Analisando *code smells* no tratamento de exceções em TypeScript

Robson Oliveira Pereira Junior, Leopoldo Motta Teixeira

[1]Instituto de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7803 – 50.740-530 – Recife – PE – Brasil

(ropj, lmt)@cin.ufpe.br

Abstract. *Exception Handling is a mechanism to handle events and errors that modify the application execution flow, and this is the common way to handle these scenarios in some languages. Using this method, we can avoid unexpected code behavior and give the user a better understanding of the problems, which can help identify and fix them. Given this, errors must be handled best, avoiding the so-called 'bad smells' and other inappropriate treatments. After all, they introduce bugs, vulnerabilities, code readability issues, and more into the code. However, this task isn't simple, even for an expert developer. This paper aims to study exception handling in TypeScript projects, the occurrence of bad smells and anti-patterns, and to analyze if the use of linters impacts these occurrences.*

Resumo. *O tratamento de exceções é um mecanismo para lidar com erros e eventos que alteram o fluxo de execução de uma aplicação. Usando este método, podemos evitar ações inesperadas do código, apresentar respostas estruturadas para o usuário, e auxiliar na identificação e correção do problema. Diante disso, é fundamental que os erros sejam tratados da melhor forma evitando os chamados "bad smells" e outros tipos de tratamentos inapropriados, afinal, são inseridos, no código, bugs, vulnerabilidades, problemas de legibilidade, entre outros. Entretanto, isso não é uma tarefa simples, mesmo para um desenvolvedor experiente. Este trabalho busca estudar o tratamento de exceções em projetos de código aberto em TypeScript, a ocorrência dos bad smells e também analisar se a utilização de linters impacta nessas ocorrências.*

1. Introdução

Segundo o IEEE [IEEE 1990], podemos definir Engenharia de software como uma aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. Utilizando desses conceitos, podemos construir ferramentas cada vez mais eficientes, seguras e práticas, mesmo em sistemas de alta complexidade. Com a propagação dos mercados de tecnologia, temos não só o crescente aumento na competitividade das áreas de atuação, mas também do nível de exigência sobre a qualidade esperada dos softwares por parte dos usuários. Diante disso, é fundamental se atentar aos padrões de desenvolvimento e às boas práticas na produção das aplicações.

No entanto, todo processo é suscetível a falhas. Roger S. Pressman [Pressman 2005] (Tradução própria) aponta que a engenharia de software é uma tecnologia em camadas e uma delas é focada na qualidade do produto. Se esse conjunto de

práticas não é bem aplicado, teremos um produto de difícil compreensão, manutenção, e até mesmo suscetível a falhas de segurança. O tratamento de exceções é uma técnica conhecida e muito utilizada para melhorar a robustez do software [D.B. de Sousa 2020]. Sua principal função é garantir que os processos respondam a eventos inesperados que poderiam encerrar sua execução e também garantir um feedback útil para os mantenedores, levando à correção de problemas.

No mecanismo de tratamento de exceções, há múltiplos padrões para diversas linguagens diferentes. Cada linguagem tem suas particularidades e isso influencia na forma como devemos lidar com possíveis erros evitando más práticas como code smells. A maior parte dos estudos que analisam práticas de tratamento de exceções são focados em projetos escritos na linguagem Java como estudo de caso. O objetivo deste trabalho é desenvolver análises sintáticas e semânticas de código em projetos de código aberto para identificação e exposição de más práticas. De acordo com o nosso conhecimento atual, nenhum estudo do tipo foi encontrado usando TypeScript como foco, portanto, por este motivo decidimos utilizá-lo. O TypeScript, um superset de JavaScript, é uma linguagem moderna, de código aberto, estaticamente tipada e muito utilizada em aplicações web.

Para atingir o objetivo do trabalho, aprimoramos e utilizamos a ferramenta *exception-miner* para minerar os *smells* definidos no projeto. Mineramos 56 repositórios TypeScript dentre os mais populares do GitHub, extraindo cerca de 5000 smells. Por meio dos resultados, vimos que os *smells* mais comuns são os *empty catch* (59%) e *wrapped catch*(26%). Para finalizar, observamos o número de problemas relacionados a tratamento de exceção encontrados de projetos com e sem a utilização de linters. No entanto, verificamos que existe uma alta dispersão entre os dados do grupo de projetos com linter, além de diferenças significantes em relação a tamanhos de projeto dos dois grupos. Considerando isso, o resultado não teve significância estatística e seria necessário um retrabalho no conjunto dos dados para avaliar projetos mais equivalentes de ambas as partes. Ainda assim, vimos que dos 20 maiores projetos selecionados, 80% utilizam linters, simbolizando sua importância e uso por parte da maioria dos desenvolvedores.

Em resumo, as principais contribuições desse trabalho são: i) criar o primeiro estudo exploratório com as principais práticas de tratamento de erros em projetos TypeScript; ii) listar *code smells*, explicá-los e sugerir possíveis refatorações providenciando conhecimento para desenvolvedores acerca das práticas de tratamento de exceção em TypeScript.

2. Background

Essa seção apresenta uma visão geral dos conceitos relevantes para o tema abordado nesse trabalho. Primeiramente abordaremos a linguagem TypeScript, sua estrutura e funcionamento, depois abordaremos os conceitos sobre tratamentos de exceção e seu fluxo de ocorrência.

2.1. TypeScript

Em 2023, TypeScript foi declarada como a terceira linguagem mais popular do Github [Daigle 2023]. Sua utilização tem se expandido e consolidado ao longo dos últimos anos, o que reforça sua popularidade. Apesar disso, não se encontram materiais

acadêmicos que tratem de boas práticas em tratamento de erros especificamente em TypeScript. TypeScript é uma linguagem de código aberto que estende o Javascript adicionando tipagem estática. Ou seja, podemos especificar tipos na atribuição de valores em seu código. Tendo tipos conhecidos, conseguimos evitar diversos erros inesperados e difíceis de depurar que comumente ocorrem em JavaScript. Todavia, por ser um superset de JavaScript, após a compilação, a detecção e a manipulação de erros ainda ficará por conta do loop de eventos que organiza e executa o fluxo de processos do Javascript gerado da transpilação do código.

A Listing 1 apresenta uma estrutura de código em TypeScript que mostra a criação de uma classe de erro personalizada que pode ser utilizada para lançar exceções e construir um fluxo de feedback e logs mais eficiente. Da mesma forma, o TypeScript permite ao desenvolvedor trabalhar com tipagem estática de erros no tratamento da exceção de forma simplificada.

Listing 1. TypeScript example

```
class DivisionByZeroError extends Error {
  constructor(message: string) {
    super(message);
    this.name = "ValidationError";
  }
}

function init(value: number) {
    if(number === 0) throw new DivisionByZeroError('Check␣your␣
       value')
    return earthWeight / value
}
```

2.2. Exception Handling

O tratamento de exceção, na engenharia de software, é fundamental para garantir respostas a eventos inesperados durante a execução de programa de forma a evitar o encerramento da mesma. É essencial na garantia da qualidade do software empregando robustez e confiabilidade, permitindo que os programas se recuperem devidamente dos problemas.

2.2.1. Exception Handling Mechanisms

Exceções indicam erros e anomalias durante a execução do programa. Lidar com uma exceção depende exclusivamente se seu código pode/deve se recuperar de forma esperada. Caso a recuperação não seja possível, é interessante que se deixe o erro ser propagado pelo fluxo de execução.

Após a transpilação para JavaScript, o controle do fluxo do programa é definido pelo conjunto de declarações que o JavaScript permite fazê-lo. Essas declarações permitem que o desenvolvedor consiga contornar e ditar o fluxo de execução do código sempre que um erro ocorre. As formas como podemos fazer isso em TypeScript serão abordadas a seguir.

2.2.2. Blocos try-catch-finally

Os blocos `try-catch` dividem o código de forma a criar um caminho de execução para o caso de lançamento de exceções e, assim, permitindo o bom controle da execução do programa. De acordo com a documentação do Mozilla [MDN 2023], o bloco `try-catch` é subdividido em três blocos:

- `try`: bloco contendo uma ou mais instruções a serem verificadas.
- `catch`: bloco que contem instruções a serem executadas caso alguma exceção seja lançada durante a execução do bloco `try`.
- `finally`: contem as declarações que devem ser executadas após as execuções dos blocos anteriores. Sua principal função é a de limpeza de estados.

Por exemplo, a Listing 2 apresenta um trecho de código que ilustra a escrita de um arquivo dentro de um bloco `try-catch` para lidar com qualquer exceção que seja lançada. Por fim, demonstra como a instrução *finally* é utilizara para limpar estados, no caso, fechar a conexão com o arquivo após a execução do bloco `try-catch`.

Listing 2. try-catch-finally example

```
openMyFile();
try {
  writeMyFile(theData) // This may throw an error
} catch (e: any) {
  handleError(e) // If an error occurred, handle it
} finally {
  closeMyFile() // Always close the resource
}
```

2.2.3. Promise-based

Considerando código assíncrono, temos formas diferentes de lidar com o fluxo de execução do código. A primeira é usar o padrão de tratamento de promessas que envolve o uso do `then...catch`. A segunda é tornar o código síncrono com a utilização do `async...await` em conjunto com os blocos `try...catch`.

A Listing 3 demonstra o uso de um bloco `then-catch` promise-based. Nesse caso, as instruções internas são executadas após a finalização da função *doSomething()* que é assíncrona. Já na Listing 4 podemos ver como o mesmo código pode ser feito de maneira síncrona usando o bloco `try-catch`.

Listing 3. Código assíncrono

```
1 doSomething()
2    .then(( result ) => doSomethingElse(result))
3    .then(( finalResult ) => {
4       console.log(` result : ${ finalResult }`)
5    })
6    .catch(( err ) => failureCallback (err));
```

Listing 4. Código síncrono

```
1 async function foo() {
2    try {
3       const result = await doSomething()
4       const finalResult = await
            doSomethingElse(result);
5       console.log(` result : ${ finalResult
            }`)
6    } catch (error : any) {
7       failureCallback (error)
8    }
9 }
```

Neste trabalho, focamos na análise de tratamento de exceções para fluxos síncronos de código.

2.3. Tratamento de exceção: *Code Smells*

Um *code smell* é uma indicação superficial que normalmente corresponde a um problema mais profundo do sistema [Fowler and Beck 1999]. Ou seja, algo que precisa ser investigado e que não necessariamente é algo ruim.

Para aprofundar nosso trabalho, precisamos primeiramente definir quais *code smells* são mais comumente usados por desenvolvedores. Rocha et al. [Rocha et al. 2020] apresentam uma lista de 23 práticas ruins relacionadas ao tratamento de exceções em Java e alguns deles eram válidos para tratamentos em outras linguagens, como é o caso do TypeScript. Também, também buscamos em regras usadas pelo linter mais famoso para JavaScript, o ESLint[ESLint 2023], alguns dos smells listados. O trabalho de Chin-Yun et al. [C. Hsieh 2017] também foi fundamental tanto como base de *smells* abordados quanto para refatorações propostas.

Assim, definimos os seguintes smells do contexto de tratamento em TypeScript:

- ***empty catch***: consiste em usar o bloco de tratamento de exceção (`try-catch`) mas deixar o bloco **catch** vazio (um exemplo pode ser encontrado na Tabela 1). Isso significa capturar a exceção mas não tratá-la. Muitos desenvolvedores fazem uso desse artifício somente para evitar a parada da execução do código em caso de erro. Porém, isso também significa que o erro não está sendo tratado e nem alertado de alguma maneira, o que pode levar a um problema maior e aumento da dificuldade de detecção do problema futuramente.
- ***wrapped catch***: esse *code smell* consiste em não utilizar o erro lançado pelo **catch**. Isso significa que o a exceção e o rastreamento de pilha do erro (stack trace) não são usados no bloco onde deveriam ser tratados. Essas informações são fundamentais para identificar o erro e sua origem e também gerar feedback mais adequado ao erro exposto e, nesse caso, são perdidas.
- ***throw literal***: é um smell que consiste em lançar uma exceção sem utilizar de algum dos tipos de erros padrões do JavaScript que derivam do objeto Error também do JavaScript. Quando lançamos uma exceção através do **throw** utilizando

um objeto de *Error*, isso permite utilizar a *stack trace* referente ao erro. Porém, se utilizarmos o **throw** com um literal, perdemos essa rastreabilidade do erro.

- ***useless catch***: consiste em blocos `try-catch` onde o *catch* só faz relançar a exceção que foi obtida. Isso é equivalente a não usar o bloco `try-catch`, afinal a exceção será lançada de qualquer forma sem nenhuma alteração e ainda diminui o custo de execução do código. Então, se for relançar a exceção no *catch*, é preferível que mais algo seja feito antes como registrar o erro ou fazer limpeza de estados.
- ***error reasignment***: apesar de ter uma ocorrência mínima dentre os projetos analisados, é importante abordar sua problemática. Esse *code smell* consiste em reatribuir o erro capturado no *catch*. Isso é potencialmente problemático pois ao reatribuir a variável do erro, você pode perder sua rastreabilidade.

Tabela 1. Code smells

Nome	Exemplo	Refatoração

Useless catch	try {
	  doSomething()
	} catch (error: any) {
	  throw error
	}
		try {
		  doSomething()
		} catch (error: any) {
		  doSomethingBeforeRethrow()
		  throw error
		}

Empty catch	try { doSomething() } catch (error) {}	doSomething()

Error reasignment	try {
	  doSomething()
	} catch (error: any) {
	  error = { 'value': 2 }
	  console.error(error)
	}
		try {
		  doSomething()
		} catch (error: any) {
		  const err = new Error(
		  { 'value': 2 },
		  { cause: error }
		  )
		  console.error(err)
		}

throw literal	throw "error" // or 0, null, undefined ...	throw new Error("error")

Wrapped catch	try {
	  doSomething()
	} catch (error: any) {
	  throw new Error(
	  'Error in doSomething'
	  )
	}
		try {
		  doSomething()
		} catch (error: any) {
		  throw new Error(
		  'Error in doSomething',
		  { cause: error }
		  )
		}

3. Estudo Empírico

Diversos estudos sobre tratamentos de erros em diversas linguagens abordam o impacto que as boas práticas de codificação tem nas qualidades do software a ser desenvolvido [A. Zaidman and Shang 2018]. Diante disso, nosso estudo visa investigar as práticas de tratamento de exceção em códigos modernos usando TypeScript e a ocorrência de *smells*. Para tal, buscamos responder as perguntas de estudo descritas abaixo.

RQ1. Quais são os anti patterns mais encontrados em projetos de código aberto em TypeScript?

Seguindo nossa lista definida de *code smells* para um código moderno em TypeScript, iremos analisar as ocorrências nos projetos selecionados. Afinal, a presença desses smells pode levar a problemas futuros [A. Zaidman and Shang 2018]. Além disso, esses resultados centralizados podem ajudar desenvolvedores a terem ciência da presença desses problemas e também em como solucioná-los.

RQ2. Existe diferença na ocorrência de code smells em projetos TypeScript com ou sem *linters*?

Os *Linters* são ferramentas de análise de código estático e são auxiliares poderosos para o desenvolvedor durante a construção de um software. Sua função principal é sinalizar erros de sintaxe, erros de semântica e até estilísticos de um código visando diminuir a chance de erros não capturados. Portanto, é importante saber o impacto que essas ferramentas trazem em projetos que as utilizam, o que pode reforçar sua utilização e aprimoramento. Nesse trabalho, iremos avaliar se a presença dos linters impactam diretamente no número de problemas encontrados relacionados a tratamento de exceção.

3.1. Projetos utilizados

Nosso estudo avalia as práticas de tratamento de exceção, bugs e falhas de código em projetos de código aberto em TypeScript. Para isso, iniciamos nosso estudo coletando 35 projetos escolhidos com certa restrição para termos maior possibilidade de exito nos resultados. Essas restrições são:

- Deve ser um projeto open source disponível publicamente no GitHub;
- Deve estar ainda ativo e recebendo atualizações;
- Deve ser escrito em maior parte do código por TypeScript;
- Deve ter considerável popularidade, então buscamos por projetos com mais de 10 mil estrelas dadas por usuários;
- Deve ter no mínimo 40 contribuidores, visando buscar uma maior variedade de perfis de desenvolvedores;

Posteriormente, para análise de resultados para a RQ2, adicionamos mais 22 projetos que se enquadram nos critérios mas que não utilizam linters.

A Tabela 2 apresenta os projetos selecionados para o estudo. Nela há dados sobre os projetos, como nome, número de estrelas, número de contribuidores, número de *issues* abertas, número de *forks* do repositório e o número de *pull requests* feitos.

Tabela 2. Projetos

Nome	stars	contribuidores	issues abertas	forks	Pull Requests	linter
freeCodeCamp	384789	98	271	34854	35640	sim
developer-roadmap	268375	100	742	36352	2953	nao
vue	206534	100	356	33568	2510	nao
vscode	156882	99	7511	27324	28051	sim
TypeScript	97145	98	5394	12104	17203	sim
angular	94072	99	1394	24481	26529	sim
material-ui	91042	96	1570	31166	21600	sim
ant-design	89669	95	977	44756	16589	sim
puppeteer	86337	100	299	8944	5351	sim
storybook	82283	99	1691	8929	11996	sim
realworld	77863	80	54	7126	816	sim
excalidraw	69920	100	1262	6158	4388	sim
code-server	64594	100	314	5291	1810	sim
vite	63777	100	381	5623	5528	nao
nest	63329	97	46	7317	7783	sim
redux	60328	100	38	15267	2518	sim
socket.io	59900	218	102	10100	852	nao
grafana	59644	97	3698	11444	46582	sim
strapi	59238	98	516	7397	9795	sim
docusaurus	51915	100	263	7739	5836	sim
ui	51433	100	548	2606	946	sim
nuxt	50988	100	948	4657	5327	sim
ionic-framework	50264	98	362	13515	7342	sim
zustand	41042	100	3	1249	724	sim
appwrite	40200	328	574	3600	3621	nao
styled-components	40000	332	230	2500	1299	nao
fuels-ts	39586	45	79	1100	1125	sim
react-hook-form	39055	100	1	1945	2335	sim
query	38946	100	37	2572	2352	sim
hexo	38200	100	74	4684	1248	sim
prisma	36596	96	2889	1392	7085	sim
lerna	35260	100	345	2235	1303	sim
formik	33373	100	690	2763	1319	sim
appsmith	31026	96	3443	3298	13236	sim
solid	30700	159	64	858	411	nao
zod	29340	100	303	1009	711	sim
postcss	28100	381	19	1500	891	nao
editor.js	26230	69	530	1964	1042	sim
table	23800	397	167	3000	1123	nao
SwitchHosts	22400	40	365	2400	103	nao
Babylon.js	22160	98	79	3297	11810	sim
driver	20300	41	41	937	148	nao
OI-wiki	17800	824	116	3400	4512	nao
ionicons	17300	46	255	2100	270	nao
vConsole	16400	55	61	2900	199	nao
amis	15900	203	1002	2300	5248	nao
flatpickr	15900	246	663	1400	667	nao
chatgpt-api	15600	127	59	2100	233	nao
gitmoji	15100	105	15	795	1158	nao
qinglong	13900	52	45	2700	139	nao
brotli	13000	99	43	1200	730	nao
pinia	12100	232	28	948	823	nao
catppuccin	12100	97	34	220	272	nao
clash-verge-rev	11800	42	168	811	58	nao
fresh	11700	238	170	573	1258	nao
ky	11300	92	55	325	203	nao

Tabela 3. Capturas do conjunto de dados

Erro	Projeto	Caso
Useless catch	ionic-framework	`safeCall = (handler: any, arg?: any) => { if (typeof handler === 'function') { const jmp = config.get('_zoneGate', defaultGate); return jmp(() => { try { return handler(arg); } catch (e) { throw e; } }); } return undefined; }`
Empty catch	TypeScript	`function runCode(code: string, state: TestState, fileName: string): void { ... let sourceMapSupportModule: SourceMapSupportModule | undefined; try { sourceMapSupportModule = require("source-map-support"); } catch { // do nothing } ... }`
Error reasignment	appsmith	`if (!ExecutionMetaData .getExecutionMetaData().enableJSFnPostProcessors) { let result; try { result = fn.call(this, ...args); return result; } catch (e: any) { e = addMetaDataToError(e, name, fn.toString()); throw e; } }`
throw literal	grafana	`function labelJoinRenderer(model: QueryBuilderOperation, def: QueryBuilderOperationDef, innerExpr: string) { if (typeof model.params[1] !== 'string') { throw 'The separator must be a string'; } const separator = `"${model.params[1]}"`; return `${model.id}(${innerExpr}, "${model.params[0]}", ${separator}, "${model.params.slice(2).join(separator)}")`; }`
wrapped catch	vscode	`function safeStringify(obj: any, replacer: JSONStringifyReplacer | null): string { try { return JSON.stringify(obj, <(key: string, value: any) => any>replacer); } catch (err) { return 'null'; } }`

A Tabela 3 apresenta alguns exemplos de *code smells* retirados de alguns dos projetos utilizados no nosso estudo.

3.2. Extração de *code smells*

A primeira etapa do processo de extração era o estudo da *Abstract Syntax Tree* (AST) do TypeScript geradas pela ferramenta `tree-sitter` [Brunsfeld 2024] para elaboração de consultas que façam sentido para o nosso contexto.

Uma vez definidas as consultas de extração, precisamos utilizá-las para ler códigos em TypeScript que, posteriormente, será usado varredura dos repositórios. A ferramenta usada e aprimorada neste trabalho é o `exception-miner`. O *exception-miner* é uma ferramenta que analisa funções em Python e extrai fragmentos de código que tratam exceções usando técnicas de análise de ASTs. Para isso, ela utiliza o `tree-sitter` [Brunsfeld 2024] para formular as queries em código AST.

Essa ferramenta foi desenvolvida e mantida pelo grupo de pesquisa Software Testing and Analysis Research (STAR - Centro de Informática da UFPE), em particular pelo discente de doutorado Jairo Souza, orientado pelo Prof. Leopoldo Teixeira e co-orientador deste trabalho. Como a intenção da ferramenta é se tornar pública e disponível para a comunidade de pesquisadores e desenvolvedores interessados em estudar o aprimoramento de tratamento de exceções, este trabalho também visa contribuir com a expansão da ferramenta para uma nova linguagem, o TypeScript.

4. Análise e resultados

Nesta seção, iremos abordar os resultados encontrados com 5430 *code smells* em mais de 1.7 milhão de linhas de código distribuidos em 56 grandes projetos de código aberto no GitHub e mostrar como as práticas abordadas na seção 3 são comuns mesmo nos maiores projetos open-source com vários usuários e colaboradores.

RQ1. Quais são os anti patterns mais encontrados em projetos de código aberto em TypeScript?

Os resultados obtidos e graficamente visíveis na Figura 1, o número de blocos `try-catch` onde o bloco de catch está vazio (*empty catch*) é a ocorrência mais comum com 3213 ocorrências compondo cerca de 59% dos casos. Também obtivemos *wrapped catch* com 1423 ocorrências, *throw literal* com 742 ocorrências, *Useless catch* com 50 ocorrências e, a menor taxa de erros em *error reasignment* com somente 2 ocorrências.

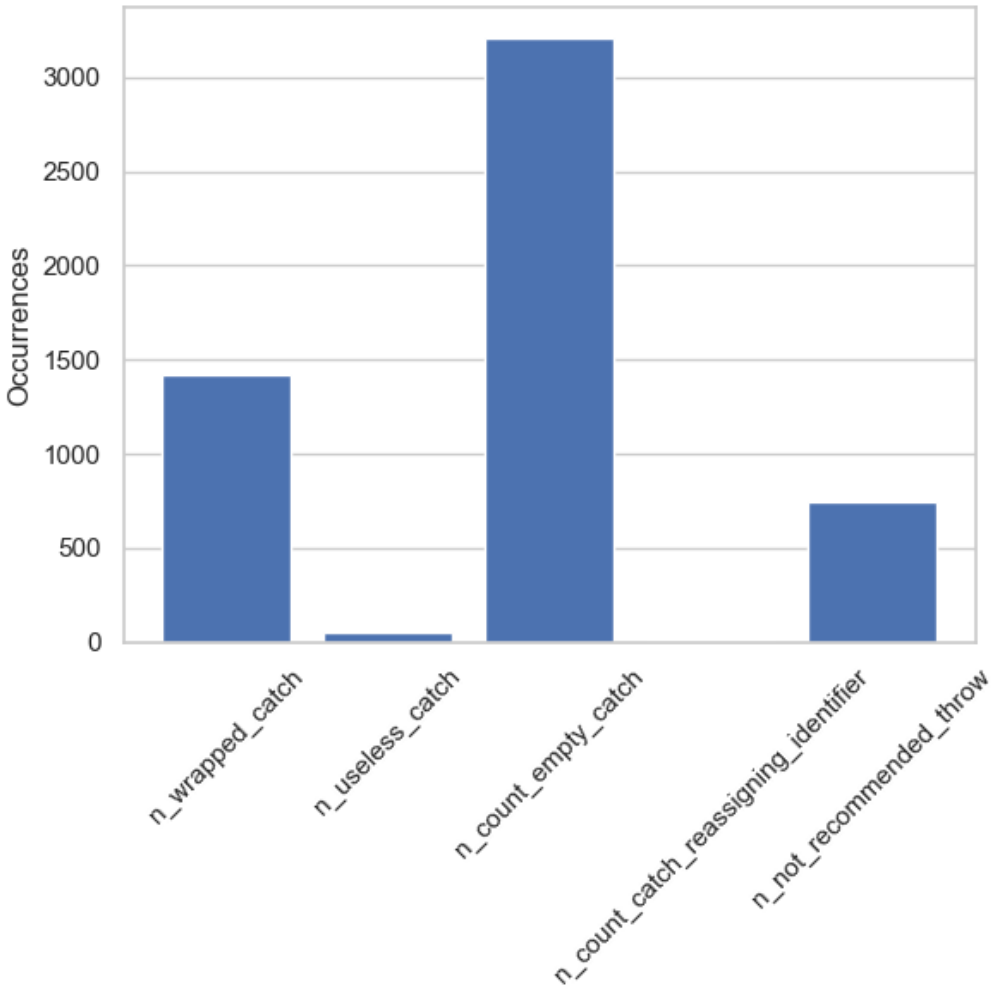

Figura 1. Número de ocorrências de erros

Com isso, podemos concluir que os erros mais comuns cometidos pelos desenvolvedores são: *empty catch* e *wrapped catch*, o que indica que os desenvolvedores comumente ignoram os erros que são capturados enquanto evitam o interrompimento da execução do código. Adicionalmente, o erro com menor número de ocorrências é o *reassigning identifier* com somente duas.

RQ2. Existe diferença significativa em code smells entre projetos com ou sem linters?

Fizemos uma varredura para detectar quantos e quais projetos utilizavam *linters* para minimizar a quantidade de erros de codigo. Em nossos resultados, verificamos que, considerando os 35 maiores projetos open source do GitHub em TypeScript, **89%** utiliza um linter para auxílio do desenvolvedor visando minimizar erros. Considerando essa discrepância, atualizamos nosso conjunto de dados com mais 21 projetos que não utilizam linters para realizar nossa avaliação.

Para analisar o impacto do uso de linters, precisamos avaliar a média de erros dos projetos com e sem o uso. Para isso, selecionamos, do conjunto de dados, aleatoriamente 25 projetos que utilizam linter(**Grupo I**) e 25 projetos que não utilizam (**Grupo II**). Fize-

mos a média dos erros dos conjuntos de projetos para cada tipo de smell abordados nesse trabalho e o resultado comparativo pode ser visto na **Figura 2**.

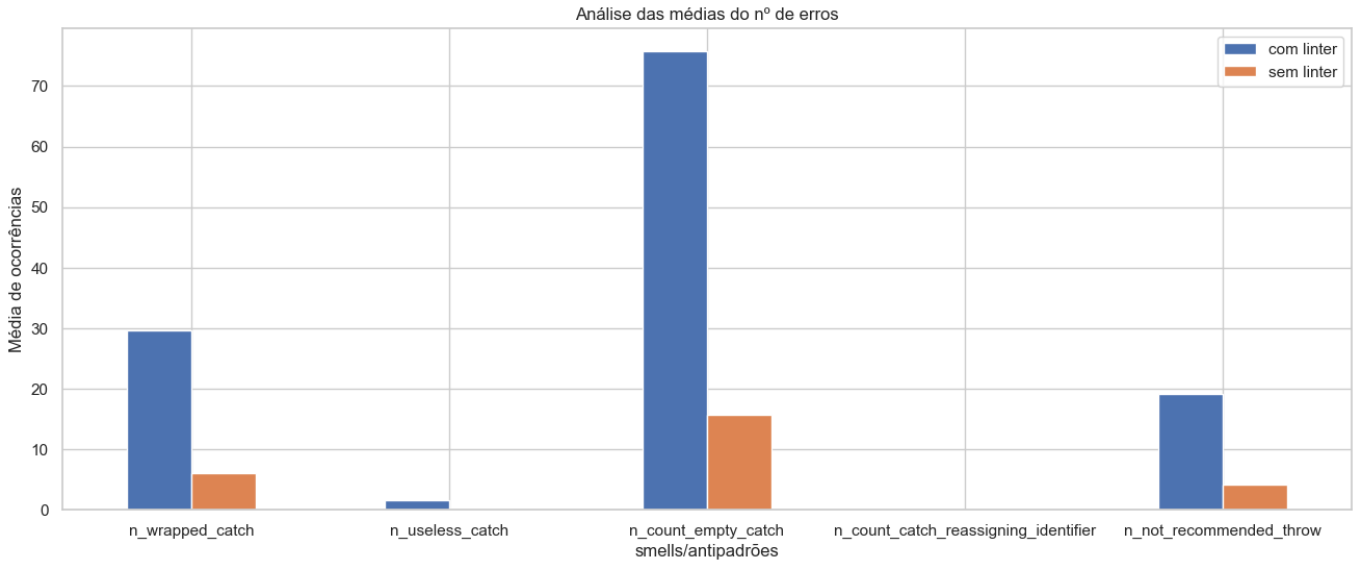

Figura 2. Comparação de projetos com/sem linters (médias)

Tabela 4. Comparação entre medidas de dispersão

Smell	Grupo	Média	Variância	Desvio Padrão
wrapped catch	C/ *linter*	29.64	2927.1	55.2
	S/ *linter*	6.04	96.5	10
useless catch	C/ *linter*	1.4	8.8	3
	S/ *linter*	0.1	0.1	0.4
empty catch	C/ *linter*	75.7	16839.7	132.4
	S/ *linter*	15.6	740.3	27.7
catch reassign	C/ *linter*	0	0	0
	S/ *linter*	0	0	0
not recommended throw	C/ *linter*	19.04	616.6	25.3
	S/ *linter*	4.2	91.4	9.7

Primeiramente é possível observar na Figura 2 que a ocorrência dos *smells* segue o mesmo padrão encontrado no **RQ1** o que mostra que os erros mais comuns nos projetos são os mesmos independente do uso de linter. Além disso, o gráfico exibe uma quantidade maior de erros nos projetos que utilizam linters o que pode sugerir que a utilização de linter não é uma garantia de resolução de erros.

Porém, analisando a variância entre os dados na Tabela 4, notamos que para os smells *wrapped catch* e *empty catch* há uma alta variação entre os projetos com uso de linters. Para entender melhor a motivação disso, observamos o desvio padrão dos dados que pode ser analisado também na Tabela 4 e pudemos perceber que pra esses *smells*, existem projetos que distoam muito da média smells encontrados considerando o **Grupo**

I inteiro. Isso é um forte indicador de que a análise geral tem uma precisão prejudicada. Adicionalmente, é possível de verificar na Tabela 2 que a maioria dos projetos que utilizam linters são projetos maiores, o que pode refletir em uma desproporção do número de smells por projeto.

Visando analisar a correlação entre o número de *smells* considerando os dois tipos de projetos (Grupos I e II), nós usamos o *Wilcoxon Rank Sum Test* [Elise Whitley 2002] para avaliar a semelhança dos resultados. Esse teste estatístico não-paramétrico é usado para testar a heterogeneidade de duas amostras independentes (em nosso estudo, as ocorrências de *smells* em projetos que usam linter ou não). Para garantir significância estatística nos resultados [Andrade 2019], iremos adotar o nível de significância 0.05 ($p < 0.05$) para esse teste.

Ademais, usamos a medida *Cliff's Delta (d)* para avaliar a diferença entre as duas distribuições. Essa medida é particularmente útil quando se deseja avaliar a magnitude da diferença entre dois conjuntos de dados. A medida varia entre -1 a 1 onde:

- O Delta próximo a **-1** indica que o primeiro grupo tende a ter valores menores que o segundo grupo;
- O Delta próximo a **1** indica que o primeiro grupo tende a ter valores maiores que o segundo grupo;
- O Delta próximo a **0** indica que não há diferença substancial entre os grupos;

Para gerar uma interpretação mais simples do tamanho de efeito do Cliff's Delta, utilizaremos uma classificação já definida na utilização dessa métrica que se divide em quatro categorias de magnitude: *insignificante*, *pequena*, *média* e *grande*.

A Tabela 5 resume os resultados da correlação de **RQ2**. A primeira coluna lista os smells abordados nesse estudo. As colunas seguintes descrevem as métricas do *Wilcoxon Rank Sum Test* (representado pelo *p-value*) e do Cliff's Delta junto à magnitude da diferença entre os conjuntos.

Tabela 5. correlation between projects with and without linters

smell	p-value	Cliff's Delta	magnitude
wrapped catch	0.005870	0.4096	média
useless catch	0.016115	0.2976	pequena
empty catch	0.006356	0.4304	média
catch reassign	0.317311	0.0400	insignificante
not recommended throw	0.002255	0.5392	grande

As células em cinza apresentam os resultados que sugerem evidência estatisticamente significativa no *Wilcoxon Rank Sum Test*, ou seja, que há uma diferença estatísticamente significativa entre as duas amostras (*p-value <0.05*).

Nós observamos que os principais erros encontrados na **RQ1** (*wrapped catch*, *empty catch*, *not recommended throw*) também apresentam aqui uma diferença mais significativa entre as amostras (*p-value* mais próximo de 0) e também com delta mais próximo da extremidade, nesse caso, do 1 (o que também indica que o **Grupo I** tende a ter

valores maiores que o **Grupo II**. A coluna *magnitude* indica o grau de diferença entre os dois grupos mostrando que, para *wrapped catch*, *empty catch*, *not recommended throw*, o grupo de projetos que utilizam linter apresenta diferença substancial sobre o o grupo dos projetos que não utilizam.

5. Trabalhos relacionados

Nesta seção, apresentamos uma breve descrição de pesquisas que foram realizadas em áreas que fundamentam nosso trabalho.

O tratamento de erros é fundamental para o bom funcionamento de um software. Visando melhorar e entender as formas como os tratamentos são feitos, diversos estudos foram criados seguindo diversas abordagens. Demonstrando a importância desse método, Lopes et al. [Lopez et al. 2021] mostram como ele pode ser fundamental para o crescimento profissional do desenvolvedor durante o aprendizado da forma como podem evitar as problemáticas causadas por erros não tratados.

Além do mais, muito trabalho de pesquisa segue o caminho de melhorar a qualidade de código através da detecção de *code smells* e sugerindo refatorações para corrigí-los [Fowler and Beck 1999]. Entretanto, poucos [C. Hsieh 2017] são realmente focados em *smells* relacionados a tratamento de erros. Dentre os mais recentes, existem trabalhos focados diretamente na investigação sobre se o tratamento é ou deve ser feito considerando específicas linguagens de programação, expondo conceitos e conhecimento que ajude os desenvolvedores a evitar cometer erros conhecidos [dos Santos 2019, C. Hsieh 2017]. Como TypeScript é um superset de JavaScript, o trabalho trazido por Luana [dos Santos 2019] ilustra dados sobre como desenvolvedores utilizam o JavaScript para lidar com erros e Chin-Yun et al. [C. Hsieh 2017] trazem possibilidades de refatoração para *code smells* nessa linguagem. Esses dados foram fundamentais para embasar esse trabalho que busca explorar o TypeScript e sua maneira de lidar com erros considerando suas possibilidades de tipagem estática.

6. Conclusão

Este trabalho apresenta um estudo exploratório sobre as práticas de tratamento de exceção em TypeScript, seus mais comuns *code smells*. Para realizar nosso estudo, coletamos *smells* de 56 repositórios open source populares do GitHub. Os resultados mostram que os desenvolvedores tendem à prática de ignorar exceções ou não tratá-las corretamente. Além disso, vimos que os problemas mais comuns de tratamento de exceção nesses repositórios são *empty catch* e *wrapped catch*.

Também fizemos uma análise comparativa sobre a utilização de linters. O resultado mostra um maior número da média de erros em projetos que utilizam linters e isso é reforçado pela análise de correlação entre os conjuntos. Contudo, também vimos que há uma alta dispersão nesse grupo de projetos que prejudica uma avaliação precisa. Ademais, considerando que dos 20 maiores projetos de código aberto analisados somente 4 não utilizam nenhum linter, sua utilização se faz relevante e é considerada pela grande maioria dos desenvolvedores.

Como trabalho futuro, planejamos expandir nossa base de dados, analisar também blocos de código assíncrono e explorar novas possibilidades de padrões em tratamento

de erros considerando tipagens estáticas dos erros. Também pretendemos aprimorar a ferramenta para outras linguagens e torná-la acessível publicamente para pesquisadores e desenvolvedores.

Referências

A. Zaidman, Y. Kamei, E. H. G. B. d. P. and Shang, W. (2018). Studying the relationship between exception handling practices and post-release defects. *Proceedings of the 15th International Conference on Mining Software Repositories*, page 564–575.

Andrade, C. (2019). The p value and statistical significance: Misunderstandings, explanations, challenges, and alternatives. *Indian J Psychol Med*, 41(3):210–215.

Brunsfeld, M. (2024). Tree-sitter. `https://tree-sitter.github.io/tree-sitter/` [Accessed: 20-10-2023].

C. Hsieh, C. L.My, K. T. H. Y. C. C. (2017). Identification and refactoring of exception handling code smells in javascript. *Journal of Internet Technology*, 18(6):1461–1471.

Daigle, K. (2023). Octoverse: The state of open source and rise of ai in 2023. `https://github.blog/2023-11-08-the-state-of-open-source-and-aif` [Accessed: 13-03-2024].

D.B. de Sousa, Maia, P. R. L. e. a. (2020). Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *J Braz Comput Soc 26*.

dos Santos, L. M. (2019). A study of javascript error handling.

Elise Whitley, J. B. (2002). Statistics review 6: Nonparametric methods. *Crit Care*, 6:509–13.

ESLint (2023). Eslint. `https://eslint.org/docs/latest/` [Accessed: 20-10-23].

Fowler, M. and Beck, K. (1999). *Refactoring - improving the design of existing code*. Addison-Wesley, 1th edition.

IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

Lopez, T., Sharp, H., Petre, M., and Nuseibeh, B. (2021). Bumps in the code: Error handling during software development. *IEEE Software*, 38(3):26–34.

MDN (2023). Control flow and error handling. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling` [Accessed: 03-03-2024].

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan, 6th edition.

Rocha, J., Melo, H., Coelho, R., and Sena, B. (2020). Towards a catalogue of java exception handling bad smells and refactorings. *Proceedings of the 25th Conference on Pattern Languages of Programs*.