UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

Um Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript

Daniela Rocha Silva
Luis Felipe Bentin Sobral

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL

JULHO DE 2017

Um Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript

Daniela Rocha Silva

Luis Felipe Bentin Sobral

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada por:

Prof. Márcio de Oliveira Barros, DSc. (UNIRIO)

Prof. Gleison dos Santos Souza, DSc. (UNIRIO)

Fábio de Almeida Farzat, MSc. (COPPE/UFRJ)

RIO DE JANEIRO, RJ – BRASIL

JULHO DE 2017

Agradecimentos

Gostaríamos de agradecer em primeiro lugar ao nosso orientador Márcio Barros, pelo o apoio, dedicação, preocupação, puxões de orelha, por ter-nos guiado muito bem durante todo o processo de construção desse estudo estando sempre próximo e disponível para auxiliar-nos em qualquer questão e, principalmente, por ter-nos encorajado do início ao fim, sendo fundamental para nos fazer acreditar em nós mesmos e que tínhamos a capacidade de concluir este trabalho apesar de todas as dificuldades de percurso e do curto período de tempo disponível. Obrigado.

Agradecemos também a todos os nossos amigos da UNIRIO que percorreram esses quatro anos e meio conosco, em especial a Amanda Rodrigues, Hugo Bertoche, Igor Balteiro, Luiz Paulo Carvalho, Matheus Costa, Victor Farias e Gian Biolchini. Valeu pessoal, vocês são demais.

Daniela Rocha Silva

Agradeço a minha mãe e meus irmãos pelo apoio essencial e pelo orgulho que têm de mim. Amo vocês. Agradeço aos meus colegas Daiane Correa, Paulo Victor Selano e Lucas Azevedo pelas sessões de terapia e pela amizade maravilhosa de vocês, seus lindos. Por último e mais importante, agradeço ao meu *partner in crime* Luis Felipe pela parceria incrível durante todo esse período de UNIRIO e por ser a coisa mais maravilhosa que eu vou levar dessa faculdade. Sofremos, mas conseguimos. Te amo.

Luis Felipe Sobral

Agradeço primeiramente a minha família pelo apoio que sempre me deram. Agradeço também a todos os amigos que fiz nesse tempo da UNIRIO, e por todos os momentos bons e as felicidades que passamos juntos. Aos amigos que fiz na vida, Alexia Ferreira, Flávio Crelier, Gabriel Santos, Glenda Carvalho, Gustavo Kusdra, que me apoiaram durante todo o árduo processo de escrita deste trabalho. Por último, mas não menos importante – a mais importante na verdade – a minha grande amiga e parceira Daniela, que não só esteve ao meu lado nesse trabalho como em todos os outros da UNIRIO, e principalmente fora dela, sempre me oferecendo um ombro amigo quando precisei. Te amo.

RESUMO

A linguagem JavaScript surgiu na década de 1990 com o propósito de permitir a codificação de comportamento dinâmico e interativo no *front-end* das páginas Web. No entanto, desde a criação o JavaScript tem sido cada vez mais utilizado em contextos diversos. Com esta ampliação do contexto de uso da linguagem, diferentes tipos e estilos de código-fonte surgiram e devido à flexibilidade de desenvolvimento da linguagem, padrões e boas práticas de desenvolvimento foram sendo esculpidos pela experiência própria de cada desenvolvedor, sem uma norma declarada oficialmente.

Essa monografia tem como objetivo realizar um estudo para caracterizar o uso da linguagem JavaScript do ponto de vista da estrutura do código-fonte dos projetos desenvolvidos nesta linguagem. Foram selecionadas diferentes métricas baseadas na estruturação da linguagem, que foram aplicadas na análise de uma ampla gama de projetos de código aberto. A monografia detalha a linguagem analisada, suas características e formas de uso. São também descritas as principais ferramentas de apoio ao estudo, o processo de coleta de dados e os resultados da análise.

Palavras-chave: JavaScript, análise estrutural de código, NPM, Esprima

ABSTRACT

The JavaScript programming language was introduced in the 1990s to allow the encoding of dynamic and interactive behavior on the front-end of Web pages. However, since its creation JavaScript has been increasingly used in a variety of contexts. With this widening of the context of language use, different types and styles of source code have emerged and due to the flexibility of language development, standards and good development practices have been sculpted by each programmer's own experience, without any officially declared standard.

This monograph aims to conduct a study to characterize the use of JavaScript programming language from the point of view of the source code structure of the projects developed in this language. We selected different metrics based on language structure, which were applied in the analysis of a wide range of open source projects. This monograph details the analyzed programming language, its characteristics and ways of usage. It also describes the main tools to support the study, the data collection process and the results of the analysis.

Keywords: JavaScript, structure code analysis, NPM, Esprima

Conteúdo

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivo 2

1.3 Estrutura do texto 2

2 A Linguagem JavaScript **4**

2.1 Introdução 4

2.2 A história por trás da linguagem JavaScript 4

2.3 A linguagem JavaScript 8

2.3.1 Callbacks 10

2.3.2 Closures 11

2.4 O padrão de normas ECMA 12

2.5 Ferramentas de apoio 13

2.5.1 O servidor Node.js 13

2.5.2 O repositório NPM 14

2.5.3 O pacote Esprima 16

2.6 Considerações finais 18

3 Análise Estrutural de Programas JavaScript **20**

3.1 Introdução 20

3.2 Medidas de interesse 20

3.3 Coleta de dados 23

3.3.1 Busca de pacotes no NPM 24

3.3.2 Captura do código-fonte dos pacotes 25

3.3.3 Eliminação de arquivos e pacotes desnecessários 26

3.3.4 Eliminação de *forks* 26

3.4 Cálculo das medidas de interesse 27

3.4.1 Eliminação de pacotes triviais 29

3.5 Considerações finais 29

4 Resultado das Análises **30**

4.1 Introdução 30

4.2 Representação dos resultados 30

4.3 Resultados por pacote 31

4.4 Resultados por arquivo 35

4.5 Resultados por função 38

4.6 Considerações Finais 44

5 Conclusão **45**

5.1 Contribuições 45

5.2 Limitações 45

5.3 Trabalhos futuros 46

Lista de Figuras

2.1 Representação da interface inicial do DOM level 0 6

2.2 Representação da estrutura inicial do DOM level 0 7

2.3 Representação da estrutura de uma árvore DOM 7

2.4 Gráfico de popularidade de linguagens de programação em repositórios hospedados no GitHub 8

2.5 Propriedades de configuração para o *parse* 17

2.6 Exemplo de árvore sintática criada 18

3.1 Modelo de dados para o armazenamento dos pacotes identificados no NPM 24

3.2 Árvore de diretórios de pacotes na máquina local 26

4.1 Boxplots com outliers das medidas por pacote 34

4.2 Boxplots sem outliers das medidas por pacote 35

4.3 *Boxplot* sem *outliers* das medidas por arquivo 37

4.4 *Boxplot* sem *outliers* por função 41

Lista de Tabelas

2.1 Tipos de nós encontrados no Esprima 18

4.1 Tabela de estatísticas descritivas por pacote 32

4.2 Matriz de correlação das medidas por pacote 34

4.3 Tabela de estatísticas descritivas por arquivo 36

4.4 Matriz de correlação das medidas por arquivo 38

4.5 Tabela de estatísticas descritivas por função 38

4.6 Matriz de correlação das medidas por função 41

4.7 Frequência dos nós sintáticos por função 43

Capítulo 1. Introdução

1.1 Motivação

Na década de 90, no início da *Web*, a linguagem JavaScript foi criada voltada para o desenvolvimento *front-end*, para solucionar problemas de interação do usuário final com as páginas *Web*, que na época eram estáticas e não permitiam mais do que navegar através de links e enviar informações através de formulários. Desde então, JavaScript passou a ser a principal linguagem para desenvolvimento *front-end* de sistemas de software disponibilizados na *Web*, implementada em quase todos os navegadores. Devido à competição de mercado na época, a linguagem desenvolveu suas funcionalidades rapidamente e teve um crescimento grande em um curto período de tempo.

Por ser uma linguagem não tipificada, o desenvolvimento em JavaScript é muito flexível. Essa flexibilidade fez com que os desenvolvedores começassem a implementá-la em outros cenários, como em servidores no *back-end* das aplicações *Web*, desprendendo a linguagem da ideia de somente permitir uma melhor interação visual do navegador com o usuário. Neste novo cenário, a ferramenta mais utilizada atualmente é o Node.js, que traz consigo um gerenciador de pacotes (ou módulos), o NPM, que dispõe de um repositório com mais de 400 mil pacotes de código aberto escritos em JavaScript [4][1].

Porém, a flexibilidade da linguagem tem um lado negativo: desenvolvedores não têm um padrão de referência para a programação JavaScript. Não há informação disponível sobre o que é um pacote grande nesta linguagem, como seus recursos são utilizados, quão complexos são os pacotes desenvolvidos ou quão difícil é manter estes pacotes? O desempenho da linguagem depende do julgamento de quem está programando e gargalos desta natureza são constantes em JavaScript. A flexibilidade excessiva também dificulta

[1] Embora a documentação do NPM cite a existência de mais de 400 mil pacotes, o mecanismo de busca do gerenciador permitiu localizar apenas em torno de 34 mil pacotes distintos.

e torna custosa a depuração dos pacotes em JavaScript. A própria comunidade de desenvolvedores tenta definir padrões - com base na experiência de cada desenvolvedor com a utilização da linguagem - e criar uma documentação que auxilie no desenvolvimento em JavaScript, mas ainda não existe um padrão oficial de desenvolvimento.

1.2 Objetivo

Percebendo a inexistência de padrões e boas práticas de desenvolvimento em JavaScript, foi realizado esse estudo para caracterizar a forma com que a linguagem é utilizada atualmente. A ideia foi colher uma ampla base de código-fonte em JavaScript, analisar seu conteúdo e apresentar estatísticas baseadas em métricas de interesse para os desenvolvedores. Para a base de código-fonte, escolhemos pacotes retirados do repositório do NPM[2], a fim de coletar códigos de ampla utilização, que estejam sendo constantemente testados e tenham a aceitação da comunidade de desenvolvedores web, em sua maioria, caracterizando bem o momento atual de desenvolvimento na linguagem.

1.3 Estrutura do texto

Este documento descreve as etapas em que o estudo foi realizado, começando pela coleta de códigos-fonte, a criação de programas para analisar estes códigos e extrair as métricas selecionadas, até a análise estatística dos relatórios resultantes da análise. Todas as etapas do processo serão detalhadas neste documento, que está dividido da seguinte forma:

- Capítulo 2 - A Linguagem JavaScript: apresenta a linguagem JavaScript e as ferramentas auxiliares utilizadas no estudo;
- Capítulo 3 - Análise Estrutural de Pacotes JavaScript: apresenta a definição e o detalhamento das etapas do estudo;
- Capítulo 4 - Resultado das Análises: apresenta o resultado das análise de código-fonte JavaScript a as estatísticas relacionadas às métricas selecionadas.

[2]https://www.npmjs.com/

- Capítulo 5 - Conclusão: Considerações finais sobre a linguagem e o estudo realizado sobre ela.

Capítulo 2. A Linguagem JavaScript

2.1 Introdução

JavaScript (JS) é uma linguagem de programação originalmente *client-side*, ou seja, voltada para a criação de programas em computadores que solicitam informações de servidores e desenvolvida para permitir o comportamento dinâmico de páginas *Web* – respostas do navegador às ações do usuário – e torná-las interativas. Criada por Brendan Eich em 1995, a serviço da Netscape, JavaScript foi originalmente desenvolvida sob o nome de Mocha, em um período de apenas 10 dias. Foi uma revolução para a época, pois antes disso os navegadores ainda eram estáticos e os *websites* existentes tinham uma interatividade limitada com o usuário: páginas que apresentavam seu conteúdo exatamente como foi criado para ser exibido no navegador. Navegar através de links e enviar informações através de formulários era basicamente tudo o que se podia fazer.

Este capítulo está dividido em cinco seções além dessa introdução. A seção 2.2 fala sobre a história do JavaScript e de sua criação. A seção 2.3 fala sobre os conceitos que a linguagem traz, sua sintaxe, algumas funcionalidades e utilizações principais. A seção 2.4 fala sobre as normas e padrões ECMA, se referindo a especificação da linguagem de *script* que o JavaScript implementa e o porquê desta necessidade de padronização. A seção 2.5 traz as principais ferramentas utilizadas neste projeto. A seção 2.6 consolida as principais questões abordadas neste capítulo e apresenta suas conclusões.

2.2 A história por trás da linguagem JavaScript

A Netscape Communications (ou somente Netscape) foi uma empresa norte americana de serviços de informática. Fundada em 1994 com o objetivo de explorar a *World Wide Web*[1] que estava surgindo, ganhou popularidade com a criação do navegador Netscape Navigator, que em pouco tempo se tornou o navegador mais popular e dominante da

[1] a.k.a. *the Web*; techterms.com/definition/www

década de 90 (posteriormente viria a perder sua hegemonia para o recém criado Internet Explorer, da Microsoft, na chamada "Guerra dos Navegadores"[2]). Em 2003 foi comprada pela AOL que, apesar do anúncio da descontinuação do navegador Netscape Navigator (2007), ainda utiliza amplamente a marca Netscape. Atualmente a Mozilla[3] é considerada a sucessora da Netscape, trazendo a influência do Netscape Navigator em seu navegador atual e principal produto, o Firefox[4] [8].

Com a grande expansão do uso de seu maior produto (o Navigator), em pouco tempo a Netscape chegou à conclusão de que, para sua verdadeira e sólida ascenção, a *Web* teria que se tornar mais dinâmica. No cenário da época, um navegador tinha que fazer uma requisição ao servidor para obter qualquer resposta, o que pesava bastante em seu desempenho e na experiência de usabilidade do usuário. Foi aí que a Netscape viu a possibilidade do desenvolvimento de uma nova linguagem de programação *Web* voltada para o lado do cliente, que resolvesse esses problemas de dinamismo do navegador – a falta de elementos da página que tenham movimento, mudem de cor ou apresentem qualquer outro comportamento dinâmico – e tivesse uma sintaxe menos complexa do que as linguagens padrões para a época (como Java e C++), sendo mais fácil de utilizar e aprender.

Em 1995, a Netscape contratou Brendan Eich para criar esta "linguagem dos sonhos"que daria a Netscape o reconhecimento que ela ainda procurava. Nos 10 dias que seguiram, entre 5 e 15 de maio daquele ano, Eich criou apenas o núcleo inicial do que viria a ser a linguagem JavaScript: sua demo foi um JS *console* que fazia um *post* para uma URL JavaScript: "<form method='post' action='javascript:...' >". Apesar de concisa, a demo apresentada possuía *design* e *layout* (Figura 2.1), representados numa estrutura *flat display-list-like* (Figura 2.2) sem hierarquia, denominada "DOM Level 0"[11][26].

Como definido pela W3C[5], o DOM (*Document Object Model*, ou Modelo de Objeto de Documento) é uma interface neutra – em termos de plataforma e linguagem – que permite que programas e *scripts* acessem e atualizem dinamicamente o conteúdo, a estrutura e

[2]docente.ifrn.edu.br/moisessouto/disciplinas/autoria-web/a-verdadeira-historia-da-internet-guerra-dos-navegadores

[3]www.mozilla.org/

[4]www.mozilla.org/pt-BR/firefox/?utm_medium=referral&utm_source=firefox-com

[5]www.w3.org/

o estilo dos documentos *Web* [20]. É um modelo multi-plataforma que representa como as marcações em HTML, XHTML e XML são organizadas e lidas pelo navegador que as utiliza. Uma vez indexadas, estas marcações se transformam em elementos de uma árvore (Figura 2.3) que pode ser manipulada através de programas ou *scripts* [10]. O DOM Level 0 criado pela Netscape era bem simples: oferecia acesso a alguns elementos HTML e sua indexação inicial consistia em "document.links,forms,anchors,images", oferecendo acesso a poucos elementos HTML, como *links* e (um pouco mais tarde) imagens, com uma capacidade especial para codificar ações de formulário.

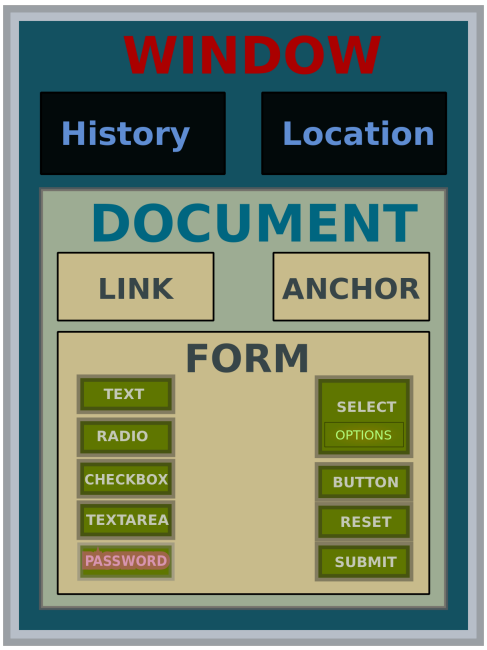

Figura 2.1: Representação da interface inicial do DOM level 0.[6]

[6]Disponível em: pt.wikipedia.org/wiki/Modelo_de_Objeto_de_Documentos

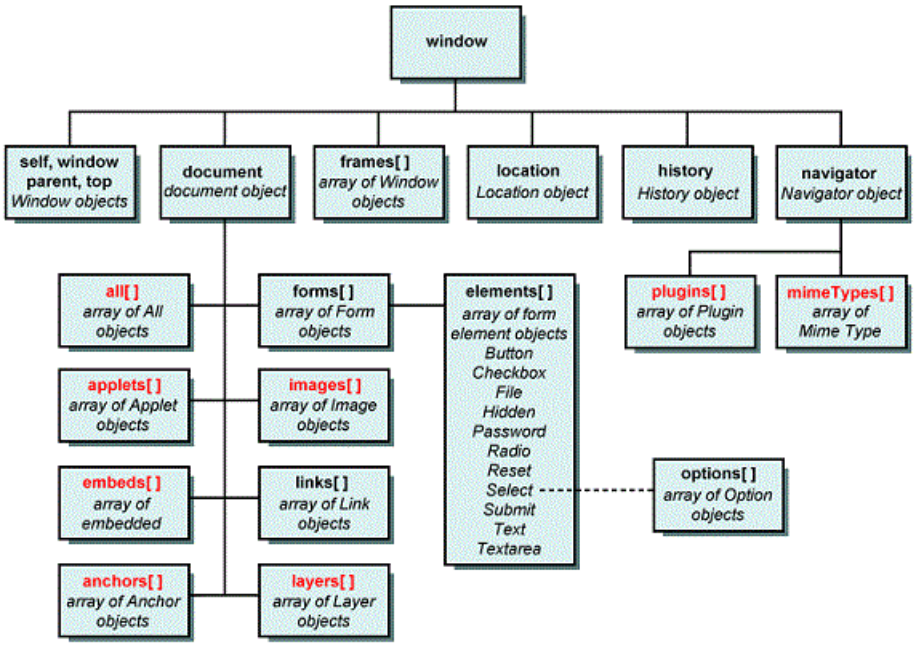

Figura 2.2: Representação da estrutura inicial do DOM level 0 [26].

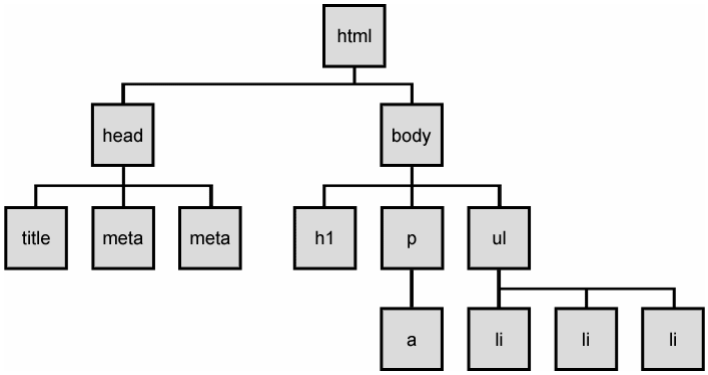

Figura 2.3: Representação da estrutura de uma árvore DOM [10].

Eich produziu a semente que acabou crescendo para o JavaScript moderno, decorrente de muito trabalho empregado ao longo desses 22 anos de existência da linguagem. O original Mocha posteriormente teve seu nome modificado para LiveScript, que foi o nome oficial da linguagem quando esta foi lançada pela primeira vez na versão beta do navegador Netscape 2.0, em setembro de 1995. Mais tarde, a linguagem teve seu nome alterado novamente em uma aliança com a Sun Microsystems – criadora do Java – em dezembro do mesmo ano, fazendo com que o nome JavaScript fosse adotado não por similaridade com a linguagem Java, mas sim por estratégia de marketing e divulgação da marca.

Atualmente, JavaScript é um dos alicerces da *Web* como a conhecemos. Segundo o *ranking* mais atual (maio/2017) do PYPL de linguagens de programação mais populares

mundialmente, a linguagem JavaScript se encontra no Top 5, sendo uma das únicas que teve crescimento na comparação com o ano anterior (2016) [24]. A classificação PYPL é baseada na frequência de consultas do Google para tutoriais sobre uma linguagem. Em 2015, o GitHub[7] – o maior site de hospedagem de projetos git na nuvem, onde reside a maior parte dos maiores projetos open-source do mundo, como Docker e o NPM – divulgou um gráfico (Figura 2.4) de popularidade de linguagens utilizadas em seus repositórios, no qual a linguagem JavaScript aparece em primeiro lugar desde meados de 2012.

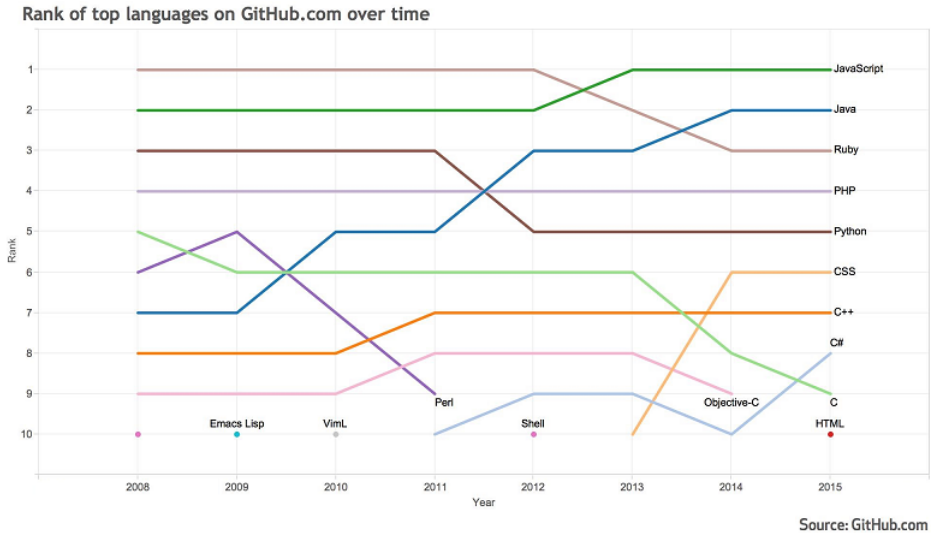

Figura 2.4: Gráfico de popularidade de linguagens de programação em repositórios hospedados no GitHub (Agosto/2015) [13].

2.3 A linguagem JavaScript

Desde o início, a ideia central da *Web* era de ser uma plataforma aberta para todos. Sendo assim, era de suma importância que a sua principal linguagem de programação também carregasse estes conceitos de código aberto e fácil aprendizado. A principal vantagem do JavaScript era a competitividade que ele fomentava no mundo da *Web*: todos os navegadores batalhavam por uma implementação mais rápida da linguagem, o que acarretou em um crescimento imediato e em um aperfeiçoamento constante. Nessa briga pelo domínio da *Web*, os desenvolvedores e os usuários saíam ganhando. Diferente da internet rígida e restrita que existia antes – cheia de ferramentas proprietárias, *plug-ins*

[7] github.com/

de terceiros e de desempenho ruim –, o JavaScript trouxe a liberdade e a flexibilidade que os navegadores (e os usuários!) estavam precisando tanto.

Tecnicamente, JavaScript é uma linguagem de *script*[8], interpretada, baseada em objetos e protótipos, multi-paradigma e dinâmica, suportando além do estilo orientado a objeto, os estilos imperativo e funcional. Uma linguagem interpretada é aquela em que a análise do código fonte acontece no mesmo momento da execução do código, sem uma compilação[9] prévia. O conceito de linguagem baseada em objetos e protótipos provém do conceito de orientação a objeto, porém difere em um sentido: cada novo objeto é criado clonando-se objetos existentes, que são chamados de protótipos. Nas linguagens orientadas a objetos, tradicionalmente o comportamento das instâncias é definido em classes que guardam uma coleção de métodos e atributos, sendo a adição de novos comportamentos feita através da extensão da classe já existente, exigindo que para cada objeto exista uma classe com a sua definição. A programação orientada a protótipos privilegia a relação entre objetos para uma posterior divisão em classes. Através de um objeto já existente, criam-se instâncias e só depois esses objetos criados podem ser classificados em uma estrutura similar ao modelo de classes, não obrigatóriamente.

Na Computação, um paradigma caracteriza o estilo, conceitos e métodos de uma linguagem, para descrever situações e processos e para resolver problemas, cada um deles servindo melhor para programação em áreas de aplicação específicas [15]. Denominar uma linguagem como multi-paradigma, significa que ela suporta paradigmas de múltiplos estilos de linguagens (por exemplo, combinando orientação a objetos com aspectos de linguagem funcional). O paradigma da Orientação a Objeto é o mais difundido atualmente. Nele implementam-se classes gerais que definem objetos (instâncias destas classes), suas propriedades (os atributos) e seus comportamentos (os métodos), assim como os relacionamentos entre eles. O paradigma Imperativo é aquele que expressa o código através de comandos ao computador. É focado na mudança de estados das variáveis. O paradigma Funcional tem seu foco na avaliação de funções, como na matemática, expressando o código através delas e evitando a mutação de estado [27].

[8] searchwindevelopment.techtarget.com/definition/scripting-language

[9] processo de transformação do código fonte em código alvo, ou seja, "tradução"do código para linguagem de máquina

Além de carregar uma sintaxe clara, de fácil e rápida associação, uma das principais qualidades do JavaScript é a facilidade de testar a linguagem de forma rápida, o que auxilia o desenvolvedor, permitindo que este veja o resultado do seu código em tempo quase real [17]. JavaScript também possui tipagem dinâmica, ou seja, não é necessário definir o tipo das variáveis ao declará-las (para isso basta usar alguma das palavras reservadas disponíveis para declaração de variáveis em JavaScript – como *var*, *let* ou *const*). Em JavaScript, as funções são objetos de primeira classe, ou seja, as funções são do tipo *Object* e podem ser usadas como entidades de primeira classe como qualquer outro objeto (*String*, *Array*, *Number*, etc.). Entidades de primeira classe são indivíduos que suportam todas as operações geralmente disponíveis para outras entidades: ser passado como um argumento, retornado de uma função, modificado e atribuído a uma variável, o que dá a possibilidade, por exemplo, de atribuição de funções inteiras a variáveis ou ter funções como valor de retorno da execução de outras funções. Este conceito possibilita também funções receberem outras funções como parâmetro (o que passa pelo conceito de *callbacks*, que é o foco da subseção 2.3.1), e até mesmo funções com seu escopo definido dentro de um contexto léxico – um bloco, o corpo de uma função – que tem acesso a todos os outros contextos externos que as englobam (passando pelo conceito de *closures*, abordado em detalhes na subseção 2.3.2).

2.3.1 Callbacks

Funções de *callback* são provavelmente a técnica de programação funcional mais utilizada em JavaScript [5]. Uma função de *callback* é uma função que é passada para outra função como um parâmetro, sendo executada (na maioria das vezes) dentro dessa outra função. Não há uma regra que obrigue a execução da função *callback* ser feita dentro desta primeira função que a recebeu como parâmetro, o que permite que ela seja passada a frente como parâmetro para diversas outras funções ou até mesmo nem seja executada.

Geralmente programas orientados a eventos – como os navegadores *Web* – consistem de um único *loop*, chamado de *event loop* (ou *loop* de eventos), que espera por eventos e repassa os mesmos para o manipulador do evento, executando tudo serialmente. Ao executar uma operação de longa duração dentro de um *loop* de eventos, o processo bloqueia, ou seja, para de processar outros eventos enquanto aguarda a conclusão da operação. Os *callbacks* têm como função permitir a execução de eventos assíncronos, evitando o

bloqueio em operações de longa duração como na situação exemplificada acima.

Quando passamos uma função de *callback* como argumento para outra função, o *callback* é executado em algum ponto dentro do corpo da função que o contém, como se este tivesse sido definido dentro desta função que o contém. Isso significa que uma função de *callback* é também uma *closure*, que é o conceito abordado na subseção seguinte.

2.3.2 Closures

Uma *closure* é a combinação de uma função e o ambiente léxico dentro do qual essa função foi declarada [18]. Pode ser exemplificada como uma função interna a outras funções e que tem acesso aos escopos destas funções externas. Uma *closure* possui três níveis de escopo: possui acesso ao seu próprio escopo (variáveis definidas no corpo da função *closure*), possui acesso ao escopo da função externa e possui acesso ao escopo global do programa, não só as variáveis, mas também aos parâmetros de cada um [3].

Assim como os *callbacks*, *closures* também são parte do conceito de assincronismo do JavaScript. Elas têm acesso às variáveis da função externa mesmo após esta ter sido encerrada (ou retornada, como se fala na programação), o que permite que a *closure* seja chamada posteriormente na execução do código. Isso é possível pois quando uma função é executada em JavaScript, ela utiliza a mesma cadeia de escopo que estava disponível quando esta foi criada, o que significa que mesmo após a função externa ter encerrado, a *closure* interna ainda tem acesso a este escopo total.

Closures armazenam referências às variáveis de seus escopos externos, e não o valor, tendo assim acesso aos valores atualizados destas variáveis, permitindo que o programador associe dados a uma função que opera nestes dados. Situações onde esta característica é útil são constantes na *Web* orientada a evento: é definido algum comportamento e então associa-se este comportamento a um evento que é disparado por alguma ação, *geralmente anexado como um callback*.

2.4 O padrão de normas ECMA

Após a criação do JavaScript, a Microsoft criou, em agosto de 1996, uma linguagem idêntica para ser usada no Internet Explorer 3 – o JScript[10]. A Microsoft foi processada pela Netscape alegando que a sua concorrente estaria utilizando táticas monopolistas para ganhar o mercado de navegadores. Foi então que enxergou-se a necessidade de desenvolver um padrão para linguagens de *script* para os navegadores.

O ECMAScript (ES) é a especificação da linguagem de *script* que o JavaScript implementa, ou seja, é a descrição formal e estruturada de uma linguagem de script, sendo padronizada pela ECMA International[11] – associação criada em 1961 dedicada à padronização de sistemas de informação e comunicação [22]. A ECMA coordena um grupo que tem participação colaborativa de empresas que implementam o *run-time* do JS, como Mozilla, Google, Microsoft e Apple, além da participação de desenvolvedores web que representam as comunidades de desenvolvimento [17].

Em 2015 foi lançado o ECMAScript 2015 (também chamado ECMAScript 6, ou ES6) que consiste na primeira fase do projeto *Harmony*: projeto para fundir ambas as vertentes de especificações ECMAScript que existiam (ES3.1 e ES4), a fim de padronizar o desenvolvimento. O ES6 foi a maior mudança para a linguagem JavaScript desde a sua criação e teve como principal objetivo tornar a linguagem mais flexível, enxuta e fácil de se aprender e trabalhar, tornando-a mais próxima de outras linguagens orientadas a objeto. Dentre as principais mudanças, temos: criação de novos tipos de dados (*Map*, *WeakMap*, *Set*, *WeakSet*), novas maneiras de iterar objetos e coleções, declaração de variáveis com *let* e *const*, modularização e estrutura de classes, operadores *rest* e *spread*[12]. A versão ECMAScript 2016 (ES7) foi a última fase da versão *Harmony* e trouxe mais recursos, como operadores exponenciais e outras *features* que ainda não estão estabilizadas na versão. Alguns navegadores ainda não suportam totalmente as versões ES6 e ES7, porém é possível transpilar (transformar o código fonte escrito em uma linguagem, para outra linguagem que tenha um nível similar de abstração) códigos escritos em ambas as versões

[10] msdn.microsoft.com/pt-br/library/72bd815a%28v=vs.100%29.aspx

[11] ecma-international.org/

[12] codingwithspike.wordpress.com/2016/06/11/javascript-rest-spread-operators/

para ES5 através de bibliotecas como o Babel[13].

2.5 Ferramentas de apoio

2.5.1 O servidor Node.js

O exemplo mais significativo da aplicação da linguagem JavaScript no lado do servidor é o Node.js[14] (também referido como Node somente). É uma plataforma construída sobre o motor JavaScript do Google Chrome[15] [19]. A plataforma roda em uma Máquina Virtual JavaScript V8[16], que é um interpretador JavaScript escrito em C++ que pode ser incorporado a qualquer aplicação, não estando restrito a um navegador. Apesar de não ser ainda hoje a plataforma mais usada em seu domínio (PHP ainda domina o lado do servidor em aplicações *Web*, seguida das linguagens ASP.NET e Java [29]), o crescimento da utilização da plataforma Node.js é significativo. Em um período de 5 anos (2010-2015), mais de 190.000 módulos Node foram adicionados pelos desenvolvedores. Isso supera toda a contribuição ao repositório CPAN[17] do Perl dos últimos 20 anos e ultrapassa também o Java Maven Central[18], apesar do Node.js ter uma base de desenvolvedores menor [25].

Ideal para aplicações em tempo real com troca intensa de dados, a plataforma Node.js facilita a construção de aplicações de rede escaláveis. Diferente de outras plataformas em JavaScript que rodam no servidor, o Node faz uso de um modelo de I/O direcionado a eventos não bloqueantes, ou seja, cada conexão dispara um evento executado dentro da *engine* do Node, não bloqueando uma *thread* a cada conexão, significando que os pedidos serão feitos e entregues quando estiverem prontos. Ele modifica a maneira como as conexões são tratadas no servidor, o que permite que estas sejam feitas sem alocação desnecessária de memória, aumentando o número de conexões concorrentes que os servidores podem manipular e aumentando a velocidade de tráfego na *Web*.

[13] babeljs.io/
[14] nodejs.org/en/
[15] www.google.com/chrome/
[16] motor de execução JavaScript que a Google usa com seu navegador Chrome
[17] Repositório Central do Perl; www.cpan.org/
[18] Repositório Central do Java; search.maven.org/

Quando foi criada a linguagem JavaScript, a ideia de seu uso no servidor podia parecer absurda, mas algumas características da linguagem mostram esta capacidade de utilização e suas vantagens. O lado cliente e o lado servidor agem da mesma maneira e respondem aos mesmos tipos de estímulos. Por mais que de diferentes tipos, ambos também respondem a eventos: o clique de um botão é um exemplo de evento do tipo cliente; uma conexão criada é um exemplo de evento do tipo servidor. Acontece que a linguagem JavaScript é uma ótima linguagem para programação orientada a eventos, pois permite funções anônimas e *closures* [19]. Funções anônimas em JavaScript são funções que são criadas utilizando o operador de função e em tempo de execução, ou seja, são criadas dinamicamente a medida em que aparecem no código, diferente das funções declaradas da forma tradicional que são movidas para o topo do código e criadas antes do resto do código ser executado.

Além das vantagens de produtividade que uma linguagem de *script* traz ao desenvolvimento de aplicações, sendo estas inerentemente mais produtivas do que as linguagens de programação mais pesadas como C e C ++ [21], como dito na seção 2.3 deste capítulo, a sintaxe da linguagem JavaScript é bastante clara, de fácil e rápida associação e bem familiar para quase todos que já tiveram algum contato com programação. Todas essas características do JavaScript possibilitam ao Node reduzir o tempo de desenvolvimento das aplicações, ainda obtendo a mesma funcionalidade que outras plataformas.

2.5.2 O repositório NPM

Segundo Farrer et al. (2008) um módulo é "um grupo de comandos, constituindo um trecho de algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo". O Node é um servidor que carrega o conceito de módulos, que podem ser adicionados no núcleo deste (as funções dos módulos do Node estão documentadas na página da API do Node[19]). Em Node, podemos definir módulos como sendo pacotes – bibliotecas ou sub-programas – que executam tarefas bem definidas, ajudam a organizar seu código em partes separadas com responsabilidades limitadas e podem ser injetados em outros trechos de código como dependências e/ou executados independentemente.

[19] nodejs.org/api/

NPM é o nome reduzido de Node Package Manager (Gerenciador de Pacotes do Node). É composto de:

1. Um repositório online para publicação e armazenamento de projetos de código aberto para o Node.js;
2. Um utilitário de linha de comando que interage com este repositório online, possibilitando a instalação de pacotes, gerenciamento de versões, dependências e a publicação de pacotes de código aberto.

O NPM conta com mais de 400 mil pacotes de código aberto em seu repositório, podendo estes ser encontrados no portal de busca do NPM[20]. Contudo, nosso processo de coleta de códigos-fonte (apresentado no Capítulo 3) somente conseguiu identificar cerca de 34 mil pacotes distintos no repositório, sendo esta limitação explicada posteriormente neste documento.

O NPM é o gerenciador de pacotes JavaScript mais popular. Seu número de pacotes registrados chega a ser o dobro da quantidade de pacotes registrados em outros gerenciadores. O número de pacotes instalados por usuários chega a 18 bilhões. Uma vantagem a ser destacada do NPM é a velocidade com que este baixa os pacotes, sendo 75% mais rápido que fazer o download direto do repositório do GitHub [4].

A utilização do NPM é fácil e intuitiva. Sabendo o nome do pacote que se quer instalar, com um único comando é possível instalar esta dependência. Este comando – que é o principal comando do NPM – é o *npm install*, seguido do nome do(s) pacote(s) desejado(s). Este comando é utilizado para instalar pacotes como dependências do projeto. Um projeto pode ter diversas dependências necessárias ao seu funcionamento. Na hora de rodar o projeto em um ambiente que não tenha estas dependências instaladas, não é necessário instalar todas as dependências uma a uma. O NPM conta com um arquivo de configuração que guarda a lista de todas as dependências do projeto em questão, além de outras informações sobre o mesmo (autor, versão, link do repositório remoto, etc.). O arquivo tem o nome padrão de *package.json* e é utilizado pelo NPM para, por exemplo, identificar quais são as dependências necessárias àquele projeto. Assim, tendo em seu projeto este

[20] www.npmjs.com/search

arquivo *package.json* basta executar o comando *npm install* no diretório em que o arquivo se encontra – sem nenhuma opção ou nomes de pacotes especificados logo após –, e toda a sua lista de dependências será instalada pelo próprio NPM.

Após todas as dependências instaladas, é o momento de utilizá-las em seu projeto. O método embutido *require* do Node é a maneira que o Node utiliza para importar módulos (pacotes) auxiliares existentes em arquivos separados. O *require* lê o arquivo JavaScript, interpreta o script e em seguida retorna o conteúdo do objeto *exports* existente no arquivo [7]. O *exports* consiste em um objeto que encapsula o conteúdo executável de um módulo – como variáveis, funções, etc. –, tornando-o visível a qualquer escopo que o incorpore, de modo que este possa ser acessado e utilizado no arquivo que o importar.

O método *require* pode ser utilizado tanto na importação de módulos auxiliares vindos do diretório '*./node_modules/*' (diretório de destino dos pacotes de dependência instalados pelo NPM), como na importação de módulos personalizados, locais e criados exclusivamente para o projeto em questão. Um dos módulos auxiliares de maior importância utilizado neste projeto foi o mysql[21]. Este módulo consiste em um *driver* Node para MySQL, escrito em JavaScript e que não requer compilação. Ele permite a troca de dados entre o pacote desenvolvido em Node e o banco de dados MySQL, através de conexões e fazendo uso de *SQL queries* que podem ser dinamicamente construídas, dando mais liberdade ao programa que o utiliza.

2.5.3 O pacote Esprima

O Esprima é um pacote escrito em JavaScript que possibilita análises léxicas e sintáticas de programas JavaScript. Ele funciona em diversos ambientes, incluindo o Node.js. Para utilizá-lo através do Node, é necessário injetá-lo através do comando *require* e usar a sua função *parse*. Esta função recebe como parâmetro o conteúdo de um arquivo JavaScript e a partir dele constrói um objeto em formato de árvore sintática que representa a estrutura sintática deste conteúdo. Caso o conteúdo lido não seja válido, uma exceção é dada e o erro indicado.

[21] github.com/mysqljs/mysql

A escolha do Esprima como ferramenta se deu por alguns motivos. O primeiro é a forma como ele retorna o conteúdo de um arquivo JavaScript, como uma árvore sintática (Figura 2.6). Dessa forma se torna fácil percorrer o conteúdo do arquivo e saber o que cada parte significa dentro do código. Um segundo motivo é ele já ser escrito em JavaScript o que facilita na comunicação com o Node.js.

A função de *parse* do Esprima permite que, além do conteúdo, seja passado como parâmetro também um objeto com as configurações, como mostra a Figura 2.5. As propriedades importantes ao projeto foram *loc* e *token*. A primeira indica que para cada nó da árvore gerada deve ser incluída uma propriedade que indica onde este nó se encontra no código do programa analisado. Essa propriedade é um objeto que contém os números das linhas inicial e final, bem como os números das colunas inicial e final do nó.

A segunda propriedade indica que deve ser incluído no objeto de retorno da função *parse* uma nova propriedade, um vetor em que cada elemento é um *token* do conteúdo do programa classificado com um tipo, sendo eles: *Keyword, Identifier, Punctuator, Numeric*. Cada *token* também pode incluir suas linhas inicial e final, bem como colunas inicial e final, caso a propriedade de *loc* esteja ativada. Essa propriedade será importante no cálculo de operadores e operandos de cada função, pois será a partir do tipo do *token* que este será classificado entre operador ou operando.

Name	Type	Default	Description
sourceType	String	"script"	Define the program type: a script or a module
jsx	Boolean	false	Support JSX syntax
range	Boolean	false	Annotate each node with its index-based location
loc	Boolean	false	Annotate each node with its column and row-based location
tolerant	Boolean	false	Tolerate a few cases of syntax errors
token	Boolean	false	Collect every token
comments	Boolean	false	Collect every line and block comment

Figura 2.5: Propriedades de configuração para o *parse* [9].

Uma árvore é um grafo direcionado. Cada elemento é denominado de nó. Existe um nó, chamado de raiz, que tem zero ou mais sub-árvores, denominadas filhos. Cada filho pode ter zero ou mais filhos e assim por diante.

Cada nó da árvore sintática do Esprima é um objeto da linguagem JavaScript. Esse objeto tem a propriedade *type*, que indica o seu tipo, e outras propriedades que podem ser configuradas como indicado na Figura 2.5. Os tipos de nó encontrados nos Esprima

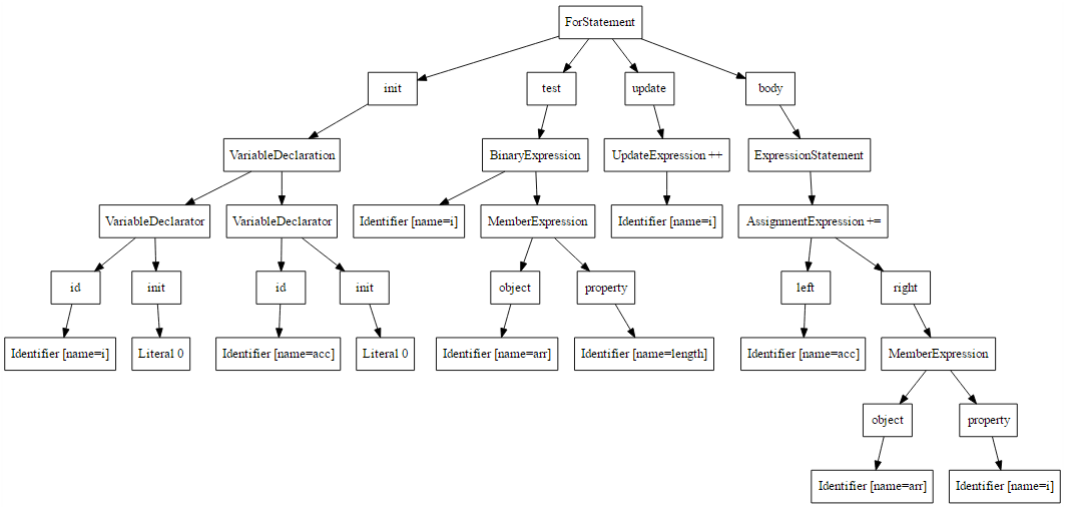

Figura 2.6: Exemplo de árvore sintática criada.

estão apresentados na Tabela 2.1.

Classificação	Tipos
BindingPattern	ArrayPattern; AssignmentPattern; BindingPattern; RestElement; ObjectPattern
Expression	ThisExpression; Identifier; Literal; ArrayExpression; SpreadElement; ObjectExpression; Property; FunctionExpression; ArrowFunctionExpression; ClassExpression; ClassBody; MethodDefinition; TaggedTemplateExpression; TemplateElement; TemplateLiteral; MemberExpression; Super; MetaProperty; NewExpression; CallExpression; UpdateExpression; UnaryExpression; BinaryExpression; LogicalExpression; ConditionalExpression; YieldExpression; AssignmentExpression; SequenceExpression
Statement	BlockStatement; BreakStatement; ContinueStatement; DebuggerStatement; DoWhileStatement; EmptyStatement; ExpressionStatement; ForStatement; ForInStatement; ForOfStatement; FunctionDeclaration; IfStatement; LabeledStatement; ReturnStatement; SwitchStatement; SwitchCase; ThrowStatement; TryStatement; CatchClause; VariableDeclaration; VariableDeclarator; WhileStatement; WithStatement
Scripts and Modules	Import; ImportDeclaration; ImportSpecifier; ImportDefaultSpecifier; ImportNamespaceSpecifier; ExportAllDeclaration; ExportDefaultDeclaration; ExportNamedDeclaration

Tabela 2.1: Tipos de nós encontrados no Esprima

2.6 Considerações finais

Como visto neste capítulo, a linguagem JavaScript é uma linguagem recente, que foi sendo desenvolvida e aprimorada com o tempo pela própria comunidade de desenvolvedores que a utilizava. Padrões de linguagem foram definidos, mas ainda assim existem muitas divergências de implementações em JavaScript, dada a flexibilidade da linguagem, o que levantou a necessidade de um estudo de caracterização como este.

O Capítulo 3 apresentará o processo de análise que utilizamos neste projeto, expondo

as métricas abordadas pelo estudo e o porquê da escolha destas. Ele fala sobre todas as etapas percorridas desde a coleta das informações dos pacotes para uma base de dados, passando pelo tratamento desta base até sua consolidação na base final que foi utilizada nas análises, assim como a descrição da coleta dos códigos fonte e seu armazenamento.

Capítulo 3. Análise Estrutural de Programas JavaScript

3.1 Introdução

Com a intensificação do seu uso nos diversos cenários relatados no capítulo 2.1, a linguagem JavaScript deixou de ser somente um acessório para a construção de *websites* e vem se tornando cada vez mais importante no desenvolvimento de softwares mais complexos. No entanto, as características do bom uso da linguagem não são consensuais na comunidade de desenvolvedores que a utiliza. Dessa forma, torna-se importante realizar um estudo estrutural de códigos JavaScript utilizados atualmente pela comunidade desenvolvedora, que nos permita de caracterizar como a linguagem vem sendo utilizada e que possa ajudar os desenvolvedores na criação de seus programas de forma que a escrita dos códigos-fonte se torne mais uniforme.

Este capítulo está dividido em quatro seções além dessa introdução. Na seção 3.2 são apresentadas as medidas usadas para caracterizar a estrutura de programas JavaScript. Na seção 3.3 é apresentado o processo de coleta dos pacotes. A seção 3.4 explica como foram calculadas as medidas de interesse a partir do código-fonte dos pacotes. Finalmente, a seção 3.5 apresenta as considerações finais deste capítulo.

3.2 Medidas de interesse

Para caracterizar a estrutura de programas JavaScript foram selecionadas algumas medidas que poderiam ser coletadas a partir do código-fonte. Estas medidas foram divididas em três níveis: informações por pacote, por arquivo e por função.

As medidas de interesse para os pacotes foram o seu número de linhas de código, o número de funções declaradas, o número de variáveis declaradas, o número de arquivos e o seu tamanho em bytes. As mesmas medidas foram coletadas por arquivo, com excessão do número de arquivos. As medidas de interesse para as funções foram o seu número de

linhas de código, o número de variáveis declaradas, o número de chamadas de funções na função sob análise, o número de funções declaradas no escopo da função, sua complexidade ciclomática, seu volume de Halstead e o seu *maintainability index*. Para cada função também foi coletado o número de vezes que cada tipo de nó aparece em sua árvore sintática (seção 2.5.3).

O número de linhas de código de um arquivo é relacionado com o esforço necessário para o seu desenvolvimento e a sua compreensão. É desejável que os arquivos de código-fonte sejam pequenos em *loc*, porém um tamanho muito reduzido por arquivo pode levar a um número exagerado de arquivos em um pacote, o que também dificulta a compreensão da implementação como um todo. Conhecida a média de linhas de código dos arquivos JavaScript, os desenvolvedores de um novo pacote podem avaliar se seus arquivos estão próximos à média ou muito acima dela [14], levando-os a considerar uma decomposição dos maiores arquivos. O mesmo racional também se aplica ao número de linhas de código de uma função. Nesse estudo foram consideradas todas as linhas de código-fonte que estão expressas no escopo de um arquivo ou função, incluindo comentários, linhas em branco e declarações.

O número de funções em um arquivo de código-fonte JavaScript indica quantas funções são declaradas neste arquivo. De forma similar ao número de linhas de código, a presença de um grande número de funções torna um arquivo mais complexo de compreender e, consequentemente, de manter. Sendo assim, a comparação do número de funções declaradas em um arquivo que esteja em desenvolvimento com a média de funções em uma grande amostra de programas JavaScript pode ajudar o desenvolvedor a decidir sobre a refatoração do seu código ou a divisão do mesmo entre diferentes arquivos.

Como explicado na seção 2.3.2, uma função em JavaScript pode conter a declaração de outras funções como parte do seu escopo. Assim, para cada função foi coletado quantas funções são declaradas no seu escopo. O número de funções declaradas dentro de cada função pode ser visto como um indicativo da dificuldade de manutenção do código da função, em especial em uma linguagem em que o uso de *callbacks* e *closures* é intenso.

O número de variáveis declaradas por arquivo, ou seja, o número de variáveis globais (que não pertençam ao escopo de uma função) indica o volume e a diversidade de informações que representam o estado do arquivo. O desenvolvedor responsável pela manutenção

deste arquivo precisa estar ciente dos valores esperados nestas variáveis quando altera o código-fonte do arquivo. Sendo assim, quanto maior o número de variáveis no arquivo, mais complexa será a sua manutenção. Mais uma vez, o mesmo racional se aplica ao número de variáveis declaradas dentro de cada função.

O número de chamadas de função indica quantas vezes funções são invocadas dentro do código-fonte de uma função. Uma função chamar outra função pode indicar uma boa segregação do código, com as partes integrantes de uma funcionalidades divididas entre as funções. Isto tende a simplificar a reutilização e a manutenção do código. O número de chamadas de funções também demonstra a intensidade da colaboração de uma função com as outras funções do pacote, além de estar relacionado com o número de *callbacks*. Valores muito altos denotam que há um excesso de colaboração entre as operações. Um número alto de *callbacks* declarados como *closures*, por exemplo, indica que o código é complexo, uma vez que estes *callbacks* serão executados assincronamente e terão acesso às variáveis declaradas no escopo da função onde são chamados, podendo causar um *callback hell* [28].

O número de chamada de funções também pode ser relacionado com o acoplamento entre as funções, como visto na métrica *Coupling Factor* (CF). Essa métrica é calculada como o fator das dependências de chamadas de função observadas no código-fonte pelo número total de dependências possíveis entre as funções. CF é uma das medidas incluídas nas métricas MOOD (*Metrics for Object Oriented Design*). Altos valores de CF devem ser evitados, pois indicam alta complexidade, redução de encapsulamento e potencial de reuso, além de limitar o entendimento e a manutenção do código [1]. Nesse estudo toda e qualquer chamada de função foi considerada, não somente chamadas a funções distintas, ou seja, se uma função x fizer chamada a função y duas vezes, contabilizaremos duas chamadas de função.

A complexidade ciclomática é uma medida que representa o número de caminhos independentes em uma função e indica o número máximo de testes que devem ser realizados para garantir que todos os comandos da função foram executadas pelo menos uma vez [23]. Ela é calculada como o número de decisões simples acrescido de um. Para calcular a complexidade ciclomática foi considerado o número de ocorrências de comandos do tipo *if*, *for*, *while*, *do-while*, expressões condicionais e casos de *switch*. Um maior número de caminhos independentes indica que a função tem mais chances de apresentar erros em seu

código-fonte.

O volume de uma função faz parte das medidas de complexidade propostas por Maurice Halstead como um meio de determinar a complexidade de um trecho de código-fonte com base no número de operadores e operandos usados no código [6]. O volume (V) é calculado pela equação 3.1, em que N é o somatório do número de operadores e operandos usados na função e n é o somatório do número de operadores e operandos distintos.

$$V = N \times \log_2 n \tag{3.1}$$

O *maintainability index*[1] (MI, ou índice de manutenibilidade) é um índice que varia de 0 a 100 e representa a facilidade com que um código pode ser mantido. Este índice é calculado por função e um valor alto significa maior facilidade na manutenção da função [16]. O índice é calculado pela equação 3.2, em que V é o volume, CC a complexidade ciclomática e *loc* é o número de linhas de código da função sob análise.

$$MI = MAX(0, 100 \times \frac{171 - 5,2 \times \log_{10} V - 0,23 \times CC - 16,2 \times \log_{10} loc}{171}) \tag{3.2}$$

3.3 Coleta de dados

As medidas citadas na seção anterior serão utilizadas para caracterizar as informações estruturais sobre programas JavaScript. No entanto, para que esta caracterização represente a realidade dos programas desenvolvidos nesta linguagem, precisamos de um grande número de programas para utilizar como base do estudo. Foi feita então uma coleta de pacotes do NPM, seguindo uma série de passos descritos nas próximas subseções.

[1] https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/

3.3.1 Busca de pacotes no NPM

O objetivo inicial do projeto era coletar todos os pacotes disponíveis no NPM. Foram coletados 34.726 pacotes, no entanto esse número foi reduzido ao longo do refinamento dos dados, resultando ao final em um conjunto de 18.659 pacotes que foram efetivamente analisados.

O processo se iniciou com a coleta de todas as informações disponíveis no site do NPM sobre os pacotes existentes no repositório do NPM. Um programa JavaScript foi criado para fazer a busca no site por todas as letras do alfabeto, de A-Z, e para cada pacote retornado na busca foram coletadas as informações de: nome, autor, estrelas, versão, descrição e *tags*. As estrelas refletem a avaliação daquele pacote, de acordo com os usuários, enquanto que as tags são palavras-chave relacionadas àquele pacote e seu conteúdo - como a tecnologia e padrões utilizados, dentre outros fatores. A cada pacote encontrado na consulta foi feita uma nova consulta à página do pacote para que também fosse coletado o link do seu repositório remoto no GitHub. Tais informações foram armazenadas em um banco de dados MySQL segundo o modelo da Figura 3.1. Inicialmente, o banco continha 129.076 entradas, cada uma representando um pacote, não distintos.

Como dito na seção 2.5.2, o repositório do NPM conta com mais de 400 mil pacotes, porém na época em que foi feita a coleta destes para este projeto, a busca do site do NPM estava com problemas e não retornava todos os pacotes existentes no repositório. Este problema foi corrigido recentemente pela organização responsável pelo site [2], mas a coleta dos pacotes para o projeto ficou defasada, sendo coletados somente 34.726 pacotes distintos ao todo.

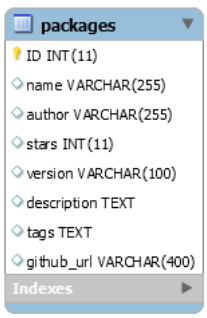

Figura 3.1: Modelo de dados para o armazenamento dos pacotes identificados no NPM.

3.3.2 Captura do código-fonte dos pacotes

A intenção inicial foi usar os nomes dos pacotes coletados para instalar via NPM cada um deles na máquina em que a análise seria realizada. Feita a instalação, um programa JavaScript foi criado para procurar os códigos-fonte de cada pacote e movê-los para um diretório específico para que fossem utilizados na análise posterior. Por fim, o programa removeria o pacote da máquina a fim de não manter nenhum material desnecessário. Porém, foi verificado que instalar e remover repetidamente diversos pacotes poderia deixar rastros (lixo de memória) que acabariam pesando no armazenamento da máquina.

Foi então desenvolvida uma segunda solução: a criação de um arquivo .bat (arquivo de comandos do Windows) onde seriam inseridos comandos usados para clonar os repositórios do GitHub para a máquina local, utilizando a url do GitHub de cada pacote. Após uma breve análise de desempenho, foi acordado que esta seria a melhor solução para o recolhimento dos dados necessários. Porém, no decorrer do estudo percebemos que o NPM armazena em seu repositório somente os códigos-fonte compiláveis, diferente do repositório no GitHub que mantém o código completo, causando assim um maior trabalho de tratamento dos dados coletados.

Devido à limitação do número de arquivos e/ou subdiretórios para um diretório no Windows (que não deve passar de 1000 arquivos e/ou subdiretórios), foi necessário pensar em uma estrutura de diretórios que permitisse o armazenamento dos pacotes de modo a não infringir esta limitação e que também facilitasse a procura de um pacote dado o seu id (chave primária de identificação no banco de dados). A estrutura final desses diretórios na máquina se deu da seguinte forma: dado o id de um pacote no banco de dados, o repositório clonado seria salvo em um diretório com nome igual ao resultado da equação *id % 1000*. Assim, nenhum diretório excederia o limite de 1000 subdiretórios e a facilidade de acesso a partir do identificador do banco de dados estaria garantida.

Antes da clonagem dos repositórios foi feito um tratamento para remover informações duplicadas do banco de dados. Como dito da seção 3.3.1 inicialmente, o banco continha 129.076 pacotes. Removidos os pacotes com nomes duplicados, restaram 34.726 pacotes. Por fim, foram removidos os pacotes com url do GitHub duplicadas, restando 31.912 pacotes. Destes, 5.347 não continham a url do GitHub e foram removidos. Assim, restaram 26.565 pacotes para prosseguir na análise. Destes, foram clonados em torno de 25.000

pacotes, já que alguns projetos antigos não existiam mais e outros exigiam credenciais de acesso para a clonagem. A estrutura de diretórios resultante desta etapa pode ser vista na Figura 3.2.

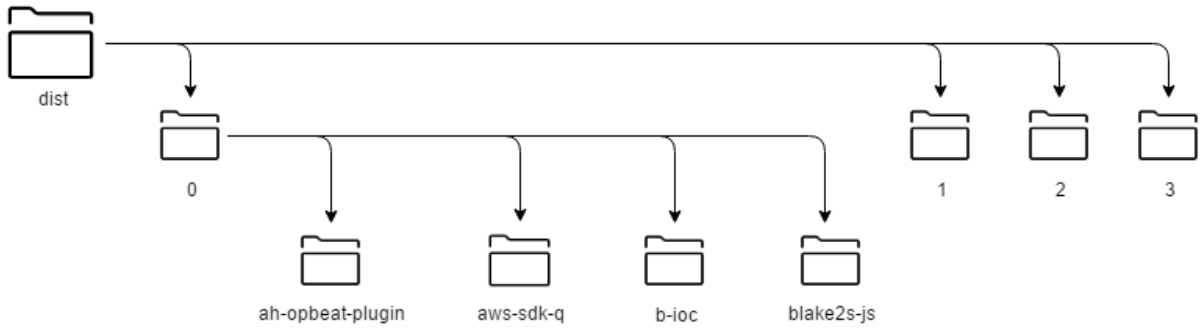

Figura 3.2: Árvore de diretórios de pacotes na máquina local.

3.3.3 Eliminação de arquivos e pacotes desnecessários

Com todos os repositórios clonados e organizados como descrito acima, a próxima fase foi remover diretórios de dependências e do próprio histórico de versionamento do *git*, já que não são dados que farão parte das análises realizadas neste projeto. Foram eliminados também todos os arquivos com extensão diferente de '.js' (extensão dos arquivos JavaScript). Os arquivos que terminavam '\min.js' ou '-min.js' também foram excluídos, já que estes são versões minificadas de outros arquivos de código-fonte. Alguns dos pacotes não continham arquivos JavaScript e consequentemente foram eliminados. Ao final desta etapa, ficamos com 23.709 repositórios clonados.

3.3.4 Eliminação de *forks*

Dando continuidade à coleta de dados, notou-se que alguns diretórios continham os mesmos arquivos ou arquivos muito similares a outros diretórios. Isso acontece porque alguns dos repositórios do GitHub são *forks* de outros. De acordo com o manual do GitHub, um *fork* é uma cópia de um repositório [12]. *Forks* permitem experimentar livremente mudanças sem afetar o projeto original, por exemplo, para propor mudanças ao projeto original [12]. O próximo passo foi listar e eliminar os repositórios que foram identificados como *forks*. Foram encontrados 517 *forks*, o que resultou na remoção da mesma quantidade de pacotes, reduzindo o escopo do estudo para 23.192 pacotes.

3.4 Cálculo das medidas de interesse

Após a coleta dos dados, a próxima etapa foi calcular as medidas de interesse para os arquivos de código-fonte. Um programa JavaScript foi criado para montar as árvores sintáticas de cada arquivo de código-fonte e calcular as medidas de interesse. Também foi gerado um arquivo .bat que percorre a estrutura de diretórios onde os pacotes foram armazenados e executa o programa de cálculo das medidas.

O programa de cálculo das medidas recebe o *path* do diretório de um pacote, o *id* do pacote no banco de dados e o nome do pacote. Ele percorre todos os seus subdiretórios, recursivamente, procurando por aquivos JavaScript (identificados pela extensão '.js'). Cada arquivo encontrado é lido e seu *parse* é realizado usando o Esprima. Caso o Esprima consiga montar sua árvore sintática, esta é analisada e as medidas especificadas na seção 3.2 são coletadas. Caso contrário, armazena-se o *path* do arquivo que não foi processado em um arquivo externo, junto com o respectivo erro indicado pelo Esprima.

Para os pacotes em que algum erro foi encontrado, foi decidido que todas as medidas coletadas seriam descartadas, ou seja, o pacote seria desconsiderado das análises. Em alguns deles nenhum dos arquivos de código-fonte puderam ter sua árvore sintática criada. Já outros pacotes tiveram seu arquivo de medidas criado, porém algum erro na criação da árvore foi identificado em um ou mais arquivos. Em alguns desses pacotes a árvore sintática não conseguiu ser gerada devido ao erro de *Unexpected token*, ou seja, algum caractere do código JavaScript não foi colocado no lugar correto e o Esprima não conseguiu validar a sintaxe do código para criar a árvore.

Em outros casos, o problema foi encontrado ao percorrer os nós da árvore usando a biblioteca Estraverse[2]. Estes casos foram identificados porque o Esprima possibilita a validação e criação da árvore sintática de códigos JSX, mas o Estraverse não reconhece certos nós utilizados nesta variação do JavaScript, resultando em erros quando nós do tipo *JSXElement* são encontrados no percurso da árvore sintática. Outros pacotes tiveram outros tipos de erros na criação da árvore sintática. Todos estes erros resultaram na exclusão de 2.305 pacotes, reduzindo a amostra para um total de 20.887 pacotes.

[2]github.com/estools/estraverse

Dada a árvore sintática resultante de um arquivo de código-fonte, o programa a percorre, nó por nó, procurando nós que representem declarações de funções e montando uma pilha com estas declarações. Toda vez que uma declaração de função é encontrada, um objeto é criado para representar a função declarada e guardar as seguintes informações sobre ela:

- nome;
- o número de vezes que cada nó do Esprima aparece dentro do escopo da função, representado por um vetor de inteiros;
- o número de funções escritas no escopo da função, representado por um vetor de funções "filhas";
- suas linhas inicial e final;
- suas colunas inicial e final;
- os operadores distintos que aparecem no escopo da função, bem como o número total de operadores (não distintos);
- os operandos distintos que aparecem no escopo da função, bem como o número total de operandos (não distintos).

Esse objeto é inserido na pilha e em seguida é verificado o tipo do nó atual no percurso da árvore, contabilizando este nó no vetor que representa a quantidade de nós de cada tipo que aparecem no escopo da última função adicionada na pilha. Quando todos os filhos de um nó são percorridos, é chamada uma função de saída do nó. Esta função retira a última função da pilha e, caso não houvesse mais funções na pilha, ela é considerada como uma função "pai"e adicionada a um novo vetor. Caso contrário, a função é adicionada ao vetor de funções "filhas"da função que ocupa o topo da pilha.

Após percorrer todos os nós da árvore, temos um vetor com todas as funções "pais"conhecidas. Para cada uma destas funções, e para cada uma das suas funções "filhas", é calculada a quantidade de operadores (total e distintos) e a quantidade de operandos (total e distintos). Tal cálculo é feito através da propriedade *tokens*, retornada pelo *parser* do Esprima. Por fim, um arquivo texto é criado para cada pacote analisado contendo as medidas especificadas na seção 3.2 para cada função e arquivo de código-fonte. As informações por

pacote são consideradas como o somatório das informações dos arquivos componentes do pacote.

3.4.1 Eliminação de pacotes triviais

Concluído o cálculo das medidas de interesse, os resultados foram analisados para verificar se todos os pacotes tinham relevância para o estudo. Notou-se que alguns pacotes eram compostos por somente um arquivo de código-fonte e que este não continha nenhum código JavaScript relevante. Após uma breve discussão sobre o tamanho de arquivos similares, decidiu-se então que os pacotes com tamanho menor que 1000 *bytes* seriam descartados das análises por representar projetos muito pequenos e sem relevância ao estudo. Esta restrição levou à eliminação de 1.938 pacotes, restando 18.849 pacotes para análise.

Mesmo com a limitação do tamanho, percebeu-se que alguns pacotes tinham arquivos com muitas linhas de código que não são consideradas relevantes para o estudo, ou seja, que não continham informações que influenciassem as medidas de interesse. Um exemplo de pacote deste tipo é o *brittanica-b*, que tem 105 arquivos e 1.424.690 linhas de código. Porém, este pacote não tem nenhuma declaração de função e nenhuma variável. Seu código-fonte não implementa nenhuma funcionalidade, sendo composto por declarações, como um *ASCII-art* e um *exports* de diversos objetos totalizando 95.000 linhas de código em um único arquivo. Para evitar que pacotes como este influenciassem os resultados da análise, colocamos uma segunda restrição, eliminando da análise pacotes que não tenham nenhuma declaração de função. Foram excluídos mais 290 pacotes, resultando em 18.659 pacotes disponíveis para a análise.

3.5 Considerações finais

Este capítulo apresentou as medidas de interesse para caracterização da estrutura de programas JavaScript e o processo de coleta e organização dos dados que foi aplicado para colher as medidas selecionadas de 18.659 pacotes do NPM. Os resultados da análise e as suas conclusões serão discutidos no próximo capítulo.

Capítulo 4. Resultado das Análises

4.1 Introdução

Nesse capítulo são discutidos estatisticamente os resultados obtidos através das análises descritas no capítulo 3. Foram geradas tabelas, matrizes de correlação e *boxplots* para apresentar os resultados.

A análise estatística dos resultados da análise estrutural dos pacotes JavaScript foi dividida em cinco seções além desta introdução. A Seção 4.2 descreve a representação estatística dos resultados, citando os tipos de representação que foram utilizados. A Seção 4.3 apresenta os resultados por pacote. A Seção 4.4 apresenta os resultados por arquivo. A Seção 4.5 apresenta os resultados por função. Finalmente, a Seção 4.6 apresenta as considerações finais deste capítulo.

4.2 Representação dos resultados

A represetação dos resultados da análise estatística realizada neste capítulo foi feitas através de gráficos *boxplot* e estruturas de tabelas e matrizes. As tabelas relacionam as medidas de interesse descritas na Seção 3.2, informando seus valores mínimo, máximo, média, mediana e desvio padrão. Em algumas situações, dados complementares são adicionados às tabelas e são explicados caso-a-caso.

As matrizes de correlação representam a força da relação entre pares de variáveis utilizando o coeficiente de Spearman. Este coeficiente é uma estatística não-paramétrica, ou seja, não supõe a existência de uma relação linear entre as variáveis. O coeficiente está no intervalo entre -1 e +1. Coeficientes próximos a -1 implicam em uma relação forte e inversa entre as variáveis (o valor de uma variável cresce quando o valor da outra diminui); coeficientes próximos a +1 implicam em uma relação forte e direta entre as variáveis (o valor de uma variável cresce quando o valor da outra cresce); por fim, coeficientes próximos de zero implicam em uma relação fraca entre as variáveis.

Um *boxplot* é um gráfico que apresenta o valor mínimo, o primeiro quartil, a mediana, o terceiro quartil e o valor máximo de uma série de dados. Dado um *boxplot*, é possível visualizar estatísticas descritivas sobre a distribuição dos dados e seus *outliers*, ou seja, dados que fogem à distribuição analisada. *Boxplots* são úteis por serem capazes de representar graficamente um resumo da distribuição dos dados sem ocupar muito espaço, podendo ser justapostos para mostrar diferentes distribuições.

Para os cálculos dos resultados apresentados neste capítulo foi utilizado o software de estatística R. R é uma linguagem e um ambiente de programação que foi criado em 1993 para fins de computação estatística e geração de gráficos. Hoje em dia, as grandes empresas utilizam R para análises de grandes volumes de dados (*Big Data Analytics*) e aprendizagem de máquina (*Machine Learning*[1])[30]. R oferece ao usuário diversos recursos para a manipulação de dados, cálculos e projeção de gráficos. Neste estudo, utilizamos os métodos de armazenamento dos dados (*data.frames*, *data.tables* e matrizes), operações para cálculos utilizando vetores de dados e operações para exportação de gráficos para arquivos em formato *PDF* e *PNG*.

4.3 Resultados por pacote

A Tabela 4.1 apresenta as estatísticas descritivas geradas para cada medida de interesse aplicável no nível de pacotes, considerando os 18.659 pacotes analisados. Assim como nas próximas tabelas, a coluna *min* representa o valor mínimo da medida, a coluna *25%* representa o primeiro quartil, a coluna μ representa o valor médio da medida, a coluna σ representa o desvio padrão da medida, a coluna *med* representa a mediana da medida, a coluna *75%* representa o terceiro quartil, a coluna *95%* representa o 95% percentil e a coluna *max* representa o valor máximo da medida.

Em relação ao número de arquivos por pacote, vemos que a maioria dos pacotes têm poucos arquivos, já que a média do número de arquivos por pacote é 11 e a mediana é 4. Além disso, 95% dos pacotes analisados têm menos de 40 arquivos. Porém, alguns pacotes são muito maiores, chegando ao limite máximo de 2.806 arquivos em um mesmo pacote.

[1] https://pt.coursera.org/learn/machine-learning

	min	25%	μ	σ	med	75%	95%	max
Número de arquivos	1	2	11	41	4	8	38	2.806
Linhas de código	1	142	3.342	19.729	353	1.044	11.149	1.101.324
Número de variáveis	0	7	224	1.267	20	65	775	53.501
Número de funções	1	11	258	1.197	29	92	967	28.552
Tamanho (bytes)	1.000	4.018	133.218	832.195	10.705	34.762	430.676	40.189.915

Tabela 4.1: Tabela de estatísticas descritivas por pacote

Esses *outliers* fazem com o que o desvio padrão do número de arquivos seja muito maior do que a média e a mediana.

Em relação ao número de linhas de código, vemos que ele varia muito entre os pacotes, existindo pacotes com apenas uma linha de código, enquanto outros chegam a ter mais de 1 milhão de linhas. De todo modo, percebemos que grande parte dos pacotes são pequenos, tendo em média cerca de 3.000 linhas de código e observando-se apenas 5% de arquivos com mais de 11.000 linhas. Também é importante ressaltar que alguns arquivos JavaScript são minificados, ou seja, não há espaços em branco, quebras de linhas ou identação de código no arquivo. Estes arquivos, mesmo que muito grandes, possuem apenas uma linha de código. A minificação é uma técnica utilizada para reduzir o tamanho de arquivos JavaScript, especialmente quando estes são transferidos pela Web.

Já sobre o número de variáveis é possível notar que há pacotes sem declaração de variável, enquanto que outros têm mais de 50.000 variáveis. A mediana do número de variáveis tem valor baixo, de 20 variáveis por pacote, ainda que a média e o desvio padrão tenham valores mais altos. Mais uma vez se observa que uns poucos pacotes muito grandes arrastam a média para cima e alargam o desvio padrão, ainda que o número de variáveis declaradas nos pacotes próximos ao corte de 5% seja também muito alta (775 variáveis).

Como mencionado na Seção 3.4.1, descartamos pacotes sem declaração de função. Assim, temos que o número mínimo de funções nos pacotes analisados é igual a 1. O número máximo, mais uma vez é alto, aumentando a média e o desvio padrão, enquanto a mediana permanece relativamente baixa (29 funções por pacote). O mesmo raciocínio se aplica ao tamanho dos pacotes: a maior parte dos pacotes têm tamanho pequeno, mas o desvio padrão é 6,5 vezes maior que a média, que é 12 vezes maior que a mediana.

Com esses dados é possível relacionar o número de variáveis com os valores de outras

medidas. Por exemplo, se dividirmos o número médio de variáveis pelo número médio de arquivos, observamos que em média são declaradas 20 variáveis por arquivo. Do mesmo modo, ao comparar o número médio de variáveis com o número médio de linhas de código notamos que a cada 14 linhas de código uma variável é declarada. E, por fim, relacionamos o número médio de variáveis com o número de funções e percebemos que poucas variáveis são declaradas por função – em média, apenas uma variável por função.

Para cada medida, analisamos em que pacotes foram observados os seus maiores valores e notou-se que a maioria destes valores pertenciam a pacotes diferentes. O número máximo de arquivos é referente ao pacote *dojox*, que faz parte do *Dojo Toolkit*, cujo objetivo é agilizar o desenvolvimento de aplicações Web. Seu número de linhas de código é de 330.154 e seu tamanho é de 10,4 MB.

O valor máximo para o número de linhas de código se refere ao pacote *array-last*, que tem como objetivo retornar o último ou os n últimos elementos de um vetor. Este pacote tem 12 Mb e apenas 12 arquivos. A explicação para este tamanho e número de linhas de código é a existência de um arquivo no pacote com 999.999 linhas, contendo apenas um *module.exports* de um vetor de todos os números de 1 até 999.999.

O número máximo de variáveis pertence ao pacote *cesium-buildings*, que é um plugin para visualizar objetos em formato 3D, principalmente prédios e construções. O pacote tem 34 arquivos, 623.319 linhas de código e 26,6 Mb de tamanho. Por fim, o pacote *regl*, que tem por objetivo simplificar a programação WebGL, tem o maior número de funções e também o maior tamanho. O pacote tem 206 arquivos e 394.539 linhas de código.

Com relação às linhas de código, todos os quatro pacotes se encontram entre os 15 maiores, sendo os pacotes *array-last* e *cesium-buildings* o 1^o e o 2^o com mais linhas de código, respectivamente. Quanto ao tamanho, dos quatro pacotes, somente o pacote *regl* se encontra entre os 10 maiores. O pacote *array-last* é o segundo maior entre os quatro, aparecendo em 12^o entre todos os pacotes analisados.

As Figuras 4.1 e 4.2 mostram, respectivamente, os *boxplots* com e sem *outliers* para cada uma das medidas de interesse por pacote. Estes gráficos devem ser interpretados da seguinte maneira: (i) a linha na base de cada gráfico representa o valor mínimo; (ii) a linha na base do retângulo central representa o primeiro quartil, ou seja, entre o mínimo e o primeiro quartil se encontram 25% dos dados; (iii) a linha no centro do retângulo central

representa a mediana; (iv) a linha no topo do retângulo central representa o terceiro quartil; e (v) a linha no topo de cada gráfico representa o valor máximo, descartados os *outliers* quando aplicável.

Percebe-se que os retângulos centrais dos *boxplots* apresentados na Figura 4.1 são irreconhecíveis, dado o número dos *outliers* ali representados e os valores muito díspares das medidas observadas nestes projetos. Por outro lado, é interessante notar na Figura 4.2 que as medidas tem dispersão proporcionais, ou seja, as distâncias entre os trechos consecutivos dos *boxplots* guardam proporções semelhantes.

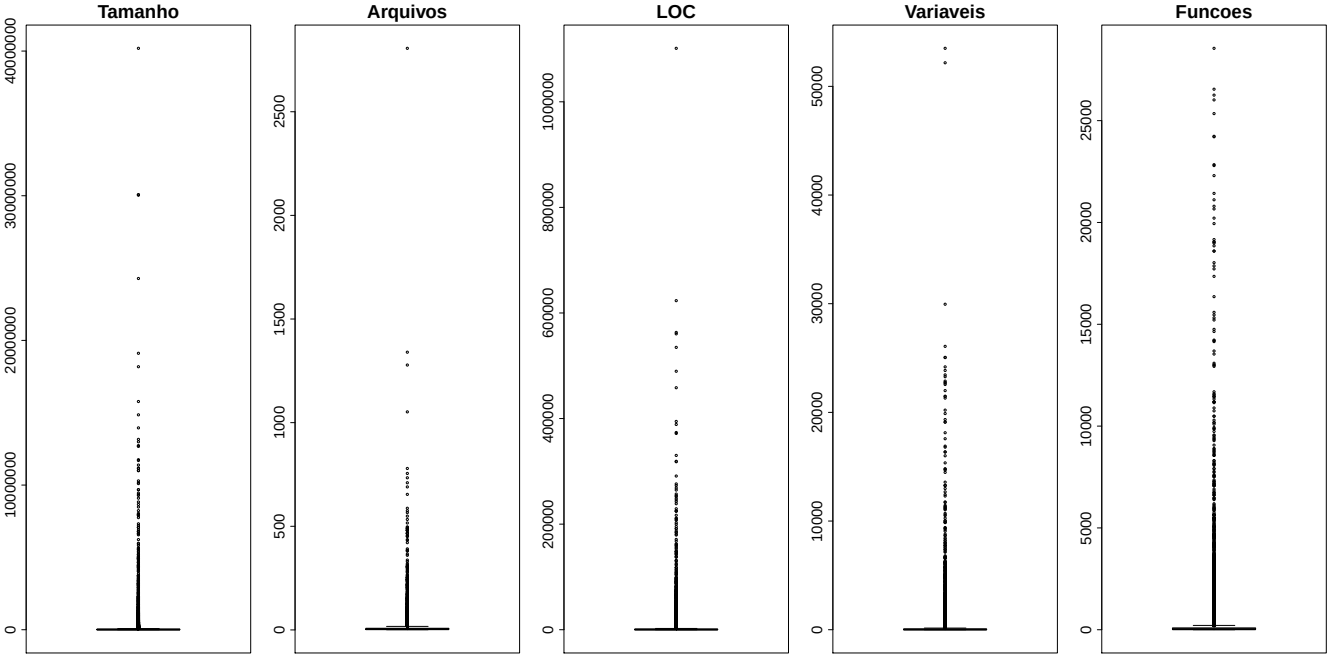

Figura 4.1: Boxplots com outliers das medidas por pacote.

	Número de arquivos	Linhas de código	Número de variáveis	Número de funções	Tamanho
Número de arquivos	-	0.71	0.59	0.65	0.68
Linhas de código	-	-	0.86	0.89	0.97
Número de variáveis	-	-	-	0.87	0.88
Número de funções	-	-	-	-	0.90
Tamanho	-	-	-	-	-

Tabela 4.2: Matriz de correlação das medidas por pacote

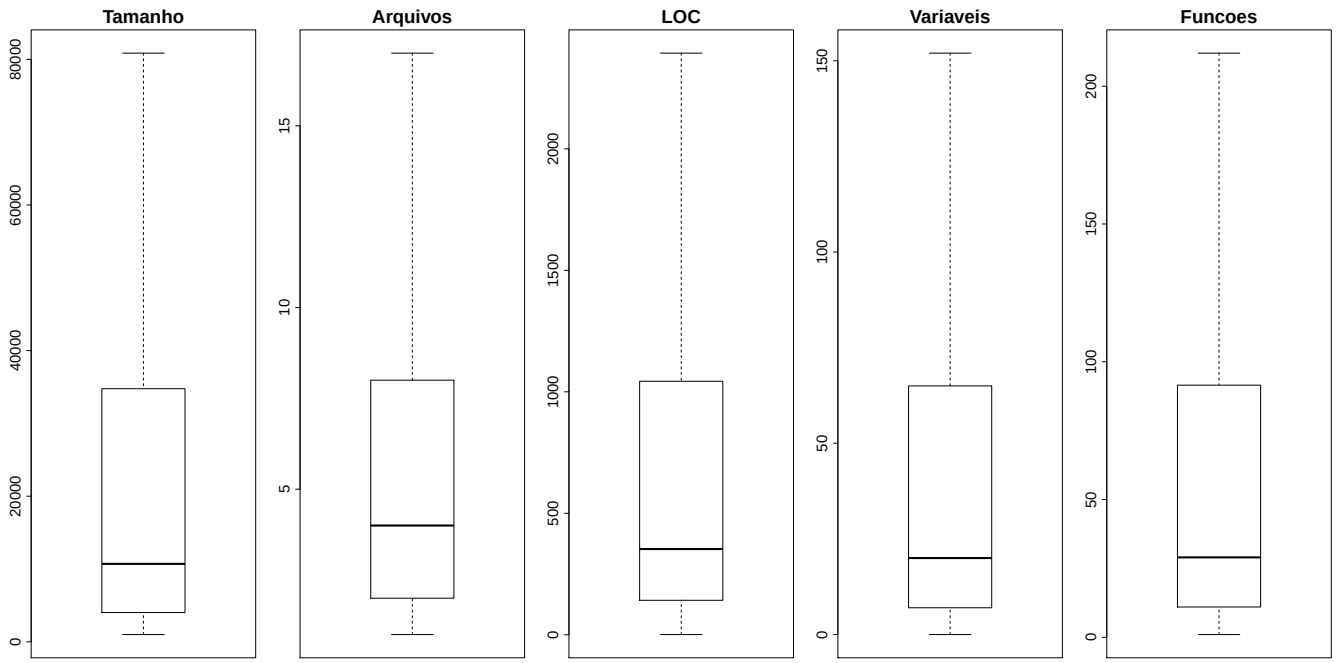

Figura 4.2: Boxplots sem outliers das medidas por pacote.

Analisando a matriz de correlação entre as diferentes medidas, apresentada na Tabela 4.2, podemos ver que o maior coeficiente se dá entre o número de linhas de código e o tamanho do pacote, ou seja, quanto maior for o tamanho do pacote, maior será o seu número de linhas de código.

Uma correlação importante é entre o número de linhas de código e o número de funções. Um coeficiente de 0.89 indica que quanto maior o seu número de linhas de código, mais funções um pacote terá. Isto explica a razão muito próxima entre a média e a mediana do número de linhas de código (9,47) e do número de funções (8,90), indicando que os desenvolvedores JavaScript tendem a escrever funções com poucas linhas de código.

Outra correlação interessante ocorre entre o número de arquivos e o tamanho do pacote (0.68). Este valor intermediário indica que um pacote grande nem sempre terá muitos arquivos, podendo então ter poucos arquivos grandes.

4.4 Resultados por arquivo

Nesta seção são apresentadas as estatísticas descritivas para cada medida aplicadas por arquivo. Ao todo, foram considerados 208.573 arquivos. A Tabela 4.3 ilustra os resultados.

	min	25%	μ	σ	med	75%	95%	max
Linhas de código	0	20	298,96	3.430,75	53	133	633	999.999
Número de variáveis	0	0	20,03	172,19	2	8	49	14.480
Número de funções	0	1	23,07	167,04	3	11	38	4.562
Tamanho (bytes)	0	608	11.917,67	104.149,96	1.679	4.504	26.100	11.888.903

Tabela 4.3: Tabela de estatísticas descritivas por arquivo

O primeiro fator a ser notado na Tabela 4.3 é que o mínimo para todas as medidas é equivalente a 0. Entende-se o porquê de algumas medidas, como número de funções e número de variáveis serem iguais a zero. Alguns arquivos apenas incluem um *module.exports* que não contém nenhuma função ou variável, porém esses arquivos não podem ser descartados, pois ainda são essenciais para que o pacote realize suas funções.

Em relação ao número de linhas de código, percebemos que o desvio padrão é um número muito alto, indicando que há uma grande dispersão em relação à média entre os arquivos. É possível perceber que os arquivos são pequenos em sua maioria. A mediana para o número de linhas de código é de 53, ou seja, 50% dos arquivos têm menos que 53 linhas. Percebe-se também que os valores discrepantes, os *outliers*, elevam os valores de média e desvio padrão, já que 75% dos arquivos têm menos de 133 linhas de código. O pacote com o maior arquivo, em número de linhas de código, é o *array-last*.

Sobre o número de variáveis, nota-se que alguns arquivos não tem declaração de variáveis. A média de declarações de variáveis se encontra em torno de 20, com desvio padrão em torno de 172, ambos valores muito altos em relação à mediana (2). Mais uma vez isso explicado pelo grande número de *outliers* que arrastam os valores de média e desvio padrão para muito acima do valor da mediana. O arquivo com maior número de variáveis pertence ao pacote *cesium-buildings*.

Assim como o número de variáveis, existem arquivos que não tem nenhuma declaração de função. É interessante notar que 25% dos arquivos têm no máximo uma função declarada e que 75% deles têm menos de 11 funções. Novamente vemos que a média e o desvio padrão têm valores muito altos, devido aos poucos arquivos com um número muito alto de funções, chegando a mais de 4.000 funções em um mesmo arquivo. O valor máximo de número de funções em um mesmo arquivo se encontra no pacote *box2d*, uma *engine* de simulação de física para jogos.

Por fim, nota-se que o tamanho da maior parte dos arquivos é muito pequeno. Em média, um arquivo de código-fonte JavaScript tem 11 Kb, número que pode ser considerado alto se comparado à mediana de apenas 1 Kb. 75% dos arquivos têm menos de 4,5 Kb e apenas 5% têm tamanho maior do que 26 Kb. O desvio padrão de 104 Kb é novamente puxado pra cima pelos *outliers*, como por exemplo um arquivo com mais de 11 Mb de dados, do pacote que contém o arquivo com maior tamanho é o *array-last*.

Assim como nas análises por pacote, observamos distribuições assimétricas para todas as características analisadas. Há uma maior concentração das características dos arquivos em torno dos valores mais baixos, enquanto a distribuição apresenta uma cauda longa em direção a valores muito grandes. Isto indica que projetos de software devem se comparar com as medianas que foram observadas neste estudo, uma vez que as médias são amplificadas pelos valores extremos e o desvio padrão não pode ser facilmente interpretado devido à assimetria das distribuições.

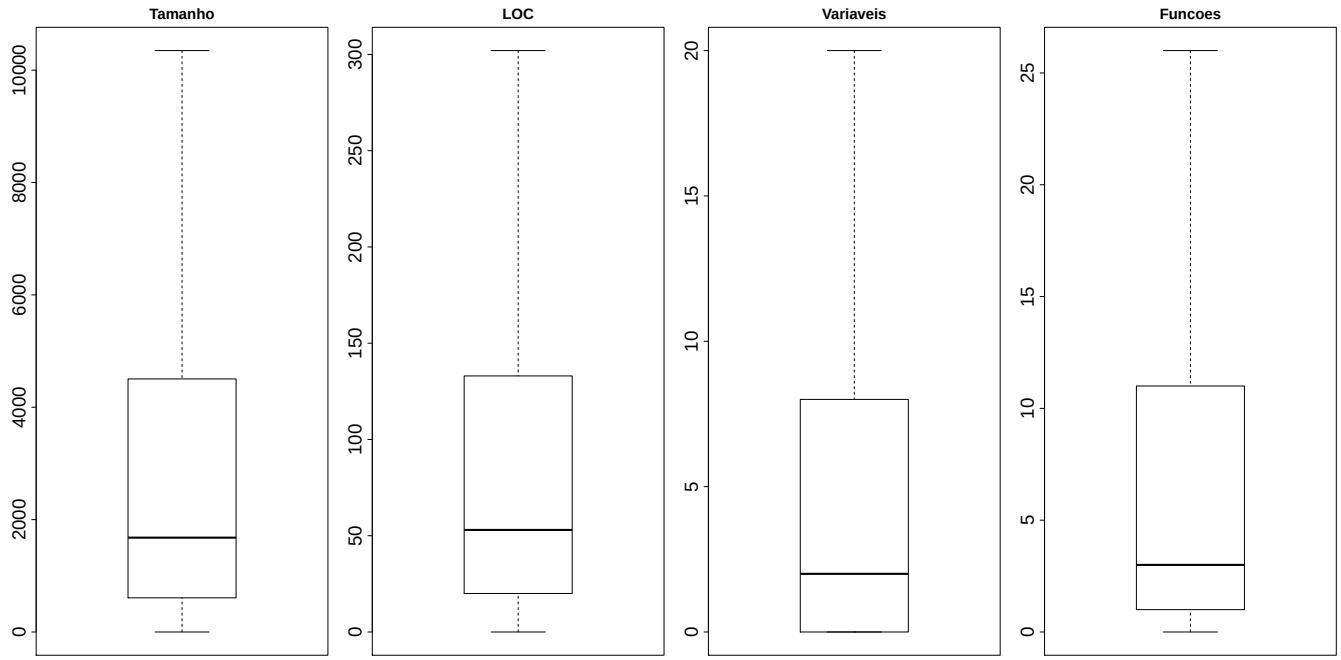

Figura 4.3: *Boxplot* sem *outliers* das medidas por arquivo.

A Figura 4.3 apresenta os *boxplots* sem *outliers* das medidas de interesse por arquivo. *Boxplots* com *outliers* não são apresentados porque os IQR das distribuições são muito pequenos em relação aos valores máximos de cada característica. Isto leva os *boxplots* com *outliers* a ficarem planos próximo ao eixo X, sem apresentar qualquer informação útil. Quando removemos os *outliers*, percebe-se claramente a assimetria na distribuição das diversas características, com medianas próximas aos valores mínimos.

	Linhas de código	Número de variáveis	Número de funções	Tamanho
Linhas de código	-	0,69	0,73	0,90
Número de variáveis	-	-	0,80	0,70
Número de funções	-	-	-	0,72
Tamanho	-	-	-	-

Tabela 4.4: Matriz de correlação das medidas por arquivo

Os coeficientes de correlação apresentados na Tabela 4.4 indicam que as relações entre as características observadas são fortes, sendo mais forte a correlação entre o número de linhas de código e o tamanho dos arquivos: conforme esperado, quanto maior o número de linhas de código, maior o tamanho do arquivo. A correlação entre o número de funções e o número de linhas de código indica que quanto mais linhas de código, mais funções o arquivo terá, reforçando a tendência de que as funções escritas em JavaScript tenham poucas linhas de código.

4.5 Resultados por função

Nesta seção são apresentadas as análises realizadas por função implementada nos arquivos JavaScript. O conjunto de todas as funções, de todos os pacotes considerados, chega a 4.811.111 funções. A Tabela 4.5 apresenta as estatísticas descritivas de cada uma das medidas coletadas para essa parte.

	min	25%	μ	σ	med	75%	95%	max
Linhas de código	1	3	25,99	429,3	5	13	63	317.354
Número de variáveis	0	0	0,87	3,0	0	1	4	1.314
Chamadas de funções	0	1	3,21	70,7	2	4	11	152.945
Declarações de funções	0	0	0,82	13,2	0	0	3	12.510
Volume de Halstead	2	41	313,88	6.384,5	93	232	938	6.585.482
Complexidade ciclomática	1	1	2,20	5,2	1	2	6	5.352
Maintainability index	0	60	68,98	15,1	71	80	90	98

Tabela 4.5: Tabela de estatísticas descritivas por função

Em relação ao número de linhas de código, nota-se que a média entre as funções é de 26 linhas, o que é um número alto para funções. Já a mediana tem valor igual a 5, indicando

que grande parte das funções são pequenas e podem ser mais facilmente reutilizadas e mantidas. Este argumento é reforçado ao observarmos que 75% das funções têm menos de 13 linhas de código. O desvio padrão, assim como a média, é influenciado pelos *outliers* e possui um valor muito alto.

A função com o maior número de linhas de código pertence ao pacote *react-timeseries-table*. A funcionalidade do pacote é montar uma tabela para renderizar objetos de outro pacote, denominado *pond*. A função em análise não tem nenhuma variável em seu escopo, nenhuma outra função é chamada por ela, assim como não tem nenhuma função escrita dentro dela. O número excessivo de linhas de código ocorre porque a função é um simples *module-exports* com um objeto que contém diversos vetores.

O número de variáveis por função é bem pequeno, com mediana igual a zero, ou seja, 50% das funções não tem variáveis declaradas em seu escopo (o que é próprio de funções pequenas). Seu desvio padrão também é pequeno, o que indica que não há muita dispersão do número de variáveis entre as funções. Isso pode ser notado também ao ver que 95% das funções têm menos de quatro variáveis. A função com o maior número de variáveis pertence ao pacote *d2protocol*, o qual implementa o protocolo *dofus 2.0* em JavaScript. A função tem 60.805 linhas de código, 1.314 chamadas a outras funções e 1.314 funções declaradas dentro do seu código.

Sobre o número de chamadas de função, observamos que cada função chama poucas outras funções. A média é de 3,21 chamadas, enquanto a mediana é de duas chamadas. Apenas 25% das funções fazem chamadas a mais de quatro funções. O uso de declaração de funções dentro de outras funções, as denominadas *funções filhas*, não é algo que ocorre com muita frequência, mesmo considerando-se a estreita conexão da linguagem com o uso de *callbacks* e *closures*. Sua média é menor que 1 e sua mediana igual a zero. Apenas 25% das funções declaram outras funções entre as suas linhas de código e apenas 5% declaram mais de três funções. Em compensação, o pacote *telehash-c* possui 12.510 funções filhas.

Em relação ao Volume de Halstead temos que a mediana é equivalente a 93. Logo, pela equação pelo qual o volume é determinado, Eq. 3.1, é possível deduzir que o número de operadores e operandos das funções não é muito alto. Assim temos que 50% das funções não são tão complexas e isso facilita o entendimento e manutenção delas. Contudo, algumas funções têm volumes altos e, consequentemente, alta complexidade. Por exemplo,

o pacote *benchmark-octane*, um *benchmark* para *engines* de execução JavaScript, possui a função com o maior volume (6.585.482) entre os pacotes analisados.

Sobre a complexidade ciclomática, nota-se que há poucos caminhos independentes a serem seguidos na maior parte das funções JavaScript, já que o seu valor médio é em torno de 2 e sua mediana é 1. Seu desvio padrão não é alto, sendo somente cinco vezes maior que a mediana. É possível notar também que apenas 5% das funções têm mais do que seis caminhos independentes, o que contribui para reforçar que a maior parte das funções JavaScript possui uma implementação bastante simples.

Por fim, temos o *Maintainability Index*. De acordo com o MI, Uma função é considerada boa para manutenção se tiver um índice entre 20 e 100 (99.5% das funções analisadas), moderada se tiver índice entre 10 e 19 (0.3% das funções analisadas) e ruim se seu índice for entre 0 e 9 (0.2% das funções analisadas) [16]. Na anáise percebe-se que algumas funções tem este índice igual a 0, o que indica que sua manutenção não será fácil. Porém, a maioria das funções tem um bom índice: 75% delas têm um índice maior que 60, 25% delas têm valor maior que 80 e 5% têm valor maior que 90. Nenhuma no entanto chega a 100, sendo o valor máximo de 98 em diversas funções. Sendo assim, a maior parte das funções JavaScript parece ser de fácil manutenção.

Os sete primeiros *boxplots* na Figura 4.4 representam as distribuições sem *outliers* das medidas de interesse por função. O último *boxplot* representa a distribuição do *Maintainability Index* com *outliers* (abaixo da linha que representa o valor mínimo sem *outliers*). Percebe-se que neste caso as distribuições são muito distintas, variando desde situações onde a maior parte dos dados se concentra em torno do zero (funções filhas) até distribuições com relativamente poucos *outliers* (MI). Exceto pelo MI, todas as distribuições apresentam a assimetria e concentração perto dos valores mínimos que foram previamente observadas para os arquivos e pacotes.

A Tabela 4.6 apresenta a matriz de correlação das diferentes medidas por função. A relação com mais força é entre o número de linhas de código e o *Maintainability index*, mas é importante notar que é uma relação inversa, ou seja, quanto maior o número de linhas de código, menor será o *MI*. Esta relação não é novidade, pois a fórmula do MI já demonstra a relação inversa (e logarítmica) entre as medidas. Funções com muitas linhas de código serão de manutenção mais difícil. Essa mesma análise também serve para as

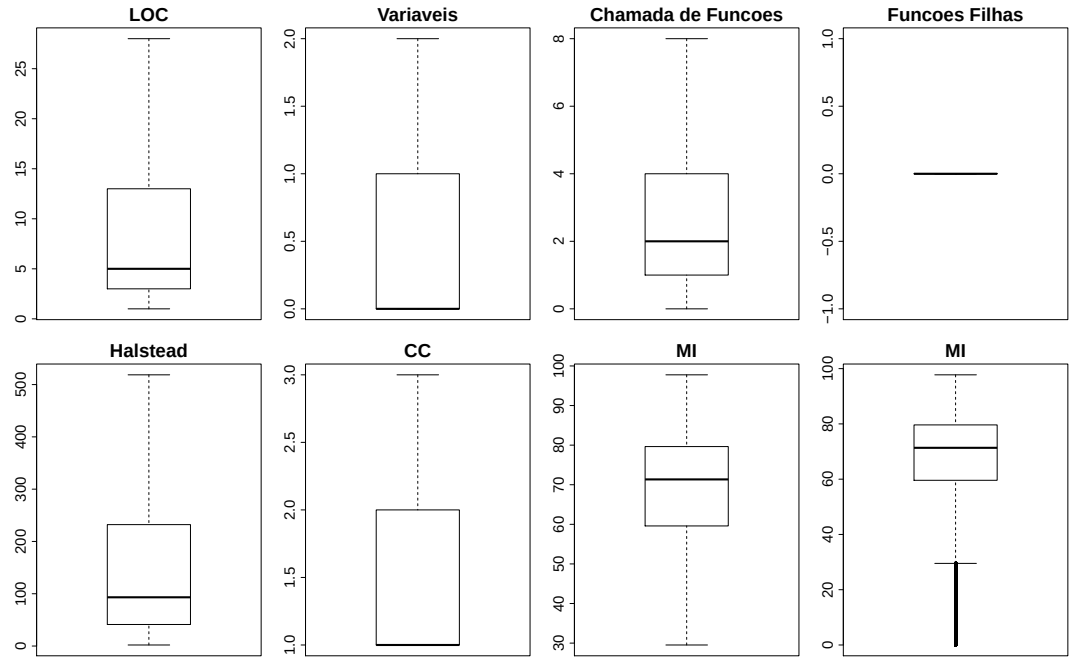

Figura 4.4: *Boxplot* sem *outliers* por função.

	Linhas de código	Número de variáveis	Chamadas de funções	Declarações de funções	Volume de Halstead	Complexidade ciclomática	Maintainability index
Linhas de código	-	0.42	0.39	0.410	0.451	0.306	-0.97
Número de variáveis	-	-	0.48	0.218	0.660	0.459	-0.50
Chamadas de funções	-	-	-	0.138	0.659	0.402	-0.49
Declarações de funções	-	-	-	-	0.047	-0.071	-0.36
Volume de Halstead	-	-	-	-	-	0.640	-0.61
Complexidade ciclomática	-	-	-	-	-	-	-0.41
Maintainability index	-	-	-	-	-	-	-

Tabela 4.6: Matriz de correlação das medidas por função

relações entre volume de Halstead e *MI* e entre a complexidade ciclomática e *MI*, já que estas medidas também entram com sinal negativo no cálculo do índice de manutenção.

Uma relação com força moderada ocorre entre o volume de Halstead e o número de variáveis, indicando que quanto maior o número de variáveis, maior será o volume da função. Isso ocorre porque cada declaração de variável aumenta o número de operandos da função, número esse que é utilizado para o calcular o volume da função.

	min	25%	μ	σ	med	75%	95%	max
AssignmentExpression	0	0	1,92	15,16	0	1	8	9.250
ArrayExpression	0	0	0,43	54,4	0	0	1	63.405
BlockStatement	1	1	1,89	5,2	1	2	5	8.557
BinaryExpression	0	0	24,85	26,10	0	1	8	9.052

	min	25%	μ	σ	med	75%	95%	max
BreakStatement	0	0	0.06	1,33	0	0	0	579
CallExpression	0	1	3,21	70,67	2	4	11	152.945
CatchClause	0	0	0,01	0,14	0	0	0	24
ConditionalExpression	0	0	0,21	0,94	0	0	1	330
ContinueStatement	0	0	0	0,19	0	0	0	124
DoWhileStatement	0	0	0,01	0,46	0	0	0	448
EmptyStatement	0	0	0	0,39	0	0	0	408
ExpressionStatement	0	0	2,89	72,12	1	3	10	152.945
ForStatement	0	0	0,09	0,45	0	0	1	88
ForInStatement	0	0	0,02	0,18	0	0	0	16
FunctionDeclaration	0	0	0,18	0,38	0	0	1	1
FunctionExpression	0	1	0,81	0,38	1	1	1	1
Identifier	0	5	20,82	105,16	10	21	69	155.112
IfStatement	0	0	0.77	3,70	0	1	4	5.351
Literal	0	0	6,59	262,70	1	4	17	190.807
LabeledStatement	0	0	0	0,12	0	0	0	78
LogicalExpression	0	0	0,58	2,63	0	0	3	788
MemberExpression	0	1	6,69	34,85	3	7	24	41.863
NewExpression	0	0	0,17	12,71	0	0	1	25.123
ObjectExpression	0	0	0,47	17,60	0	0	2	21.562
Property	0	0	1,32	60,80	0	0	4	66.123
ReturnStatement	0	0	0,67	1,08	1	1	2	220
SequenceExpression	0	0	0,13	0,80	0	0	1	472
SwitchStatement	0	0	0,01	0,12	0	0	0	17
SwitchCase	0	0	0,07	2,21	0	0	0	802
ThisExpression	0	0	1,08	4,20	0	1	5	3.985
ThrowStatement	0	0	0,04	0,28	0	0	0	50
TryStatement	0	0	0,02	0,15	0	0	0	24
UnaryExpression	0	0	0,60	33,93	0	0	3	44.323
UpdateExpression	0	0	0,14	0,79	0	0	1	401
VariableDeclaration	0	0	0,86	2,99	0	1	4	1.314
VariableDeclarator	0	0	1,39	5,39	0	1	6	3.139

	min	25%	μ	σ	med	75%	95%	max
WhileStatement	0	0	0,03	0,32	0	0	0	96
WithStatement	0	0	0	0,01	0	0	0	5
ArrayPattern	0	0	0	0,02	0	0	0	19
ObjectPattern	0	0	0	0,03	0	0	0	18
ArrowFunctionExpression	0	0	0,01	0,51	0	0	0	501
TaggedTemplateExpression	0	0	0	0.09	0	0	0	212
Super	0	0	0	0,03	0	0	0	9
YieldExpression	0	0	0	0,12	0	0	0	52
AssignmentPattern	0	0	0	0,05	0	0	0	24
RestElement	0	0	0	0,02	0	0	0	16
MethodDefinition	0	0	0	0,23	0	0	0	308
TemplateElement	0	0	0,01	0,39	0	0	0	309
TemplateLiteral	0	0	0	0,22	0	0	0	225
SpreadElement	0	0	0	0	0	0	0	10
ForOfStatement	0	0	0	0,03	0	0	0	10

Tabela 4.7: Frequência dos nós sintáticos por função

A Tabela 4.7 mostra as estatísticas da frequência de cada tipo de nó do Esprima por função. Analisando as tabelas, vemos que a maioria dos tipos têm mediana equivalente a 0. Um fato digno de nota é que o tipo *FunctionDeclaration* tem mediana igual a 0, e dado o seu 95% percentil apresentado, vemos que somente 5% das funções em JavaScript são declaradas da maneira tradicional – sem atribuição à variável, indicando o nome da função e sem ser declarada *inline* na chamada de outra função. As funções analisadas são em sua maioria do tipo *FunctionExpression*, o que explica a mediana deste tipo ser equivalente a 1. Pelas estatísticas analisadas também notamos que o tipo de nó com maior dispersão entre as funções é o *Literal*, dado o seu desvio padrão.

4.6 Considerações Finais

Após as análises dos pacotes, arquivos e funções JavaScript, algumas conclusões podem ser tiradas para cada uma das partes. Por pacote, vemos que o comum é que estes sejam pequenos, com poucas linhas de código, como visto na maioria dos pacotes analisados. Podemos afirmar o mesmo sobre o seu número de arquivos e funções.

É possível concluir que a maioria dos arquivos é pequena, no número de linhas e consequentemente no tamanho em *bytes*. Também concluímos que a maioria dos arquivos de código-fonte JavaScript declara menos de 11 funções e 8 variáveis, caracterizando arquivos pequenos e provavelmente de fácil manutenção.

Por fim, em relação às funções percebe-se que o padrão é que elas também sejam pequenas, como na maioria das funções analisadas. Permanecendo esse número pequeno, outros valores serão influenciados e se tornaram melhores, como o volume de Halstead e o MI, o que indicaria menos complexidade na funções, maiores chances de reutilização e mais facilidade na manutenção da função. Observamos também que, sendo o número de mediana da complexidade ciclomática das funções analisadas igual a 1, somado a informação de que a maioria das funções em JavaScript tem poucas linhas de código, é possível afirmar que o paradigma que mais caracteriza o desenvolvimento em JavaScript atualmente é o funcional, sendo essa afirmação aplicável a pelo menos 50% das funções analisadas.

Capítulo 5. Conclusão

5.1 Contribuições

A principal contribuição do estudo reportado neste projeto são os resultados obtidos e consequentemente o melhor entendimento de como é organizado um código-fonte JavaScript atualmente. Podemos observar que a grande maioria dos componentes de código-fonte JavaScript é pequena e simples, muito embora existam pacotes, arquivos e mesmo funções que ultrapassam em muito qualquer limite que se defina como simples, contendo milhares de variáveis, declarações de funções, linhas de código, operandos, operadores ou caminhos independentes para a execução do código.

Percebe-se claramente que todas as características observadas nos três níveis (pacote, arquivo e função) apresentam uma distribuição assimétrica e densa à esquerda, concentrando a maior parte dos valores observados mais perto do valor mínimo do que da média. A cauda longa à direita das distribuições, embora pouco densa, é muito extensa e arrasta a média e o desvio padrão para valores mais altos, permanecendo a mediana e mesmo o terceiro quartil com valores baixos para quase todas as características.

5.2 Limitações

Não foi possível encontrar uma maneira eficiente de coletar os dados, como uma API própria do NPM que retornasse as informações necessárias para calcular as métricas pretendidas. Assim, implementamos uma automação que utilizou o mecanismo de busca do site para coletar estas informações.

Como dito na Seção 3.3, apesar de o repositório conter mais de 400.000 pacotes, houveram algumas limitações devido a falhas no mecanismo de busca no site do NPM que impediram a coleta de dados sobre todos os pacotes, limitando este estudo a cerca de 34.000 pacotes. Posteriormente, os problemas no mecanismo de busca foram corrigidos, mas já havíamos encerrado a etapa de coleta de dados.

Outra limitação foi identificada durante o processo de cálculo das medidas de interesse, seção 3.4: alguns códigos-fonte continham programação na linguagem JSX. Apesar do Esprima identificar nós do tipo JSX, o pacote utilizado para percorrer as árvores sintáticas não reconhecia os tipos de nós utilizados para identificar esta linguagem. Assim, mais de 500 pacotes tiveram seu código-fonte coletado, mas foram descartados do estudo.

Outros pacotes foram eliminados devido a erros de sintaxe em ao menos um dos seus código-fonte, impossibilitando que o Esprima criasse sua árvore sintática e, consequentemente, que as medidas selecionadas fossem coletadas para o arquivo e pacote em questão.

5.3 Trabalhos futuros

Estudos futuros podem ser planejados e executados a partir dos resultados reportados neste trabalho. Uma sugestão seria utilizar as métricas e conclusões deste estudo para definir um padrão prático de desenvolvimento em JavaScript, que imponha restrições sobre o código-fonte de modo a impedir que este se torne um dos *outliers* em nossas análises. O padrão também poderia ser personalizado para os diferentes ambientes em que a linguagem JavaScript é utilizada.

Outra análise interessante seria comparar as características estruturais de programas JavaScript que utilizem diferentes *frameworks*. Por exemplo, será que programas desenvolvidos usando o *framework* Angular para o desenvolvimento do lado cliente de aplicações *Web* são estruturalmente diferentes de aplicações que usem o *framework* React, desenvolvido com o mesmo fim? Será que as características estruturais mudam entre programas JavaScript desenvolvidos para o lado cliente e programas desenvolvidos para o lado servidor?

Também seria interessante analisar se determinadas características estruturais estão presentes em maior ou menor frequência em arquivos que são alterados para resolver um mesmo *issue* ou que são salvos ao mesmo tempo com frequência no sistemas de controle de versão. Esta análise pode ser feita coletando outras informações dos repositórios dos projetos sob análise.

Referências

[1] Fernando Brito e Abreu; Rogério Carapuça. «Object-Oriented Software Engineering: Measuring and Controlling the Development Process». Em: McLean, VA, USA: 4th Int. Conf. on Software Quality, 1994.

[2] The npm Blog. *Better Search is Here!* URL: http://blog.npmjs.org/post/154912817335/better-search-is-here.

[3] Richard Bovell. *Understanding JavaScript Closures With Ease*. URL: http://javascriptissexy.com/understand-javascript-closures-with-ease/.

[4] PAUL BROWN. *State of the Union: npm*. URL: https://www.linux.com/news/event/Nodejs/2016/state-union-npm.

[5] Christopher Buecheler. *Understanding JavaScript Callbacks*. URL: https://closebrace.com/tutorials/2017-01-17/understanding-javascript-callbacks.

[6] IBM Knowledge Center. *Halstead Metrics*. URL: https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm.

[7] *Como Funciona a Função Require do NodeJs?* URL: http://nodebr.com/como-funciona-a-funcao-require-do-node-js/.

[8] Mozilla Corporation. *History of the Mozilla Project*. URL: https://www.mozilla.org/en-US/about/history/details/.

[9] Esprima. *Chapter 2. Syntactic Analysis (Parsing)*. URL: http://esprima.readthedocs.io/en/latest/syntactic-analysis.html.

[10] Alysson Franklin. *Tenha o DOM*. URL: https://tableless.com.br/tenha-o-dom/.

[11] Jaydson Gomes. *O 22 Anos do Javascript, Contados Pelo Seu Criador Brendan Eich*. URL: https://braziljs.org/blog/os-22-anos-javascript-contados-pelo-seu-criador-brendan-eich/.

[12] GitHub Help. *Fork A Repo*. URL: https://help.github.com/articles/fork-a-repo/.

[13] Alyson La. *Language Trends on GitHub*. URL: https://github.com/blog/2047-language-trends-on-github.

[14] Michele Lanza e Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[15] Manjunath M. *The Concept of Multi-Paradigm Programming Languages*. URL: https://msritse2012.wordpress.com/2013/01/31/the-concept-of-multi-paradigm-programming-language-manjunath-m/.

[16] Microsoft. *Code Metrics Values*. URL: https://msdn.microsoft.com/en-us/library/bb385914.aspx.

[17] Eduardo Mustafa. *JavaScript - 20 Anos de História e Construção da Web*. URL: https://imasters.com.br/front-end/javascript/javascript-20-anos-de-historia-e-construcao-da-web/?trace=1519021197.

[18] Mozilla Developer Network. *Closures*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.

[19] *O que é Node.js?* URL: http://nodebr.com/o-que-e-node-js/.

[20] W3C Org. *Document Object Model (DOM)*. URL: https://www.w3.org/DOM/.

[21] John K. Ousterhout. *Scripting: Higher Level Programming for the 21st Century*. URL: https://www.tcl.tk/doc/scripting.html.

[22] Diego Pinho. *O ECMAScript 6 e o Futuro do Javascript*. URL: https://imasters.com.br/front-end/javascript/o-ecmascript-6-e-o-futuro-do-javascript/?trace=1519021197&source=single.

[23] Roger S Pressman. *Software Engineering: a practitioner's approach*. 2010.

[24] *PYPL PopularitY of Programming Language*. URL: http://pypl.github.io/PYPL.html.

[25] Ben Rossi. *The Rise of Node.js and Why it Will Rule Enterprise Software Development for at Least a Decade*. URL: http://www.information-age.com/rise-nodejs-and-why-it-will-rule-enterprise-software-development-least-decade-123460405/.

[26] TechnologyWK. *The Document Object Model*. URL: www.technologyuk.net/internet/world-wide-web/document-object-model.shtml.

[27] Kennedy Tedesco. *Linguagens e Paradigmas de Programação*. URL: https://www.treinaweb.com.br/blog/linguagens-e-paradigmas-de-programacao/.

[28] UWD. *Evitando Callback Hell no Node.js*. URL: https://udgwebdev.com/evitando-callback-hell-no-node-js/.

[29] w3techs. *Usage of server-side programming languages for websites*. URL: https://w3techs.com/technologies/overview/programming_language/all.

[30] Alyona Zakurdaeva. *Top 10 Programming Languages in 2016/2017 by GitHub*. URL: https://yalantis.com/blog/top-10-programming-languages-in-2016-2017/.