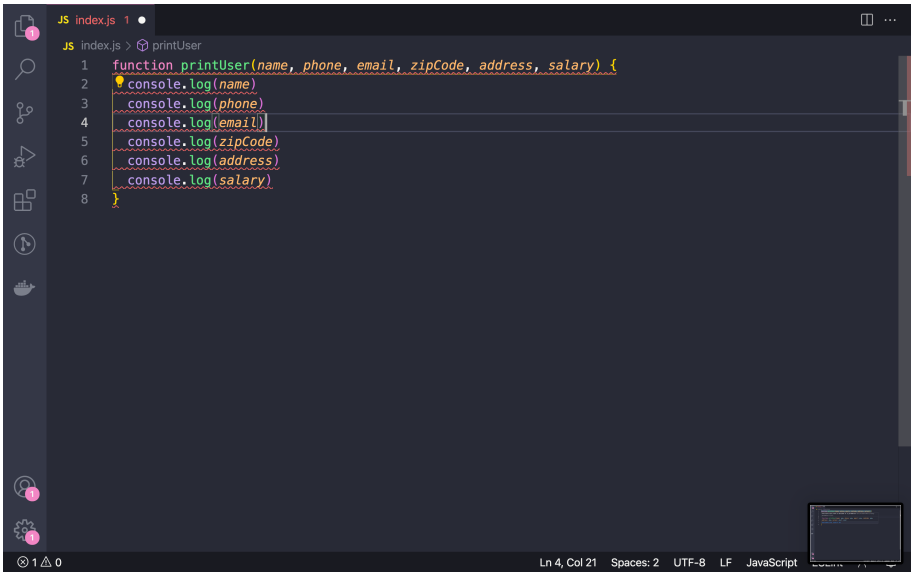

Figura 3 – Exemplo de código com *code smell*

A Figura 4, mostra um exemplo de notificação encaminhada ao usuário.

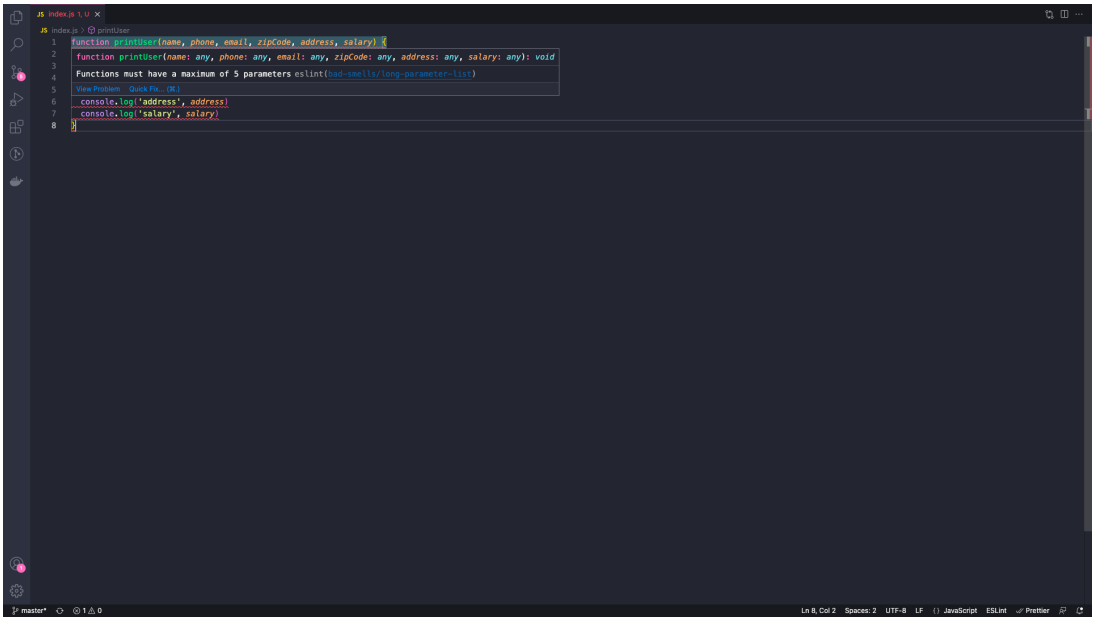

Figura 4 – Exemplo de notificação do usuário

A Figura 5 exibe a página de documentação após o usuário clicar no link do erro.

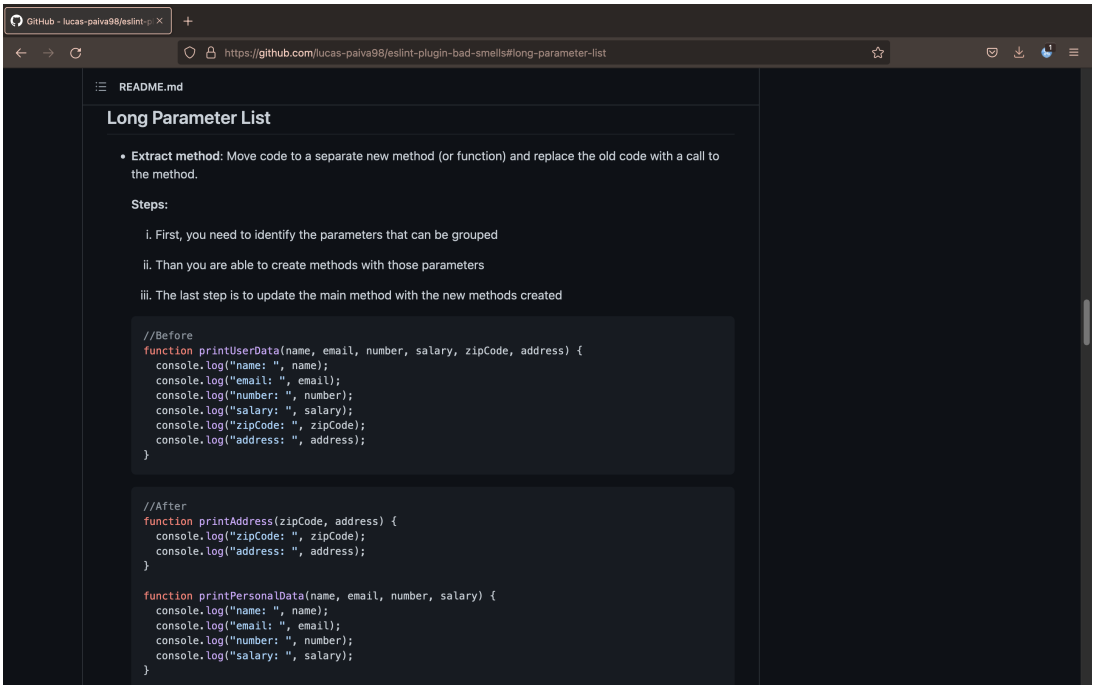

Figura 5 – Exemplo de documentação

5.4 Considerações Finais

Esse capítulo mostrou a arquitetura de *Linter Bad Smells*, como ela está estruturada, bem como suas principais funcionalidades. Foram apresentados também exemplos práticos que exibem, respectivamente, um exemplo de código sem a presença de *code smell* e um exemplo onde a ferramenta encontra um *code smell*. Além disso, foi mostrado uma tela com a estrutura das pastas de *Linter Bad Smells.*

Capítulo 6 conclui este trabalho.

6 Conclusão

O Trabalho de Fim de Curso (POC), relatado nesta monografia mostrou todo o processo de desenvolvimento da ferramenta *Linter Bad Smells*, destacando os melhoramentos propostos no POC 2 para essa ferramenta de *linter*.

Linter Bad Smells faz uma análise de projetos implementados em *JavaScript* buscando por cinco *code smells*: *Long Method*, *Long Parameter List*, *Long Message Chain*, *Nested Callbacks* e *Empty Catch*.

No projeto do POC 2 foi realizada a adição de *links* nos erros que aparecem no editor de texto que levam o usuário para a página da documentação da ferramenta e para a refatoração do erro indicado pelo *linter*.

O objetivo principal desta ferramenta é aumentar a qualidade de código desenvolvido pelos engenheiros de software. A partir dela, os engenheiros de software terão contato com conceitos de qualidade de código que ajudará a difundir esses princípios na comunidade de *JavaScript* e, assim, auxiliá-los a aumentar a excelência dos projetos de software desenvolvidos.

Referências

[Babel 2021]BABEL. 2021. Disponível em: <https://github.com/babel/eslint-plugin-babel>.

[Bertrán 2009]BERTRÁN, I. M. Avaliação da qualidade de software com base em modelos uml. Tese (Doutorado) — Dissertação da PUC-RJ. Rio de Janeiro, RJ, Brasil. Pontifícia Universidade . . . , 2009.

[Brown et al. 1998]BROWN, W. H. et al. AntiPatterns: refactoring software, architectures, and projects in crisis. [S.l.]: John Wiley & Sons, Inc., 1998.

[Checker 2021]CHECKER, C. S. 2021. Disponível em: <https://github.com/streetsidesoftware/vscode-spell-checker/>.

[Chidamber e Kemerer 1994]CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. IEEE Transactions on software engineering, IEEE, v. 20, n. 6, p. 476–493, 1994.

[ESLint 2021]ESLINT. 2021. Disponível em: <https://eslint.org/>.

[ESLint 2022]ESLINT. 2022. Disponível em: <https://www.npmjs.com/package/eslint>.

[Fard e Mesbah 2013]FARD, A. M.; MESBAH, A. Jsnose: Detecting javascript code smells. In: IEEE. 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). [S.l.], 2013. p. 116–125.

[Filó, Bigonha e Ferreira 2015]FILÓ, T. G. S.; BIGONHA, M. A. S.; FERREIRA, K. A. M. A catalogue of thresholds for object-oriented software metrics. In: International Conference on Advances and Trends in Software Engineering SOFTENG. Proceedings of International Conference on Advances and Trends in Software Engineering. [S.l.: s.n.], 2015. p. 1.

[Fowler]FOWLER, K. B. M. [S.l.: s.n.].

[Fowler e Beck 1999]FOWLER, M.; BECK, K. Refactoring: Improving the Design of Existing Code. [S.l.]: Addison-Wesley, 1999.

[Gong et al. 2015]GONG, L. et al. Dlint: Dynamically checking bad coding practices in javascript. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. [S.l.: s.n.], 2015. p. 94–105.

[Hooks 2021]HOOKS, R. 2021. Disponível em: <https://github.com/facebook/react/tree/master/pack
plugin-react-hooks>.

[Lanza e Marinescu 2006]LANZA, M.; MARINESCU, R. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. [S.l.]: Springer Science & Business Media, 2006.

[Linter 2021]LINTER, S. 2021. Disponível em: <http://www.sublimelinter.com/en/v3.10.10/about.htr

[Lorenz e Kidd 1994]LORENZ, M.; KIDD, J. Object-oriented software metrics: a practical guide. [S.l.]: Prentice-Hall, Inc., 1994.

[Marinescu 2002]MARINESCU, R. Em Measurement and Quality in Object-Oriented Design. Tese (Doutorado) — University of Timisoara, 2002.

[Nguyen et al. 2012]NGUYEN, H. V. et al. Detection of embedded code smells in dynamic web applications. In: IEEE. 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. [S.l.], 2012. p. 282–285.

[Pressman e Maxim 2016]PRESSMAN, R.; MAXIM, B. Engenharia de Software-8ª Edição. [S.l.]: McGraw Hill Brasil, 2016.

[Pressman 2006]PRESSMAN, R. S. Engenharia de software. 6ª edição. ed. [S.l.]: MacGraw Hill, Rio de Janeiro, 2006.

[Prettier 2021]PRETTIER. 2021. Disponível em: <https://prettier.io/>.

[React 2021]REACT. 2021. Disponível em: <https://www.npmjs.com/package/eslint-plugin-react>.

[Souza 2016]SOUZA, P. A utilidade dos valores referência de métricas na avaliação da qualidade de softwares orientados por objeto. In: Anais do XIV Workshop de Teses e Dissertações em Qualidade de Software (WTDQS 2016). [S.l.: s.n.], 2016. p. 9.