Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Átomos de Confusão em JavaScript e sua influência na compreensão de código

Fillype Alves do Nascimento

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2022

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Átomos de Confusão em JavaScript e sua influência na compreensão de código

Fillype Alves do Nascimento

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

MSc Walter Lucas Monteiro
Universidade de Brasília

MSc Luís Amaral
Universidade de Brasília

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 21 de novembro de 2022

Dedicatória

Dedico este trabalho à minha família que esteve presente e colaborou de inúmeras formas com meus estudos. Também dedico aos bons amigos que fiz durante todo o período em que estive na Universidade de Brasília os quais, mais do que ninguém, acompanharam os melhores e piores momentos que se passaram durante toda a graduação. Dedico ainda aos colegas do mercado de tecnologia que me mentoraram e aconselharam profissionalmente em diferentes etapas do curso. E uma dedicatória especial a muitos docentes, funcionários e terceirizados da UnB (principalmente do Departamento de Ciência da Computação) que tiveram apreço cuidadoso por minha jornada universitária.

Agradecimentos

Agradeço a Deus por ter me concedido forças para me dedicar a esta formação. Agradeço aos meus pais Francisco e Domingas, ao meu irmão Pedro Lucas, e aos familiares mais próximos com os quais compartilhei muito do que vivi. Obrigado por sempre apoiarem emocionalmente e financeiramente meus estudos.

Reconheço aqui o empenho de todos os que se dedicam para construir e entregar um ensino público gratuito e de qualidade, focado na comunidade acadêmica, e no avanço da universidade pública. Diante dos inúmeros elogios, críticas e questionamentos, tecidos muitas vezes por quem que não conhece essa realidade, não seria justo eu, aluno da UnB, concluir minha formação sem agradecer pelo empenho das pessoas que fazem a diferença nessa universidade. Isto me incentivou a continuar nessa jornada desafiadora até o fim.

Agradeço imensamente ao Prof. Dr. Rodrigo Bonifácio, o qual tive a oportunidade de ser aluno em diversas disciplinas e foi meu orientador neste trabalho de conclusão de curso. Com grande paciência e cuidado, ele me orientou e auxiliou na idealização e execução desta pesquisa ao contribuir com seu conhecimento acadêmico, científico e de mercado, trazendo grande valor a este trabalho.

Agradeço aos colegas de curso e de profissão Caio Oliveira e Adriano Torres e ao professor Fernando Castor do Centro de Informática da UFPE por terem fornecido insumos que auxiliaram na execução da pesquisa realizada. Agradeço também ao meu amigo Christian Luis M. Costa pelo apoio a nível técnico com as tecnologias utilizadas no *survey*.

Agradeço a estas comunidades que apoiaram respondendo o *survey*: alunos da UnB, colegas de trabalho e membros da CJR. Agradecimento especial à comunidade JavaScript no Reddit que contribuiu com muitas respostas no *survey* e apresentou considerações relevantes para a qualidade da pesquisa. Agradeço a todos os produtores de conteúdo *tech*, gratuitos ou pagos, que encontrei em diversas plataformas *online*.

Por fim, deixo meu agradecimento sincero a todos os amigos de curso e de vida que me apoiaram moral e emocionalmente para que eu concluísse este trabalho. Obrigado por terem me ouvido e sido pontos de apoio.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Uma das principais tarefas a serem desempenhadas por um programador durante o desenvolvimento de *software* é a leitura e compreensão de código. Em um contexto de criação, evolução ou manutenção de *software*, um código fácil de compreender pode facilitar bastante as tarefas a serem realizadas. Faz-se necessário então que toda atividade que envolva a escrita de código tenha, em seu cerne, a preocupação com a escrita de código simples de entender, eliminando barreiras que comprometam a compreensão sua compreensão. Este trabalho apresenta uma análise dos impactos que construtos sintáticos presentes na linguagem JavaScript podem causar na compreensão de código. Esses contrutos, denominados *átomos de confusão*, foram apresentados em trechos de código a dezenas de programadores através de um *survey*, onde os mesmo eram solicitados a predizer a saída esperada quando o código for executado. Os trechos também foram apresentados em uma versão sem a presença de tais construtos. O intuito era entender se a presença deles no código levaria os programadores a não compreenderem o que o código faz. Os resultados foram analisados sob métodos estatísticos e mostraram que alguns dos átomos analisados de fato tornam o código mais confuso de entender.

Palavras-chave: Átomos de Confusão, Compreensão de código, JavaScript

Abstract

During software development, one of the main tasks to be performed by a programmer is reading and understanding code. In the context of creating, upgrading, or maintaining software, easily comprehensible software can make things a lot easier. Therefore, every effort involved in writing code must commit to writing clean code as a principle, eliminating every barrier that might affect negatively code comprehension. This study presents an analysis of the impacts of certain syntactic constructs allowed by JavaScript, on code comprehension. These constructs, so-called 'atoms of confusion', were presented to dozens of programmers via a web survey, in which they were prompted to predict the output of code snippets containing or not these constructs. The main objective was to understand if the presence of these constructs in a source code could lead the programmers to misunderstand the real behavior of the code. The results were analyzed under statistical methods and revealed that certain constructs can make source code difficult to understand, being considered atoms of confusion.

Keywords: Atoms of Confusion, Code Comprehension, JavaScript

Sumário

1 Introdução **1**
1.1 Problema . . . 2
1.2 Objetivos . . . 2
1.3 Estrutura Deste Trabalho . . . 3

2 Referencial Teórico **4**
2.1 Compreensão de Código . . . 4
2.2 Átomos de Confusão . . . 5
2.3 Átomos Analisados . . . 6

3 Metodologia **14**
3.1 Questões de Pesquisa . . . 14
3.2 Desenho Experimental . . . 14
3.2.1 O *survey* utilizado na coleta de respostas . . . 14
3.2.2 Quadrados Latinos . . . 16
3.2.3 Estratégia de Execução do *survey* . . . 18

4 Resultados **19**
4.1 Dados coletados na pesquisa . . . 19
4.1.1 Escolaridade e Experiência . . . 19
4.2 Análise das respostas dos participantes . . . 20
4.2.1 Corretude das respostas . . . 22
4.2.2 Análise estatística sobre a corretude das respostas . . . 24
4.2.3 Tempo gasto para responder as questões . . . 25
4.2.4 Análise estatística sobre o tempo gasto para responder as questões . . . 27
4.3 Respondendo as Questões de Pesquisa . . . 29
4.3.1 Resposta à Questão de Pesquisa 1 . . . 29
4.3.2 Resposta à Questão de Pesquisa 2 . . . 29
4.3.3 Resposta à Questão de Pesquisa 3 . . . 29
4.4 Ameaças à validação dos resultados . . . 30

5 Conclusão **31**
5.1 Trabalhos futuros 32

Referências **33**

Lista de Figuras

3.1 Exemplo de construção de Quadrado Latino. 17

4.1 'Escolaridade' dos participantes. 20
4.2 'Anos de experiência com programação' dos participantes. 20
4.3 Distribuição da quantidade de respostas erradas para cada participante. . . 24
4.4 Distribuição do tempo em segundos gasto para responder corretamente as questões (em escala logaritmica). 26

Lista de Tabelas

4.1 Diferença na quantidade de respostas corretas entre trechos confusos e limpos 23
4.2 Resultados dos Testes de Hipóteses relacionados à corretude das respostas. Asteriscos (*) indicam diferença estatística significante. 25
4.3 Diferença no tempo gasto (em segundos) para responder as questões 27
4.4 Resultados dos Testes de Hipóteses relacionados ao tempo gasto para responder as questões. Asteriscos (*) indicam diferença estatística significante. 28

Lista de Abreviaturas e Siglas

CJR Empresa Júnior de Computação da UnB.

UFPE Universidade Federal de Pernambuco.

UnB Universidade de Brasília.

Capítulo 1

Introdução

O conceito de 'abstração' em computação consiste em uma representação simplificada de um tópico complexo, sob determinado contexto. Em se tratando de *software*, abstrações implementadas por linguagens de programação procuram simplificar o processo de escrita e favorecer a compreensão de código por programadores. Tais códigos fonte posteriormente terão seu nível de abstração reduzido ao serem traduzidos (por compiladores ou interpretadores) para linguagem de máquina, tornando-se sequências de instruções em formato binário, o qual um computador de fato compreende. Entretanto, a linguagem de máquina é humanamente difícil de ser compreendida. Nesse contexto, o cuidado com a compreensão de código se mostra um elemento central durante a codificação, considerando-se as características da linguagem utilizada.

Cada linguagem de programação possui um nível de abstração correspondente ao seu propósito. Ainda, diversas linguagens concedem ao programador a opção de escolher entre mais de uma estrutura sintática para um mesmo fim, sendo umas estruturas mais abstratas que outras e ainda assim todas sintaticamente corretas. Contudo, ao optar pela utilização de uma sintaxe mais abstrata, o programador pode introduzir padrões de codificação que dificultem a compreensão do código escrito. Trabalhos anteriores, como o de Gopstein [1], estudaram e almejaram classificar trechos de código que demonstram ser confusos de entender mas que possuem uma versão sintaticamente equivalente menos confusa. Tais trechos de código são denominados "átomos de confusão".

Trabalhos recentes analisaram o impacto de átomos de confusão na compreensão de trechos de código escritos nas linguagens C, C++ e Java [1, 2]. Porém, características de estruturas sintáticas presentes na linguagem JavaScript expuseram a necessidade de explorar sobre o tema para a mesma. O grupo envolvido na pesquisa deste trabalho de conclusão de curso foi pioneiro na condução de estudos sobre átomos de confusão em JavaScript [3].

1.1 Problema

Um código difícil de compreender também significa um código mais difícil de manter, testar, documentar e evoluir [4]. Não importa qual a necessidade de se compreender o código, a tarefa final será mais facilmente atingida se estiver claro para o programador qual o comportamento do código em questão. Os átomos de confusão, entretando, se apresentam como um desafio à compreensão do código em diversas linguagens de programação diferentes. Neste sentido, faz-se necessário explorar os impactos dos átomos considerando as especificidades presentes em outras linguagens, como o JavaScript.

Com o intuito de expandir o entendimento sobre átomos de confusão para a linguagem JavaScript, o grupo envolvido neste trabalho de conclusão de curso conduziu, anteriormente, um experimento independente, considerando um plano experiental diferente do proposto por Gopstein [1]. Utilizando 10 dos 15 átomos de confusão apresentados na pesquisa original [1], Oliveira e Torres [3] executaram um *survey* voltado para programadores com experiência prática em JavaScript, utilizando trechos de código escritos nessa linguagem. Para cada trecho de código, o participante era solicitado a informar a saída esperada ao executá-lo. Para 9 dos 10 átomos avaliados, houve mais respostas corretas para os trechos de código onde os átomos de confusão não estavam presentes, indicando possível melhora na compreensão do código na ausência do átomo.

O JavaScript possui características próprias da linguagem que não estão presentes em C, C++ ou Java. Tal fato abre precendentes para que outros padrões de código, permitidos pela linguagem JavaScript, possam ser considerados átomos de confusão. Neste contexto, este trabalho procura expandir o experimento anteriormente realizado [3] através do estudo de átomos adicionais. Aqui, são considerados todos os 15 átomos da pesquisa de Gopstein [1] que podem ser escritos em JavaScript (isto exclui átomos que lidam com ponteiros e macros de pré-processador) e 9 novos átomos, totalizando 24 átomos analisados.

1.2 Objetivos

O objetivo geral deste trabalho é investigar se a utilização de padrões de código exclusivos da linguagem JavaScript (além dos átomos já explorados em estudos anteriores [1, 3, 2]) podem ser fonte de confusão na compreensão do código. Esta investigação se dá através de uma réplica do trabalho de Oliveira e Torres [3], porém lançando mão de um plano experimental com sutis diferenças. Foram almejadas a revisão e a evolução da infraestrutura do experimento citado [3].

As perguntas a serem respondidas são:

1. Trechos de código que contém átomos de confusão resultam em mais erros na predição do output que trechos de código onde os átomos foram removidos?
2. Trechos de código que contém átomos de confusão exigem mais tempo do programador para predizer o output?
3. Os resultados desta pesquisa reforçam os resultados apresentados no estudo anterior para um mesmo conjunto de átomos em JavaScript?

Ao responder essas perguntas, pode-se expandir a noção de átomos de confusão à linguagem JavaScript, generalizando ou descartando, neste contexto, sua influência na compreensão de código.

1.3 Estrutura Deste Trabalho

No Capítulo 2 é apresentado o Referencial Teórico da pesquisa. É explorada a relevância da compreensão de código e são delineados detalhes dos trabalhos utilizados como referência na análise de átomos de confusão.

O Capítulo 3 traz o detalhamento da metodologia utilizada neste trabalho, bem como a explicação das decisões tomadas e as melhorias trazidas ao replicar o estudo de Oliveira e Torres [3].

Os resultados da pesquisa são apresentados no Capítulo 4. Lá serão exploradas as análises realizadas sobre as respostas ao *survey* e também é onde as *Research Questions* definidas neste capítulo serão respondidas.

Concluindo este trabalho, o Capítulo 5 traz a sumarização do que foi discutido ao longo de toda a pesquisa.

Capítulo 2

Referencial Teórico

Este capítulo, nas Seções 2.1 e 2.2, busca apresentar ao leitor uma contextualização sobre os principais temas abordados neste trabalho. Ao final do capítulo, a Seção 2.3 apresenta os átomos de confusão analizados na pesquisa.

2.1 Compreensão de Código

O conceito de compreensão de código está intimamente ligado à manutenção de software [5]. Entender o que um código faz é essencial ao programador que deseja evoluir, corrigir ou até mesmo documentar esse código. A compreensão de código se mostra um elemento ainda mais importante ao considerar que pelo menos 50% do tempo empenhado nas atividades citadas anteriormente é alocado apenas para entender o código em questão [6]. É necessário mitigar as barreiras que fazem da compreensão do código algo complicado e demorado, a fim de dinamizar tarefas relacionadas à sua escrita e manutenção [7].

A compreensão de código pode ser abordada de diversas formas durante o desenvolvimento de um *software*. De fato, estudos já buscaram construir *frameworks* que tornassem o processo de facilitar a compreensão de código, durante a escrita, algo sistemático e reprodutível em passos acertados [8]. Entretanto, isto não é algo simples de ser feito, pois, mensurar o quanto um código é compreensível, pode ser feito de diversas formas [9]. Ainda, a compreensão individual de um programador sobre um código pode ter influência da sua experiência profissional ou do contexto no qual está inserido, além da sua familiaridade com o software objeto de trabalho/estudo e suas particularidades. Considerando a individualidade de programadores/equipes, podem existir diversas abordagens nas quais a compreensão de código pode ser afetada negativa ou positivamente.

Ao longo do tempo, vários tipos de alterações podem ser realizados num código fonte. O esforço empenhado em manter a qualidade desse código está ligado a diversas atividades que devem fazer parte da rotina do programador, como manter a documentação

do *software* atualizada [6]. A omissão em realizar tais atividades pode impactar diretamente na compreensão do código e da solução implementada, principalmente se o código em questão apresentar alta complexidade [6]. Neste contexto, a aplicação de boas práticas em programação [10] funciona como um direcionador para facilitar todas as tarefas relacionadas ao desenvolvimento de *software*, independente da linguagem utilizada para escrever o código e do nível de complexidade associado a ele. Compreender esse código não deveria ser uma tarefa complexa.

Linguagens de programação são como linguagens utilizadas para comunicação entre seres humanos. Elas também possuem estruturas sintáticas e semânticas. Assim como na língua portuguesa, por exemplo, uma linguagem de programação pode apresentar duas estruturas sintáticas corretas que, apesar de diferentes na escrita, são equivalentes em semântica. A preferência por uma ou outra versão na escrita do código vai depender das individualidades do programador, citadas anteriormente. Entretanto, uma dessas opções pode ser, em algum grau, mais complexa de compreender, o que pode afetar negativamente o entendimento do código e de seu comportamento por parte de outro programador (Átomo de Confusão), ou propriamente de quem o escreve, em algum outro momento.

2.2 Átomos de Confusão

A ideia de que certas estruturas sintaticamente complexas podem levar a dificuldade de compreensão do código foi apresentada como Átomos de Confusão no trabalho de Gosptein et. al.[1]. Tal trabalho se deu em torno de um *survey* que contou com a participação de programadores com vários níveis de experiência e de escolaridade. O *survey* consistia em pedir para os participantes avaliarem diferentes trechos de código escritos em C/C++ e predizer a saída esperada ao executá-los. Esses trechos foram separados em duas categorias distintas, a saber, uma com trechos considerados 'confusos' (onde a estrutura sintática complexa (átomo) estava presente) e uma com trechos considerados 'limpos' (onde o átomo em questão foi removido). Analisando as respostas, os autores mensuraram os impactos que um trecho de código confuso pode ter na compreensão de código escrito em C/C++.

Utilizando o trabalho de Gopstein et. al. como referência, o trabalho de Oliveira e Torres [3] executou um experimento para trechos de código escritos na linguagem JavaScript. Ao lançar mão da semelhança entre estruturas sintáticas de C/C++ e JavaScript, conseguiu-se reaproveitar 10 dos 15 átomos analizados no trabalho de Gopstein et. al. [1]. O critério de seleção para os átomos reaproveitados foi escolher aqueles nos quais foi

encontrada diferença estatística significativa caracterizando melhora na compreensão de código ao comparar quantidade de respostas corretas (com e sem a presença do átomo).

Assim como no trabalho de Gopstein et. al. [1], o trabalho de Oliveira e Torres também apontou melhora na compreensão para 9 dos 10 átomos analizados (escritos em JavaScript). O átomo que obteve menor vantagem ao ter seu *output* previsto apresentou um aumento de 7% mais respostas corretas quando o átomo foi removido. Já o átomo com maior maior vantagem, apresentou um aumento de 132% mais respostas corretas nos trechos de código onde o átomo foi removido.

O estudo de Oliveira e Torres [3] obteve sucesso ao reforçar o conceito de átomos de confusão, expandindo o estudo e mensurando os impactos de tais átomos, originalmente escritos em C/C++, para a linguagem JavaScript. Os átomos considerados confusos para C/C++, majoritariamente, também se mostraram confusos para JavaScript. O fato de as linguagens compartilharem estruturas sintáticas semelhantes favorece o resultado obtido ao reaproveitar os átomos da pesquisa de Gopstein et. al. [3]. Entretanto, a linguagem JavaScript possui estruturas sintáticas exclusivas (resultantes da natureza Orientada a Objetos da linguagem), inexistentes em C/C++, os quais também podem se apresentar como átomos de confusão. Estes porém, em sua maioria, não foram analisados no trabalho de Oliveira e Torres.

Este trabalho de graduação busca replicar a pesquisa realizada em [3], considerando um plano experimental bastante semelhante, mas utilizando 14 átomos adicionais além dos testados na pesquisa anterior, totalizando 24 tipos de átomos analisados. Ao todo, 15 dos 24 dos átomos utilizados aqui também foram avaliados no trabalho de Gopstein. O critério de seleção dos 9 novos átomos considerou características particulares da linguagem JavaScript, tais como: o modo como a linguagem permite lidar com acessos de propriedades em objetos, o uso de *arrow functions*, o acesso a elementos de um *array*, entre outros. A explicação sobre os átomos analisados pode ser encontrada na seção 2.3.

2.3 Átomos Analisados

Os 24 átomos analisados neste trabalho incluem os 10 átomos analisados em [3] e 14 átomos adicionais. Pelo menos 6 dos 24 átomos apresentam estruturas sintáticas exclusivas da linguagem JavaScript.

1. *Implicit Predicate*: este átomo assume a existência de um predicado a ser avaliado mesmo que ele não esteja explicitamente declarado em código. Por exemplo:

```
1    if (9 % 3) {
2        console.log('rest');
3    }
```

No programa acima, o resultado da operação de resto vai ser interpretado como *true* se for diferente de zero, e *false* caso contrário. Para tornar o predicado explícito:

```
if (9 % 3 != 0) {
    console.log('rest');
}
```

2. *Infix Operator Precedence*: o JavaScript possui dezenas de operadores para os quais existem vários níveis de precedência e associatividade, as quais podem não ser óbvias para o programador. Por exemplo:

```
0 && 1 || 2
```

pode ser reescrito assim:

```
(0 && 1) || 2
```

3. *Post Increment/Decrement*: os operadores de pós incremento/decremento alteram o valor da variável em 1 e retornam o valor antigo. São duas operações onde a ordem de execução pode não ser clara. Por exemplo:

```
x = y++
```

pode ser reescrito explicitando as operações realizadas:

```
x = y
y += 1
```

4. *Pre Increment/Decrement*: os operadores de pré incremento/decremento são semelhantes aos de pós incremento/decremento, porém, primeiro alteram o valor da variável em 1 e depois retornam o novo valor. Por exemplo:

```
a = ++b
```

pode ser reescrito explicitando as operações realizadas:

```
b += 1
a = b
```

5. *Constant Variables*: representam uma camada de abstração ao não avaliarem um valor propriamente dito, mas sim uma variável. Por exemplo:

```
let a = 7
console.log(a)
```

pode ser reescrito sem a utilização da variável:

```
console.log(7)
```

6. *Indentation no Braces*: ao omitir chaves e não apresentar indentação, o escopo de certos blocos de código pode ficar confuso. Por exemplo:

```
if (x)
if (y)
z = z - 1;
else
z = z - 2;
```

pode ser reescrito inserindo chaves para delimitar o escopo das estruturas:

```
if (x) {
if (y) { z = z - 1; }
else { z = z - 2; }
}
```

7. *Ternary Operator*: executa, em uma única expressão, controle de fluxo que é comumente realizado por uma estrutura do tipo *if...else*, atribuindo à variável um ou outro valor. Por exemplo:

```
f = (a == 1) ? 'verdadeiro' : 'falso';
```

pode ser reescrito como uma estrutura condicional padrão:

```
if (a == 1) {
    f = 'verdadeiro';
} else {
    f = 'falso';
}
```

8. *Arithmetic As Logic*: operadores aritméticos resultam em valores não *booleanos* mas são interpretados como tal em certas situações. Por exemplo:

```
if ((a - 2) * (b - 3)) {
    console.log('inside')
}
```

pode ser reescrito utilizando operadores lógicos, não deixando dúvidas sobre a semântica:

```
if ((a !== 2) && (b !== 3)) {
    console.log('inside')
}
```

9. *Comma Operator*: utilizado para inserir mais de uma operação na mesma linha, causando confusão por ser pouco utilizado e ter um comportamento por vezes desconhecido. Por exemplo:

```
x = ( y += 2; y )
```

pode ser reescrito para explicitar as operações realizadas:

```
y += 2;
x = y;
```

10. *Assignment As Value*: o operador de atribuição altera o valor de uma variável mas também retorna algo. Por não ser tão comum utilizar esse retorno, pode ser confuso ver sua utilização. Por exemplo:

```
v1 = v2 = 1
```

pode ser reescrito para explicitar todas as operações:

```
v2 = 1
v1 = v2
```

11. *Logic As Control Flow*: operadores lógicos são utilizados para conjunção ou disjunção lógica, mas devido ao *short circuit*, podem ser utilizados para execução condicional de um bloco de código. Por exemplo:

```
a && f();
```

pode ser reescrito para expressar a execução da função se o valor da variável for válido para tal:

```
if (a) {
    f();
}
```

12. *Repurposed Variables*: quando uma mesma variável é utilizada para diferentes propósitos ao longo do programa, tornando difícil prever seu real significado. Por exemplo:

```
let x = 1;
x = "Hello"
```

pode ser reescrito pra não mudar o valor da variável:

```
let x = 1;
let y = "Hello";
```

13. *Dead, Unreachable, Repeated*: representa código repetido ou não executado, ou seja, sem nenhum efeito funcional. Pode causar confusão por motivos óbvios: se o código está ali, deveria haver um motivo para ele ser executado. Por exemplo:

```
let v1 = 7;
v1 = 5
```

a variável v1 possui duas atribuições, uma logo após a outra, tornando a primeira atribuição uma operação inútil. Logo, pode ser reescrito com apenas a segunda atribuição:

```
let v1 = 5;
```

14. *Change Literal Encoding*: representa uma mudança na representação de um número quando se pretende utilizar outro formato. Formatos menos convencionais podem ser confusos. Por exemplo:

```
let x = 013;
console.log(x);
```

o código acima possui um número em representação octal (precedido de um 0). O valor da variável x pode ser confundido com um inteiro em formato decimal. Para evitar a confusão, pode-se reescrever o mesmo número diretamente em representação decimal:

```
let x = 11;
console.log(x);
```

15. *Ommited Curly Braces*: representa a omissão das chaves ao definir um bloco de código de alguma estrutura. Por exemplo:

```
let a = 15;
if (a) funcX(); funcY();
```

pode ser confuso definir ao certo o escopo da estrutura condicional acima, que pode ser reescrita inserindo as chaves:

```
let a = 15;
if (a) { funcX(); } funcY();
```

16. *Type Conversion*: o JavaScript realiza a conversão implícita de tipos quando necessário. O tipo resultante dessa conversão, entretanto, pode não ser claro para o programador. Por exemplo:

```
let v1 = 2 + { p1: 2 };
```

o código acima pode ser reescrito explicitando a conversão a ser realizada:

```
let v1 = (2).toString() + { p1: 2 }.toString();
```

17. *Indentation With Braces*: semelhante ao átomo *Indentation with Braces*, aqui também não é apresentada indentação, mas as chaves não são omitidas, delimitando claramente o escopo. Ainda assim, pode causar confusão. Por exemplo:

```
if (x) {
    if (y) { z = z - 1; }
else { z = z - 2; }
}
```

Ajustar a indentação do código acima pode auxiliar na compreensão do escopo das estruturas condicionais:

```
    if (x) {
        if (y) { z = z - 1; }
        else { z = z - 2; }
    }
```

18. *Automatic Semicolon Insertion*: por ser opcional na maior parte dos casos, o ponto-e-vírgula por vezes pode ser omitido onde não deveria, deixando o compilador em dúvida sobre como proceder. Por exemplo:

```
    let v1 = 9
    (v1 += 3)
    console.log(v1)
```

    o código acima produz um erro. Para que o código entre parêntesis seja interpretado como um bloco independente, ele deve ser precedido de um ponto-e-vírgula:

```
    let v1 = 9;
    (v1 += 3);
    console.log(v1);
```

19. *Property Access*: existem duas formas de se acessar propriedades de objetos em JavaScript: utilizando a notação de ponto ou acessando as propriedades como na estrutura de dados Dicionário. A segunda forma não é tão convencional para esse propósito, o que pode causar confusão. Por exemplo:

```
    let a = { p1: 7, p2: 14 }
    a['p1'] = 13
```

    no código acima, utilizar a notação de ponto, como na maioria das linguagens orientadas a objetos, pode reduzir a confusão:

```
    let a = { p1: 7, p2: 14 }
    a.p1 = 13
```

20. *Arrow Function*: representa a utilização preferencial das *arrow functions* do JavaScript em relação à declaração padrão de funções anônimas. Por exemplo:

```
    let F = x => x + 2;
    let y = F(6);
```

    o código acima pode ser reescrito utilizando uma declaração mais próxima à padrão para funções anônimas em JavaScript.

```
    let F = function(x) {
        return x + 2;
    }
    let y = F(6);
```

21. *Array Spread*: este operador é bastante útil para performar diversas tarefas como cópia de elementos de um array para outro ou a concatenação de arrays. Possui uma notação pouco usual, que pode causar confusão por falta de familiaridade. Por exemplo:

```
let a1 = [1,2,3]
let a2 = [4,5]
let a3 = [..a1, ..a2]
```

o operador ... no código acima insere no array *a3* todos os elementos dos *arrays a1* e *a2*. Para facilitar esta operação, o código pode ser reescrito desta forma:

```
let a1 = [1,2,3]
let a2 = [4,5]
let a3 = [a1[0], a1[1], a1[2], a2[0], a2[1]]
```

22. *Object Spread*: bastante similar ao átomo *Array Spread*, o operador de ... permite aproveitar os valores já existentes em outro objeto, sobrescrevendo propriedades já existentes. Por exemplo:

```
let obj1 = { p1:7, p2:9, p3:12 }
let obj2 = {..obj1, p3: 14}
```

o código acima pode ser reescrito explicitamente quais propriedades serão copiadas para o novo objeto:

```
let obj1 = { p1:7, p2:9, p3:12 }
let obj2 = { p3:14 }
obj2.p1 = obj1.p1
obj2.p2 = obj1.p2
```

23. *Array Destructuring*: utiliza a ideia de casamento de padrões ao acessar valores de um *array*. Exige que se tenha conhecimento sobre o tamanho do *array*. É uma forma pouco usual de acessar os elemtos, o que pode causar confusão. Por exemplo:

```
let [v1,,v3] = [1,2,3]
console.log(v1, v3);
```

o código acima pode ser alterado para realizar o acesso aos valore do *array* diretamente:

```
let v1 = [1,2,3]
console.log(v1[0], v1[2]);
```

24. *Object Destructuring*: segue a mesma lógica do átomo *Array Destructuring*, porém aplicada a propriedades de objetos, fazendo casamento de padrões com os nomes das propriedades. Por exemplo:

```
let obj = { p1:4, p2:5 };
let {p1, p2, p3 = 8} = obj;
```

o código acima pode ser alterado para acessar diretamente os valores de *obj*:

```
let obj = { p1:4, p2:5 };
let v1 = obj.p1;
let v2 = obj.p2;
let v3 = 8;
```

Capítulo 3

Metodologia

Neste capítulo é apresentada a metodologia utilizada para mensurar o impacto de 24 átomos de confusão na compreensão de trechos de código escritos na linguagem JavaScript.

3.1 Questões de Pesquisa

O objetivo principal desta pesquisa é investigar o impacto de átomos de confusão na compreensão de código escrito em JavaScript. Essa investigação ém feita através da reprodução do estudo feito por Oliveira e Torres [3], considerando um plano experimental semelhante e átomos adicionais. Para atingir o objetivo, assim como no estudo reproduzido, foi conduzido um *survey* onde os resultados basearam a investigação e irão ajudar a responder as *Research Questions* apresentadas no Capítulo 1 deste trabalho.

3.2 Desenho Experimental

O experimento foi delineado em torno de um *survey* que contou com a participação de programadores com diferentes níveis de experiência e escolaridade. O intuito é analisar a quantidade de respostas corretas e o tempo gasto para responder as questões quando comparados trechos de código com e sem a presença dos átomos de confusão.

3.2.1 O *survey* utilizado na coleta de respostas

Os átomos escolhidos como candidatos desta pesquisa foram inseridos em uma aplicação *web* para a coleta das respostas. A aplicação utilizada foi reaproveitada do trabalho de Oliveira e Torres [3], onde foram feitas algumas alterações de forma a atender o desenho experimental atual considerando os átomos adicionais. Além de atender os objetivos do desenho experimental atual, as alterações realizadas na aplicação almejaram evoluir a

infraestrutura utilizada, mas mantendo a arquitetura do *survey* inicialmente utilizada no experimento anterior.

Na primeira etapa do *survey* era apresentado ao usuário um *modal*[1] com instruções para responder o *survey*, informando como o mesmo funcionava. Aqui era solicitada do participante atenção máxima para responder às perguntas e também que não se utilizassem fontes externas (por exemplo, interpretadores online) durante a participação. Para cada página com questões apresentadas ao participante, eram mantidos rastreios para saber se o mesmo mudou de abas ou janelas.

A próxima etapa do *survey*, apresentada ao fechar o *modal*, consistia em uma breve coleta de dados demográficos do participante, onde o mesmo era perguntado sobre sua idade, nível de escolaridade e experiência com programação. O e-mail, apesar de solicitado, era um dado opcional e requisitado apenas para encaminhar os resultados ao participante após a conclusão do estudo aqui apresentado. O usuário era solicitado também a marcar uma *checkbox*[2] informando ciência de que os dados coletados seriam utilizados apenas para fins acadêmicos.

A partir de então, o *survey* de fato era iniciado. Eram apresentadas, sequencialmente, 12 questões para cada participante, cada questão com um trecho de código. Na página de cada questão havia um campo de texto onde o participante deveria escrever a resposta contendo a saída esperada após executar o trecho de código apresentado. Caso o participante não soubesse ou não quisesse responder à questão, havia, na mesma página, um botão *"I do not know"* ("Eu não sei"), o qual tratava a resposta do participante como incorreta e o direcionava para a próxima questão.

Cada trecho de código apresentado nas questões estava em formato de imagem, extraídas como capturas de tela a partir de um editor de texto. Com isso, objetivou-se mitigar que o participante pudesse facilmente copiar o código e utilizar fontes de apoio externas para responder a questão. Ao responder uma questão e clicar no botão *"Next"* ("Próximo"), a resposta do participante era registrada na base de dados e o mesmo era direcionado à próxima página contendo outro trecho de código, até que se completassem as 12 questões.

Como no *survey* conduzido por Oliveira e Torres [3], não foram fornecidos aos participantes *feedbacks* sobre o tempo gasto para responder cada questão. Entretanto, como forma de demonstrar apreço às sugestões apresentadas pelos participantes da primeira execução do *survey*, após responder as 12 questões, uma página contendo as respostas do participante e a resposta esperada era apresentada para que o mesmo pudesse, de imediato, ponderar sua performance.

[1] `https://blog.hubspot.com/website/modal-web-design`
[2] `https://developer.mozilla.org/pt-BR/docs/Web/HTML/Element/Input/checkbox`

3.2.2 Quadrados Latinos

Após selecionar os 24 átomos candidatos participantes da pesquisa, foram escritos os trechos de código separados em duas categorias distintas, a saber, os que continham átomos de confusão os sem a presença dos átomos. Como cada átomo tinha, portanto, sua versão "confusa" e sua versão "limpa", totalizando 48 trechos de código a serem analisados pelos participantes. Assim como no trabalho de Oliveira e Torres [3], houve a preocupação com o tempo gasto para responder o *survey* em sua totalidade. Logo, dos 48 trechos de código disponíveis, os participantes eram convidados a responder 12, dois trechos a mais que na pesquisa anterior. Dos 12 átomos apresentados a cada participante, 6 continham átomos de confusão e os outros 6 não continham.

A fim de minimizar as chances de o participante perceber se estava respondendo um trecho de código confuso ou limpo, a ordem em que as 12 questões eram apresentadas era aleatória. Além disso, era necessário reduzir as chances de um participante ser influenciado por suas respostas a questões anteriores. A fim de previnir que um mesmo participante respondesse a versão confusa e também a versão limpa de um mesmo átomo, foi lançada mão da utilização do *design* dos Quadrados Latinos [11], como Oliveira e Torres fizeram no experimento anterior [3].

O desenho experimental dos Quadrados Latinos, aplicado ao caso em estudo, consiste na criação de uma matriz 2x2 onde são distribuídas as questões a serem respondidas. Cada linha da matriz representa um participante e cada coluna representa a presença ou ausência de átomo de confusão. As questões são distribuídas de forma que nenhum valor se repete na mesma linha ou coluna, formando conjuntos complementares. Por exemplo, um participante X, que representa a primeira linha de um quadrado, é solicitado a responder os trechos [1,2,3,4,5,6] e essas questões contém átomos de confusão. Esse mesmo participante só será solicitados a responder as versões limpas dos átomos [7,8,9,10,11,12]. Enquanto isso, um participante Y, que representa a segunda linha do mesmo quadrado do participante X, irá responder as versões limpas dos átomos [1,2,3,4,5,6] e responderá as versões confusas dos átomos [7,8,9,10,11,12]. Dessa forma, todas os 24 trechos de código estarão presentes num quadrado e serão respondidas ali apenas uma vez. A figura Figura 3.1 traz uma representação visual do conceito dos Quadrados Latinos para 24 trechos de códigos que podem estar presentes em um quadrado.

Um problema que pode ocorrer na construção dos quadrados é quando um participante deixa a pesquisa sem responder todas as suas questões. Ao fazer isso, o quadrado é invalidado e desconsiderado dos resultados, resultando em perda de respostas. A fim de mitigar tais perdas, foi utilizado um método de imputação de dados. Ao invés de descartar completamente um quadrado se um dos participantes envolvidos nele parasse de responder a pesquisa antes de completar todas as 12 questões, ele pode ser parcialmente aproveitado.

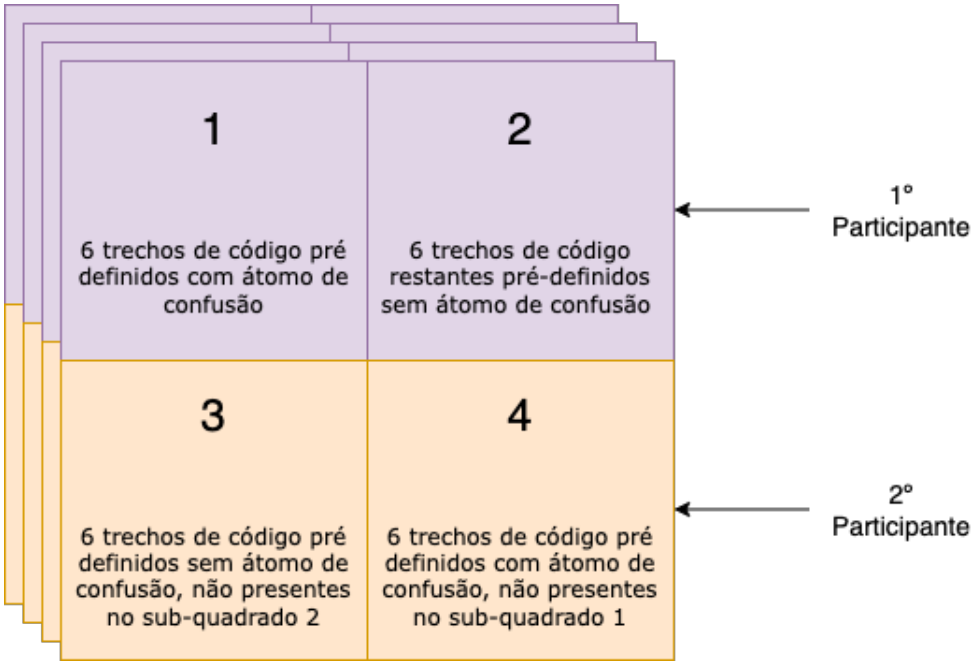

Figura 3.1: Exemplo de construção de Quadrado Latino.

Se o outro participante envolvido no mesmo quadrado houver respondido as 12 questões, essa linha do quadrado será aproveitada na construção de um novo quadrado, fazendo um casamento entre quadrados que tiverem as mesmas questões envolvidas. Esta foi uma das alterações de desenho experimental que divergiu do trabalho de Oliveira e Torres [3]. Linhas incompletas em quaisquer quadrados entretanto foram descartadas, ou seja, se um participante não respondeu todas as questões, suas respostas não serão consideradas na análise dos resultados.

Outra alteração de desenho experimental deste trabalho em relação ao trabalho de Adriano e Torres [3] está na seleção dos átomos que compõem um quadrado. No experimento anterior, havia um total de 20 trechos de código (10 confusos e 10 limpos). Destes, 10 (5 confusos e 5 limpos) eram selecionados aleatoriamente para compor um quadrado. Aqui, não há seleção aleatória. Os 48 trechos de código analisados foram divididos em 4 subconjuntos distintos pré-definidos contendo 12 trechos cada (6 confusos e 6 limpos). Dessa forma, cada subconjunto é atribuído a um participante num quadrado, totalizando os 24 trechos de código a serem respondidos por quadrado. Tal decisão foi tomada para evitar que alguns trechos de código fossem muito e outros fossem pouco respondidos ao serem selecionados aleatoriamente, já que a quantidade de trechos a serem avaliados cresceu consideravelmente em relação à pesquisa anterior.

3.2.3 Estratégia de Execução do *survey*

Survey Piloto

Com o intuito de validar as alterações realizadas na aplicação *web* do *survey* antes executado no trabalho de Oliveira e Torres [3], foi rodado uma versão informal piloto cujos objetivos foram:

1. Encontrar erros na aplicação após as alterações e/ou erros nos dados coletados.
2. Obter *feedbacks* sobre o survey e sobre as questões apresentadas.
3. Entender quanto tempo de fato seria empenhado pelos participantes para responder o survey.

Participaram do piloto colegas também alunos de graduação da UnB, colegas de outras faculdades e de trabalho. Sugestões sobre a interface foram apresentadas e, depois de implementadas, o *survey* estava pronto para ser executado oficialmente.

Survey aberto publicado em redes sociais

Para a execução oficial do *survey*, foi utilizada a mesma estratégia de Oliveira e Torres [3] de publicar a pesquisa em espaços voltados à comunidade JavaScript em redes sociais. O *survey* foi publicado em um dos maiores fóruns do *Reddit*[3] voltado para a comunidade JavaScript, onde obteve-se um engajamento expressivo de participantes respondendo e fazendo comentários e sugestões sobre o *survey*. Também foi publicado no *Twitter*[4], porém como não há um espaço específico para tais comunidades nesta rede social, o alcance foi menor que o observado no Reddit. O *survey* também foi encaminhado para outros colegas de trabalho e de graduação. No total, 235 participantes responderam o survey, resultando em um aproveitamento em torno de 60 quadrados válidos e aptos a serem analisados.

[3]O Reddit é um fórum de discussão online. As *threads* de discussão, *subreddits*, são organizadas por assunto. Para a pesquisa, o *survey* foi publicado em 3 *subreddits* voltados para JavaScript.

[4]O *Twitter* é uma rede social onde os usuário publicam *tweets*, em formato de pequenos textos de até 140 caracteres. A comunidade *tech* costuma ser bastante ativa na rede, por isso, também foi utilizada para veicular o *survey*, marcando na publicação perfis de referência em tecnologia na rede, com o intuito de ganhar visibilidade.

Capítulo 4

Resultados

Nesta seção são apresentados os resultados do *survey*. Estes resultados são utilizados na tentativa de responder as três perguntas apresentadas na Seção 1.2.

4.1 Dados coletados na pesquisa

Os seguintes dados foram coletados dos participantes:

- E-mail (opcional);
- Idade;
- Escolaridade;
- Anos de experiência com programação.

O e-mail foi tratado como opcional, seguindo as sugestões dos participantes do trabalho de Oliveira e Torres [3]. Ainda assim, foi solicitado com o intuito de fornecer ao participante um retorno sobre os resultados desta pesquisa.

4.1.1 Escolaridade e Experiência

As figuras abaixo apresentam histogramas com a distribuição dos participantes de acordo com sua escolaridade e anos de experiência com programação, respectivamente.

De acordo com a Figura 4.1, mais da metade (aproximadamente 73%) dos participantes possuía grau de Bacharelado ou Mestrado. Apenas 1 participante possuía grau de Doutorado. Isto indica que a maior parte dos participantes possuía graduação formal direcionada a programação, enquanto 28% possuíam grau de Ensino Médio ou curso universitário sem diplomação.

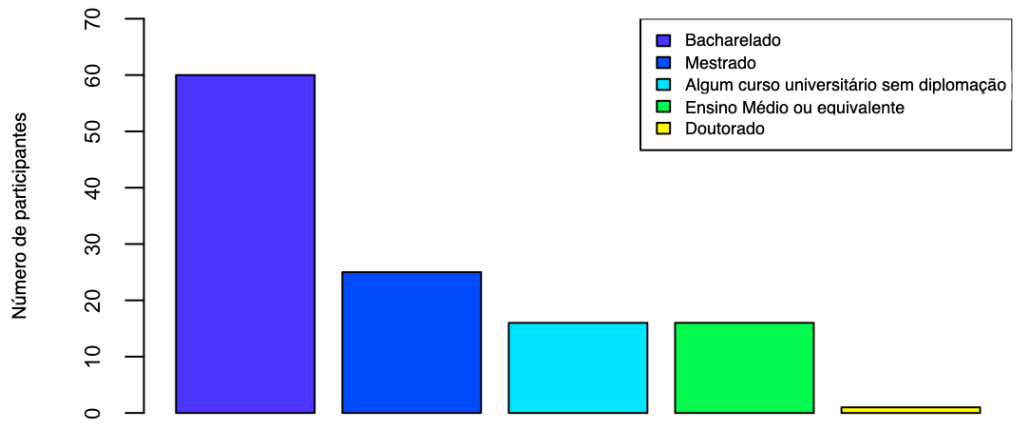

Figura 4.1: 'Escolaridade' dos participantes.

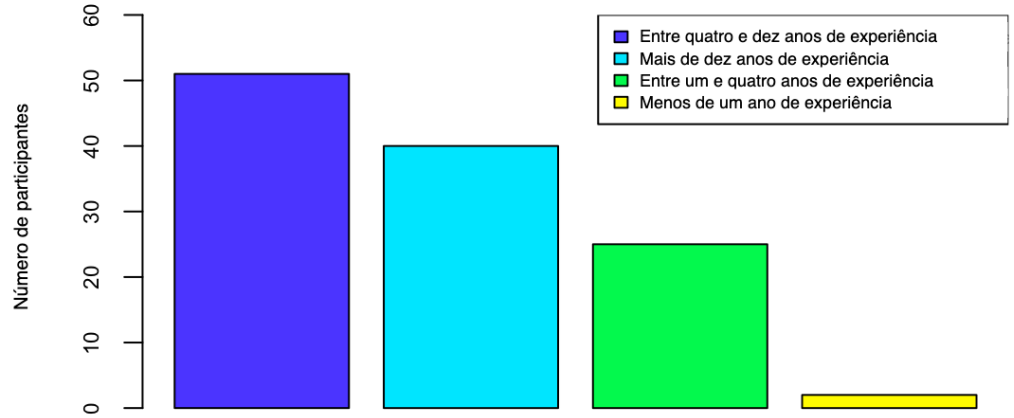

Figura 4.2: 'Anos de experiência com programação' dos participantes.

Já a Figura 4.2 evidencia que quase todos os participantes possuíam pelo menos 1 ano de experiência com programação, enquanto 77% possuía pelo menos 4 anos de experiência prévia. Ainda, aproximadamente 21% possuíam entre 1 e 4 anos de experiência.

4.2 Análise das respostas dos participantes

Como definido na Metodologia (Capítulo 3), cada participante foi solicitado a responder 12 questões, cada uma contendo um trecho de código. Destas, 6 eram versões limpas e 6 eram versões confusas de um átomo. Algumas métricas foram coletadas para as questões:

- Corretude da resposta, indicando se o participante acertou ou errou a questão ao tentar predizer corretamente a saída esperada para o trecho de código analisado. Se

o participante selecionasse o botão *"I do not know"* ("Eu não sei"), a reposta era registrada como incorreta;

- Tempo empenhado para responder a questão;
- Um *booleano* que identificava se a questão possuía ou não um átomo de confusão;
- Um identificador para relacionar a questão com o átomo que estava sendo analisado naquele trecho de código;
- O endereço de IP da máquina do participante para previnir que o participante tentasse responder a pesquisa mais de uma vez naquele instante.

Como no estudo de Oliveira e Torres [3], foi utilizado um valor *booleano* para indicar se o participante trocou de aba ou de janela durante a questão. A premissa para a necessidade de tal variável era a suposição de que trechos de código com a presença do átomo de confusão levariam a maior tempo de resposta, o qual poderia ser ainda maior se o participante saísse da página do survey para tentar acessar algum outro material externo como apoio. Essa hipótese, entretanto, não se validou, como detalhado na Seção 4.2.3. A análise sobre a variável que indica troca de janelas então, foi descartada, como no estudo anterior.

No total, 120 quadrados latinos foram criados durante a execução do survey. Seguindo o desenho experimental do trabalho de Oliveira e Torres [3], descartando os quadrados incompletos (que não tinham 24 respostas), sobraram 36 quadrados a serem analisados, pouco mais de 25% do total inicial de quadrados criados. Utilizando a imputação de dados citada no Capítulo 3, foi possível formar 23 novos quadrados a partir de junção de linhas completas de quadrados incompletos, ou seja, onde ao menos um dos participantes respondeu todas as 12 questões, esse participante se tornou candidato a compor um novo quadrado. Foram identificados os quadrados que poderiam ser unidos para formar um novo utilizando o critério citado. Para selecionar como a combinação iria acontecer, utilizou-se um *script* em Javascript no qual eram fornecidas listas dos quadrados que poderiam ser combinados, e a combinação acontecia de forma aleatória, a fim de evitar viéses na seleção. O Programa 4.1 apresenta o script utilizado para indicar a combinação dos quadrados.

Como resultado do processo de imputação de dados, totalizou-se 59 quadrados latinos completos, aptos a serem analisados, o que corresponde a analisar as respostas de 118 participantes, num total de 1416 respostas aos trechos de código. Diferente do estudo de Oliveira e Torres [3], não houve uma mesma quantidade de respostas para todos os átomos, visto que apenas metade de todos os trechos de código analisados na pesquisa estavam disponíveis em cada quadrado formado seguindo o critério descrito na Seção 3.2.2.

Programa 4.1 Script escrito em JavaScript para combinar quadrados

```
function buildSquareCombination(firstSet, secondSet) {
    let finalCombination = []

    while(firstSet.length > 0 && secondLength.length > 0) {
        let firstSetIndex = Math.floor(Math.random() *
            firstSet.length)
        let secondSetIndex = Math.floor(Math.random() *
            secondSet.length)

        finalCombination.push([firstSet[firstSetIndex], secondSet
            [secondSetIndex]])

        firstSet.splice(firstSetIndex, 1)
        secondSet.splice(secondSetIndex, 1)
    }

    console.log("FINAL SQUARE COMBINATION:", finalCombination)
    console.log("Squares not combined from first Set:", firstSet)
    console.log("Squares not combined from second Set:",
        secondSet)
}
```

4.2.1 Corretude das respostas

A tabela 4.1 apresenta uma análise sobre quantidade de respostas corretas para cada átomo. São apresentadas as colunas "NºCorretas" que mostram a quantidade de respostas corretas para cada átomo analizado, para as duas categorias de trechos de código ("Com átomo" e "Sem átomo"). A coluna "$\Delta(\%)$" apresenta, em porcentagem, a variação entre os valores apresentados nas duas colunas citadas acima (ou seja, qual a variação de respostas corretas a trechos de código sem átomo em relação a trechos de código com átomo). Ou seja, mostrando a porcentagem de re Dos 24 átomos apresentados, 11 apresentaram melhora de pelo menos 10% na quantidade de respostas corretas quando o átomo não estava presente no trecho de código. Os átomos *Type Conversion* e *Pre Increment* foram os que mais apresentaram variação na quantidade, com mais do que o dobro de respostas corretas quando o átomo não estava presente.

De maneira surpreendente, o átomo *Pre Increment* assumiu uma posição não esperada inicialmente, visto que no trabalho de Oliveira e Torres [3], não foi evidenciada uma diferença tão grande na quantidade de respostas corretas com e sem o átomo (7% no experimento anterior e 111% neste experimento). Talvez o trecho de código que contém esse átomo, neste trabalho, possa ter corroborado para a expressiva diferença no resultado, já que não era o mesmo do utilizado no experimento anterior. Tal fato poderia ser explorado em um novo trabalho.

Ainda, dos 24 átomos analisados, 7 apresentaram diferença negativa na quantidade de respostas corretas para trechos com e sem a presença do átomo. Ou seja, Para esses 7 átomos, houve mais respostas corretas **quando o átomo estava presente**. O átomo que mais chamou a atenção na análise dos resultados foi o *Object Spread*, um átomo exclusivo de JavaScript, não analisado em trabalhos anteriores, que apresentou uma diferença de 31% mais respostas corretas quando o átomo estava presente.

Os átomos *Comma Operator* e *Automatic Semicolon Insertion*, que apresentaram as maiores diferenças na quantidade de respostas corretas no trabalho de Oliveira e Torres, também apresentaram diferenças relevantes neste trabalho, com, respectivamente, 76% e 37% mais respostas corretas para os trechos de código onde os átomos estavam ausentes.

Átomo	Nº**Corretas** Com átomo	Nº**Corretas** Sem átomo	**Δ(%)**
Type Conversion	6	15	150
Pre Increment	9	19	111
Change Literal Encoding	11	20	81
Comma Operator	13	23	76
Repurposed Variables	8	13	62
Automatic Semicolon Insertion	8	11	37
Object Destructuring	13	17	30
Arrow Function	14	18	28
Post Increment	21	25	19
Indentation With Braces	27	32	18
Assignment As Value	17	19	11
Logic As Control Flow	32	35	9
Array Destructuring	33	35	6
Arithmetic As Logic	18	19	5
Constant Variables	33	34	3
Ommited Curly Braces	29	30	3
Indentation No Braces	16	16	0
Ternary Operator	35	34	-3
Property Access	34	33	-3
Implicit Predicate	34	31	-9
Dead, Unreachable, Repeated	36	33	-9
Array Spread	37	34	-9
Infix Operator Precedence	20	18	-10
Object Spread	13	9	-31

Tabela 4.1: Diferença na quantidade de respostas corretas entre trechos confusos e limpos

O gráfico *box plot* [12] apresentado na Figura 4.3 mostra que houve queda na média da quantidade de respostas incorretas quando os átomos não estavam presentes. Além disso, o conjunto de respostas para os trechos de código onde um átomo não estava presente (limpos) apresentou menor dispersão, indicando que código sem a presença de construtos confusos é mais fácil de compreender corretamente.

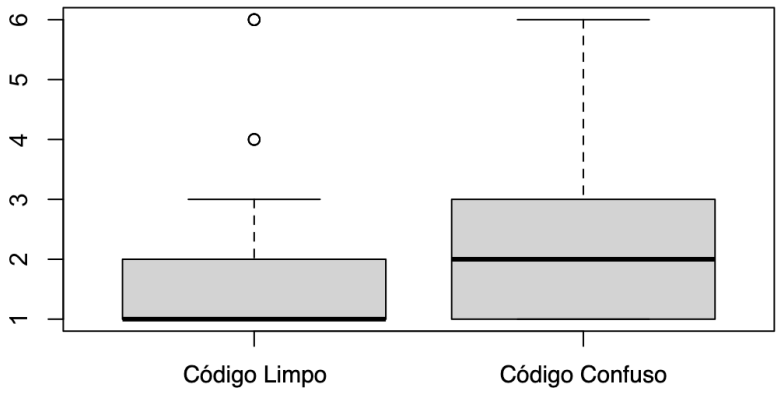

Figura 4.3: Distribuição da quantidade de respostas erradas para cada participante.

4.2.2 Análise estatística sobre a corretude das respostas

Foi utilizado o teste *Qui-quadrado de Pearson* [13] para investigar se existe diferença estatística significante nas distribuições de repostas corretas e incorretas, dado que um átomo tem duas versões, uma limpa e uma confusa. Os p-valores para os testes estão relacionados na Tabela 4.2. Em 4 dos 24 átomos analisados (*Change Literal Encoding*, *Pre Increment*, *Type Conversion* e *Comma Operator*), os resultados apontam uma diferença estatística significante (apresentam **p-valor $<$ 0.05**), indicando que as versões confusas desses trechos de código impactaram a compreensão de código negativamente. Destes 4 átomos, apenas o átomo *Comma Operator* também apresentou diferença estatística significante no trabalho de Oliveira e Torres [3], indicando resultados convergentes para o impacto do átomo na compreensão de código.

Os intervalos de confiança reportados para 2 dos 4 átomos citados acima (*Change Literal Encoding* e *Pre Increment*) foram bastante grandes, não permitindo afirmar com segurança que estudos futuros indiquem porcentagem parecida para melhora na predição correta de respostas para as versões limpas desses átomos. Isto pode explicar o por que, por exemplo, o átomo *Pre Increment* teve uma discrepância tão grande desse parâmetro entre o estudo de Oliveira e Torres [3] e o estudo realizado neste trabalho. Ainda assim, para os 4 átomos onde foi encontrada diferença estatística significante, também foi observado um limite inferior para o intervalo de confiança maior ou igual a 1. Isto indica que, com um nível de confiança de 95%, pode-se afirmar que um programador tende a cometer menos erros na predição do output quando analiza as versões limpas dos códigos.

O Tamanho de Efeito [14] indica a importância prática das chances de um evento acontecer em relação ao outro. Ele foi medido para as versões limpas dos trechos de código de acordo com a corretude das respostas, usando a Razão de Probabilidades (ou

Razão de Chances). Os resultados também se encontram na Tabela 4.2, na coluna "Razão de Chances". Para 2 dos átomos, *Change Literal Encoding* e *Pre Increment*, a Razão de Probabilidades apresentou valores elevados. Analisando o caso do átomo *Change Literal Encoding*, por exemplo, quando se compara as respostas dos átomos limpos e dos confusos, temos uma Razão de Probabilidades de 16.99. Em termos práticos, isto significa que as chances de que o programador responda corretamente a versão limpa deste trecho de código são 16.99 vezes maiores do que as chances de ele responder corretamente a versão confusa.

Átomo	**Qui-quadrado**	**Razão de Chances**	**p-valor**	**Intervalo de Confiança**
Change Literal Encoding	0.004991	16.99	**0.0036256***	(1.97, 823.34)
Pre Increment	0.0032197	11.83	**0.0025015***	(2.02, 131.04)
Type Conversion	0.013555	5.95	**0.01262***	(1.38, 29.64)
Arrow Function	0.2771	2.92	0.27716	(0.54, 20.73)
Comma Operator	0.038677	2.91	**0.037981***	(1.05, 8.35)
Repurposed Variables	0.21704	2.58	0.21672	(0.65, 11.02)
Object Destructuring	0.30551	2.56	0.30576	(0.54, 14.29)
Assignment As Value	0.65924	2.19	0.6628	(0.27, 27.16)
Logic As Control Flow	0.47768	2.17	0.47987	(0.42, 14.49)
Indentation With Braces	0.27086	2.15	0.27072	(0.63, 8.08)
Automatic Semicolon Insertion	0.53524	1.76	0.5359	(0.45, 7.28)
Array Destructuring	0.70857	1.75	0.71104	(0.31, 12.20)
Arithmetic As Logic	1	1.57	1	(0.16, 20.81)
Post Increment	0.48142	1.55	0.48176	(0.56, 4.36)
Constant Variables	1	1.28	1	(0.25, 7.06)
Ommited Curly Braces	1	1.16	1	(0.34, 3.99)
Indentation No Braces	1	1.00	1	(0.19, 5.30)
Property Access	1	0.78	1	(0.14, 3.97)
Ternary Operator	1	0.73	1	(0.10, 4.68)
Implicit Predicate	0.51437	0.53	0.51609	(0.10, 2.31)
Object Spread	0.35399	0.47	0.3543	(0.11, 1.86)
Dead, Unreachable, Repeated	0.42758	0.37	0.43033	(0.03, 2.46)
Infix Operator Precedence	0.59913	0.31	0.606	(0.01, 4.24)
Array Spread	0.35477	0.23	0.35799	(0.00, 2.52)

Tabela 4.2: Resultados dos Testes de Hipóteses relacionados à corretude das respostas. Asteriscos (*) indicam diferença estatística significante.

4.2.3 Tempo gasto para responder as questões

A Tabela 4.3 apresenta os resultados na análise da diferença de tempo gasto, em segundos, para responder corretamente as questões, com e sem átomo de confusão. Aqui, os átomos *Change Literal Encoding*, *Assignment As Value* e *Logic As Control Flow* foram os que

mais apresentaram diferença de tempo para responder corretamente as questões, apresentando mais de 60% de redução no tempo médio gasto para fornecer uma resposta correta quando o átomo não estava presente. O átomo *Change Literal Encoding* é um átomo não exclusivo de JavaScript, mas que não foi analisado no experimento de Oliveira e Torres [3], apresentando redução de 75% no tempo médio gasto empenhado para apresentar uma resposta correta.

Dos 24 átomos analizados, 13 apresentaram redução do tempo médio gasto para fornecer uma resposta correta quando o átomo não estava presente no trecho de código. O átomo *Logic as Control Flow* aparece como o terceiro átomo com a maior diferença, com redução no tempo médio de aproximadamente 65%, reforçando a ideia apresentada nos resultados do estudo de Oliveira e Torres [3].

Dentre os átomos que apresentaram o efeito inverso do esperado, apontando maiores tempos médios para apresentar uma resposta correta quando na ausência do átomo, estão o *Object Destructuring*, o *Ternary Operator* e o *Implicit Predicate*. Estes 3 átomos apresentaram aumento de mais do que o dobro do tempo para responder corretamente a questão quando o átomo estava ausente. Chama a atenção o átomo *Implicit Predicate*, que apresentou tempo médio quatro vezes maior para predizer a saída corretamente na ausência do átomo, indo na contramão dos resultados apresentados no estudo de Oliveira e Torres, onde o mesmo átomo apresentou redução de quase 30% no tempo médio empenhado para responder corretamente quando o átomo estava ausente.

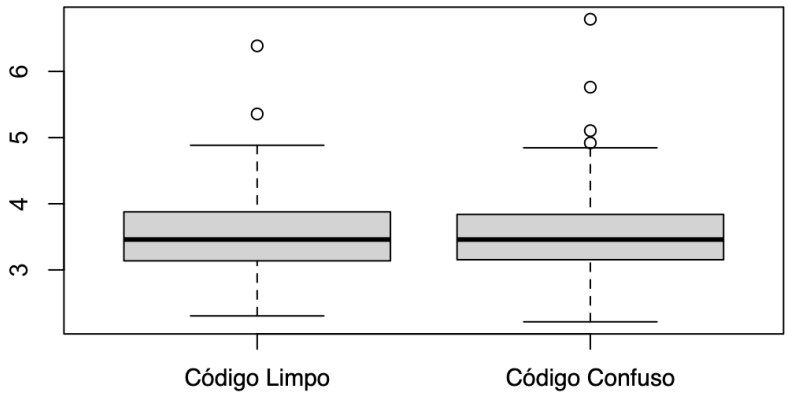

Figura 4.4: Distribuição do tempo em segundos gasto para responder corretamente as questões (em escala logaritmica).

A Figura 4.4 apresenta a dispersão dos tempos médios gastos para responder corretamente as questões. O tempo no gráfico é apresentado em escala logarítmica. Indo na contramão do que se pensou inicialmente, não houve diferença relevante na média entre os tempos gastos na presença e na ausência do átomo. A dispersão também se apresentou

Átomo	**Tempo Médio** Com átomo	**Tempo Médio** Sem átomo	**Δ(%)**
Change Literal Encoding	58.81	14.27	-75.74
Assignment As Value	105.06	31.81	-69.72
Logic As Control Flow	140.86	49.73	-64.69
Repurposed Variables	145.51	76.20	-47.63
Type Conversion	52.40	29.55	-43.61
Infix Operator Precedence	47.55	27.98	-41.15
Constant Variables	25.23	15.83	-37.26
Arithmetic As Logic	45.49	33.64	-26.05
Indentation With Braces	38.25	28.29	-26.05
Comma Operator	80.37	62.89	-21.75
Automatic Semicolon Insertion	76.68	66.12	-13.78
Ommited Curly Braces	26.74	24.76	-7.41
Dead, Unreachable, Repeated	18.07	16.78	-7.14
Pre Increment	67.64	71.23	5.29
Object Spread	58.80	64.47	9.63
Indentation No Braces	34.19	40.11	17.30
Arrow Function	30.19	37.55	24.40
Array Spread	38.13	47.44	24.41
Post Increment	39.68	52.92	33.37
Array Destructuring	22.64	33.77	49.16
Property Access	42.99	65.47	52.30
Object Destructuring	31.68	63.69	101.04
Ternary Operator	30.33	73.77	143.24
Implicit Predicate	27.31	141.95	419.85

Tabela 4.3: Diferença no tempo gasto (em segundos) para responder as questões

bastante semelhante, exceto por algumas amostras no conjunto de tempo empenhado em responder código confuso, que desviaram bastante da distribuição.

4.2.4 Análise estatística sobre o tempo gasto para responder as questões

Foi utilizado o teste *Mann-Whitney U* para investigar a hipótese nula (H0) de que os programadores gastam o mesmo tempo para predizer corretamente a saída de ambas versões limpa e confusa de um átomo. Assim como no teste *Qui-quadrado* realizado na Seção 4.2.2, o objetivo é encontrar os átomos que tiveram diferença estatística significante que confirme a hipótese alternativa (H1) de que os programadores gastam mais tempo para predizer corretamente trechos de código confusos em comparação aos limpos.

A Tabela 4.4 apresenta os resultados do teste *Mann-Whitney U*, e indica que a hipótese inicial (H0) deve ser refutada para 7 dos 24 átomos analisados, *Comma Operator*, *Object Destructuring*, *Logic as Control Flow*, *Indentation With Braces*, *Post Increment*, *Constant*

Variables e *Property Access*. Aqui também foi analisado o Tamanho de Efeito, porém utilizando o *Delta de Cliff*. Os valores negativos para o *Delta de Cliff* na Tabela 4.4 indicam que os participantes gastaram menos tempo para predizer corretamente a saída de átomos em sua versão limpa. Para os átomos *Comma Operator*, *Indentation With Braces* e *Constant Variables*, a análise permite concluir que os participantes **levaram mais tempo para predizer corretamenta a saída das versões confusas desses átomos**. Já para os átomos *Object Destructuring*, *Logic as Control Flow*, *Post Increment* e *Property Access*, os resultados sugerem que os participantes **gastaram menos tempo para predizer corretamente a saída das versões confusas dos átomos**.

Átomo	**Teste Mann-Whitney U**	**Delta de Cliff**
Change Literal Encoding	0.052071	-0.35
Pre Increment	0.12581	0.28
Type Conversion	0.2501	-0.21
Arrow Function	0.88131	-0.03
Comma Operator	**0.0024913***	-0.40
Repurposed Variables	0.94049	0.02
Object Destructuring	**0.0092569***	0.46
Assignment As Value	0.2501	-0.21
Logic As Control Flow	**0.046225***	0.27
Indentation With Braces	**0.00036196***	-0.47
Automatic Semicolon Insertion	0.23985	-0.22
Array Destructuring	0.087513	0.23
Arithmetic As Logic	0.69037	-0.07
Post Increment	**0.033081***	0.28
Constant Variables	**0.000016407***	-0.56
Ommited Curly Braces	0.68324	-0.06
Indentation No Braces	0.90098	0.02
Property Access	**0.0016611***	0.41
Ternary Operator	0.0605	0.25
Implicit Predicate	0.28405	0.14
Object Spread	0.72745	-0.07
Dead, Unreachable, Repeated	0.1836	-0.18
Infix Operator Precedence	0.98015	0.01
Array Spread	0.55237	0.08

Tabela 4.4: Resultados dos Testes de Hipóteses relacionados ao tempo gasto para responder as questões. Asteriscos (*) indicam diferença estatística significante.

4.3 Respondendo as Questões de Pesquisa

4.3.1 Resposta à Questão de Pesquisa 1

> (RQ1)Trechos de código que contém átomos de confusão resultam em mais erros na predição do output que trechos de código onde os átomos foram removidos?

Como mostrado na Tabela 4.1, em 16 átomos analisados houve aumento na quantidade de respostas corretas para trechos de código onde o átomo de confusão não estava presente. Com variação de 3% a 150% mais respostas corretas, 16 dos 24 átomos analizados mostraram potencial de tornar o código mais difícil de compreender quando presentes. A análise apresentada na Seção 4.2.2 indica que existe relação entre a ausência do átomo de confusão e a chance de predição correta da saída do trecho de código analisado. Portanto, pode-se concluir que a resposta para a RQ1 é positiva e está alinhada com o experimento anterior, apesar de ser confirmada para átomos diferentes.

4.3.2 Resposta à Questão de Pesquisa 2

> (RQ2)Trechos de código que contém átomos de confusão exigem mais tempo do programador para predizer o output?

Como é possível observar na Tabela 4.3 e na Figura 4.4, a variância no tempo gasto para responder corretamente a um trecho de código é pequena, assim como no trabalho de Oliveira e Torres [3]. Apesar disso, a análise apresentada na seção 4.2.4 mostra existe relação entre o tempo gasto para apresentar uma resposta correta e presença do átomo de confusão para alguns trechos analisados. A resposta para a RQ2 portanto é positiva e também está alinhada com o encontrado no experimento anterior, apesar de também ser confirmada para átomos diferentes.

4.3.3 Resposta à Questão de Pesquisa 3

> (RQ3)Os resultados desta pesquisa reforçam os resultados apresentados no estudo anterior para um mesmo conjunto de átomos em JavaScript?

Apesar de o átomo *Comma Operator* ter sido o único que se confirmou como átomo de confusão entre as pesquisas, outros átomos como o *Pre Increment* apresentaram uma discrepância relevante entre o trabalho de Oliveira e Torres [3] e este trabalho. Portanto, a resposta para a RQ3 é inconclusiva, e abre caminhos a serem explorados para entender o por que da diferença entre os dois estudos para o mesmo subconjunto de átomos.

4.4 Ameaças à validação dos resultados

O *survey* utilizado neste trabalho foi conduzido *online* com participantes desconhecidos de toda a comunidade JavaScript, espalhados por vários países. Assim como trabalho de Oliveira e Torres [3], não há como garantir que as informações demográficas informadas por todos os participantes estão corretas, o que poderia levar a diferentes conclusões das apresentadas aqui. Ainda, apesar dos esforços em direcionar os participantes a responderem ao *survey* por conta própria, não há como garantir que não houve a utilização de fontes externas para responder as perguntas apresentadas. Isto representa ameaça às validade das distribuições de dados e hipóteses analisadas nas Seções 4.2.2 e 4.2.4.

Ainda, a pré-seleção de átomos realizada como informado na Seção 3.2.2 pode ter agregado, de maneira inconsciente, questões de elevado nível de dificuldade. Tal fato pode ter elevado o esforço de tempo necessário para responder as questões presentes em alguns quadrados, e pode ter sido um dos motivos pelos quais muitos pararam de responder ao *survey* antes de completar as 12 questões.

Capítulo 5

Conclusão

Os resultados analisados e apresentados no Capítulo 4 se mostraram satisfatórios para atender ao objetivo inicial do trabalho. Foi possível expandir o estudo sobre átomos de confusão para átomos além dos já estudados em experimentos anteriores, considerando também átomos exclusivos de JavaScript.

Em relação à corretude das respostas analisadas, os átomos ***Change Literal Encoding***, ***Pre Increment***, ***Type Conversion*** e ***Comma Operator*** se confirmaram átomos de confusão, porém nenhum deles é exclusivo de JavaScript. De maneira geral, analisando a quantidade de respostas corretas para cada categoria de trecho de código (com e sem o átomo de confusão), aqueles com a ausência do átomo de confusão tiveram mais respostas corretas. Dos átomos confirmados citados acima, 2 apresentaram resultados bastante interessantes considerando que foram analisados também no estudo de Oliveira e Torres [3]. O átomo *Comma Operator* reiterou os resultados apresentados no estudo anterior e se confirmou um átomo de confusão novamente. Já o átomo *Pre Increment* apresentou uma divergência considerável entre o estudo anterior e o aqui apresentado, se tornando um átomo de confusão. Tal característica, pode levantar questionamentos sobre a estabilidade dos resultados encontrados para os átomos estudados entre diferentes execuções da pesquisa.

Em relação ao tempo empenhado para predizer corretamente a saída de um trecho de código, os trechos de código com a presença dos átomos de confusão levaram mais tempo para serem respondidos. Os átomos ***Comma Operator***, ***Indentation With Braces*** e ***Constant Variables*** apresentaram diferença estatística significativa que corrobora a hipótese de que átomos limpos são respondidos mais rapidamente. Um fato interessante evidenciado pelos resultados da pesquisa é o de que os átomos ***Object Destructuring*** e ***Property Access*** levaram mais tempo para serem respondidos corretamente na versão em que o átomo estava ausente. Ambos são átomos exclusivos da linguagem JavaScript, e foram introduzidos apenas nesta pesquisa.

5.1 Trabalhos futuros

Como trabalhos futuros, existe a possibilidade de explorar a influência das métricas "Experiência" e "Escolaridade" nas análises sobre corretude das respostas e sobre o tempo gasto para responder as questões. Seria possível estudar se participantes com nível de escolaridade mais elevado tendem a responder corretamente as questões onde os átomos de confusão estão presentes na mesma proporção em que respondem corretamente as questões onde os átomos não estão presentes. Ainda, o trabalho de Posnett et. al. [15] apresenta um estudo interessante ao propor um modelo para avaliar a compreensão/legibilidade de um código, um conceito que, utilizado em estudos futuros, poderia trazer resultados promissores na análise sobre átomos de confusão.

Alguns fatores podem ter contribuído para os resultados encontrados neste trabalho e também se apresentam como possibilidades a serem exploradas em trabalhos futuros. Por exemplo, seria interessante investigar o por que da discrepância de resultados entre átomos avaliados em ambos os estudos. Acredita-se que os trechos de código utilizados possam ter interferência, por serem diferentes. Ainda, o tempo entre a execução do *survey* e consequentemente entre a realização dos estudos também pode ser um fator contribuinte para os resultados divergentes. O fato de o JavaScript estar em constante ascenção no mercado e ser a linguagem mais utilizada do mundo[1] nos últimos anos pode ser mais um dos motivos para divergência de resultados. Uma linguagem moderna e cada dia mais difundida como o JavaScript pode disseminar entre sua comunidade programadora noções sobre padrões e estruturas sintáticas únicas da linguagem, que com o tempo podem não ser mais percebidas como fontes de confusão.

[1] `https://octoverse.github.com/2022/top-programming-languages`

Referências

[1] Gopstein, Dan, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K. C. Yeh e Justin Cappos: *Understanding misunderstandings in source code*. Em *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, página 129–139, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450351058. https://doi.org/10.1145/3106237.3106264.

[2] Gopstein, Dan, Hongwei Henry Zhou, Phyllis Frankl e Justin Cappos: *Prevalence of confusing code in software projects: Atoms of confusion in the wild*. Em *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, página 281–291, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450357166. https://doi.org/10.1145/3196398.3196432.

[3] OLIVEIRA, Caio e Adriano Rodrigues Figueiredo TORRES: *On the impact of atoms of confusion in javascript code*. https://bdm.unb.br/handle/10483/27600, 2019.

[4] Podcast, Fronteiras da Engenharia de Software: *13: Compreensão de código com fernando castor (ufpe) by fronteiras da engenharia de software*, Jul 2021. https://anchor.fm/fronteirases/episodes/Compreenso-de-Cdigo-com-Fernando-Castor-UFPE-e12ssjs, Acesso em: 25 de Out. de 2022.

[5] Reuter, Jessica Susan: *What is program comprehension?*, Oct 2022. https://www.wise-geek.com/what-is-program-comprehension.htm, Acesso em: 6 de Nov. de 2022.

[6] Al-Saiyd, Nedhal A.: *Source code comprehension analysis in software maintenance*. Em *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, páginas 1–5, 2017.

[7] Nishizono, Kazuki, Shuji Morisakl, Rodrigo Vivanco e Kenichi Matsumoto: *Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners*. Em *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, páginas 473–481, 2011.

[8] Tilley, S.R., S. Paul e D.B. Smith: *Towards a framework for program understanding*. Em *WPC '96. 4th Workshop on Program Comprehension*, páginas 19–28, 1996.

[9] Wyrich, Marvin, Justus Bogner e Stefan Wagner: *40 years of designing code comprehension experiments: A systematic mapping study*, 2022. https://arxiv.org/abs/2206.11102.

[10] Neri: *Boas práticas de programação: Organizando o código e seus pacotes*. https://www.devmedia.com.br/boas-praticas-de-programacao/21137, 2011. Acesso em: 25 de Out. de 2022.

[11] P., Box George E, J. Stuart Hunter e William G. Hunter: *Statistics for experimenters design, innovation, and Discovery*. Wiley-Interscience, 2005.

[12] Claudio, Claudio: *Box plot: O que é e como analisar e interpretar esse gráfico?*, May 2022. https://www.escolaedti.com.br/o-que-e-um-box-plot, Acesso em: 3 de Nov. de 2022.

[13] Turney, Shaun: *Chi-square (^{2}) tests: Types, formula amp; examples*, Nov 2022. https://www.scribbr.com/statistics/chi-square-tests/, Acesso em: 16 de Nov. de 2022.

[14] Ramsey, Fred L. e Daniel W. Schafer: *The statistical sleuth: A course in methods of data analysis*. Brooks/Cole, Cengage Learning, 2013.

[15] Posnett, Daryl, Abram Hindle e Premkumar Devanbu: *A simpler model of software readability*. Em *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, página 73–82, New York, NY, USA, 2011. Association for Computing Machinery, ISBN 9781450305747. https://doi.org/10.1145/1985441.1985454.