Luís Henrique Soares Monteiro

Um plugin para a ferramenta ESLint que verifica Test Smells

Belo Horizonte, Minas Gerais
2025

Luís Henrique Soares Monteiro

Um plugin para a ferramenta ESLint que verifica Test Smells

Relatório Final POC 2

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

Orientador: Thiago Ferreira de Noronha
Coorientador: Natã Goulart da Silva

Belo Horizonte, Minas Gerais
2025

Sumário

1 Introdução

No contexto da engenharia de software, as boas práticas de programação abrangem uma variedade de aspectos, incluindo convenções de nomenclatura, organização de código, padrões de projeto e práticas de teste. Ao aderir a essas práticas, os desenvolvedores podem melhorar a qualidade do código, facilitar a colaboração em equipe e reduzir a incidência de bugs e erros.

Detectar e corrigir problemas associados às práticas de teste, ou test smells, representa um desafio significativo na engenharia de software contemporânea. Os test smells são indicadores de problemas nos testes automatizados, como redundâncias, complexidade excessiva, baixa cobertura ou má estruturação. Identificar tais práticas é crucial, pois testes de baixa qualidade podem comprometer a confiabilidade do software e aumentar a carga de manutenção.

O ESLint (2013) é uma ferramenta de código aberto amplamente utilizada na comunidade de desenvolvimento de software JavaScript. Ele funciona como um linter, ou seja, uma ferramenta de análise estática que verifica o código-fonte em busca de problemas, erros ou padrões problemáticos, de acordo com um conjunto de regras configuráveis. Além disso, é altamente configurável e permite que os desenvolvedores personalizem as regras de verificação de acordo com as necessidades específicas de seus projetos. Ele pode ser integrado facilmente em fluxos de trabalho de desenvolvimento, tanto em ambientes locais quanto em sistemas de integração contínua.

De tal forma, foi possível desenvolver uma ferramenta integrada ao ESLint (2013) que faz verificação de código fonte buscando especificamente problemas em código de teste, problemas esses que são passíveis de detecção estática.

1.1 Objetivos Gerais

Este projeto busca promover a importância das boas práticas de programação na engenharia de software e apresentar uma solução prática para aprimorar a qualidade dos testes automatizados em JavaScript por meio do desenvolvimento de um plugin para o ESLint (2013). Essa abordagem visa elevar o padrão de qualidade do código produzido, beneficiando tanto os desenvolvedores quanto os usuários finais das aplicações.

Neste contexto, o plugin desenvolvido para o ESLint (2013) desempenha um papel crucial, já que estende as capacidades da ferramenta para detectar "test smells", ou seja, indicadores de problemas nos testes automatizados.

1.2 Objetivos Específicos

Durante a etapa da POC 2, os objetivos foram desenvolver os seguintes aspectos:

- Implementação de mais 5 regras de verificação. São elas:
  - *Sleepy Test*: Método que invoca um método .sleep() ou variações dele, o que pode causar comportamentos inesperados, já que o tempo de processamento pode variar em diferentes dispositivos. (também é possível que o usuário defina um treshold para esse valor).
  - *Resource Optimism*: Método de teste faz uma suposição otimista de que o recurso externo (por exemplo, arquivo), utilizado pelo método de teste, existe.
  - *Eager Test*: Um único método de teste contém múltiplas chamadas para vários métodos de produção, resultando em dificuldades na compreensão e manutenção do teste.
  - *Constructor Initialization*: Uma classe de teste que contém uma declaração de construtor. Idealmente, o conjunto de testes não deve ter um construtor. A inicialização dos campos deve ser no método setUp() ou variações do mesmo.
  - *Unknown Test*: Um método de teste que não contém uma única instrução de asserção. É possível que um método de teste seja escrito sem uma instrução de asserção. Em tal cenário, possivelmente o framework de teste mostrará o método como aprovado se as instruções não resultarem em uma exceção, quando executadas.
- Escrita de documentação detalhada com as boas práticas para todas as regras implementadas. O objetivo é demonstrar na própria mensagem de erro retornada pela ferramenta uma descrição, seu possível impacto na qualidade do código, bem como um link para uma documentação de boas práticas associadas à regra, que estará diponível no repositório da ferramenta.
- Adição de uma sugestão de correção automática, quando possível. O plugin pode corrigir automaticamente algumas regras identificadas, especialmente quando a solução é clara, previsível e segura para ser aplicada sem intervenção manual. Essa etapa também inclui o estudo e avaliação de quais das regras implementadas possibilitam esse tipo de correção.

2 Referencial Teórico

Alguns estudos foram utilizados como base para o entendimento e a criação dessa proposta, os principais são (DAMASCENO..., 2018), que aborda a relação entre a qualidade do código e testes e (NGUYEN..., 2012), que aprofunda a pesquisa sobre detecção de práticas ruins de programação em aplicações web dinâmicas. Os principais conceitos empregados neste trabalho são:

- Test Smells: [Test... (2018)] são descritos como sendo vários padrões de refatoração comuns em testes de software. Essa referência explora os diferentes tipos de test smells, sendo estes os que serão abordados pela nova ferramenta:
  - *Assertion Roulette*: Ocorre quando um método de teste possui múltiplas asserções não documentadas. Múltiplas declarações de asserção em um método de teste sem uma mensagem descritiva impactam a legibilidade / compreensibilidade / manutenção, pois não é possível entender o motivo da falha do teste.
  - *Conditional Test Logic*: uso de condicionais (if-else e instruções de repetição) que constroem cenários onde blocos de código podem não ser executados.
  - *Exception Handling*: descrito quando o caso de teste possui instruções de try, catch e throw
  - *Ignored Test*: o caso de teste não é executado devido ao uso de uma propriedade ".skip" ou similiar.
  - *Redundant Print*: O método de teste contém uma instrução que imprime o valor de uma variável no console. Esta é uma declaração redundante que pode ter sido adicionada por um desenvolvedor, para fins de depuração, no momento da escrita do método de teste
  - *Sensitive Equality*: O uso do valor padrão retornado por um método toString() de objetos, para realizar comparações de strings, corre o risco de falha no futuro devido a mudanças na implementação de objetos do método toString().
  - *Sleepy Test*: Método que invoca um método .sleep() ou variações dele, o que pode causar comportamentos inesperados, já que o tempo de processamento pode variar em diferentes dispositivos.
  - *Resource Optimism*: Método de teste faz uma suposição otimista de que o recurso externo (por exemplo, arquivo), utilizado pelo método de teste, existe.
  - *Eager Test*: Um único método de teste contém múltiplas chamadas para vários métodos de produção, resultando em dificuldades na compreensão e manutenção do teste.

  - *Constructor Initialization*: Uma classe de teste que contém uma declaração de construtor. Idealmente, o conjunto de testes não deve ter um construtor. A inicialização dos campos deve ser no método setUp() ou variações do mesmo.
  - *Unknown Test*: Um método de teste que não contém uma única instrução de asserção. É possível que um método de teste seja escrito sem uma instrução de asserção. Em tal cenário, possivelmente o framework de teste mostrará o método como aprovado se as instruções não resultarem em uma exceção, quando executadas.

- O VSCode... (2021) é uma extensão para o Visual Studio Code que verifica a ortografia do texto em arquivos de código-fonte e de texto. Ele destaca palavras que estão fora do dicionário do idioma configurado, permitindo que os desenvolvedores identifiquem e corrijam erros de digitação e ortografia enquanto escrevem código ou texto. Essa extensão é útil para manter a qualidade e legibilidade do código, além de melhorar a precisão e profissionalismo da comunicação escrita..

- JSNose (2013) é uma ferramenta de análise estática desenvolvida para JavaScript, focada na detecção de más práticas de codificação e problemas de segurança no código-fonte. Ele funciona examinando o código JavaScript em busca de padrões problemáticos, vulnerabilidades conhecidas e potenciais erros de programação. O JSNose ajuda os desenvolvedores a identificar áreas de melhoria no código JavaScript, como variáveis não utilizadas, funções inseguras, uso inadequado de APIs e práticas que podem levar a vulnerabilidades de segurança. Ao destacar esses problemas, o JSNose auxilia os desenvolvedores a escrever código mais seguro e robusto.

Tais tecnologias desempenham papéis cruciais na monitorização e análise de código estático, e foram utilizadas como base em conjunto com os casos de teste referenciados para implementar a ferramenta.

3 Metodologia

Como a primeira versão do plugin já havia sido desenvolvida na primeira etapa deste projeto, a ideia foi evoluir e aprimorar tanto a qualidade das possibilidades de avaliação do ESLint (2013) como a quantidade, já que os novos casos de teste abordados passaram a ser cobertos pela nova versão.

Para a execução do projeto, os seguintes passos foram seguidos:

- Implementação dos novos casos de teste: Desenvolver o código para atender os novos requisitos da aplicação, usando como base as ferramentas similares previamente analisadas.
- Teste individual de cada test smell: Nesta etapa, executar uma série de testes individuais em cada um dos novos casos de teste escolhidos, de maneira à garantir que as validações façam o que é esperado.
- Documentação e publicação de nova versão: Descrever de maneira detalhada todos os test smells propostos, com descrição, impacto e um link para a documentação de boas práticas associadas à cada uma das regras cobertas pela aplicação.
- Adição de correção automática : Estudo e implementação de correção automática para as regras que possibilitam esse tipo de ação. A documentação deve detalhar quais das regras atendem esses requisitos e como a correção é feita.
- Publicação de nova versão: Por fim, publicar uma nova versão da ferramenta no gerenciador de pacotes npm (2010) para que a comunidade possa testar e utilizar.

4 Atividades Conduzidas

4.1 POC1

Desde o início do POC1, as atividades conduzidas para a realização do projeto seguiram essa ordem:

- Estudos e implementação do linter: A primeira etapa do projeto envolveu um estudo sobre o ESLint e as regras a serem implementadas. Durante este período, o tempo foi dedicado a entender como o ESLint funciona, sua arquitetura de plugins e a Árvore... (), que é uma representação estruturada do código-fonte em uma forma de árvore, onde cada nó da árvore corresponde a uma construção sintática do código. O ESLint (2013) usa a AST para analisar o código de maneira eficiente, permitindo a aplicação de regras de linting ao percorrer e interpretar essa estrutura. Essa análise baseada em Árvore... () facilita a detecção de padrões de código, erros e "smells"de programação de forma precisa e flexível e as melhores práticas para criar regras personalizadas. Após compreender o funcionamento do ESLint, foi iniciada implementação das regras escolhidas. As regras foram definidas com base em "test smells"comuns que podem comprometer a qualidade dos testes. Elas são:
  - assertion-roulette: Detecta múltiplas asserções em um único teste.
  - conditional-test-logic: Detecta lógica condicional em testes.
  - ignored-test: Detecta testes ignorados usando .skip ou .only
  - exception-handling: Detecta uso inadequado de tratamento de exceções em testes.
  - redundant-print: Detecta uso desnecessário de declarações console.log em testes.
  - sensitive-equality: Detecta comparações sensíveis em strings que podem causar falhas inesperadas em diferentes versões da linguagem Javascript.
- Teste individuais: Para garantir a robustez das regras implementadas, foram programados testes unitários para cada regra. Utilizando o framework de teste Mocha (), foram criados cenários de teste que cobrem tanto casos válidos quanto inválidos para todas as regras. Esses testes asseguram que as regras funcionem conforme o esperado e ajudem a manter a qualidade do código
- Testes em larga escala: O plugin foi testado em projetos open-source já existentes. Este passo foi crucial para avaliar o desempenho e a eficácia do plugin em diferentes bases de código. A aplicação do plugin em projetos reais permitiu identificar possíveis

melhorias e ajustes necessários para aumentar a precisão das regras. Abaixo seguem as etapas e scripts que foram utilizados para acelerar a coleta e execução de testes em larga escala. O código fonte dos scripts pode ser encontrado na Seção 6:

- Seleção de repositórios com a API do Github: Utilizando a API do github, um script 6.1 que permite encontrar repositórios foi desenvolvido. Ele busca por repositórios que possuam ao menos 500 estrelas, 100 forks e utilizem a linguagem "Javascript". Após encontrar essa lista de repositórios, o script procura, nos repositórios retornados, por arquivos de configurações que identifiquem frameworks de teste. Os frameworks abrangidos pela ferramenta até o momento são: Jest, Mocha e Karma. Além dos arquivos de teste, o script também filtra por repositórios que possuam arquivos com variações da palavra 'test'. Foram selecionados os 30 primeiros repositórios, considerando que muitos são projetos grandes e mantidos por toda a comunidade de desenvolvimento. Esses respositórios serviram de base para validar o funcionamento da ferramenta. Abaixo a lista:

Nome	**Estrelas**	**Forks**	**Referência**
next.js	121887	26069	(NEXT..., 2024)
create-react-app	102050	26635	(CREATE-REACT-APP, 2024)
three.js	99680	35156	(THREE..., 2024)
iptv	79286	1885	(IPTV, 2024)
mermaid	68025	5946	(MERMAID, 2024)
reveal.js	67113	16684	(REVEAL..., 2024)
github-readme-stats	65735	21202	(GITHUB-README-STATS, 2024)
webpack	64262	8730	(WEBPACK, 2024)
express	64091	14196	(EXPRESS, 2024)
Chart.js	63736	11873	(CHART..., 2024)
markdown-here	59516	11267	(MARKDOWN-HERE, 2024)
jquery	58972	20622	(JQUERY, 2024)
angular.js	58907	27550	(ANGULAR..., 2024)
gatsby	55160	10333	(GATSBY, 2024)
uptime-kuma	53266	4805	(UPTIME-KUMA, 2024)
prettier	48809	4234	(PRETTIER, 2024)
cypress	46500	3145	(CYPRESS, 2024)
dayjs	46368	2269	(DAYJS, 2024)
serverless	46323	5693	(SERVERLESS, 2024)
marktext	45798	3408	(MARKTEXT, 2024)
meteor	44174	5160	(METEOR, 2024)
zx	42517	1080	(ZX, 2024)
Leaflet	40671	5783	(LEAFLET, 2024)
materialize	38862	4746	(MATERIALIZE, 2024)
video.js	37613	7406	(VIDEO..., 2024)
htmx	35725	1205	(HTMX, 2024)
fullPage.js	35079	7294	(FULLPAGE..., 2024)
koa	35036	3222	(KOA, 2024)
JavaScript	31867	5464	(JAVASCRIPT..., 2024)
fastify	31468	2223	(FASTIFY, 2024)

Tabela 1 – Projetos populares no GitHub

- Download dos repositórios para ambiente local: Foi criado um script para clonar os repositórios selecionados para uma pasta local.
- Instalação do plugin nos diretórios de cada repositório: Também foi executado um script 6.2 para instalar o plugin, bem como adicionar o arquivo de configuração do ESLint em todos os repositórios.
- Execução do plugin em larga escala: Shell script 6.3 para iterar pelos diretórios filhos diretos da pasta de repositórios, rodar o comando que executa o plugin e salva os resultados obtidos em um arquivo 'result.json'.
- Análise dos Resultados: Script 6.4 em python que processa os arquivos 'result.json' em cada um dos repositórios, extrai a contagem de mensagens de erro exibidas pelo plugin e gera um relatório com a contagem de cada mensagem de

erro exibida por regra avaliada, bem como um relatório específico para cada repositório.

- Documentação e publicação: A documentação do projeto foi descrita e publicada no gerenciador de pacotes (NPM, 2010), acessível através da URL (NPM-ESLINT-TEST-SMELLS, 2024). Esta documentação inclui informações sobre a instalação, configuração e uso da ferramenta. Também é possível acessar o código-fonte e documentação através do github: (ESLINT-TEST-SMELLS, 2024)

4.2 POC2

A partir do POC2, as seguintes etapas foram seguidas para concluir a entrega do projeto:

  - Implementação das novas regras do linter: Essa etapa consistiu na implementação do código para atender os novos requisitos da aplicação, usando como base as ferramentas similares previamente analisadas. As novas regras são:
    * Sleepy Test: Método que invoca um método .sleep() ou variações dele, o que pode causar comportamentos inesperados, já que o tempo de processamento pode variar em diferentes dispositivos..
    * Resource Optimism: Método de teste faz uma suposição otimista de que o recurso externo (por exemplo, arquivo), utilizado pelo método de teste, existe.
    * Eager Test: Um único método de teste contém múltiplas chamadas para vários métodos de produção, resultando em dificuldades na compreensão e manutenção do teste.
    * Constructor Initialization: Uma classe de teste que contém uma declaração de construtor. Idealmente, o conjunto de testes não deve ter um construtor. A inicialização dos campos deve ser no método setUp() ou variações do mesmo.
    * Unknown Test: Um método de teste que não contém uma única instrução de asserção. É possível que um método de teste seja escrito sem uma instrução de asserção. Em tal cenário, possivelmente o framework de teste mostrará o método como aprovado se as instruções não resultarem em uma exceção, quando executadas.
  - Teste individual de cada test smell: Nesta etapa, foram executadas uma série de testes individuais em cada um dos novos casos de teste escolhidos, de maneira à garantir que as validações façam o que é esperado.

- Documentação e publicação de nova versão: Todos os test smells foram documentados, com descrição e um link para a página de boas práticas no github, associadas à cada uma das regras cobertas pela aplicação, bem como outras fontes. Entretanto, ao contrário do que era o objetivo inicial, a mensagem de erro retornada pela ferramenta não descreve toda a documentação associada à regra pois isso sobrecarregava a mensagem com excessivas informações para o usuário, algo que poluía a tela da interface de desenvolvimento utilizada. De tal forma, a documentação detalhada fica disponível por meio de um link exibido na própria mensagem de erro que o plugin exibe.
- Adição de correção automática : Foi realizada uma etapa de estudo e implementação de correção automática para as regras que possibilitassem esse tipo de ação. Nessa etapa, algumas dificuldades foram encontradas no que tange a correção automática. Como as correções dependem de uma manipulação direta no código-fonte do usuário, e em javascript diferentes pacotes são utilizados para manipular uma série de recursos iguais, a variedade de padrões com os quais é possível realizar as mesmas operações dificulta a implementação de uma correção automática, de modo que ela precisa verificar e garantir resiliência em diversos cenários no código. Por esse motivo, a limitação de tempo só permitiu que pudessem ser automatizadas com segurança as correções para as regras 'Redundant-print' e 'Ignored-test', que consistem em remover o código detectado, já que é possível garantir ele não é crucial e sua remoção não causa impacto à aplicação.
- Publicação de nova versão: Por fim foi feita a publicação de uma nova versão da ferramenta no gerenciador de pacotes npm (2010) para que a comunidade possa testar e utilizar. A última versão da ferramenta pode ser acessada na página do npm-eslint-test-smells (2024).

5 Conclusão e Resultados

O projeto resultou no desenvolvimento de uma ferramenta robusta e acessível, o eslint-test-smells (2024), que auxilia desenvolvedores na identificação e correção de "test smells"em códigos JavaScript. Durante a primeira etapa, foram implementadas as regras inicialmente planejadas, e a ferramenta foi submetida a testes em larga escala em projetos open-source - detalhes sobre essa etapa podem ser vistos no relatório final do POC1 (2024). Essa validação prática demonstrou a eficácia e a precisão das regras, além de fornecer insights para ajustes pontuais. Uma documentação inicial foi elaborada, detalhando a instalação, configuração e uso do plugin. Abaixo um exemplo de uso da regra 'Assertion roulette', em que o código de teste usa mais de uma asserção para verificar um único cenário:

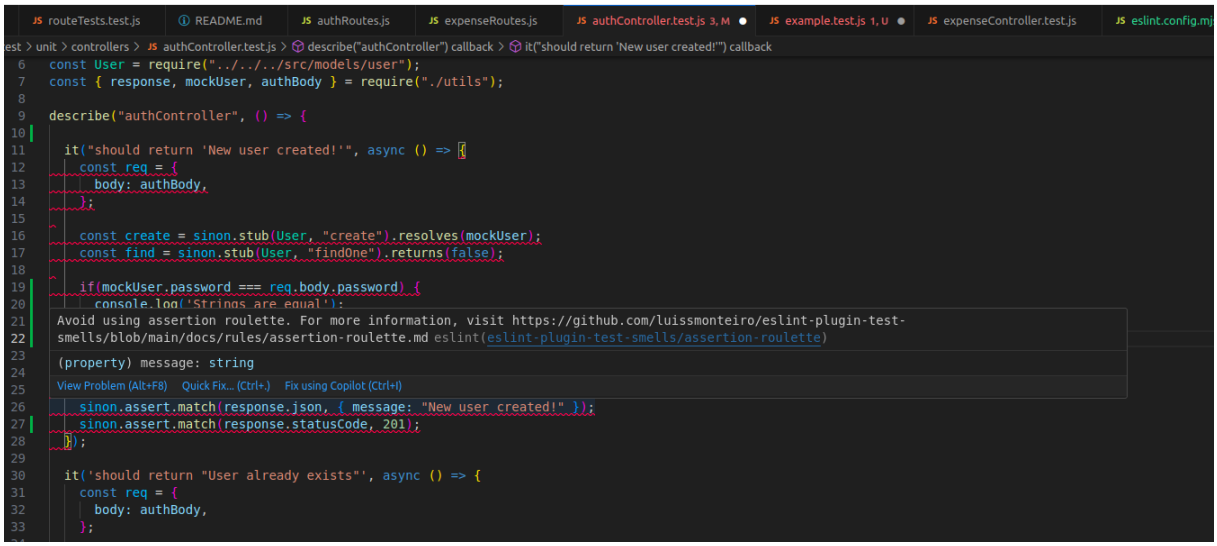

Figura 1 – Assertion Roulette

Na segunda etapa, novas regras foram desenvolvidas e integradas à ferramenta, expandindo sua capacidade de identificar padrões prejudiciais em testes automatizados. Paralelamente, a documentação foi aprimorada, passando a incluir explicações detalhadas de cada regra e exemplos práticos de uso. Também foi criada uma documentação abrangente sobre os casos de teste cobertos, tanto válidos quanto inválidos, permitindo que os desenvolvedores compreendam melhor como a ferramenta funciona. Após esses aprimoramentos, uma nova versão do plugin foi publicada no gerenciador de pacotes NPM, consolidando a ferramenta como uma solução para a comunidade open-source. Abaixo, um exemplo de como a correção automática pode funcionar, usando a regra 'Ignored-test':

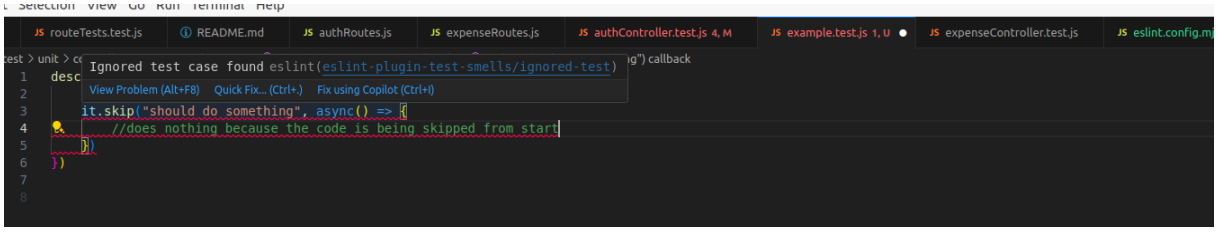

Figura 2 – Ignored test 1

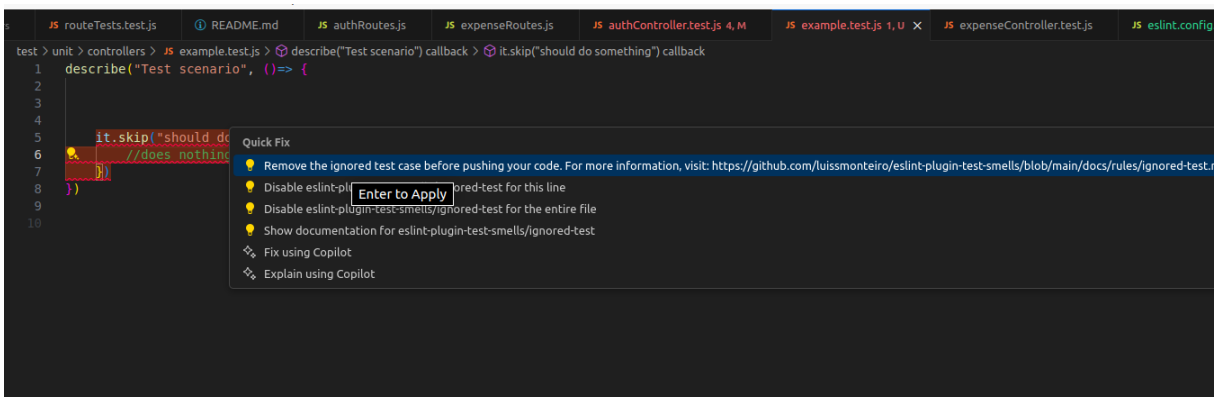

Figura 3 – Ignored test 2

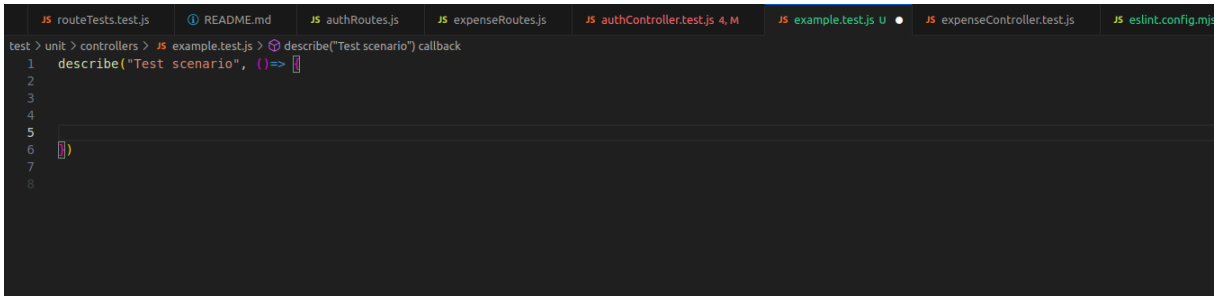

Figura 4 – Ignored test 3

Os resultados obtidos reforçam a relevância do projeto no contexto da engenharia de software, especialmente para equipes que desejam garantir a qualidade de seus testes e reduzir potenciais problemas em aplicações de larga escala. A publicação como open-source no GitHub e NPM torna a ferramenta amplamente acessível e permite que ela evolua com o apoio da comunidade de desenvolvedores, gerando um impacto positivo contínuo no ecossistema JavaScript..

Referências

ANGULAR.JS. 2024. Disponível em: <https://github.com/angular/angular.js>. 10

CHART.JS. 2024. Disponível em: <https://github.com/chartjs/Chart.js>. 10

CREATE-REACT-APP. 2024. Disponível em: <https://github.com/facebook/create-react-app>. 10

CYPRESS. 2024. Disponível em: <https://github.com/cypress-io/cypress>. 10

DAMASCENO, H., Bezerra, C., Campos, D., Machado, I., Coutinho, E. (2023). Test smell refactoring revisited: What can internal quality attributes and developers' experience tell us?. Journal of Software Engineering Research and Development, 11(1), 13:1 – 13:16. https://doi.org/10.5753/jserd.2023.3195. 2018. 5

DAYJS. 2024. Disponível em: <https://github.com/iamkun/dayjs>. 10

ESLINT. 2013. Https://eslint.org/. 3, 7, 8

ESLINT-TEST-SMELLS. 2024. Disponível em: <https://github.com/luissmonteiro/eslint-plugin-test-smells>. 11, 13

EXPRESS. 2024. Disponível em: <https://github.com/expressjs/express>. 10

FASTIFY. 2024. Disponível em: <https://github.com/fastify/fastify>. 10

FULLPAGE.JS. 2024. Disponível em: <https://github.com/alvarotrigo/fullPage.js>. 10

GATSBY. 2024. Disponível em: <https://github.com/gatsbyjs/gatsby>. 10

GITHUB-README-STATS. 2024. Disponível em: <https://github.com/anuraghazra/github-readme-stats>. 10

HTMX. 2024. Disponível em: <https://github.com/bigskysoftware/htmx>. 10

IPTV. 2024. Disponível em: <https://github.com/iptv-org/iptv>. 10

JAVASCRIPT Algorithms. 2024. Disponível em: <https://github.com/TheAlgorithms/JavaScript>. 10

JQUERY. 2024. Disponível em: <https://github.com/jquery/jquery>. 10

JSNOSE. 2013. <https://people.ece.ubc.ca/aminmf/SCAM2013.pdf>. Acesso em 25 de março de 2024. 6

KOA. 2024. Disponível em: <https://github.com/koajs/koa>. 10

LEAFLET. 2024. Disponível em: <https://github.com/Leaflet/Leaflet>. 10

MARKDOWN-HERE. 2024. Disponível em: <https://github.com/adam-p/markdown-here>. 10

MARKTEXT. 2024. Disponível em: <https://github.com/marktext/marktext>. 10

MATERIALIZE. 2024. Disponível em: <https://github.com/Dogfalo/materialize>. 10

MERMAID. 2024. Disponível em: <https://github.com/mermaid-js/mermaid>. 10

METEOR. 2024. Disponível em: <https://github.com/meteor/meteor>. 10

MOCHA. Https://github.com/mochajs/mocha. 8

NEXT.JS. 2024. Disponível em: <https://github.com/vercel/next.js>. 10

NGUYEN, Hung Nguyen, Hoan Nguyen, Tung Nguyen, Anh Tien, Nguyen. (2012). Detection of embedded code smells in dynamic web applications. 10.1145/2351676.2351724. 2012. 5

NPM. 2010. Disponível em: <https://docs.npmjs.com/about-npm>. 7, 11, 12

NPM-ESLINT-TEST-SMELLS. 2024. Disponível em: <https://www.npmjs.com/package/eslint-plugin-test-smells>. 11, 12

POC1. 2024. Disponível em: <https://monografias.dcc.ufmg.br/monografia/um-plugin-para-a-ferramenta-eslint-que-verifica-test-smells/>. 13

PRETTIER. 2024. Disponível em: <https://github.com/prettier/prettier>. 10

REVEAL.JS. 2024. Disponível em: <https://github.com/hakimel/reveal.js>. 10

SERVERLESS. 2024. Disponível em: <https://github.com/serverless/serverless>. 10

TEST Smells. 2018. <https://testsmells.org/pages/testsmells.html#ConditionalTestLogic>. Acesso em 27 de de 2024. 5

THREE.JS. 2024. Disponível em: <https://github.com/mrdoob/three.js>. 10

UPTIME-KUMA. 2024. Disponível em: <https://github.com/louislam/uptime-kuma>. 10

VIDEO.JS. 2024. Disponível em: <https://github.com/videojs/video.js>. 10

VSCODE spell checker. 2021. <https://github.com/streetsidesoftware/vscode-spell-checker/>. Acesso em 27 de março de 2024. 6

WEBPACK. 2024. Disponível em: <https://github.com/webpack/webpack>. 10

ZX. 2024. Disponível em: <https://github.com/google/zx>. 10

ÁRVORE de sintaxe abstrata. Https://github.com/estree/estree. 8

6 Apêndice

6.1 Script que lista seleciona repositórios passíveis de teste no Github

```
import requests
from github import Github

# Fun   o para buscar projetos do GitHub com base em crit rios
    espec ficos
def buscar_projetos_github(linguagem, estrelas_minimas, forks_minimos,
    per_page=100):
    url = f"https://api.github.com/search/repositories?q=language:{
        linguagem}+stars:>={estrelas_minimas}+forks:>={forks_minimos}&
        sort=stars&order=desc&per_page={per_page}"
    headers = {'Accept': 'application/vnd.github.v3+json'}
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        return response.json()['items']
    else:
        return []

# Fun   o para verificar se um reposit rio cont m testes unit rios
def verificar_testes_unitarios(repo):
    try:
        # Listar arquivos e diret rios na raiz do reposit rio
        contents = repo.get_contents("")
        test_dirs = ['test', 'tests', '__tests__']
        test_files = ['jest.config.js', 'mocha.opts', 'karma.conf.js']

        for content_file in contents:
            if content_file.type == 'dir' and content_file.name.lower()
                in test_dirs:
                return True
            if content_file.type == 'file' and content_file.name.lower()
                in test_files:
                return True

        # Se a verifica   o inicial falhar, verificar recursivamente
            subdiret rios
        for content_file in contents:
            if content_file.type == 'dir':
                sub_contents = repo.get_contents(content_file.path)
```

```
                for sub_content_file in sub_contents:
                    if sub_content_file.type == 'dir' and
                       sub_content_file.name.lower() in test_dirs:
                        return True
                    if sub_content_file.type == 'file' and
                       sub_content_file.name.lower() in test_files:
                        return True
    except Exception as e:
        print(f"Erro ao verificar o reposit rio {repo.full_name}: {e}")

    return False

# Configura  o inicial
linguagem = "JavaScript"
estrelas_minimas = 500
forks_minimos = 100
token = 'Token removido por quest es de seguran a'

g = Github(token)

# Buscar projetos
projetos = buscar_projetos_github(linguagem, estrelas_minimas,
   forks_minimos)

# Filtrar projetos que possuem testes unit rios
projetos_com_testes = []
for projeto_info in projetos:
    repo = g.get_repo(projeto_info['full_name'])
    if verificar_testes_unitarios(repo):
        projetos_com_testes.append(projeto_info)

# Exibir os projetos selecionados
for projeto in projetos_com_testes:
    print(f"Nome: {projeto['name']}, URL: {projeto['html_url']}, Stars:
       {projeto['stargazers_count']}, Forks: {projeto['forks_count']}")
```

6.2 Instala ESLint test-smells em todos os diretórios

```
#!/bin/bash
ARQUIVO_CONFIG="home/luis/Documents/eslint.config.mjs"

PASTA_REPOSITORIOS="/home/luis/Documents/test_repos"

# Itera sobre cada diret rio filho direto em PASTA_REPOSITORIOS
for DIR in "$PASTA_REPOSITORIOS"/*; do
    if [ -d "$DIR" ]; then
```

```
        echo "Instalando␣ESLint␣no␣reposit rio:␣$DIR"
        cd "$DIR"
        cp $ARQUIVO_CONFIG .
        npm install ~/Documents/eslint-test-smells
    fi
done

echo "Processo␣conclu do."
```

6.3 Executa plugin em cada diretório de testes

```
#!/bin/bash

# Caminho para a pasta contendo os reposit rios
PASTA_REPOSITORIOS="/home/luis/Documents/test_repos"

# Caminho para o arquivo de sa da
ARQUIVO_SAIDA="/home/luis/Documents/test_repos/eslint_output.txt"

# Limpar o conte do do arquivo de sa da se ele j  existir
> $ARQUIVO_SAIDA

# Iterar sobre cada diret rio filho direto em PASTA_REPOSITORIOS
for DIR in "$PASTA_REPOSITORIOS"/*; do
    if [ -d "$DIR" ]; then
        echo "Executando␣ESLint␣no␣reposit rio:␣$DIR" >> $ARQUIVO_SAIDA
        cd "$DIR"
        # Executar o comando ESLint e salvar o output no arquivo de
           sa da
        npx eslint . --format json --output-file result.json
    fi
done

echo "Processo␣conclu do.␣Os␣resultados␣foram␣salvos␣em␣$ARQUIVO_SAIDA"
```

6.4 Processa resultados obtidos após execução dos testes em larga escala

```
import os
import json
from collections import Counter, defaultdict

pasta_repositorios = '/home/luis/Documents/test_repos'
```

```
arquivo_saida_global = '/home/luis/Documents/scripts-eslint/novos-resultados.txt'
arquivo_saida_por_diretorio = '/home/luis/Documents/scripts-eslint/novos-resultados-por-dir.txt'

# Dicion rios para armazenar contagens de ruleIds
contador_ruleIds_global = Counter()

contador_ruleIds_por_diretorio = defaultdict(Counter)

# Obter a lista de diret rios diretamente dentro da pasta_repositorios
repositorios = [d for d in os.listdir(pasta_repositorios) if os.path.isdir(os.path.join(pasta_repositorios, d))]

# Percorrer cada reposit rio
for repo in repositorios:
    caminho_repo = os.path.join(pasta_repositorios, repo)
    caminho_result_json = os.path.join(caminho_repo, 'result.json')
    if os.path.exists(caminho_result_json):
        try:
            with open(caminho_result_json, 'r') as f:
                dados = json.load(f)
                for objeto in dados:
                    for message in objeto.get('messages', []):
                        ruleId = message.get('ruleId')
                        if ruleId:
                            contador_ruleIds_global[ruleId] += 1
                            contador_ruleIds_por_diretorio[repo][ruleId] += 1
        except json.JSONDecodeError:
            print(f"Erro ao ler o arquivo JSON em {caminho_result_json}")
        except Exception as e:
            print(f"Erro inesperado no reposit rio {caminho_repo}: {e}")

# Escrever os resultados globais nos arquivos de sa da
with open(arquivo_saida_global, 'w') as f_global:
    f_global.write("Contagens globais de ruleId:\n")
    for ruleId, contagem in contador_ruleIds_global.items():
        f_global.write(f"{ruleId}: {contagem}\n")

with open(arquivo_saida_por_diretorio, 'w') as f_diretorio:
    f_diretorio.write("Contagens de ruleId por diret rio:\n")
    for repo, contador in contador_ruleIds_por_diretorio.items():
        f_diretorio.write(f"\nReposit rio: {repo}\n")
        for ruleId, contagem in contador.items():
```

```
47                 f_diretorio.write(f"␣␣{ruleId}:␣{contagem}\n")
48
49 print(f"Processo␣conclu do.␣Os␣resultados␣globais␣foram␣salvos␣em␣{
       arquivo_saida_global}␣e␣os␣resultados␣por␣diret rio␣foram␣salvos␣em␣
       {arquivo_saida_por_diretorio}")
```