Associação Propagadora Esdeva
Centro Universitário Academia – UniAcademia
Curso de Engenharia de Software
Trabalho de Conclusão de Curso – Artigo

# ANALISANDO A EVOLUÇÃO DA BIBLIOTECA JQUERY POR MEIO DO SONARQUBE

*Rafael de Carvalho Clemente Oliveira*[1]
*Centro Universitário Academia, Juiz de Fora, MG*

*Tassio Ferenzini Martins Sirqueira*[2]
*Centro Universitário Academia, Juiz de Fora, MG*

Linha de Pesquisa: Engenharia de Software

**RESUMO**

Atualmente, muitos sites usam a linguagem *JavaScript* em sua programação do lado do cliente e, de acordo com dados da *W3Tech*, a maioria absoluta usa a biblioteca *jQuery*. Considerando a importância desta biblioteca e seu uso, este trabalho se propõe a realizar uma análise histórica de como o *jQuery* é mantido e evoluído. A metodologia foi estruturada em leitura ad hoc de trabalhos relacionados e busca de versões estáveis no repositório. Posteriormente, a ferramenta *SonarQube* analisou as versões, e os resultados foram integrados e estruturados em forma de tabela. Ao longo do estudo, 83 versões estáveis foram analisadas e as métricas estudadas são bugs, *code smell*, blocos duplicados e porcentagem de código duplicado. Durante o estudo, uma análise experimental de engenharia de software foi realizada por meio de testes estatísticos. Os resultados são apresentados e discutidos individualmente para cada métrica *jQuery*. No geral, descobriu-se que a biblioteca cresceu ao longo do tempo, com base no número de linhas de código. No entanto, o número de bugs e *code smell* não acompanhou o mesmo crescimento, embora ainda tenha aumentado, enquanto as duplicatas de código e os blocos duplicados permaneceram estáveis. Esta análise é essencial para demonstrar o estado atual do *jQuery* e apresentar indicadores para os desenvolvedores que fazem uso dele.

**Palavras-chave:** jQuery, Evolução, JavaScript, SonarQube.

***ABSTRACT***

*Currently, many sites use the JavaScript language in their client-side programming, and according to data from W3Tech, the absolute majority use the jQuery library. Considering the importance of this library and its use, this work proposes to perform a historical analysis of how jQuery is maintained and evolved. The methodology was structured in ad hoc reading of related works and searching for stable versions in the repository. Subsequently, the SonarQube tool analyzed the versions, and the results were integrated and structured in table form. Throughout the study, 83 stable versions were analyzed, and the metrics studied are bugs, code smell, duplicate blocks, and percentage of duplicate code. During the study, an experimental software engineering*

[1] Discente do Curso de Engenharia de Software do Centro Universitário Academia – UniAcademia. E-mail: rafaelxeng@gmail.com.
[2] Docente do Curso de Engenharia de Software do Centro Universitário Academia. Orientador.

*analysis was performed using statistical tests. The results are presented and discussed individually for each jQuery metric. Overall, it was found that the library grew over time, based on the number of lines of code. However, the number of bugs and code smells did not follow the same growth, although they still increased, while code duplicates and duplicate blocks remained stable. This analysis is essential for demonstrating the current state of jQuery and presenting indicators for developers who make use of it.*

## 1 INTRODUÇÃO

Na metade dos anos 1990, a Internet estava começando a se expandir e formar uma comunidade global. Consequentemente, crescia uma demanda por uma linguagem de *scripting* que fosse executada diretamente no computador do usuário (*client-side*), ao invés de depender de validações e respostas do servidor (*server-side*), (W3SCHOOLS, 2021). Vale lembrar que as conexões de um usuário médio da Web eram lentas, principalmente se comparadas com os padrões atuais.

Portanto, com o intuito de reduzir o tempo gasto com validações no servidor e tornar a navegação mais fluida, a *Netscape Communications*, empresa até então líder, focou seus esforços para desenvolver uma linguagem de *scripting client-side*. Foi em 1995 que Brendan Eich, que trabalhava na Netscape, criou a linguagem *JavaScript*. Desenvolvida especificamente para execução em *browsers*, a programação em *JavaScript* tornou-se mais adequada à programação de interfaces de usuário baseadas em eventos para navegadores Web (GIZAS *et al.*, 2014).

De acordo com a pesquisa da W3Techs[3] (2021), a maioria dos sites da World Wide Web – mais de 97% – conta com a linguagem *JavaScript* em sua programação *client-side*. Em outra análise da mesma pesquisa, concluiu-se que a mais utilizada biblioteca de *JavaScript* nos websites é a *jQuery*, com 77,8% de uso absoluto entre os domínios analisados, representando cerca de 95,7% de participação no *market share*. A segunda biblioteca de *JavaScript* mais utilizada é a *Bootstrap*, com 22,1% de uso absoluto. É importante destacar que um site pode utilizar mais de uma linguagem e bibliotecas simultaneamente.

Essa dominância e popularidade do *jQuery*, segundo Gizas *et al.* (2014), deve-se a duas características principais: (i) seu mecanismo de localização de elementos de página ser baseado em seletores CSS, somado ao (ii) suporte para extensões, e à diversidade de plugins disponíveis, além da facilidade de criá-los. Além disso, conforme Gizas *et al.* (2014), a técnica chamada de *implicit iteration*, que dispensa a utilização de muitos construtores de *loops*, pois interage diretamente com um grupo de objetos, e não individualmente. Assim, o código é consideravelmente reduzido e simplificado.

Criada em 2006 pelo engenheiro de software John Resig, jQuery é uma biblioteca livre e de código aberto, cuja principal função é de simplificar a sintaxe de manipulação de interfaces DOM (*Document Object Model*) (W3SCHOOLS, 2021), e conta com métodos que facilitam a utilização de técnicas como a associação assíncrona entre JavaScript e XML(*Extensible Markup Language*), conhecido como (OPENJS FOUNDATION, 2021). No geral, jQuery proporciona uma programação *JavaScript* simplificada, encurtando e possibilitando maior eficiência aos códigos que utilizam seus métodos. Ainda que novas bibliotecas e *frameworks* de *JavaScript*, como *Angular*, *React* e *Vue*, estejam em ascensão, ainda possuem uma expressão ínfima se comparados com o *jQuery* nos domínios da Internet.

[3] W3Techs. Disponível em: https://w3techs.com/technologies/overview/client_side_language. Acesso em 21 de jun. 2021.

Verificando como o *jQuery* tem evoluído ao longo do tempo, nota-se a importância em executar uma análise histórica de sua qualidade. Realizar uma análise da qualidade de qualquer produto pode se tornar um assunto delicado, pois trata-se de um conceito amplo e relativo, onde as métricas adotadas podem limitar ou expandir o escopo da análise.

De acordo com a norma ISO/IEC 25000 (2014), Qualidade de Software é definida como a capacidade de um produto de software de satisfazer necessidades, explícitas e implícitas, quando utilizado sobre condições específicas.

A fim de examinar a qualidade de um software, faz-se necessária a utilização de métricas adequadas para cada situação e atributos específicos, como a capacidade de manutenção, eficiência e compatibilidade de um determinado código. Portanto, a decisão correta quanto à escolha dessas métricas torna-se crucial para a detecção e prognóstico de códigos com má qualidade.

Neste trabalho foi utilizado a plataforma *SonarQube*[4], para auxiliar na análise da qualidade do código do *jQuery*. Essa plataforma foi escolhida por ser um software de código aberto, destinado a inspeção contínua da qualidade do código, passível de detecção de *bugs*, *code smell* e vulnerabilidades de segurança. A plataforma foi desenvolvida pela empresa *SonarSource*[5], utilizada no contexto de ambientes de integração contínua. Com suporte para mais de 25 linguagens de programação, a função principal do *SonarQube* é de detectar os problemas no código, incorporando suas próprias configurações e regras, tanto de análise estática e dinâmica, mas também possibilitando a adição de novas regras (MARCILIO *et al.,* 2019).

O SonarQube considera três diferentes categorias de problemas nos códigos examinados: (i) *bug*, (ii) *vulnerabilidade* e (iii) *code smell*. Referente à confiabilidade, a ferramenta pode detectar automaticamente um bug que esteja causando uma falha ou resultado inesperado num programa. Quanto à segurança do código, o *SonarQube* busca por vulnerabilidades, que são pontos fracos no software que podem ser explorados para causar problemas ao sistema. Por fim, concernente à capacidade de manutenção, procura-se por code smells – problemas que não são nem um bug, nem uma vulnerabilidade, mas que podem complicar e prolongar todo o processo de manutenção e atualização de um código. Um *code smell* é uma indicação superficial que geralmente corresponde a um problema mais profundo no sistema (FOWLER, 2018).

Além desta introdução, na Seção 2 são abordados os trabalhos relacionados, advindos de uma busca *AD HOC*. Na Seção 3, são expostos os detalhes da metodologia utilizada durante a pesquisa. Na Seção 4, apresenta-se a análise do jQuery e uma discussão sobre os resultados. Na Seção 6 comenta-se as limitações deste trabalho e por fim, na Seção 6, são expostas as considerações finais, juntamente com os possíveis trabalhos futuros.

## 2 TRABALHOS RELACIONADOS

Tendo em vista a dominante presença histórica, tanto do *JavaScript*, quanto da biblioteca *jQuery*, na programação de aplicações Web, realizou-se uma busca AD HOC para identificar trabalhos que estão preocupados com a evolução do *jQuery*.

Com o intuito de conciliar a pesquisa em engenharia de software com os profissionais que trabalham com desenvolvimento de software, Graziotin e Abrahamsson (2013) propuseram um modelo de pesquisa para comparação analítica de *frameworks* de *JavaScript*. No entanto, os

[4] SonarQube. Disponível em: https://www.sonarqube.org/. Acesso em 21 de jun. 2021.

[5] SonarSourece. Disponível em: https://www.sonarsource.com/. Acesso em 21 de jun. 2021.

autores enfatizam a carência de pesquisas que ajudariam profissionais a escolher a melhor biblioteca/*framework* para situações específicas, e por fim, fazem um pedido de ação para que futuros trabalhos sejam realizados para enriquecer o assunto.

Outras pesquisas chegaram a comparar as bibliotecas de *JavaScript* (ROSALES-MORALES; ALOR-HERNÁNDEZ e JUÁREZ-MARTINEZ, 2011; MARIANO, 2017), porém, não contam com uma ênfase na evolução da biblioteca. Logo, os trabalhos anteriores tiveram o objeto de estudar *JavaScript* e a biblioteca *jQuery*, em contextos específicos.

A fim de entender as causas e impactos das falhas de *JavaScript* em aplicações Web, Oscariza Jr *et al.* (2013) realizaram um estudo empírico com mais de 300 relatórios de *bugs* de *client-side JavaScript*. A análise das falhas de *JavaScript* a partir de mensagens de erros no console e por análise estática, ressaltaram a importância da análise de *bugs*, pois os relatórios

conforme Oscariza Jr *et al.* (2013), apresentaram informações vitais sobre a causa fundamental de uma falha e o seu comportamento esperado. Concluiu-se que a maioria das falhas analisadas, assim como grande parte das falhas consideradas de alto nível de severidade, estão relacionadas ao DOM. Sendo tal característica uma das principais funcionalidades do *jQuery,* isto é, facilitar o acesso e manipulação dos DOMs.

Gizas, Christodoulou e Pomonis (2013) publicaram um trabalho examinando a performance e qualidade do jQuery, porém com uma amostra limitada das versões analisadas. Foram adotados três tipos diferentes de métricas para testes de qualidade: quanto ao tamanho, à complexidade e à capacidade de manutenção. Para a condução dos testes, foram utilizadas as ferramentas *Cloc*[6] e *Jsmeter*[7]. Os autores mencionam a pré-existência de uma pequena e genérica literatura sobre a comparação, análise e teste de qualidade de linguagens de programação, reafirmando a necessidade de produção de trabalhos, métricas e propostas para aprimorar a análise qualitativa, não somente dos programas que utilizam *jQuery*, mas também qualquer outra biblioteca de *JavaScript*.

Neste trabalho, utilizou-se a ferramenta *SonarQube* para analisar a evolução da biblioteca *jQuery*. Verifica-se na Figura 1, uma linha do tempo apresentando a ordem cronológica em que os trabalhos foram apresentados.

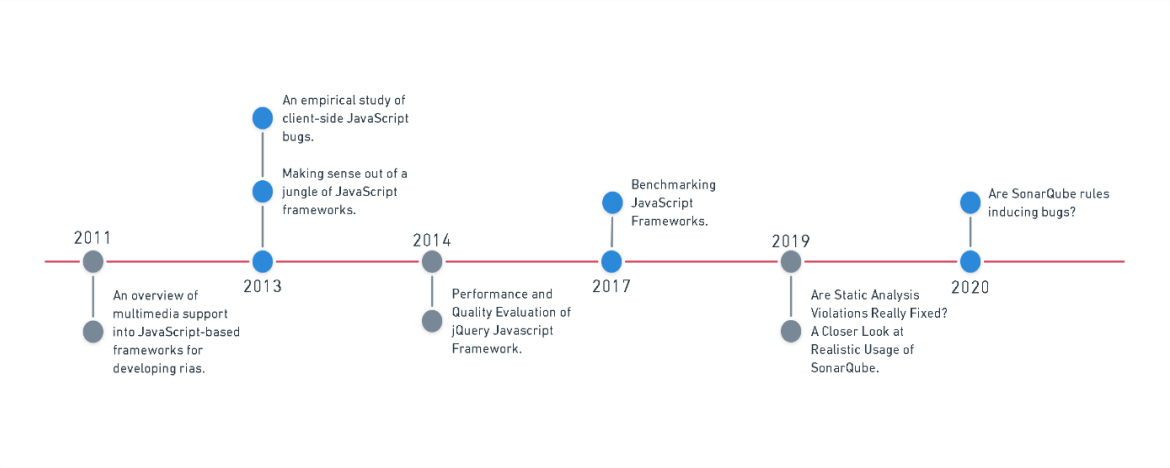


Figura. 1. Linha do tempo dos trabalhos relacionados. Fonte: Elaboração própria.

[6] CLOC. Disponível em: http://cloc.sourceforge.net/. Acesso em 21 de jun. 2021.

[7] JSMeter. Disponível em: https://github.com/PhoSor/jsmeter-online. Acesso em 21 de jun. 2021.

## 3 METODOLOGIA

A metodologia da pesquisa iniciou-se com uma leitura ad hoc sobre o assunto, seguida pela separação dos trabalhos relacionados, realizando a integração e estruturação das amostras coletadas. Além disso, definiu-se o uso da ferramenta SonarQube, que conforme Lenarduzzi et al. (2019) permite investigar questões relacionadas aos bugs, crescimento do software, blocos duplicados, às regras de segurança e manutenção. Com base no SonarQube, decidiu-se as métricas a serem utilizadas e por fim, a análise experimental de dados via métodos estatísticos, seguindo modelo descrito por Sirqueira *et al.* (2020). O fluxo metodológico encontra-se na Figura 2.

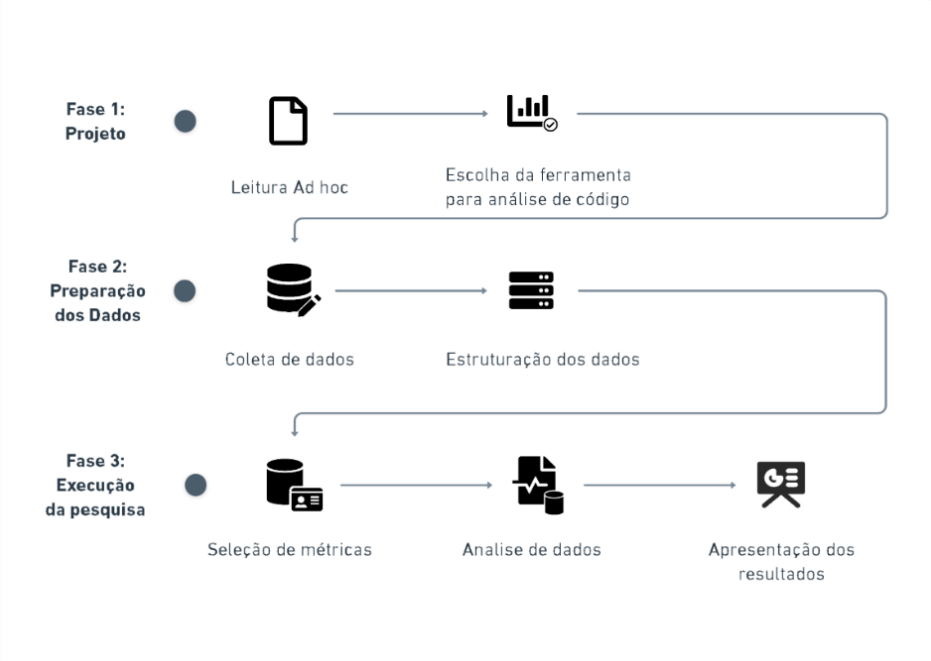


Figura. 2. Metodologia do trabalho. Fonte: Elaboração própria.

Com o objetivo de analisar historicamente o jQuery, a coleta de dados foi realizada por meio do repositório oficial da biblioteca no GitHub[8]. Foram extraídas 83 versões do *jQuery*, abrangendo desde a versão 1.0, datada de agosto de 2006, até a versão mais atual, 3.6, lançada em março de 2021.

O *jQuery* utiliza o versionamento semântico para enumerar os seus lançamentos. Dessa forma, temos o versionamento dividido em três partes, identificados da seguinte forma: “Maior. Média.Menor”, isto é, “Maior” representa mudanças incompatíveis com a versão anterior biblioteca, “Média”, funcionalidades adicionadas mantendo compatibilidade com a versão Maior, e “Menor”, que representa a correção de falhas mantendo a compatibilidade.

A partir dos dados coletados e organizados, importou-se as amostras para o software *SonarQube* e realizada a análise estática dos códigos. As métricas utilizadas foram de confiabilidade (*bugs*), capacidade de manutenção (*code smells*), e blocos duplicados, essa última abrangendo tanto o percentual de código duplicado, como a quantidade total de blocos

[8] jQuery. Disponível em: https://github.com/jquery/jquery. Acesso em 21 de jun. 2021.

duplicados. Ressalta-se que, não foi utilizado vulnerabilidade devido a uma limitação da ferramenta.

A metodologia da análise experimental dessa pesquisa foi fundamentada em métodos estatísticos, seguindo a árvore de decisões proposta por Sirqueira *et al.* (2020). Respondendo positivamente à questão inicial "A amostra possui mais de 30 valores?", seguimos para o próximo passo que sugere a realização do Teste de Kolmogorov-Smirnov (K-S). O teste K-S avalia a semelhança entre a distribuição de duas amostras, também indicando similaridade da distribuição entre uma amostra em relação a uma distribuição clássica. Dessa forma, podemos verificar a normalidade das amostras com base nas hipóteses:

- **H0** (hipótese nula): As amostras apresentam distribuição normal.
- **H1** (hipótese alternativa): As amostras não apresentam distribuição normal.

Diante dos resultados de cada métrica, foram aplicados os métodos estatísticos correspondentes, sugeridos pela árvore de decisões. Ressalta-se que, como foram analisadas três diferentes versões, todas as análises foram utilizando métodos com um fator(Variável Analisada) e mais de dois tratamentos(Versões do *jQuery*).

## 4 ANÁLISE E DISCUSSÃO

O *SonarQube* nos oferece uma considerável quantidade de métricas para diferentes características e focos analíticos, porém, neste trabalho não utilizamos todas as métricas disponíveis, mas sim uma diminuta quantidade devido ao núcleo do estudo que se concentra na evolução de software. As métricas escolhidas são apresentadas a seguir, com suas respectivas *strings* atribuídas no SonarQube entre parênteses:

- **Bugs** (bugs): número total de problemas definidos como bugs;
- **Code smells** (code_smells): número total de problemas definidos como code smells;
- **Duplicated blocks** (duplicated_blocks): número total de blocos de linhas duplicados;
- **Lines of code** (loc): número total de linhas que contêm pelo menos um caractere que não seja um espaço em branco, tabulação, nem parte de um comentário.

Ao examinar o código-fonte, toda vez que ocorre uma violação de um dado conjunto de regras adotado pela análise no *SonarQube*, o programa marca a violação como uma *issue* – um problema no código. As *issues* são categorizadas em três diferentes tipos (*Bug*, *Vulnerability* e *Code Smell*) e em cinco diferentes níveis de severidade: (i) *blocker*, (ii) *critical*, (iii) *major*, (iv)*minor* e (v) *info*.

Um problema classificado como tendo severidade **Blocker** é um bug com alta probabilidade de impactar o comportamento da aplicação e precisa ser corrigido imediatamente. **Critical** são as *issues* que precisam ser revisadas o mais rápido possível e consiste ou em bugs com baixa probabilidade de impacto no comportamento da aplicação, ou em alguma vulnerabilidade. Definidos como **Major** são as falhas de qualidade que podem atrapalhar fortemente a produtividade do desenvolvedor, como blocos duplicados e parâmetros não utilizados. Por sua vez, um problema **Minor** atrapalha ligeiramente o desenvolvedor, que é o caso de linhas de código que não precisam ser tão longas. **Info** são apenas descobertas dentro do código que não são nem um bug, nem uma falha de qualidade.

Somados aos diferentes níveis de severidade, é possível entender como o *SonarQube* interpreta diferentes problemas no código ao saber como define os tipos de regras, sendo quatro: (i) *Bugs, (ii) Vulnerability, (iii) Security Hotspot* e *(iv) Code Smell*.

São feitas diferentes perguntas e suas respostas categorizam o tipo de regra. Caso a regra seja sobre partes do código que estão comprovadamente erradas, ou com maior

probabilidade de estar errada, trata-se de uma regra do tipo Bug. Logo, infere-se que *bugs* categorizados pelo *SonarQube* como problemas explícitos, com código que impactam o comportamento esperado do software e que precisam ser corrigidos urgentemente. Se as regras analisadas estabelecem violações relacionadas a um código que pode ser explorado por um hacker ou que apresenta informações sensíveis de segurança, elas são classificadas como regras de *Vulnerability* e de *Security Hotspot*, respectivamente. As regras do tipo *Code Smell*, portanto, são aquelas que não estão classificadas como um bug, ou como um problema de segurança, mas estão atreladas à manutenção do código fonte.

Além disso, informações sobre o código-fonte examinado, envolvendo os blocos duplicados e linhas totais de código também são considerados problemas de manutenção. Para a análise de *bugs* e *code smells*, o SonarQube não garante que não sejam retornados nenhum falso positivo, evitando que o desenvolvedor tenha que ficar na dúvida se o problema precisa realmente ser corrigido. Isso acaba sendo uma limitação da ferramenta.

Para um bloco de código em *JavaScript* ser considerado como bloco duplicado (métrica *duplicated blocks*), é necessário atender a dois requisitos: (i) possuir ao menos 100 entradas sucessivos e duplicados, e que (ii) esses entradas estejam distribuídos em pelo menos 10 linhas de código.

Vale ressaltar que o SonarQube disponibiliza perfis personalizados, apresentando configurações pré-estabelecidas e que podem ser adequadas para diferentes naturezas de análise, sendo customizados os parâmetros. Nesse estudo foi utilizado o perfil *default*, ou seja, com as configurações padrão de regras definidas pelo software e que foram supracitadas. Todas as regras padrão utilizadas pelo SonarQube para análise estática de *JavaScript* podem ser acessadas no SonarSource.com[9].

Com base nas métricas e regras empregadas no estudo, são apresentados a seguir os resultados discutidos[10].

Entre as versões 1.0 e 1.2 (Figura 3), identificamos um padrão na primeira porção de todos os gráficos apresentados. A quantidade de *issues* e duplicações até a versão 1.1 são limitadas pelo próprio tamanho do código-fonte, que não superam o valor de 10 mil linhas. Porém, entre a versão 1.1 e 1.2 há um crescimento no tamanho do código, atingindo mais de 45 mil linhas, mas declinando na versão 1.2 para 9.357 linhas. O valor volta a crescer a partir da versão 1.3.

Um processo semelhante ocorre nas versões mais recentes do jQuery. Após a versão 3.0, identificam-se variações significantes nas linhas de código, evidenciadas por duas quedas nas versões 3.3 e 3.5, onde o tamanho do código reduz em 20 mil linhas. As duas baixas são adjacentes à uma subida proporcional durante a versão 3.4. Essa análise pode ser vista na Figura 3.

Ao traçar esse paralelo entre os padrões de variância de *bugs, code smells* e blocos duplicados, com a evolução das linhas totais que compõem o código fonte, é possível identificar mudança na versão média estáveis, quando apresentam correlação com a taxa de crescimento e decrescimento do tamanho do código no mesmo período analisado.

[9]Regras do SonarQube. Disponíveis em: https://rules.sonarsource.com/javascript/. Acesso em 21 de jun. 2021.
[10]Dados coletados. Disponíveis em: https://github.com/Faelxld/dados-jquery . Acesso em 20 de jun. 2021.

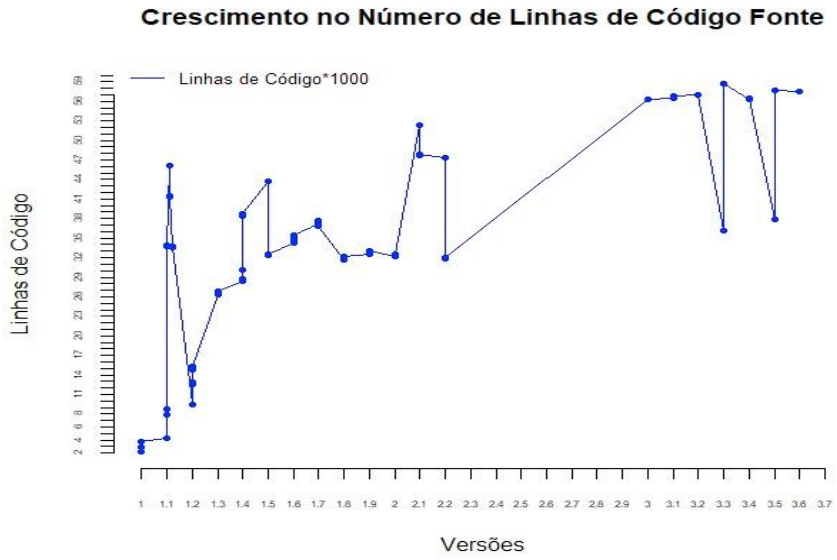


Figura. 3. Crescimento do número de linhas de código fonte. Fonte: Elaboração própria.

Na Figura 4 observa-se a quantidade de *bugs* detectados em cada amostra. Antes de atingir seu valor absoluto máximo – 183 bugs identificados na versão 1.5 – nota-se uma alta variação no número de *bugs*, que acompanha a curva de números de linhas de código, entre as versões 1.0 e 1.2. A amplitude de variação diminui e se estabiliza a partir das versões 2.0 em diante, mantendo uma quantidade de *bugs* entre 90 e 120.

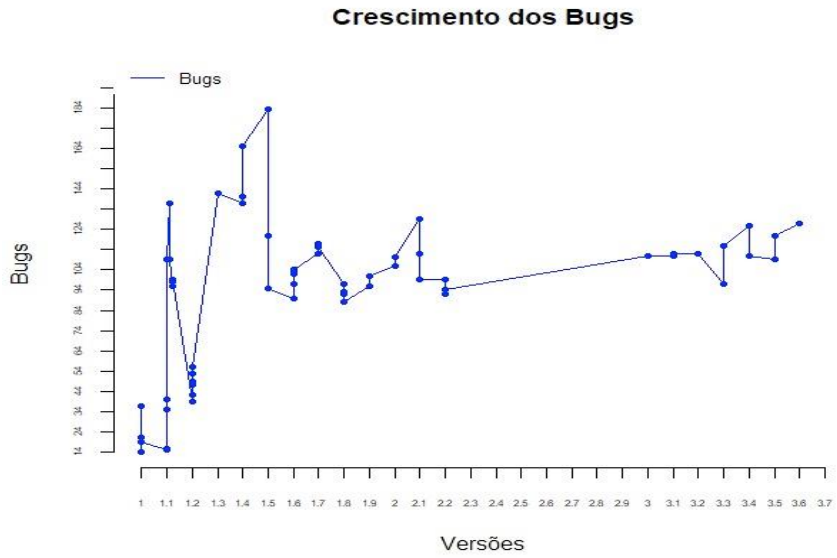


Figura. 4. Gráfico do crescimento de *bugs*. Fonte: Elaboração própria.

A evolução de *bugs* presentes nas versões também representa o mesmo padrão inicial em sua intensidade, mas, apesar de ter baixas entre as versões 3.3 e 3.5, não demonstra uma mesma taxa de alteração.

Assim como a evolução dos bugs (Figura 4), a presença histórica de *code smells* no jQuery, representada pela Figura 5, também apresenta a mesma variação que acompanha o

número de linhas de código-fonte entre as versões 1.0 e 1.2. O valor máximo também foi detectado na versão 1.5 (1200 *code smells*), tendo sua variação estabilizada durante as versões acima da 2.2.

Porém, a partir da versão 3.0 do *jQuery*, os *code smells* voltam a apresentar uma variação mais ampla e expressiva se comparada às variações de *bugs* durante a mesma faixa de amostras, podendo traçar um paralelo mais forte com o padrão final da evolução do tamanho dos códigos fonte.

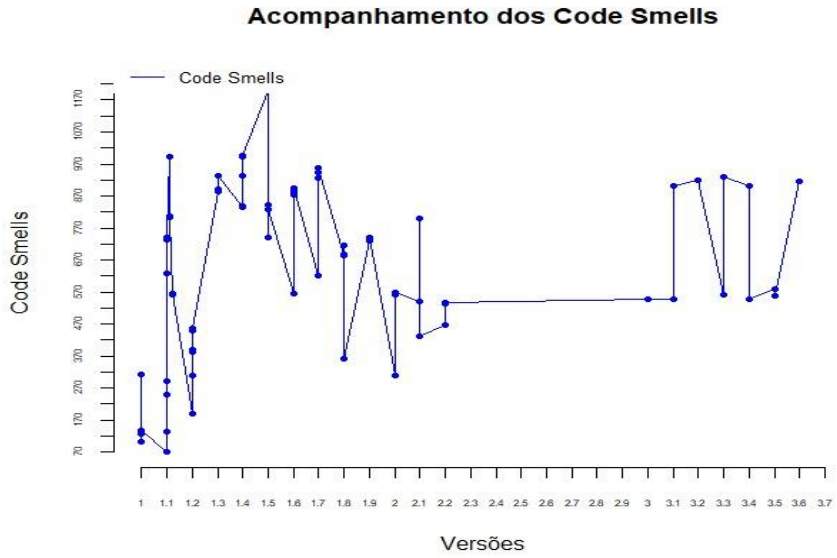


Figura. 5. Gráficos da evolução de *code smells*. Fonte: Elaboração própria.

O comportamento da curva evolutiva da métrica de Duplicações (Figura 6) é muito semelhante ao presente no gráfico de linhas totais no código-fonte. Além dos dois padrões iniciais, é percebida uma queda significativa na versão 2.2, sucedida por um crescimento até as versões 3.1.

O percentual de código duplicado teve seu maior valor na versão 2.2, atingindo 66,58% de código duplicado, mas apresenta uma média recente de cerca de 40% de duplicações.

Já a métrica de blocos duplicados (Figura 7) também apresenta semelhança com os padrões destacados anteriormente, porém teve uma taxa de redução entre as versões 1.5 e 2.0 que não é tão aparente nas curvas de tamanho do código e de duplicações. Portanto, entre as versões 1.5 e 2.0, o jQuery conseguiu reduzir a quantidade proporcional de blocos duplicados no código fonte.

Na segunda parte da análise, examina-se a hipótese de que uma suposta estabilidade, identificada através da observação visual dos gráficos, pode ser corroborada com a utilização de métodos estatísticos e os testes de hipóteses.

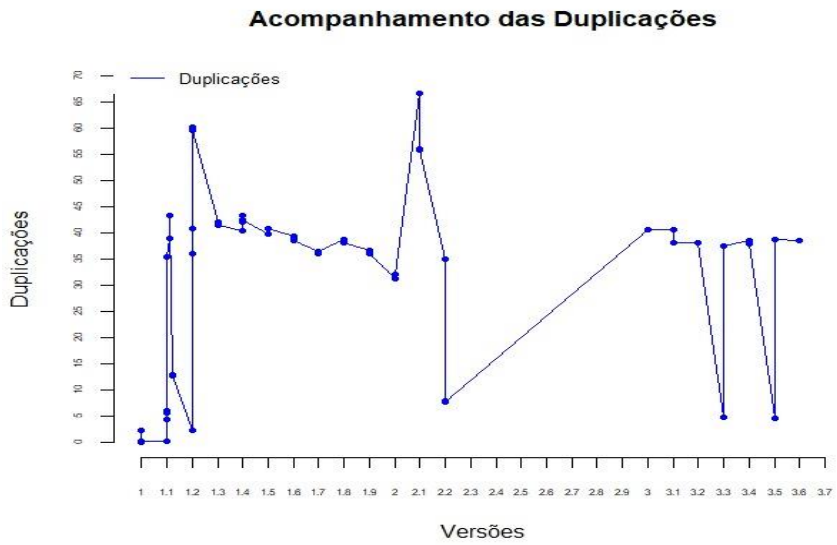


Figura. 6. Gráfico de acompanhamento das Duplicações. Fonte: Elaboração própria.

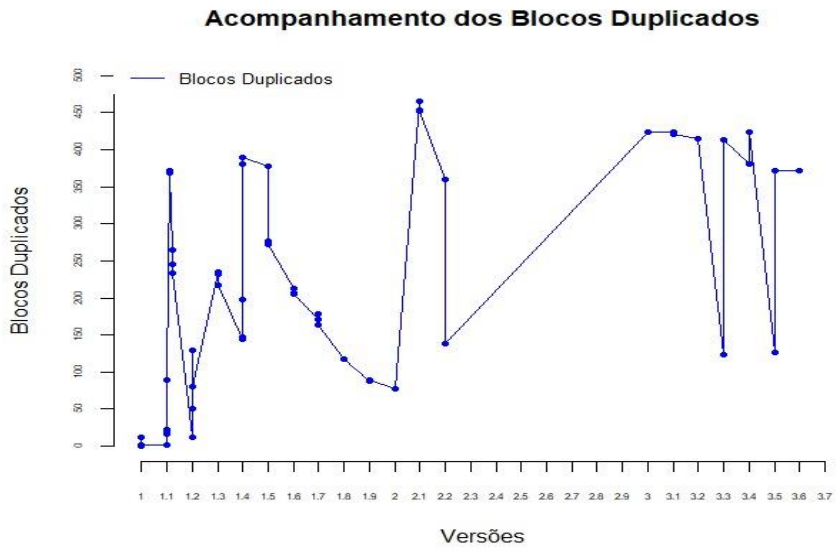


Figura. 7. Gráfico de acompanhamento dos Blocos Duplicados. Fonte: Elaboração própria.

Durante a segunda parte da análise, as versões do *jQuery* foram agregadas em grupos referentes às três versões maiores (1.x, 2.x e 3.x), cada métrica e suas medidas de tendência apresentadas em tabelas, por fim realizando a sequência de testes indicados pela árvore de decisões, com o intuito de identificar as distribuições, diferenças entre médias e demais características. Ao realizar os testes, as métricas selecionadas são os fatores, enquanto as versões são os tratamentos de cada análise de testes.

### 4.1 Bugs

Através da análise descritiva dos Bugs, têm-se os resultados apresentados na Tabela 1. Para a realização dos testes estatísticos o fator considerado foi Bugs, e o tratamento das Versões.

Tabela 1. Análise descritiva de bugs. Fonte: Elaboração própria.

| Bugs | |
|---|---|
| Média | 96,08 |
| Mediana | 104 |
| Mínimo | 14 |
| 1º Quartil | 92 |
| 3º Quartil | 113,5 |
| Máximo | 183 |

Iniciando as verificações para identificar a normalidade das amostras a partir do Teste de Kolmogorov-Smirnov (K-S), considera-se as duas hipóteses:

- H0 (hipótese nula): As amostras apresentam distribuição normal.
- H1 (hipótese alternativa): As amostras não apresentam distribuição normal.

Para consideração da hipótese nula, o nível de significância precisa ser superior ao valor estabelecido de 5% (*p-value* > 0,05). Caso seja inferior (*p-value* < 0,05), consideramos a hipótese alternativa.

Após a realização do teste K-S, obtém-se que *p-value* = 0.0008554, tem-se como resultado a aceitação da hipótese alternativa (H1), indicando que as amostras não apresentam distribuição normal. A seguinte etapa é a aplicação de um teste não-paramétrico para um fator e mais de dois tratamentos. Executamos o Teste *Kruskal-Wallis* para determinar se há diferença entre as médias no número de bugs por versão. Para isso, consideramos as seguintes hipóteses:

- H0 (hipótese nula): Não há diferença entre as médias.
- H1 (hipótese alternativa): Há diferença entre as médias.

Tendo como parâmetro o mesmo nível de significância, aceitamos novamente a hipótese alternativa no segundo teste, pois o resultado do *p-value* obtido é de 0.04053, ligeiramente inferior aos 5%. Ou seja, não apresenta estabilidade da quantidade de bugs entre as diferentes versões do *jQuery*. Com isso, para descobrirmos quais versões apresentavam semelhança, realizou-se uma comparação par-a-par utilizando o Teste de *Wilcoxon-Mann-Whitney*. O resultado pode ser visto na Tabela 2, onde observa-se que existe semelhança entre o número médio de bugs entre as versões 1 e 2; contudo na versão 3 o número de bugs não segue a média das versões anteriores.

Tabela 2. Saída do teste de Wilcoxon-Mann-Whitney para Bugs. Fonte: Elaboração própria.

| - | 1 | 2 |
|---|---|---|
| 2 | 0.414 | - |
| 3 | 0.025 | 0.012 |

### 4.2 LoC

Observando a Tabela 3, temos a análise descritiva das linhas totais de código fonte, sendo notável a diferença entre o valor mínimo, presente nas versões iniciais do *jQuery*, e o valor máximo, atingido a partir das versões 3.0 da biblioteca.

Tabela 3. Análise descritiva das Linhas de Código. Fonte: Elaboração própria.

| Linhas de Código | |
|---|---|
| Média | 32.115 |
| Mediana | 33.112 |
| Mínimo | 2.279 |
| 1º Quartil | 26.707 |
| 3º Quartil | 40.078 |
| Máximo | 58.713 |

Aplicando-se primeiramente o teste K-S para verificar a normalidade das amostras considerando as mesmas hipóteses da análise de *bugs*, o *p-value* obtido foi de 0.01175. Logo, aceitamos a hipótese alternativa (H1), e seguimos para o teste de Kruskal-Wallis para determinar a existência de diferença entre as médias dos agrupamentos, com as mesmas hipóteses da análise anterior.

Para esse teste, obteve-se o *p-value* de 1.262e-07, aceitando a hipótese alternativa (H1), uma vez que detectadas diferenças entre as médias, isto é, ao menos um grupo é diferente dos demais. Portanto, realizamos o Teste de Wilcoxon-Mann-Whitney para verificar qual grupo era diferente. O resultado (Tabela 4) demonstra que nenhum grupo possui semelhança entre si com base no número de linhas de código.

Tabela 4. Saída do teste de Wilcoxon-Mann-Whitney para LoC. Fonte: Elaboração própria.

| - | 1 | 2 |
|---|---|---|
| 2 | 0.00835 | - |
| 3 | 3e-07 | 0.00023 |

### 4.3 Code Smell

Na análise de Code Smell, a Tabela 5 apresenta a análise descritiva da variável e seguindo o mesmo fluxo das análises anteriores, realizou-se o teste de K-S para identificar se os dados seguem uma distribuição normal. Foram consideradas as seguintes hipóteses:

- H0 (hipótese nula): As amostras apresentam distribuição normal.
- H1 (hipótese alternativa): As amostras não apresentam distribuição normal.

O *p-value* resultante é de 0.1688, superando o nível de significância, sendo assim, aceita-se a hipótese nula (H0): as amostras apresentam distribuição normal.

Tabela 5. Análise descritiva de Code Smells do jQuery. Fonte: Elaboração própria.

| Code Smells | |
|---|---|
| Média | 630,4 |
| Mediana | 620 |
| Mínimo | 72 |

| | |
|---|---|
| 1º Quartil | 453,5 |
| 3º Quartil | 879,5 |
| Máximo | 1.200 |

Com base neste resultado, é necessário verificar se os dados são homocedasticos, isto é, se todos os valores da variável tiverem a mesma variância. Para essa verificação utiliza-se o Teste de Levene, considerando as seguintes hipóteses:

- H0 (hipótese nula): As amostras são homocedásticas.
- H1 (hipótese alternativa): As amostras não são homocedásticas.

Após a aplicação do teste Levene, obtém-se o *p-value* de 0.002936. Atesta-se então que as amostras não são homocedásticas, ao nível de significância de 5%. Portanto, assumindo a hipótese alternativa, seguimos a orientação da árvore de decisões de Sirqueira *et al.* (2020) para aplicar um teste não-paramétrico. O Teste *Kruskal-Wallis* para os *Code Smells*, considerou as hipóteses a seguir:

- H0 (hipótese nula): Não há diferença entre as médias.
- H1 (hipótese alternativa): Há diferença entre as médias.

Como resultado observa-se um *p-value* ligeiramente superior (0.05619) ao nível de significância estabelecido, aceitando então a hipótese nula. Assim, concluímos que não há diferença entre as médias de *code smells* presentes nas versões analisadas, mantendo-se como um fator historicamente estável entre as versões da biblioteca.

### 4.4 Duplicações

A Tabela 6 representa os valores resultantes da análise descritiva das duplicações de código presentes nos agrupamentos de versões analisadas do jQuery.

Tabela 6. Análise descritiva de Duplicações no jQuery. Fonte: Elaboração própria.

| **Duplications** | |
|---|---|
| Média | 31,19 |
| Mediana | 38 |
| Mínimo | 0 |
| 1º Quartil | 12,9 |
| 3º Quartil | 40,48 |
| Máximo | 66,58 |

Inicia-se a análise determinando se a distribuição é normal, utilizando o Teste K-S, segundo a mesma hipótese do teste anterior. Como resultado, obteve-se o *p-value* de 8.459e-06, expressivamente inferior aos níveis de significância estabelecidos. Aceita-se então, a hipótese alternativa (H1), determinando que as amostras não apresentam normalidade em sua distribuição.

O próximo passo é aplicar um teste não-paramétrico, o Teste de Kruskal-Wallis, seguindo a mesma hipótese da seção anterior. Com um *p-value* de 0.9973, consideramos a hipótese H0, determinando que as duplicações se mantêm estáveis entre as versões.

### 4.5 Blocos Duplicados

Representando também as duplicações encontradas no jQuery, a Tabela 7 exibe a estatística descritiva de uma métrica particular de duplicações, dentre as adotadas pelo *SonarQube*. Essa métrica considera os valores absolutos de blocos duplicados encontrados, sendo necessários para se enquadrar na métrica, ao menos 100 *tokens* sucessivos e duplicados, sendo eles distribuídos em, no mínimo, 10 linhas de código.

Tabela 7. Análise descritiva de Blocos Duplicados. Fonte: Elaboração própria.

| Duplications Blocks | |
|---|---|
| Média | 200,5 |
| Mediana | 171 |
| Mínimo | 0 |
| 1º Quartil | 84,5 |
| 3º Quartil | 370 |
| Máximo | 465 |

Iniciando com o teste K-S, com o objetivo de determinar se as amostras apresentam distribuição normal, conforme as hipóteses h0 e h1 a seguir:

- H0 (hipótese nula): As amostras apresentam distribuição normal.
- H1 (hipótese alternativa): As amostras não apresentam distribuição normal.

O resultado do teste K-S revela o *p-value* de 0.1668, que supera o nível de significância de 5%. Descartamos a hipótese alternativa, e seguimos a análise aceitando a hipótese nula (H0), de que a distribuição das amostras é normal.

Em seguida, realizamos o teste Levene para verificar a homocedasticidade das amostras. O *p-value* retornado foi de 0.05505, descartando novamente a hipótese alternativa e nos indicando que os dados são homocedásticos; consequentemente o próximo passo é o uso de um teste paramétrico.

Para determinar o teste utilizado, levamos em consideração a quantidade de tratamentos do estudo. Estamos considerando mais de dois tratamentos, logo adotamos o Método ANOVA, ao invés do Teste T. O Método ANOVA, também conhecido como Análise de Variância, testa a igualdade entre as médias de dois ou mais grupos, funcionando como uma alternativa paramétrica ao Teste Kruskal-Wallis, que foi aplicado em boa parte desse trabalho.

Ao aplicar o ANOVA, consideramos as seguintes hipóteses:

- H0 (hipótese nula): Não há diferença entre as médias.
- H1 (hipótese alternativa): Há diferença entre as médias.

Após a realização do teste, podemos aceitar a hipótese de que há diferença entre as médias analisadas, pois o *p-value* é 1.04e-06, valor consideravelmente inferior ao 0.05 estabelecido como nível de significância. Os blocos duplicados não demonstram estabilidade entre as diferentes versões do jQuery. Seguiu-se então para um teste par-a-par entre as versões para determinar quais eram diferentes entre si conforme o teste de Wilcoxon-Mann-Whitney. O resultado pode ser visto na Tabela 8 e indica que não existe diferença significativa a 5% entre as versões 1 e 2; e não existem também diferenças significativas entre as versões 2 e 3. Contudo, ao nível de significância de 5%, temos uma diferença significativa entre o número de bloco duplicados entre as versões 1 e 3 do jQuery.

Tabela 8. Saída do teste de Wilcoxon-Mann-Whitney para Blocos Duplicados. Fonte: Elaboração própria.

| - | 1 | 2 |
|---|---|---|
| 2 | 0.13 | - |
| 3 | 2.4e-05 | 0.49 |

## 5 Limitações da pesquisa

No que tange a proposta da pesquisa e a interpretação dos resultados, alguns fatores limitaram o estudo. Quanto à coleta de dados, as amostras – versões do jQuery – foram retiradas diretamente do repositório oficial da biblioteca no GitHub[11]. As versões *beta, alpha* e versões candidatas (rc) foram descartadas, pois analisamos somente as versões estáveis lançadas.

Outra limitação do nosso trabalho está na seleção de trabalhos relacionados, uma vez que limitou-se a uma busca *ad hoc* de artigos, abordando JavaScript e biblioteca jQuery. Dessa forma, nosso estudo foi guiado sem estudos prévios sólidos.

Um fator que restringiu uma análise mais aprofundada foi a ferramenta utilizada para a análise estática dos códigos-fonte, o *SonarQube*. Afinal, não foi possível realizar a análise e aplicação de testes nos problemas relacionados às vulnerabilidades e demais métricas de segurança do código. Também deve-se levar em consideração as regras adotadas pela ferramenta, das quais utilizamos o perfil de regras padrão. Como a ferramenta permite regras customizadas, estas podem ser mais adequadas para a análise de bibliotecas *JavaScript* e em especial do *jQuery*.

## 6 CONSIDERAÇÕES FINAIS

A metodologia com aplicação de métodos estatísticos e testes de hipótese, possibilitou entender, principalmente, a distribuição e estabilidade das métricas adotadas no trabalho. Primeiramente, geramos e interpretamos os gráficos da presença histórica de cada métrica, identificando possíveis pontos de estabilidade, a partir da comparação visual com o crescimento do tamanho do código fonte, tido como referência indissociável para a análise das outras métricas de valor absoluto. Padrões de comportamento em diferentes seções dos gráficos também foram identificados e considerados na análise comparativa.

Posteriormente, essas observações foram corroboradas através do método estatístico e testes de hipóteses, considerando um nível de significância de 5%. Na realização dos testes, um valor igual ou superior ao nível de significância, aceita-se a hipótese nula (H0). Caso o *p-value* esteja abaixo do nível de significância, aceita-se a hipótese alternativa (H1). Na análise estática, as métricas (bugs, linhas de código, *code smells*, duplicações e blocos duplicados) foram consideradas como os fatores durante as análises, enquanto os agrupamentos das versões maiores, os tratamentos.

Analisando de forma geral, as *issues* adotadas para o estudo não sofreram grandes alterações, principalmente se consideradas as versões mais recentes (a partir da 3.0). Bugs, *code smells* e duplicações mantiveram um valor proporcional ao crescimento do código fonte.

A forte presença da utilização do jQuery em aplicações web *client-side* torna esse trabalho importante, não só para os desenvolvedores do jQuery mas para a comunidade que o utiliza, pois norteia sobre o futuro da biblioteca e como a mesma vem sendo mantida e evoluída

[11] jQuery. Disponível em: https://github.com/jquery/jquery. Acesso em 21 de jun. 2021.

ao longo do tempo. O decaimento da biblioteca ou mesmo a perda de qualidade acaba impactando milhares de sites da internet.

É comum os desenvolvedores não se preocuparem com as bibliotecas que utilizam em seus projetos, mas isso é importante, dados que podem afetar a performance, a segurança e a confiabilidade do produto final. Atentarmos a evolução das bibliotecas é uma forma de nos precaver quanto a escolhas decadentes e que podem afetar o ciclo de vida de um software no futuro.

## REFERÊNCIAS

FOWLER, Martin. Refactoring: improving the design of existing code. **Addison-Wesley Professional**, 2018.

GIZAS et al. Performance and Quality Evaluation of jQuery Javascript Framework. In: **Information Engineering, vol 3**. Science and Engineering Publishing Company. 2014. p. 12-22.

GIZAS, A.; CHRISTODOULOU, S.P; POMONIS, T. Performance and Quality Evaluation of jQuery Javascript Framework. **Information Engineering. Vol. 3**. Science and Engineering Publishing Company, 2014.

GRAZIOTIN, D; ABRAHAMSSON, P. Making sense out of a jungle of JavaScript frameworks. **International Conference on Product Focused Software Process Improvement**. Springer, Berlin, Heidelberg, 2013.

ISO/IEC 25000. **Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SquaRE**. 2014.

LENARDUZZI, V. et al. Are SonarQube rules inducing bugs?. **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. IEEE, 2020.

MARCILIO et al. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. **2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).** 2019. pp. 209-219

MARIANO, C. L. **Benchmarking JavaScript Frameworks**. Masters dissertation, 2017. doi:10.21427/D72890

OSCARIZA, F. et al. An empirical study of client-side JavaScript bugs. **2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.** IEEE, 2013.

ROSALES-MORALES, V.; ALOR-HERNÁNDEZ, G.; JUÁREZ-MARTINEZ, U. An overview of multimedia support into JavaScript-based frameworks for developing rias. **CONIELECOMP 2011, 21st International Conference on Electrical Communications and Computers**. IEEE, 2011.

SIRQUEIRA, T. et al. Aplicação de Métodos Estatísticos em Engenharia de Software: Teoria e Prática. **Engenharia no Século XXI, v. 18**. Belo Horizonte: Editora Poisson, 2020. p. 229-246.

W3TECHS. **Usage statistics of client-side programming languages for websites**. 2021. Acesso em mai 2021. <https://w3techs.com/technologies/overview/client_side_language>

W3SCHOOLS. **History of JavaScript.** 2021. Acesso em jun 2021.<https://www.w3schools.in/javascript-tutorial/history-of-javascript/>

OPENJS FOUNDATION. **jQuery Offical Website**. Acesso em jun. 2021. <https://jquery.com/>

W3SCHOOLS. **What is jQuery?.** Acesso em jun. 2021. <https://www.w3schools.com/jquery/jquery_intro.asp>