UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

PEDRO AUGUSTO DE CASTRO FERREIRA

INVESTIGANDO O IMPACTO DOS *CODE SMELLS* NA MANUTENIBILIDADE DE APLICAÇÕES JAVASCRIPT E TYPESCRIPT

QUIXADÁ

2023

PEDRO AUGUSTO DE CASTRO FERREIRA

INVESTIGANDO O IMPACTO DOS *CODE SMELLS* NA MANUTENIBILIDADE DE APLICAÇÕES JAVASCRIPT E TYPESCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra.

QUIXADÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

F443i Ferreira, Pedro Augusto de Castro.
Investigando o impacto dos code smells na manutenibilidade de aplicações JavaScript e TypeScript / Pedro Augusto de Castro Ferreira. – 2023.
48 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2023.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Code smell. 2. JavaScript (Linguagem de programação de computador). 3. TypeScript. 4. Manutenibilidade. I. Título.

CDD 005.1

PEDRO AUGUSTO DE CASTRO FERREIRA

INVESTIGANDO O IMPACTO DOS *CODE SMELLS* NA MANUTENIBILIDADE DE APLICAÇÕES JAVASCRIPT E TYPESCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: ___/___/___

BANCA EXAMINADORA

Profa. Dra. Carla Ilane Moreira
Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Me. Júlio Serafim Martins
Instituto Federal do Ceará (IFCE)

Prof. Dr. Jefferson de Carvalho Silva
Universidade Federal do Ceará (UFC)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

AGRADECIMENTOS

Gostaria de agradecer a Deus, que toda honra e glória seja sempre em seu nome. Obrigado Senhor, por me abençoar e permitir a conclusão deste ciclo, pois sem Ti nada disso seria possível.

Aos meus queridos e amados pais, Flavia Dias e Eros Augusto, agradeço pelo amor, carinho, educação e tudo que abdicaram em prol do meu sucesso. Vocês são e sempre serão as maiores referências que eu tenho na minha vida, servindo sempre de inspiração no dia a dia. Com vocês, aprendi sobre fé, resiliência, e a batalhar. Não menos importante, aprendi que a educação transforma vidas. A minha Avó, Maria Tereza, um agradecimento em especial, pelo papel que exerceu na minha vida, contribuindo com a minha criação, sempre repleta de amor e carinho.

Aos amigos, Clebson Uchôa, Victor Lucas, Wilton Neto, Rafael Lima, Josué Nicholson e Isaac James, sem vocês, esta jornada não seria a mesma. Agradeço pela amizade sincera, que transbordou o dia a dia da universidade.

À Daniele Ismael, minha amada, obrigado pelos incentivos e puxões de orelhas, que fizeram com que eu chegasse ao fim deste trabalho.

Ao povo Cearense, que me acolheu tão bem na cidade de Quixadá, sempre com respeito e hospitalidade.

À Universidade Federal do Ceará — Campus Quixadá, por abrir as portas e me formar, proporcionando experiências ricas em conhecimento. Obrigado a todos os professores, servidores, colaboradores e alunos deste lugar tão belo.

À Profa. Dra. Carla Ilane Moreira Bezerra, pela excelente orientação e aos professores participantes da banca examinadora Prof. Ms. Júlio Serafim Martins e Prof. Dr. Jefferson de Carvalho Silva pelo tempo, pelas valiosas colaborações e sugestões.

"O Senhor é o meu pastor, nada me faltará"

(Salmos 23:1)

RESUMO

Em um cenário tecnológico em constante evolução, a manutenibilidade de sistemas de software se tornou um fator crítico para o sucesso a longo prazo de qualquer projeto. Principalmente no contexto de desenvolvimento web, onde linguagens como JavaScript e TypeScript desempenham um papel fundamental na criação dessas aplicações. Neste contexto, os *code smells* são vistos como ameaça a manutenibilidade e implicam na necessidade de refatoração dos sistemas. Este trabalho avalia a manutenibilidade e o impacto dos *code smells* em projetos que usam as linguagens de programação JavaScript e TypeScript. Para a realização deste estudo, dez projetos JavaScript e dez projetos TypeScript foram selecionados através da API do Github. Foram coletados e analisados os *code smells* dos projetos selecionados utilizando a ferramenta Embold e realizada a coleta de métricas de manutenibilidade utilizando a ferramenta Understand. A partir desses dados, foram obtidos os seguintes resultados: (i) a complexidade para detectar *code smells* é maior em projetos JavaScript; (ii) os *code smells Nested Callback, Shotgun Surgery* e *Message Chain* foram detectados; (iii) a base de código e a taxa de comentários dos projetos JavaScript é sempre maior em relação aos projetos TypeScript; (iv) projetos JavaScript são mais complexos em relação aos projetos TypeScript; (v) a densidade de *code smells* indicou uma pontuação melhor para os projetos JavaScript. Constatou-se que os projetos TypeScript são melhor estruturados e menos complexos em relação aos projetos JavaScript. Apesar disso, concluiu-se que a escolha isolada de utilizar TypeScript não garante bons índices de manutenibilidade e *code smells*.

Palavras-chave: *Code smell*; JavaScript; TypeScript; Manutenibilidade.

ABSTRACT

In a constantly evolving technological scenario, the maintainability of software systems has become a critical factor in the long-term success of any project. Mainly in context of web development, where languages like JavaScript and TypeScript play a key role in creating these applications. In this context, code smells are seen as a threat to maintainability and imply the need to refactor the systems. This study evaluates the maintainability and the impact of code smells in projects that use the JavaScript and TypeScript programming languages. For the realization of this study, ten projects JavaScript and ten TypeScript projects were selected via the Github API. They were collected and analyzed the code smells of the selected projects using the Embold tool and the collection of maintainability metrics was carried out using the Understand tool. From these data, the following results were obtained: (i) the complexity to detect code smells it is larger in JavaScript projects; (ii) the Nested Callback, Shotgun Surgery and Message code smells Chain were detected; (iii) the codebase and comment rate of JavaScript projects is always bigger compared to TypeScript projects; (iv) JavaScript projects are more complex in relation to TypeScript projects; (v) code smells density indicated a better score for JavaScript projects. It was found that TypeScript projects are better structured and less complex compared to JavaScript projects. Despite this, it was concluded that the choice isolated from using TypeScript does not guarantee good maintainability rates and code smells

Palavras-chave: Code smell; JavaScript; TypeScript; Maintainability.

LISTA DE FIGURAS

Figura 1 – Aba de repositórios do Embold 17
Figura 2 – Aba de issues do Embold 18
Figura 3 – Trecho de código JavaScript 18
Figura 4 – Trecho de código TypeScript 19
Figura 5 – Modelo de qualidade ISO/IEC 25010 20
Figura 6 – Tela inicial Understand Tool 21
Figura 7 – Passos para a realização do trabalho 25

LISTA DE TABELAS

Tabela 1 – Projetos selecionados 26
Tabela 2 – Ocorrência de *code smells* em projetos JavaScript 30
Tabela 3 – Ocorrência de *code smells* em projetos TypeScript 30
Tabela 4 – Soma da ocorrência de *code smells* em projetos JavaScript e TypeScript . . 31
Tabela 5 – Coleta de métricas em projetos JavaScript 32
Tabela 6 – Coleta de métricas em projetos TypeScript 33
Tabela 7 – Coleta de métricas em projetos JavaScript 34
Tabela 8 – Coleta de métricas em projetos TypeScript 35
Tabela 9 – Coleta de métricas em projetos JavaScript 36
Tabela 10 – Coleta de métricas em projetos TypeScript 37
Tabela 11 – Densidade de code smells em projetos JavaScript 38
Tabela 12 – Densidade de code smells em projetos TypeScript 39
Tabela 13 – Densidade de *code smells* dos projetos 40

LISTA DE QUADROS

Quadro 1 – *Code smells* para Orientação a Objetos definidos por Fowler e Beck (1997) coletados para TypeScript pela ferramenta Embold 15
Quadro 2 – *Code smells* para linguagem JavaScript coletados pela ferramenta Embold . 16
Quadro 3 – Comparativo entre os trabalhos relacionados e o trabalho proposto. 24

SUMÁRIO

1 INTRODUÇÃO 12
1.1 Objetivos 13
1.1.1 Objetivo geral 13
1.1.2 Objetivos Específicos 13
1.2 Organização 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 *Code smells* 15
2.1.1 Detecção de code smells 16
2.1.1.1 Ferramenta Embold *16*
2.2 JavaScript 17
2.3 TypeScript 19
2.4 Qualidade de Software 19
2.4.1 Ferramentas de medição de qualidade de software 20
2.4.2 Manutenibilidade 21
3 TRABALHOS RELACIONADOS 23
4 PROCEDIMENTOS METODOLÓGICOS 25
4.1 Selecionar sistemas JavaScript e TypeScript 25
4.2 Identificar e coletar *code smells* 27
4.3 Medir a manutenibilidade dos projetos selecionados 27
4.4 Comparar projetos JavaScript e TypeScript 27
5 RESULTADOS 29
5.1 Ocorrência de *code smells* nos projetos JavaScript e TypeScript 29
5.2 Avaliação da manutenibilidade de projetos JavaScript e TypeScript . . . 31
5.3 Densidade de *code smells* em projetos JavaScript e TypeScript 38
5.4 Ameaças à validade 40
6 CONCLUSÕES E TRABALHOS FUTUROS 42
REFERÊNCIAS 44

1 INTRODUÇÃO

Sistemas de software estão presente em vários domínios, como web, mobile, instituições financeiras e casas inteligentes (YENDURI; GADEKALLU, 2022). A presença de software nessas áreas reforça a ideia de que desenvolvedores devem seguir padrões para alcançar experiencias positivas para os usuários (NOOR, 2022). Tendo em vista que a manutenibilidade de software é essencial para o sucesso de um projeto (BUTT *et al.*, 2022), as especificações descritas na ISO (2011) tornam-se indispensaveis. Sendo assim, a refatoração de código é uma atividade importante no processo de manutenção (BAQAIS; ALSHAYEB, 2020) que consequentemente, impacta na qualidade de software. Um fator que indica a necessidade de refatoracao, é a presença de code smells (ALOMAR *et al.*, 2021).

Code smells, por sua vez, são vistos como más decisões técnicas que trazem como consequência, impactos na qualidade de software (HAN *et al.*, 2021). Originalmente este conceito foi apresentado por Fowler e Beck (1997) em 22 tipos voltados para linguagens de programação orientadas a objeto. Entretanto, esse assunto expandiu-se e pode ser observado em linguagens de programação bem populares, como JavaScript e TypeScript. Todavia, a literatura ressalta que hà diferença entre os *code smells* encontrados nessas linguagens de programação. Nesse sentido, as ferramentas para detecção de *code smells* e medição de qualidade também não são unanimidade entre essas linguagens programação (FARD; MESBAH, 2013; KAUR; DHIMAN, 2019).

Uma pesquisa realizada pelo StackOverflow (2021), constatou que JavaScript e TypeScript ocupam respectivamente a primeira e a quinta posição no *ranking* das linguagens mais populares entre os desenvolvedores de *software*. É importante destacar que enquanto o JavaScript é utilizado desde a década de 1990 (WIRFS-BROCK; EICH, 2020), o Typescrtip foi lancado pela Microsoft em 2012. Embora esta linguagem agregar mais confiabilidade para o código uma vez que, a checagem de tipos é feita enquanto o compilador converte o código para JavaScript (BOGNER; MERKEL, 2022), esta linguagem não está livre dos mesmos problemas do JavaScript e também vem sendo objeto de pesquisa em alguns estudos (FREEMAN, 2019; MIU *et al.*, 2020; FREEMAN, 2021). Dessa forma, muito tem-se discutido acerca dessas linguagens de programação e a relação com *code smells* e manutenibilidade (SALGADO, 2020; MERKEL, 2021; BOGNER; MERKEL, 2022).

De acordo com Press (1992), manutenibilidade diz respeito a “facilidade com que um sistema ou componente de software pode ser modificado para corrigir falhas, melhorar

o desempenho ou outros atributos ou adaptar-se a um ambiente alterado". Além disso, este atributo de qualidade são divididos nos seguintes aspectos: analisabilidade, modificabilidade, modularidade, reusabilidade e testabilidade (ISO, 2011). Nesse sentido, um estudo feito por Haque *et al.* (2018), identificou que os *code smells* são um dos fatores que prolongam a manutenção de software, bem como, afetam a qualidade de software com ênfase na reusabilidade. Sendo assim, o uso de métricas de qualidade, como: *Lines of Code (LOC), Executable Lines of Code (ELOC), Number of parameters (NOP)*, são úteis para detectar *code smells* e medir a manutenibilidade das aplicações. Logo, o fato do JavaScript e TypeScript estarem entre as linguagens mais populares entre os desenvolvedores, associado ao fato da manutenibilidade estar envolvida entre os principais afetados pela presença de *code smells*, evidencia a importância de analisar o impacto na manutenibilidade nessas linguagens de programação.

Dessa forma, este trabalho apresenta uma estudo exploratório do impacto dos *code smells* na manutenibilidade de projetos *open-source* em projetos JavaScript e TypeScript. Nesse estudo, serão analisados três fatores: (i) frequência dos *code smells* que ocorrem nas duas linguagens; (ii) manutenibilidade dos projetos JavaScript e TypeScript; e, (iii) densidade de *code smells* nas linguagens. Este trabalho será útil para os times de software analisarem qual das duas linguagens será menos suscetível a problemas de manutenção e quais apresentam mais degradação do código de acordo com os *code smells* identificados.

1.1 Objetivos

1.1.1 Objetivo geral

O objetivo geral deste trabalho é analisar repositórios de código *open-source* desenvolvidos nas linguagens JavaScript e TypeScript a fim de identificar e comparar *code smells* e o impacto na manutenibilidade.

1.1.2 Objetivos Específicos

a) Investigar os *code smells* que mais ocorrem em projetos JavaScript e TypeScript.
b) Medir a manutenibilidade dos projetos JavaScript e TypeScript.
c) Analisar se a densidade de *code smells* é maior em projetos JavaScript ou TypeScript.

1.2 Organização

Este trabalho segue apresentado a partir do Capítulo 2, onde são apresentados os termos e conceitos necessários para a compreensão do trabalho. No Capítulo 3 são apresentados os trabalhos relacionados com o presente trabalho. O Capítulo 4, apresenta a metodologia seguida para executar este trabalho. O Capítulo 5, apresenta os resultados obtidos a partir da execução metodologia. O capítulo 6, apresenta as conclusões e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os principais conceitos necessários para o entendimento deste trabalho. A Seção 2.1 descreve a definição e os tipos de *code smells* abordados pela literatura. Na Seção 2.2 são apresentados os conceitos básicos de JavaScript. A Seção 2.3 descreve os conceitos básicos de TypeScript. A Seção 2.4 aborda manutenibilidade como um dos atributos de qualidade de software de acordo com a ISO 25010.

2.1 *Code smells*

O termo *code smells*, foi utilizado pela primeira vez em 1999, por Fowler e Beck (1997). A principio Fowler, definia *code smells* como estruturas de código que afetam a qualidade de software e aumentam a necessidade de refatoração. Entretanto, outros termos como: *"design flaw", "smells", "disharmony"*, são usados como sinônimos para se referir ao mesmo tipo de problema (SANTOS *et al.*, 2018). *Anti-patterns* é outro termo bastante visto na literatura que, segundo Luo *et al.* (2010), se originam a partir de *design patterns* que criaram mais problemas do que soluções. *Code smells*, estão relacionados a estes problemas e ambos podem ser resolvidos através da refatoração, desencadeando melhorias na qualidade de software. Apesar de (FOWLER; BECK, 1997) definir 22 tipos de *code smells*, apenas 2 estão descritos no Quadro 1, e 1 específico para o JavaScript descrito no Quadro 2. O numero reduzido de *code smells* esta relacionado as limitações da ferramenta utilizada para detecção de *code smells*. A causa esta relacionada a problemas que envolvem os pilares da qualidade de software (e.g., manutenibilidade, modularidade, legibilidade, legibilidade do código, compreensibilidade), bem como, nas áreas de arquitetura e teste de software, conforme atesta Tahir *et al.* (2020).

Quadro 1 – *Code smells* para Orientação a Objetos definidos por Fowler e Beck (1997) coletados para TypeScript pela ferramenta Embold

Nome	Descrição
Shotgun surgery	Mudanças que são necessárias em muitos lugares se tornam difíceis de encontrar e fáceis de esquecer
Message chain	Uma função/método chama muitas funções que chamam internamente muitas outras funções.

Fonte: Adaptado de (SANTOS *et al.*, 2018)

Quadro 2 – *Code smells* para linguagem JavaScript coletados pela ferramenta Embold

Code smell	Descrição
Nested Callbacks	Aparece quando varias chamadas assíncronas são invocadas em sequencia

Fonte: Adaptado de Saboury *et al.* (2017)

2.1.1 Detecção de code smells

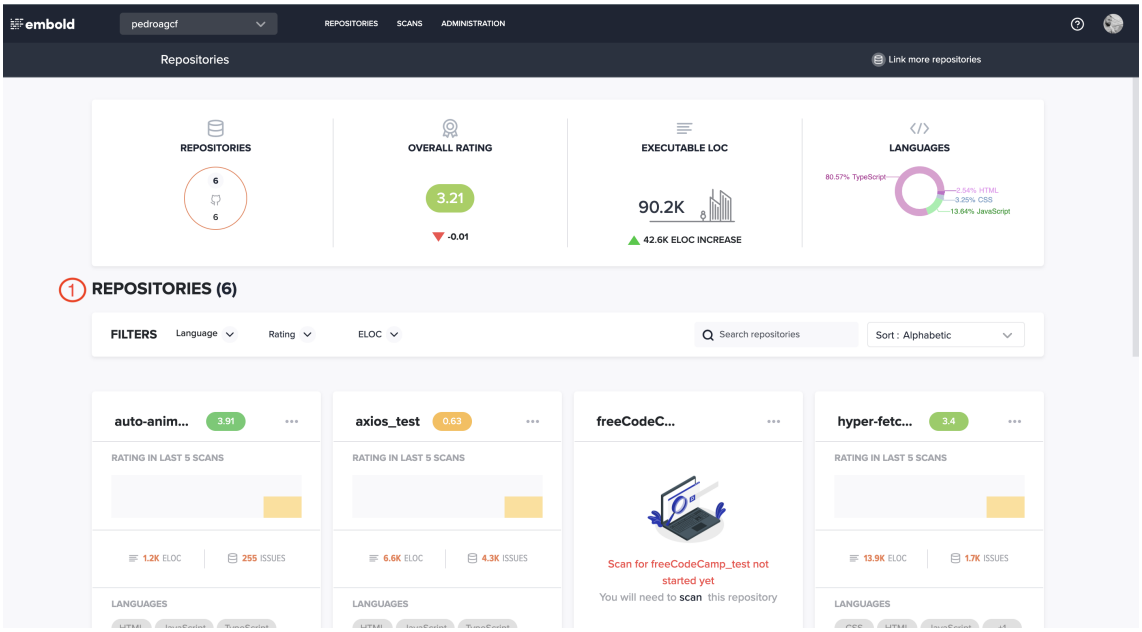
Uma revisão sistemática feita por Lacerda *et al.* (2020), evidenciou que existem mais de 162 ferramentas de detecção de *code smells*. Entretanto, a maioria se concentra na linguagem Java e não é capaz de detectar os 22 tipos de *code smells* apontados por Fowler e Beck (1997). Outro fator a ser levado em conta, é que para o cenário de projetos em JavaScript/TypeScript outros tipos de *code smells* são acrescentados, logo a ferramenta a ser utilizada precisa atender a essa especificidade. Nesse sentido, as ferramentas JSnose e TAJSlint são opções identificadas na literatura Almashfi e Lu (2020), Fard e Mesbah (2013) que atendem a esses critérios, mas que não oferecem suporte ao TypeScript. Por outro lado, a ferramenta Embold oferece suporte a detecção de *code smells* em ambas linguagens de programação JavaScript e TypeScript. Dessa forma, este presente trabalho considera o uso da ferramenta Embold coletar os *code smells* e realizar medições nos repositórios que serão analisados neste trabalho.

2.1.1.1 Ferramenta Embold

Embold é uma ferramenta de análise de código estático que aponta *anti-patterns (code smells)*, blocos de código duplicados e boas praticas de código (HOLMBERG, 2021). A ferramenta suporta várias linguagens de programação, entre elas estão C, C++, Python, Java, JavaScript e TypeScript. O Embold disponibiliza esses e outros relatórios, tais como: indicadores de performance (KPI), *bugs, issues e blueprints*, através de uma interface web. Além disso, utiliza de um conjunto de 19 métricas para inferir a qualidade do software, são elas: *Lines of Code (LOC), Executable Lines Of Code (ELOC), Lines Of Code Comments (LOC Comments), Comment Ratio (CR), Number of Methods (NOM), Number of Attributes (NOA), Lack of Cohesion Of Methods (LCOM), Number of Public Attributes (NOPA), Cyclomatic Complexity (CC), Coupling Between Objects (CBO), Depth of Inheritance Hierarchy (DOIH), Response for Class (RFC), Foreign Data Providers (FDP), Locality of Attribute Accesses (LAA), Number of Accessed Variables (NOAV), Access To Foreign Data (ATFD), Max Nesting (MN), Number of Parameters (NOP), Number of Public Methods (NOPM)* (EMBOLD, 2022).

Na Figura 1, apresenta-se a aba de repositórios da ferramenta Embold, destacando algumas características da ferramenta que serão exploradas neste trabalho. O elemento 1 apresenta a seção de repositórios, onde contem a lista dos repositórios que já foram ou ainda serão analisados.

Figura 1 – Aba de repositórios do Embold



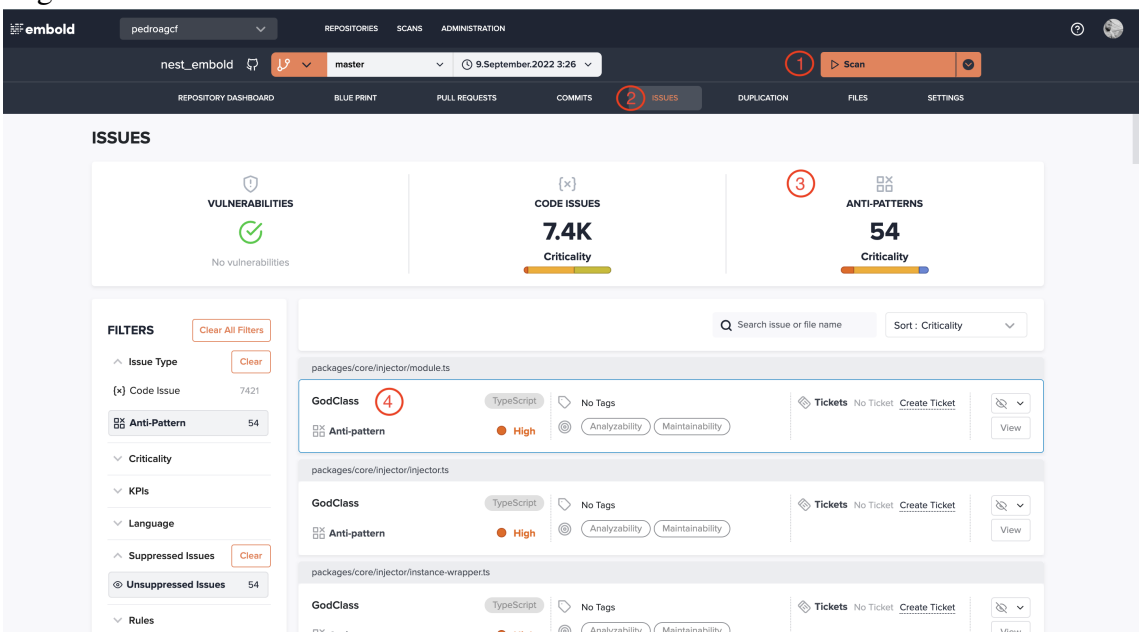
Fonte: Elaborado pelo autor.

Na Figura 4 apresentam-se os resultados obtidos após a análise do projeto. No elemento 1 observa-se o botão que ativa o *scan* do repositório. Já no elemento 2, a aba que contém os resultados. Por fim, o elemento 3 apresenta a quantidade de *code smells* encontrados, e no elemento 4, a lista detalhada dos *code smells*.

2.2 JavaScript

JavaScript é uma linguagem de programação assíncrona, dinâmica, fracamente tipada e multi-paradigma, logo suporta estilos de orientação a objetos, declarativos e imperativos (NETWORK, 2022). Inicialmente a linguagem dava suporte somente a aplicativos web no *client-side* e com o passar dos anos se estendeu ao *server-side* com a chegada do Node (NODE.JS, 2022). Além disso, se expandiu por meio de aplicações *Desktop* através do Electron *frameworks* (FOUNDATION, 2022), mobile por meio de *frameworks* como Ionic e React Native (IONIC, 2022; META, 2022). Dessa forma, JavaScript se tornou *cross-platform* e popular entre os desenvolvedores (GYIMESI *et al.*, 2019). Apesar de muitos trabalhos sobre *code smells* estarem

Figura 2 – Aba de issues do Embold



Fonte: Elaborado pelo autor.

exclusivamente no contexto de orientação a objeto (KAUR; DHIMAN, 2019), JavaScript que domina o setor web, tem seu nome vinculado ao assunto. Assim sendo, alguns estudos realizados por Johannes *et al.* (2019) sugerem 12 *code smells* diferentes dos previamente sugeridos por Fowler e Beck (1997), que são apresentados na Tabela 2. Pode-se observar no trecho de código JavaScript a tipagem fraca, mudança implícita de tipo e, os erros em forma de comentário que aparecem em tempo de execução. Embora o JavaScript ofereça flexibilidade ao desenvolvedor o código tende a apresentar erros que é percebido somente em tempo de execução.

Figura 3 – Trecho de código JavaScript

```
1  var string = 'Eu sou uma string'
2  string = 42 // Evaluates string to 42 and becomes a number
3
4  var obj = {}
5  obj.bar // Evaluates to undefined
6
7  function multiply(x){
8      return x * x
9  }
10 multiply('a') //Evaluates to NaN
```

Fonte: Elaborado pelo autor.

2.3 TypeScript

Criado em 2012 e mantido pela Microsoft, TypeScript é uma linguagem de programação que após transpilado se torna código JavaScript (MICROSOFT, 2022a). Em outras palavras, a linguagem TypeScript e um *superset* do JavaScript, ou seja, estende as funcionalidades do JavaScript, adicionando checagem de tipos. No trecho de código TypeScript pode-se notar algumas diferenças comparado ao JavaScript, dentre elas estão a tipagem forte e a checagem de tipos. Essas características contribuem para reduzir a quantidade de *bugs* e comportamentos inesperados (MICROSOFT, 2022b). Dessa forma, os desenvolvedores se sentem mais confiantes a cerca do seu próprio código, poupando tempo de investigação em código quebrado por acidente (MICROSOFT, 2022c).

Figura 4 – Trecho de código TypeScript

```
let string:String = 'Eu sou uma string'
string = 42 // Type number is not assignable to type 'string'

let obj = {}
obj.bar // Property 'bar' does not exists on type '{}'

function multiply(x: Number): Number{
    return x * x
}
multiply('a') //Argument of type 'string' is not assignable to
   parameter of type 'Number'
```

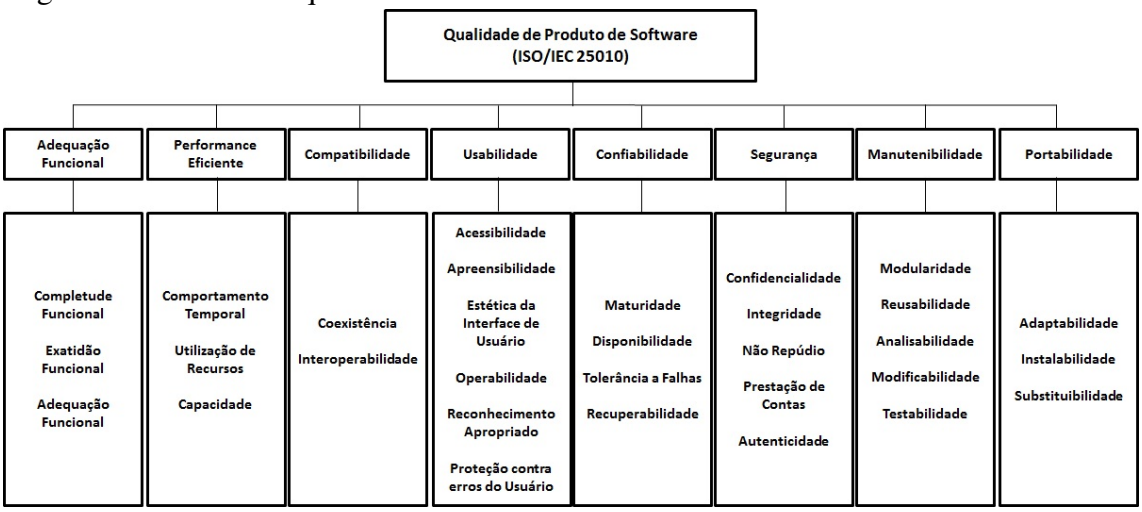
Fonte: Elaborado pelo autor.

2.4 Qualidade de Software

Segundo Crosby e Free (1979), a qualidade está diretamente ligada a conformidade com os requisitos. No entanto, esta definição é um tanto vaga quando se trata de produtos de software, visto que, para saber se há conformidade é importante definir atributos que recebam valores quantitativos e sejam inerentes ao software. Dessa forma, a ISO (2011), também conhecida como SQuaRe (*System and Software Quality Requirements and Evaluation*) se originou a partir das ISOs 9126 e 14598 e propõe um conjunto de diretrizes que são divididos entre requisitos de qualidade, modelo de qualidade, gerenciamento de qualidade, medições e avaliação, para o

contexto da qualidade de software. A Figura 5 ilustra o modelo de qualidade para características internas e externas de um produto de software (KOSCIANSKI; SOARES, 2006).

Figura 5 – Modelo de qualidade ISO/IEC 25010



Fonte: ISO (2011)

2.4.1 Ferramentas de medição de qualidade de software

Understand [1] é uma ferramenta de analise de código estático com foco em qualidade de software que possibilita a compreensão do sistema. Além disso, fornece gráficos que apresentam os resultados obtidos através das métricas que ajudam a criar e manter os códigos. O *Understand* oferece mais de 100 métricas de qualidade. Dentre elas estão: AvgCyclomatic, AvgLineCode, CountLineBlank-JavaScript, AltAvgLineCode, CountStmtDecl-Javascript Understand (2022). Uma vez que os resultados são obtidos a ferramenta permite exportar os dados em formato excel o que pode ser considerado vantajoso Gupta *et al.* (2021). Nesta secao encontra-se as métricas e suas respectivas descricoes utilizadas neste trabalho. Na Figura 6 pode-se observar a tela inicial da ferramenta.

1. *AvgLineCode*: Média do número de linhas contendo código fonte por todas as funções ou métodos aninhados.
2. *CountLineCode*: Número de linhas contendo código fonte.
3. *RatioCommentToCode*: Taxa de linhas comentadas para linhas de código.
4. *SumEssential*: Soma da complexidade essencial de todas as funções ou métodos aninhados.
5. *CountLineBlank*: Número de linhas em branco.
6. *CountLineComment*: Número de linhas contendo comentario.

[1] https://scitools.com/

7. *CountStmtDecl*: Número de declarações de instruções.
8. *CountStmtExec*: Número de instruções executaveis.
9. *AvgCyclomatic*: Média da complexidade cyclomática de todas as funções ou métodos aninhados.
10. *AvgCyclomaticModified*: Média da Complexidade ciclomática modificada para todas as funções ou métodos aninhados.
11. *AvgCyclomaticStrict*: Média da complexidade ciclomática estrita para todas as funções ou métodos aninhados.

Figura 6 – Tela inicial Understand Tool

Fonte: Understand (2022).

2.4.2 Manutenibilidade

Este atributo do modelo de qualidade de *software* é especialmente de interesse do desenvolvedor e está relacionado com a facilidade de modificação de um produto de software. Sendo assim, este atributo é composto de outros quatro subatributos, conforme atesta (KOSCIANSKI; SOARES, 2006; BEZERRA *et al.*, 2017):

– Modularidade: Nível em que um sistema ou programa de computador é composto por componentes discretos, de forma que uma mudança em um componente tem um impacto mínimo sobre os demais.

- Reusabilidade: Nível em que um produto pode ser utilizado em mais do que um sistema, ou na construção de outros produtos.
- Analisabilidade: Está relacionado com a facilidade de encontrar deficiências ou defeitos no *software*;
- Modificabilidade: Esta característica diz respeito a implementação e alteração do código Boas práticas de documentação, arquitetura e código fonte claro são eficientes na melhorias desta característica;
- Testabilidade: Permite que uma vez que o software é modificado, seja validado.

Segundo Yamashita e Moonen (2012), a manutenibilidade do software é extremamente importante visto que, promove esforço e custo significativo para o projeto. Sendo assim, o autor ainda indica os *code smells* como abordagem para avaliar alguns dos subatributos de manutenibilidade apresentados acima.

3 TRABALHOS RELACIONADOS

Na literatura foram identificados alguns trabalhos que se relacionam com este projeto, de forma mais específicas, sobre os assuntos linguagens de programação, *code smells* e manutenibilidade.

Merkel (2021), realizou um estudo envolvendo as linguagens de programação JavaScript e TypeScript com o intuito de verificar se aplicações TypeScript se sobressaem entre aplicações JavaScript em termos de qualidade de software. Sendo assim, o autor levantou 604 projetos de software e a partir disso, aplicou métricas de qualidade oriundas da API (*Aplication Programming Interface*) do SonarQube, Github e Eslint. Embora as evidências apontem que os projetos desenvolvidos em TypeScript possuam qualidade acima daqueles em JavaScript, alguns resultados contrariam as expectativas do autor. Tendo em vista os resultados obtidos e a amplitude do conceito de qualidade de software, o presente trabalho visa avaliar restritamente a qualidade em termos de manutenibilidade e para refinar os resultados, aplicando duas ferramentas de análise de qualidade do código onde somente as métricas pertencentes ao objetivo são consideradas. Dessa forma, o trabalho de Merkel (2021) irá oferecer embasamento para metodologia cientifica e os seus resultados obtidos, para fins de comparação.

O trabalho de Palomba *et al.* (2018) ressalta o impacto dos *code smells* na manutenibilidade através de investigações empíricas em larga escala, que resultaram na constatação de alguns *code smells* que afetam diretamente este pilar da qualidade de software. Apesar do trabalho do autor enfatizar o impacto dos *code smells* na manutenibilidade bem como este presente trabalho pretende fazer, a ferramenta utilizada para avaliar os *code smells* nos projetos são diferentes. Enquanto este trabalho utiliza de ferramentas de mercado para analisar os *code smells*, o autor utiliza sua própria ferramenta. Contudo, assim como o trabalho de (PALOMBA *et al.*, 2018) este presente trabalho realiza o levantamento dos *code smells* encontrados dentro do projeto.

Saboury *et al.* (2017) abordaram *code smells* na linguagem de programação JavaScript e avaliou 5 aplicações *server-side* em 537 versões diferentes com o objetivo de entender o nível de propensão a falhas. Após detectar 12 tipos de *code smells* na base previamente selecionada, o autor pontuou algumas questões para comparar arquivos com e sem *code smells* e constataram alguns tipos como os principais responsáveis pelo aumento das falhas. Além disso, o autor conduziu um questionário com 1.484 desenvolvedores JavaScript com objetivo de compreender a visão deste grupo sobre *code smells* que por sua vez, revelou diferentes tipos

de *code smells* como os que mais aumentam o risco de falha. Dessa forma, a pesquisa de Saboury *et al.* (2017) contribuiu com o presente trabalho, fornecendo a base para a análise feita pelas ferramentas que podem encontrar tipos iguais e distintos entre aplicações JavaScript e TypeScript.

Ferreira e Valente (2023) realizaram um estudo no contexto de aplicações web desenvolvidas na linguagem JavaScript com a biblioteca React, a fim de catalogar code smells específicos para esse tipo de aplicação. Além disso, ainda propuseram uma ferramenta capaz de detectar os code smells que foram previamente catalogados. Para realizar esse estudo, os autores se basearam em uma revisão cinzenta da literatura e nas informações coletadas a partir de entrevista semiestruturada, com 06 (seis) profissionais da área de desenvolvimento de software. Através da ferramenta apresentada, os autores avaliaram 10 (dez) projetos open-source que foram previamente selecionados de acordo com o nível de popularidade na plataforma Github. Contudo, os autores mantiveram o foco em aplicações web desenvolvidas exclusivamente com React e não avaliaram o impacto na manutenibilidade a partir dos code smells obtidos.

No Quadro 3 apresentam-se cinco tópicos com efeito comparativo que evidenciam a relação entre os trabalhos relacionados e o presente trabalho.

Quadro 3 – Comparativo entre os trabalhos relacionados e o trabalho proposto.

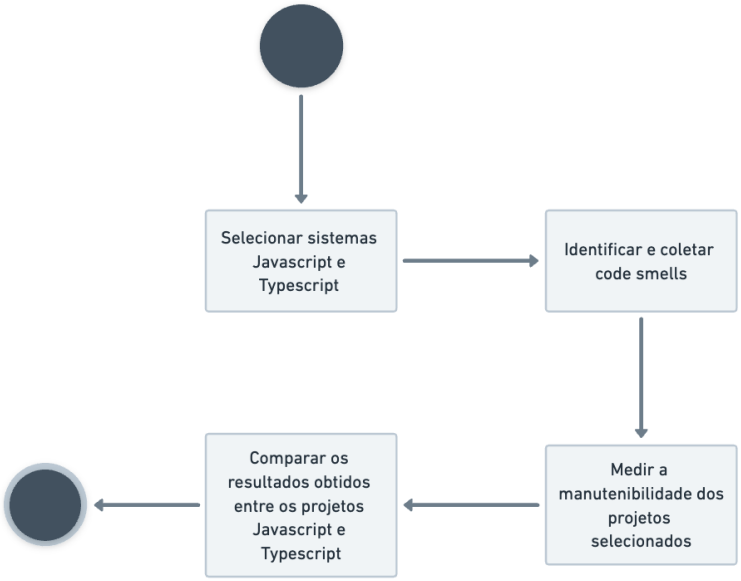
Trabalhos	**Avalia JavaScript**	**Avalia TypeScript**	**Aborda *code smells* ou manutenibilidade**	**Apresenta os tipos de *code smells***	**Compara linguagens de programação**
Trabalho Proposto	Sim	Sim	Sim	Sim	Sim
Ferreira e Valente (2023)	Sim	Não	Sim	Não	Não
Merkel (2021)	Sim	Sim	Sim	Não	Sim
Palomba *et al.* (2018)	Não	Não	Sim	Sim	Não
Saboury *et al.* (2017)	Sim	Não	Sim	Sim	Não

Fonte: Elaborado pelo autor.

4 PROCEDIMENTOS METODOLÓGICOS

Nesta seção serão apresentadas as etapas necessárias para a realização do presente trabalho. A Figura 7 apresenta uma visão geral do encadeamento entre as etapas, e as subseções a seguir descrevem mais detalhadamente cada uma delas.

Figura 7 – Passos para a realização do trabalho



Fonte: Elaborado pelo autor.

4.1 Selecionar sistemas JavaScript e TypeScript

O primeiro passo será escolher os sistemas a serem estudados neste trabalho. Para realizar esta atividade, a plataforma de mineração de dados que contem os repositórios dos sistemas JavaScript e TypeScript será o Github [1]. Uma vantagem do Github é que a plataforma disponibiliza sua própria API, que pode ser consumida de forma pública. Sendo assim, este fator promove alta flexibilidade, pois através de scripts, torna-se possível a análisar os dados obtidos a partir da API. Nesse sentido, este trabalho utilizou um *script* desenvolvido em Node [2] para desempenhar essas atividades. A primeira etapa do script consiste em obter os dados da API obedecendo os seguintes parâmetros: linguagem de programação e ordenação decrescente por nível de popularidade. Após a conclusão desta etapa, um total de 1000 repositórios da linguagem

[1] https://github.com/
[2] https://nodejs.org/en

especificada é obtida. O próximo passo do script é descartar repositórios inapropriados. Sendo assim, o script utiliza o algoritmo LDA, que no que lhe concerne, analisa a descrição de cada repositório e retorna os principais termos encontrados (ZHANG; ZHANG, 2022). A partir disso, o script desconsidera todos os repositórios que apresentarem os seguintes termos: Plugin, module, extension, API, database, framework, library. Por fim, a última etapa consiste em gerar tabelas contendo os repositórios que foram filtrados em formato de arquivo .csv. Este script é executado uma vez, recebendo como parametro a linguagem JavaScript e depois, executado novamente recebendo como parametro a linguagem TypeScript. O artefato gerado após esta atividade sao duas tabelas, contendo cerca de 350 repositorios cada. A partir deste momento uma nova rodada de analise é executada porém, de forma manual. O objetivo é investigar de forma mais minusciosa a descricao dos repositorios e as seguintes caracteristicas que o repositorio deve atender:

1. Sistemas escritos nas linguagens JavaScript e TypeScript.
2. Sistemas com mais de 1.000 linhas de código.
3. Projetos com início a partir de 2012 (ano que o TypeScript foi lançado), até 2022.

Ao fim desta etapa, chegou-se a um total de 10 projetos que utilizam a linguagem JavaScript e outros 10 projetos que utilizam a linguagem TypeScript. Na tabela 1 apresentam-se os detalhes de cada projeto.

Tabela 1 – Projetos selecionados

Projetos	Linguagem de programação	tamanho	stars	forks	watchers
arozos	JavaScript	1358909	1199	101	1199
duktape	JavaScript	76018	5554	527	5554
TILTIL$_{FRONT}$	TypeScript	71443	14	0	14
elasticlunr.js	JavaScript	51740	1952	147	1952
ImageOptim-CLI	TypeScript	43514	3328	125	3328
convnetjs	JavaScript	27806	10559	2050	10559
door-desktop	TypeScript	27528	12	1	12
WebFrontEndStack	JavaScript	26027	2325	490	2325
vscode-spell-checker	TypeScript	21756	1126	109	1126
cordova-sqlite-storage	JavaScript	17569	2131	721	2131
react-starter-kit	TypeScript	16507	21874	4140	21874
fancytree	JavaScript	15091	2668	601	2668
mapscii	JavaScript	14338	6608	227	6608
ssh2	JavaScript	11030	5103	654	5103
plato	JavaScript	9948	4537	334	4537
gmail-desktop	TypeScript	7405	682	79	682
deskreen	TypeScript	6126	14118	764	14118
shell	TypeScript	2412	4202	241	4202
FixTweet	TypeScript	1996	610	15	610
coc-snippets	TypeScript	1018	920	39	920

Fonte: elaborada pelo autor.

4.2 Identificar e coletar *code smells*

Os *code smells* serão identificados utilizando a ferramenta Embold[3] que detecta 9 tipos de *code smells* para TypeScript e detecta 4 tipos de *code smells* para projetos utilizando JavaScript. A listagem de todos os *code smells* que podem ser detectados por essa ferramenta bem como suas respectivas definições pode ser vista na Seção 2.1.

Essa ferramenta foi escolhida porque é uma ferramenta comercial que detecta *code smells* das duas tecnologias que são JavaScript e TypeScript. Além disso, ela já foi utilizada em trabalhos anteriores na literatura (HOLMBERG, 2021).

4.3 Medir a manutenibilidade dos projetos selecionados

Nessa etapa, será medida a manutenibilidade do código dos projetos escritos utilizando JavaScript e TypeScript. Para isso, serão utilizadas as ferramentas Understand[4], que é uma ferramenta que utiliza métricas para medir os atributos internos de qualidade de software e Embold, que além de detectar, oferece relatório sobre a densidade de *code smells*. Para obter essa resposta o Embold leva em consideração a seguinte formula. DC, representa a densidade de *code smells*. TC, o total de *code smells* por projeto. ELOC, métrica de qualidade que diz respeito ao número de linhas de código executáveis.

$$\text{DC} = \frac{TC}{ELOC}$$

As métricas de qualidade fornecidas pela ferramenta Understand estão descritas na seção 2.

Ambas ferramentas foram escolhidas por já terem sido utilizadas em trabalhos anteriores na literatura (HOLMBERG, 2021; GUPTA *et al.*, 2021).

4.4 Comparar projetos JavaScript e TypeScript

Nesta etapa serão comparados os resultados encontrados nos passos anteriores. Sendo assim, as métricas coletadas, o número de ocorrências de *code smells*, bem como, a densidade de *code smells*, serão registrados manualmente em forma de planilha e depois analisadas, equiparadas e exibidas através de representações visuais (gráficos e tabelas). Uma vez que os resultados individuais estão consolidados, os projetos JavaScript e TypeScript serão agrupados

[3] https://embold.io/
[4] https://www.scitools.com/

de acordo com sua respectiva linguagem de programação e a partir disso, sera apresentado a média de cada uma das respostas obtidas previamente. Dessa forma é possível equiparar os resultados e concluir o impacto na manutenibilidade em ambas linguagens.

5 RESULTADOS

Neste capitulo serão apresentados os resultados obtidos a partir das análises que foram realizadas e que estão alinhadas com o objetivo deste trabalho. Tendo em vista a forte relação entre *code smells* e a manutenibilidade de *software*, este trabalho investigou não somente a ocorrência, mas também os *code smells* que mais se repetiram de forma individual e coletiva nos projetos analisados. Além disso, este trabalho apresenta o resultado de uma série de métricas de qualidade de software aplicadas nos projetos.

5.1 Ocorrência de *code smells* nos projetos JavaScript e TypeScript

Após realizar a coleta e análise dos projetos, os *code smells Shotgun Surgery*, *Message Chain* e *Nested Callback* foram detectados. Dentre os resultados obtidos, o único *code smells* presente nas duas linguagens pesquisadas foi o *Nested Callback*. Embora o TypeScript seja definido como um *superset* do JavaScript (MICROSOFT, 2023), e através disso, adicionar recursos, ambas as linguagens de programação permitem o uso de *callbacks* (SABOURY *et al.*, 2017). A causa que explica a ocorrência desse *code smell* é o excesso de *callbacks* aninhados, implicando na manutenibilidade do código. Portanto, não está relacionado a uma característica especifica, mas a uma má decisão durante a implementação.

Contudo, o *Shotgun Surgery* e o *Message chain* foram encontrados exclusivamente na linguagem TypeScript. O *Shotgun Surgery* refere-se a um problema de código em que uma única alteração funcional requer várias modificações em diferentes partes do código. A causa deste problema está relacionado a falta de modularidade, alto acoplamento e arquitetura ruim. O *smell Message chain*, ocorre quando um objeto invoca repetidamente métodos ou acessa propriedades de outros objetos de maneira semelhante a uma cadeia (FOWLER; BECK, 1997). Isso pode levar a um código fortemente acoplado e a aumentar as dependências entre as classes.

Apesar de não serem detectados nos projetos que usam JavaScript, nenhum dos *code smells* mencionados acima estão limitados a acontecer somente em uma dessas duas linguagens de programação. Além disso, detectar *code smells* como *shotgun surgery* e *message chain* é mais desafiador em projetos JavaScript devido à tipagem dinâmica, falta de verificações em tempo de compilação e estruturas de objetos flexíveis (SABOURY *et al.*, 2017). A tipagem estática e as verificações em tempo de compilação do TypeScript oferecem melhor suporte para identificar e lidar com esses *code smells*. Diante disso, pode-se observar alguns prós e contras, ou mesmo

características individuais de uma das linguagens de programação que levam como consequência determinados *code smells*.

Lição 1: Detectar code smells em projetos JavaScript é mais difícil devido às próprias características da linguagem.

Esta seção também apresenta os dados de forma individual, facilitando o entendimento a respeito dos resultados. Nesse sentido, a tabela 4 retrata os dez projetos que foram analisados neste trabalho.

Tabela 2 – Ocorrência de *code smells* em projetos JavaScript

Projetos	Linguagem de programação	NestedCallback	ShotgunSurgery	MessageChain
arozos	JavaScript	22	0	0
duktape	JavaScript	0	0	0
elasticlunr.js	JavaScript	0	0	0
convnetjs	JavaScript	0	0	0
WebFrontEndStack	JavaScript	0	0	0
cordova-sqlite-storage	JavaScript	7	0	0
fancytree	JavaScript	2	0	0
mapscii	JavaScript	0	0	0
ssh2	JavaScript	2	0	0
plato	JavaScript	0	0	0

Fonte: elaborada pelo autor.

O primeiro fato que chama atenção é que somente quatro dos dez projetos tiveram *code smells* detectados. O projeto com maior destaque, que mais se destaca é o "Arozos"que apresenta uma contagem de vinte e dois *nested callbacks*.

Os projetos TypeScript e o número de ocorrências *code smells*, apresentam-se na tabela 3.

Tabela 3 – Ocorrência de *code smells* em projetos TypeScript

Projetos	Linguagem de programação	NestedCallback	ShotgunSurgery	MessageChain
TILTIL$_{FRONT}$	TypeScript	4	0	0
ImageOptim-CLI	TypeScript	0	0	0
door-desktop	TypeScript	0	0	0
vscode-spell-checker	TypeScript	0	0	1
react-starter-kit	TypeScript	4	0	0
gmail-desktop	TypeScript	0	0	0
deskreen	TypeScript	0	5	0
shell	TypeScript	0	6	8
FixTweet	TypeScript	0	0	0
coc-snippets	TypeScript	0	0	1

Fonte: elaborada pelo autor.

Diferente dos resultados encontrados nos projetos JavaScript, o *nested callbacks* não é maioria no grupo de projetos que utilizam TypeScript. Assim sendo, em primeiro lugar, o *code smell shotgun surgery* aparece em dois projetos distintos, totalizando doze ocorrências, e em segundo, o *Message chain*, que totaliza dez. Por fim, tem-se o *nested callback*, com um total de oito ocorrências.

Lição 2: Nested callback é o code smell mais popular entre os projetos JavaScript e TypeScript analisados.

Além dos dados apresentados de forma separada, conforme visto na tabela anterior, esta seção apresenta a ocorrência dos *code smells* nas linguagens JavaScript e TypeScript. Nesse sentido, a tabela 4 exibe esses resultados de acordo com essa perspectiva.

Tabela 4 – Soma da ocorrência de *code smells* em projetos JavaScript e TypeScript

Code Smells	JavaScript	TypeScript	Total
NestedCallback	33	8	41
ShotgunSurgery	0	12	12
MessageChain	0	10	10

Fonte: elaborada pelo autor.

De forma geral, a ocorrência de *code smells* nos projetos investigados aparece de forma bem semelhante entre as linguagens utilizadas como objeto de estudo. O motivo pelo qual os projetos JavaScript e TypeScript alcançaram esses valores diferem, pois a ocorrência de *code smells* não é uniforme entre os projetos de cada linguagem. Conforme apresentado nas tabelas 4, 3, entre os quatro projetos JavaScript, um único projeto apresentou vinte e duas ocorrências, em um total de trinta e três, enquanto entre os seis projetos TypeScript, a ocorrência de *code smells* aparece de forma distribuída, totalizando trinta ocorrências. Embora seja visto na tabela 4 a contagem de cada *code smell*, este fato isolado não classifica nenhuma das linguagens de programação como mais propensa a ter *code smell* ou pior em manutenibilidade. Este assunto é explorado com mais detalhes no tópico 5.3.

5.2 Avaliação da manutenibilidade de projetos JavaScript e TypeScript

Esta seção contempla os resultados obtidos a partir de onze métricas aplicadas aos vinte projetos investigados neste trabalho, sendo eles, dez JavaScript e dez TypeScript. A

utilização dessas métricas tem como objetivo revelar o impacto na manutenibilidade de forma individual mas também de forma coletiva, esclarecendo o quanto a manutenibilidade dos projetos JavaScript e TypeScript está impactada.

Através das métricas "AvgLineCode", "CountLineCode", "RationCommentToCode"e "SumEssential", é possível obter informações a respeito da média de linhas código do projeto, contagem das linhas de código, taxa de comentários encontrados no código e soma da complexidade essencial em todos as funções ou métodos. A tabela 5 apresenta essas métricas empregadas aos projetos JavaScript.

Tabela 5 – Coleta de métricas em projetos JavaScript

Projetos	AvgLineCode	CountLineCode	RatioCommentToCode	SumEssential
arozos	778.04	938.319	0.13	134.876
duktape	46.17	87.995	3.26	9.459
elasticlunr.js	116.19	4299.000	0.25	658.000
convnetjs	116.33	7.517	0.22	180.000
WebFrontEndStack	221.00	633.000	0.16	90.000
cordova-sqlite-storage	430.29	20.654	0.16	3.601
fancytree	316.70	82.025	0.32	10.262
mapscii	117.85	1.532	0.07	261.000
ssh2	577.55	21.947	0.08	2.677
plato	96.82	7.552	0.11	5.710

Fonte: elaborada pelo autor.

Os projetos JavaScript apresentam valores relativamente altos em relação a métrica "AvgLineCode". Sendo assim, alguns projetos tem valores acima de 200 e um projeto (arozos) atinge quase 800. Esses dados sugerem que os projetos JavaScript tendem a ter linhas de código mais longas.

Por outro lado, a métrica "RatioCommentToCode", que indica a proporção de linhas de comentário para linhas de código, revela que a maioria dos projetos JavaScript tem proporções de comentário para código relativamente baixas, com valores abaixo de 0,5. Isso sugere que os desenvolvedores em projetos JavaScript podem não priorizar comentários extensos em suas bases de código.

A métrica "SumEssential"fornece a soma dos valores de complexidade essenciais para cada projeto. Nesse sentido, a complexidade essencial diz respeito a complexidade lógica do código sem considerar o fluxo de controle. Sendo assim, os projetos JavaScript geralmente têm complexidade essencial relativamente baixa, indicando que a complexidade lógica não é significativamente alta para a maioria dos projetos. Além disso, a métrica "CountLineCode", responsável por indicar o número total de código em cada projeto, destaca que o projetos

JavaScript, como "elasticlunr.js"e "arozos", têm bases de código significativamente maiores com milhares de linhas de código. Em contraste, outros projetos, como "'convnetjs"e "mapscii", têm bases de código relativamente menores com algumas centenas de linhas de código.

De forma análoga, as métricas que foram utilizadas nos projetos JavaScript, também foram aplicadas aos projetos TypeScript. Dessa forma, tem-se os resultados apresentados na tabela 6.

Tabela 6 – Coleta de métricas em projetos TypeScript

Projetos	AvgLineCode	CountLineCode	RatioCommentToCode	SumEssential
TILTIL$_{FRONT}$	60.65	6.429	0.04	420.000
ImageOptim-CLI	33.89	610.000	0.01	76.000
door-desktop	44.21	5.217	0.08	551.000
vscode-spell-checker	70.02	31.931	0.18	4.463
react-starter-kit	49.89	2.644	0.12	182.000
gmail-desktop	79.00	1.501	0.03	146.000
deskreen	53.50	13.536	0.22	1.425
shell	185.76	10.031	0.07	1.621
FixTweet	92.09	2.026	0.17	93.000
coc-snippets	188.93	2.645	0.06	462.000

Fonte: elaborada pelo autor.

No que diz respeito aos projetos TypeScript, os resultados das métricas são apresentados na tabela 6. Sendo assim, os valores "AvgLineCode"para projetos TypeScript são geralmente mais baixos do que aqueles para projetos JavaScript, indicando que os projetos TypeScript tendem a ter códigos mais concisos em média.

A métrica “CountLineCode” mostra que os projetos TypeScript tem bases de código menores em comparação com projetos JavaScript. A maioria dos projetos TypeScript tem menos de 1.000 linhas de código, com alguns projetos menores tendo apenas algumas centenas de linhas. Em contrapartida, a métrica "RatioCommentToCode"indica que os projetos TypeScript têm taxas mais altas de comentário para código em comparação com os projetos JavaScript. Esse fator indica que os desenvolvedores parecem investir mais esforço na documentação de seu código, com muitos projetos tendo taxas de comentários acima de 0,1. Por fim, a métrica "SumEssential"mostra que os projetos TypeScript, assim como os JavaScript, geralmente têm baixa complexidade essencial. Isso sugere que os desenvolvedores TypeScript gerenciam efetivamente a complexidade lógica de seu código, resultando em bases de código mais legíveis e sustentáveis.

Lição 3: Os projetos JavaScript tendem a ter bases de código maiores com taxas de comentários relativamente baixas, enquanto os projetos TypeScript exibem bases de código menores com taxas de comentários mais altas.

A comparação dos projetos JavaScript e TypeScript com base nas tabelas fornecidas mostra diferenças no tamanho do código, proporções de comentários e complexidade essencial. Os projetos JavaScript tendem a ter bases de código maiores com taxas de comentários relativamente baixas, enquanto os projetos TypeScript exibem bases de código menores com taxas de comentários mais altas.

Conforme mencionado anteriormente, esta seção aborda os projetos sobre a perspectiva de onze métricas de qualidade. Nesse sentido, pode-se encontrar na tabela 7, o resultado da análise dos projetos JavaScript sobre as seguintes métricas: CountLineBlank, CountLineComment, CountStmtDecl e CountStmtExec.

Tabela 7 – Coleta de métricas em projetos JavaScript

Projetos	CountLineBlank	CountLineComment	CountStmtDecl	CountStmtExec
arozos	105.721	122.407	122.947	470.888
duktape	18.756	286.938	10.537	53.574
elasticlunr.js	761.000	1.093	565.000	2.067
convnetjs	202.000	382.000	358.000	1.107
WebFrontEndStack	105.000	107.000	97.000	349.000
cordova-sqlite-storage	4.084	2.958	1.909	13.328
fancytree	12.315	25.181	5.206	40.365
mapscii	259.000	114.000	292.000	743.000
ssh2	3.046	1.786	3.180	11.206
plato	1.200	807.000	3.973	15.082

Fonte: elaborada pelo autor.

Em relação a métrica CountLineBlank, os projetos JavaScript mostram uma variação considerável no número de linhas em branco, variando de algumas a mais de cem mil. Por exemplo, o projeto "arozos"tem uma alta contagem de 105.721 linhas em branco, o que pode sugerir um uso significativo de espaços em branco para formatação de código ou separação visual. Por outro lado, projetos como o "plato"têm relativamente menos linhas em branco (1.200), indicando uma estrutura de código mais concisa.

Sobre a perspectiva da métrica "CountLineComment", os projetos JavaScript exibem várias contagens de linhas de comentários, variando de algumas a mais de duzentas mil. O projeto "duktape"se destaca com uma contagem excepcionalmente alta de 286.938 linhas de comentários, indicando uso extensivo de comentários para documentação ou explicação de

código. Todavia, o projeto plato tem uma base de código substancial, mas relativamente menos linhas de comentários (807), implicando menos ênfase em comentários explícitos.

No que diz respeito aos resultados da métrica "CountStmtDecl", os projetos JavaScript demonstram diversas contagens de declarações de instrução, variando de algumas a cerca de quinhentas mil.

O projeto "elasticlunr.js"tem a maior contagem de 565.000 declarações de instrução, possivelmente indicando uma base de código com um número considerável de variáveis e funções declaradas. Em contraste, o projeto "plato"tem menos declarações de instrução (3.973), sugerindo uma estrutura de código mais simplificada e eficiente.

Por fim, os resultados da métrica "CountStmtExec"apontam que os projetos JavaScript exibem contagens variadas de execuções de instrução, variando de algumas a cerca de setecentas mil.

O projeto "mapscii"tem a maior contagem de 743.000 execuções de instruções, sugerindo uma base de código com um número substancial de funções ou loops executados. Por outro lado, projetos como "cordova-sqlite-storage"têm relativamente menos execuções de instrução (13.328), indicando uma estrutura de código potencialmente mais simples.

Semelhante ao que foi discutido sobre JavaScript, as mesmas métricas são apresentadas na tabela 8 para os projetos que utilizam TypeScript.

Tabela 8 – Coleta de métricas em projetos TypeScript

projetos	CountLineBlank	CountLineComment	CountStmtDecl	CountStmtExec
TILTIL$_{FRONT}$	759.000	279.000	1.242	597.000
ImageOptim-CLI	87.000	4.000	188.000	121.000
door-desktop	785.000	425.000	1.265	864.000
vscode-spell-checker	4.427	3.409	9.148	10.222
react-starter-kit	337.000	305.000	438.000	327.000
gmail-desktop	235.000	50.000	238.000	449.000
deskreen	2.306	2.916	2.441	3.692
shell	2.473	679.000	2.674	5.110
FixTweet	328.000	339.000	354.000	594.000
coc-snippets	215.000	100.000	842.000	1.429

Fonte: elaborada pelo autor.

Diante da métrica "CountLineBlank", os projetos TypeScript também demonstram diversas contagens de linhas em branco, com valores variando entre 2 e 785. O projeto "TILTIL-FRONT"tem a maior contagem de 759.000 linhas em branco, sugerindo uma base de código mais espaçada e organizada visualmente. Por outro lado, o projeto "deskreen"contém menos linhas em branco (2.306), sugerindo um estilo de codificação mais compacto.

Em relação a métrica "CountLineComment", os projetos TypeScript apresentam uma variação significativa, em torno de um dígito a mais de quatrocentos. Por exemplo, o projeto "vscode-spell-checker"tem 3.409 linhas de comentários, sugerindo um foco considerável na documentação do código. Por outro lado, o projeto "ImageOptim-CLI"possui apenas 4 linhas de comentário, demonstrando uma base de código mais concisa com documentação menos explícita. Em termos de declaração de instrução, que diz respeito a métrica "CountStmtDecl", as contagens de declarações de instrução também exibem variação significativa, variando de alguns a mais de oitocentos mil. O projeto "coc-snippets"tem a maior contagem de 842.000 declarações de declaração, indicando um uso potencialmente extensivo de declarações, provavelmente devido à natureza dos recursos do projeto. Entretanto, o projeto "deskreen"contém menos declarações de instrução (2.441), refletindo uma base de código mais concisa.

Por fim, os resultados da métrica "CountStmtExec"para os projetos TypeScript demonstram diferentes contagens de execuções de instrução, variando de um dígito, a mais de oitocentos mil. O projeto "door-desktop"tem a maior contagem de 864.000 execuções de instrução, possivelmente indicando uma base de código mais complexa com extensas chamadas de função ou loops. No entanto, o projeto "shell"contém menos execuções de instrução (5.110), refletindo uma estrutura de código menos complexa.

A métrica de *software Cyclomatic Complexity* é utilizada para medir a complexidade de um programa contando o número de fluxos independentes no código. Logo, valores altos indicam alta complexidade no código e promovem dificuldades na manutenção e propensão a erros no código (BUTLER, 2021). Na tabela 9, são apresentadas as métricas "AvgCyclomatic", "AvgCyclomaticModified", "AvgCyclomaticStrict"aplicadas aos projetos JavaScript.

Tabela 9 – Coleta de métricas em projetos JavaScript

Projetos	AvgCyclomatic	AvgCyclomaticModified	AvgCyclomaticStrict
arozos	2.45	2.22	3.27
duktape	1.72	1.70	1.83
elasticlunr.js	1.46	1.46	1.65
convnetjs	2.66	2.59	2.94
WebFrontEndStack	1.73	1.73	1.92
cordova-sqlite-storage	1.64	1.63	2.04
fancytree	2.62	2.52	3.60
mapscii	2.09	1.93	2.44
ssh2	2.57	2.47	3.14
plato	2.55	2.51	4.08

Fonte: elaborada pelo autor.

Os projetos JavaScript demonstram valores de Complexidade Ciclomática de mode-

rados a altos, com médias variando de aproximadamente 1,5 a 4,1. Essa informação reforça que as bases de código JavaScript geralmente têm vários pontos de decisão, loops e ramificações condicionais. Dessa forma, verifica-se que os projetos "arozos"(AvgCyclomaticStrict = 3.27) e "fancytree"(AvgCyclomaticStrict = 3.60) têm Complexidade ciclomática relativamente maior. Sendo assim, esses projetos indicam estruturas de fluxo de controle complexas e um número significativo de condições lógicas, o que pode resultar em código mais difícil de entender e manter. Por outro lado, projetos como "elasticlunr.js"(AvgCyclomatic = 1.46) e "WebFrontEndStack"(AvgCyclomatic = 1.73) possuem menor Complexidade Ciclomática. Esses dados apontam estruturas de código mais simples com menos pontos de decisão, potencialmente levando a um código mais direto e fácil de ler.

De forma semelhante, as métricas envolvendo complexidade cyclomática foram aplicadas aos projetos TypeScript e são apresentadas na tabela 10.

Tabela 10 – Coleta de métricas em projetos TypeScript

Projetos	AvgCyclomatic	AvgCyclomaticModified	AvgCyclomaticStrict
TILTIL$_{F}$*RONT*	0.93	0.92	1.03
ImageOptim-CLI	1.14	1.14	1.23
door-desktop	1.14	1.12	1.27
vscode-spell-checker	1.20	1.19	1.36
react-starter-kit	1.16	1.16	1.27
gmail-desktop	1.68	1.68	1.79
deskreen	1.10	1.09	1.15
shell	2.05	2.01	2.30
FixTweet	2.56	2.47	4.22
coc-snippets	2.31	2.28	2.61

Fonte: elaborada pelo autor.

Os projetos TypeScript geralmente exibem valores de Complexidade Ciclomática mais baixos em comparação com os projetos JavaScript, com médias variando de aproximadamente 0,9 a 4,2. Sendo assim, os projetos como TILTILFRONT (AvgCyclomatic = 0,93) e deskreen (AvgCyclomatic = 1,10) têm a menor Complexidade Ciclomática entre todos os projetos. Esses projetos provavelmente têm um fluxo de controle mais direto e menos ramificações condicionais, indicando uma base de código mais direta e potencialmente menos propensa a erros.

Em contraste, o projeto FixTweet (AvgCyclomaticStrict = 4.22) se destaca com o maior valor de *Cyclomatic Complexity* entre todos os projetos, indicando uma base de código relativamente complexa com muitos caminhos de decisão. Isso pode sugerir a presença de lógica intrincada e uso extensivo de loops ou declarações condicionais. Em média, os projetos

TypeScript tendem a ter valores de Complexidade Ciclomática mais baixos do que os projetos JavaScript. Nesse sentido, a digitação forte e a análise estática do TypeScript podem levar a bases de código mais fáceis de manter e confiáveis.

Lição 4: projetos JavaScript são maiores e potencialmente mais complexos do que os projetos TypeScript

Com base nesses resultados, o JavaScript está inclinado a possuir maior complexidade ciclomática, linhas de código, complexidade ciclomática modificada e estrita, em comparação ao TypeScript. Essas diferenças apontam que, em média, os projetos JavaScript são mais complexos e possuem mais condicionais e fluxos de execução. Em contrapartida, os projetos TypeScript tendem a ter uma contagem de linhas em branco mais alta, favorecendo uma estrutura de código mais limpa e organizada. O TypeScript também tem uma contagem maior de instruções, indicando um estilo de codificação mais expressivo e estruturado. Além disso, as bases de código TypeScript apresentam uma contagem de linhas comentadas mais alta, indicando melhor documentação e legibilidade do código. O JavaScript, no entanto, tem uma proporção maior de comentários para o código, sugerindo uma maior ênfase na documentação do projeto e legibilidade sobre a quantidade de código. É importante ressaltar que essas diferenças não são imutáveis e podem variar dependendo das práticas de codificação individuais, dos requisitos do projeto e da experiência do desenvolvedor.

5.3 Densidade de *code smells* em projetos JavaScript e TypeScript

Densidade de *code smells* refere-se à medição da concentração de *code smells* dentro de uma base de código ou projeto de software. A alta densidade de *code smells*, indica problemas na capacidade de manutenção e aumento da complexidade, enquanto a baixa densidade de *code smells* aponta uma base de código mais limpa e de fácil manutenção e com menos problemas potenciais. Nesse sentido, a tabela 11 apresenta os resultados obtidos nos projetos JavaScript.

Tabela 11 – Densidade de code smells em projetos JavaScript

Projetos	Densidade de code smells
arozos	0.023
cordova-sqlite-storage	0.338
fancytree	0.024
ssh2	0.091

Fonte: elaborada pelo autor.

A densidade relativamente baixa de *code smells* na maioria dos projetos JavaScript demonstra que os desenvolvedores prestaram atenção à qualidade do código e aos esforços de refatoração. Uma densidade de *code smells* mais baixa indica que a base de código é relativamente limpa e adere às boas práticas de codificação. Sendo assim, o projeto "arozos"se destaca com a menor densidade de *code smells* (0,023), indicando que os desenvolvedores têm sido proativos na manutenção da qualidade do código. Nesse sentido, isso pode contribuir para facilitar a manutenção do código e reduzir a dívida técnica. Entretanto, o projeto "cordova-sqlite-storage"possui a maior densidade de *code smells* (0,338) entre os projetos JavaScript. Este fator determina que pode haver áreas de preocupação na base de código que requerem atenção. Por fim, a densidade moderada de *code smells* em projetos como "fancytree"(0,024) e "ssh2"(0,091) aponta que, esses projetos sejam relativamente limpos, com um número pequeno de *code smells* que eventualmente podem removidos em busca de melhorias na qualidade de código.

De forma análoga, a densidade de *code smells* também foi medida individualmente para os projetos TypeScript.

Tabela 12 – Densidade de code smells em projetos TypeScript

Projetos	Densidade de code smells
TILTIL$_{FRONT}$	0.622
react-starter-kit	1.512
vscode-spell-checker	0.031
deskreen	0.369
shell	1.395
coc$_{snippets}$	0.378

Fonte: elaborada pelo autor.

A partir do exposto na tabela 12, é possivel observar *code smells*, com valores variando de 0,031 a 1,512. Isso sugere que a ocorrência e os tipos de *code smells* podem diferir significativamente entre os projetos TypeScript. A partir da análise dos dados, verifica-se que os projetos "react-starter-kit"(1.512) e "shell"(1.395) tem a maior densidade de code smells entre todos os projetos. Isso indica uma maior presença de code smells, o que pode exigir análises e refatorações abrangentes do código para melhorar a capacidade de manutenção do código e reduzir possíveis dívidas técnicas. Enquanto isso, o projetos como "vscode-spell-checker"(0.031), "cocsnippets"(0.378) e "deskreen"(0.369) têm densidades de *code smells* relativamente mais baixas, o que significa que eles têm menos instâncias de *code smells*. Esses projetos podem ter sido bem mantidos e aderir a práticas de codificação mais limpas. Além disso, é possível constatar que o projeto "TILTILFRONT"(0.622) possui alta densidade de *code smells*, indicando

um número significativo de *code smells* por linha de código. Esses projetos podem se beneficiar de iniciativas de refatoração de código e melhoria de qualidade para garantir a manutenção a longo prazo.

Com o intuito de medir e comparar a densidade de *code smells* nas duas linguagens de programação, a tabela 13 apresenta a média da densidade para os projetos JavaScript e TypeScript.

Tabela 13 – Densidade de *code smells* dos projetos

Linguagem de programação	Média
JavaScript	0.119
TypeScript	0.717

Fonte: elaborada pelo autor.

Lição 5: A densidade de *code smells* em projetos TypeScript é maior que em projetos JavaScript.

A partir dos dados da tabela 13 verifica-se que a densidade de *code smells* encontrados nos projetos TypeScript é superior em mais de 5 vezes em comparação com os projetos JavaScript. Embora o JavaScript apresente melhores resultados relacionados a densidade, é importante considerar que a capacidade de detecção da ferramenta, bem como, a quantidade de linhas de código entre as linguagens JavaScript e TypeScript interferem no valor dos resultados. Enquanto os projetos em JavaScript apresentam uma média de linhas de código de 281.69, o TypeScript possui 85.79. Apesar da ferramenta escolhida para detectar os code smells ser bastante versátil e uma das únicas capazes de encontrá-los nas duas linguagens de programação, o seu nível de cobertura é considerado baixo, visto que, dos seis *code smells* especificos relatados por Fard e Mesbah (2013), apenas um pode ser detectado pela ferramenta. Por outro lado, oferece uma cobertura melhor para *code smells* referentes ao TypeScript, visto que pode detectar 3 tipos diferentes. Diante disso, esses fatores aumentam a complexidade da medição.

5.4 Ameaças à validade

Ameaças à validade se referem a potenciais limitações ou fatores que podem comprometer a credibilidade e generalização dos resultados de pesquisa. Nesta seção, é identificado e discutido várias ameaças à validade da pesquisa, destacando desafios e limitações que precisam ser considerados (WOHLIN *et al.*, 2012).

Validade Interna. Em relação ao número pequeno de projetos (20), deve ser levado em consideração que poucos projetos TypeScript se encaixam no perfil estabelecido na metodologia. Para minimizar esse problema, trabalhou-se com uma quantidade reduzida de projetos, mas com os maiores números de linha de código. A coleta dos projetos ocorreu de forma híbrida. A primeira parte executada via script e a segunda parte, de forma manual. Sendo assim, as ocorrências de code smells, bem como a coleta das métricas de qualidade foram realizadas duas vezes, com o objetivo de confirmar os resultados encontrados.

Validade de Construção. Os *code smells* foram identificados automaticamente através de ferramentas, o que reduz a probabilidade de erros na detecção. No entanto, as estratégias implementadas por essas ferramentas podem representar um potencial fator de ameaça à validade, já que outras ferramentas de detecção podem adotar estratégias distintas das utilizadas pelas duas ferramentas deste estudo (MARTINS, 2017).

Validade Externa. Apesar dos code smells analisados pertençerem a diferentes domínios ou paradigmas. Os resultados servem apenas para projetos de software que utilizam JavaScript e TypeScript. Dessa forma, é preciso analisar e comparar outros paradigmas e linguagens de programação.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, realizou-se um estudo de caso para avaliar a manutenibilidade e o impacto dos *code smells* em vinte projetos open-source que utilizam as linguagens de programação JavaScript e TypeScript. Foram identificados três *code smells* conhecidos como Shotgun Surgery, Message Chain e Nested Callback. Além disso, onze métricas de qualidade foram aplicadas em cada projeto, de modo a compreender os níveis de manutenibilidade dos projetos. A estratégia utilizada neste trabalho foi comparar cada projeto, escaneando o código em busca de *code smells* e aplicando um conjunto de métricas de manutenibilidade conhecidas na literatura.

A partir da pesquisa, constatou-se que os projetos JavaScript apresentam maior ocorrência de *code smells*, sendo o Nested Callback comum entre as linguagens de programação estudadas neste trabalho. Entretanto, a ocorrência dos *code smells* Shotgun Surgery e Message Chain foi detectada exclusivamente nos projetos TypeScript. desta forma, conclui-se que o Nested Callback é o *code smells* que mais ameaça a manutenibilidade entre os projetos JavaScript e TypeScript analisados.

Verificou-se ainda, a partir das evidências apresentadas pelas métricas de qualidade de software, que em média, os projetos TypeScript apresentam números melhores em relação aos projetos JavaScript analisados. Nesse sentido, os projetos TypeScript mostraram-se menos complexos, melhor organizados e comentados. Contudo, houve alguns casos isolados onde os projetos TypeScript apresentaram índices iguais ou inferiores aos encontrados nos projetos JavaScript. Dessa forma, entende-se que apesar dos resultados favorecerem os projetos TypeScript, as práticas durante o desenvolvimento também influenciam na qualidade dos projetos.

Sobre a densidade de *code smells*, verificou-se que em projetos TypeScript, esta é superior à densidade encontrada nos projetos JavaScript. Entretanto, a discrepância da quantidade de linhas de código entre os projetos JavaScript e TypeScript dificultou a interpretação destes resultados, pois o cálculo da densidade de *code smells* é feito com base na quantidade de linhas de código e com o número de ocorrência de *code smells*. Com base neste fato, constatou-se uma dificuldade em compreender o nível de comprometimento da manutenibilidade. Apesar disso, os dados de alguns projetos isolados, que variaram tanto de forma positiva quanto negativa, levantam a ideia de que a origem dos resultados que este trabalho obteve, envolve escolhas técnicas mais profundas tomadas pelo time de desenvolvimento, ao invés de somente a decisão de qual linguagem de programação utilizar.

Trabalhos a serem realizados no futuro com base nas conclusões deste estudo incluem: i) Realizar análises em uma amostra maior de projetos implementados em JavaScript e TypeScript; ii) Replicar o estudo utilizando ferramentas que identifiquem diferentes categorias de *code smells*; iii) Desenvolver uma ferramenta com foco na detecção de *code smells* em sistemas que utilizam JavaScript e TypeScript; iv) Repetir o estudo utilizando uma variedade mais ampla de métricas para avaliar a manutenibilidade, em colaboração com desenvolvedores especializados no domínio; v) Conduzir investigações da evolução de *code smells* ao longo das publicações de *releases* em sistemas JavaScript e TypeScript.

REFERÊNCIAS

ALMASHFI, N.; LU, L. Code smell detection tool for java script programs. In: IEEE. **2020 5th International Conference on Computer and Communication Systems (ICCCS)**. [*S. l.*], 2020. p. 172–176.

ALOMAR, E. A.; MKAOUER, M. W.; NEWMAN, C.; OUNI, A. On preserving the behavior in software refactoring: A systematic mapping study. **Information and Software Technology**, Elsevier, v. 140, p. 106675, 2021.

BAQAIS, A. A. B.; ALSHAYEB, M. Automatic software refactoring: a systematic literature review. **Software Quality Journal**, Springer, v. 28, n. 2, p. 459–502, 2020.

BEZERRA, C. I.; ANDRADE, R. M.; MONTEIRO, J. M. Medidas para avaliação da manutenibilidade do modelo de features de linhas de produto de software tradicionais e dinâmicas. In: SBC. **Anais do XVI Simpósio Brasileiro de Qualidade de Software**. [*S. l.*], 2017. p. 385–399.

BOGNER, J.; MERKEL, M. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. **arXiv preprint arXiv:2203.11115**, 2022.

BUTLER, C. W. Metric based evaluation and improvement of software designs. **Journal of Software Engineering and Applications**, Scientific Research Publishing, v. 14, n. 8, p. 389–399, 2021.

BUTT, S. A.; MELISA, A.-C.; MISRA, S. Correction to: Software product maintenance: A case study. In: SPRINGER. **Computer Information Systems and Industrial Management: 21st International Conference, CISIM 2022, Barranquilla, Colombia, July 15–17, 2022, Proceedings**. [*S. l.*], 2022. p. C1–C1.

CROSBY, P. B.; FREE, Q. I. The art of making quality certain. **New York: New American Library**, v. 17, n. 4, p. 174–83, 1979.

EMBOLD. **Embold**. 2022. Disponível em: https://embold.io/. Acesso em: 13 jun. 2022.

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.

FERREIRA, F.; VALENTE, M. T. Detecting code smells in react-based web apps. **Information and Software Technology**, Elsevier, v. 155, p. 107111, 2023.

FOUNDATION, O. **Electron.js**. 2022. Disponível em: https://www.electronjs.org/. Acesso em: 25 oct. 2022.

FOWLER, M.; BECK, K. Refactoring: Improving the design of existing code. In: **11th European Conference. Jyväskylä, Finland**. [*S. l.*: *s. n.*], 1997.

FREEMAN, A. **Essential TypeScript**. [*S. l.*]: Springer, 2019.

FREEMAN, A. Understanding typescript. In: **Essential TypeScript 4**. [*S. l.*]: Springer, 2021. p. 35–41.

GUPTA, A.; SURI, B.; KUMAR, V.; JAIN, P. Extracting rules for vulnerabilities detection with static metrics using machine learning. **International Journal of System Assurance Engineering and Management**, Springer, v. 12, n. 1, p. 65–76, 2021.

GYIMESI, P.; VANCSICS, B.; STOCCO, A.; MAZINANIAN, D.; BESZÉDES, A.; FERENC, R.; MESBAH, A. Bugsjs: a benchmark of javascript bugs. In: IEEE. **2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)**. [*S. l.*], 2019. p. 90–101.

HAN, X.; TAHIR, A.; LIANG, P.; COUNSELL, S.; LUO, Y. Understanding code smell detection via code review: A study of the openstack community. In: IEEE. **2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)**. [*S. l.*], 2021. p. 323–334.

HAQUE, M. S.; CARVER, J.; ATKISON, T. Causes, impacts, and detection approaches of code smell: a survey. In: **Proceedings of the ACMSE 2018 Conference**. [*S. l.*: *s. n.*], 2018. p. 1–8.

HOLMBERG, R. V. **The Impact of AI-Based Tools on Software Development Work**. Dissertação (Master in Department of Computer science) – Lund University, Lund, Sweden, 2021.

IONIC. **Ionic framework**. 2022. Disponível em: https://ionicframework.com/. Acesso em: 25 oct. 2022.

ISO. Iec 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. **International Organization for Standardization**, v. 34, p. 2910, 2011.

JOHANNES, D.; KHOMH, F.; ANTONIOL, G. A large-scale empirical study of code smells in javascript projects. **Software Quality Journal**, Springer, v. 27, n. 3, p. 1271–1314, 2019.

KAUR, A.; DHIMAN, G. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. **Harmony search and nature inspired optimization algorithms**, Springer, p. 909–921, 2019.

KOSCIANSKI, A.; SOARES, M. d. S. **Qualidade de Software**. [*S. l.*]: NovaTec, 2006.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUÉHÉNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, Elsevier, v. 167, p. 110610, 2020.

LUO, Y.; HOSS, A.; CARVER, D. L. An ontological identification of relationships between anti-patterns and code smells. In: IEEE. **2010 IEEE Aerospace Conference**. [*S. l.*], 2010. p. 1–10.

MARTINS, J. S. **Analisando o impacto de inter-smell na manutenibilidade de linhas de produto de software**: um estudo de caso. 61 p. Monografia (TCC) – Graduação em Engenharia de Software, Universidade Federal do Ceará, Quixadà, 2017.

MERKEL, M. **Do TypeScript applications show better software quality than JavaScript applications?**: a repository mining study in github. 73 p. Monografia (TCC) – Softwaretechnik, University of Stuttgart, Stuttgart, Germany, 2021.

META. **React Native**. 2022. Disponível em: https://reactnative.dev/. Acesso em: 25 oct. 2022.

MICROSOFT. **Typescript**. 2022. Disponível em: https://github.com/microsoft/TypeScript. Acesso em: 13 jun. 2022.

MICROSOFT. **Typescript Lang**. 2022. Disponível em: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. Acesso em: 13 jun. 2022.

MICROSOFT. **Typescript Lang**. 2022. Disponível em: https://www.typescriptlang.org/why-create-typescript. Acesso em: 13 jun. 2022.

MICROSOFT. **Microsoft**. 2023. Disponível em: https://learn.microsoft.com/en-us/shows/web-wednesday/what-is-typescript. Acesso em: 21 jul. 2023.

MIU, A.; FERREIRA, F.; YOSHIDA, N.; ZHOU, F. Generating interactive websocket applications in typescript. **arXiv preprint arXiv:2004.01321**, 2020.

NETWORK, M. D. **Mozila Developer Network**. 2022. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acesso em: 25 oct. 2022.

NODE.JS. **OpenJs Foundation**. 2022. Disponível em: https://nodejs.org/en/. Acesso em: 25 oct. 2022.

NOOR, A. Improving bioinformatics software quality through incorporation of software engineering practices. **PeerJ Computer Science**, PeerJ Inc., v. 8, p. e839, 2022.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. **Empirical Software Engineering**, Springer, v. 23, n. 3, p. 1188–1221, 2018.

PRESS, I. **ANSI/IEEE Std 610.12-IEEE Standard Glossary of Software Engineering Terminology**. [*S. l.*]: IEEE, 1992.

SABOURY, A.; MUSAVI, P.; KHOMH, F.; ANTONIOL, G. An empirical study of code smells in javascript projects. In: IEEE. **2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)**. [*S. l.*], 2017. p. 294–305.

SALGADO, S. A. D. Tese, **Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics**. Porto, Portugal: [*S. n.*], 2020. 120 p.

SANTOS, J. A. M.; ROCHA-JUNIOR, J. B.; PRATES, L. C. L.; NASCIMENTO, R. S. do; FREITAS, M. F.; MENDONÇA, M. G. de. A systematic review on the code smell effect. **Journal of Systems and Software**, Elsevier, v. 144, p. 450–477, 2018.

STACKOVERFLOW. **Most popular technologies**. 2021. Disponível em: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof. Acesso em: 13 jun. 2022.

TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. **Information and Software Technology**, Elsevier, v. 125, p. 106333, 2020.

UNDERSTAND. **Understand**. 2022. Disponível em: https://www.scitools.com/. Acesso em: 25 oct. 2022.

WIRFS-BROCK, A.; EICH, B. Javascript: the first 20 years. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 4, n. HOPL, p. 1–189, 2020.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [*S. l.*]: Springer Science & Business Media, 2012.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: IEEE. **2012 28th IEEE international conference on software maintenance (ICSM)**. [*S. l.*], 2012. p. 306–315.

YENDURI, G.; GADEKALLU, T. R. A systematic literature review of soft computing techniques for software maintainability prediction: State-of-the-art, challenges and future directions. **arXiv preprint arXiv:2209.10131**, 2022.

ZHANG, Y.; ZHANG, L. Movie recommendation algorithm based on sentiment analysis and lda. **Procedia Computer Science**, Elsevier, v. 199, p. 871–878, 2022.