Análise de Métricas Estáticas para Sistemas JavaScript

Miguel Esteban Ramos[1], **Marco Tulio Valente**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

{miguel.ramos, mtov}@dcc.ufmg.br

Abstract. *JavaScript is a dynamic and prototype-based programming language that is being increasingly used in software development. Thanks to initiatives like Node.js, JavaScript is not anymore just a client-side language used to make DOM manipulations but turned itself into a language with all the features to implement full systems on both client and server side. Nevertheless, it is still hard to find tools and resources to analyze and verify the quality of JavaScript systems. This paper presents the results of the collection and analysis of 15 software metrics for a set of 50 JavaScript systems.*

Resumo. *JavaScript é uma linguagem dinâmica e baseada em protótipos que cada dia é mais utilizada para o desenvolvimento de software. Graças a iniciativas como Node.js, JavaScript deixou de ser uma linguagem que só é utilizada no lado do cliente para fazer manipulações do DOM, para se tornar uma linguagem com todas as funcionalidades para implementar sistemas completos tanto do lado do cliente quanto do lado do servidor. Mesmo assim, ainda é difícil encontrar ferramentas e recursos que permitam analisar e verificar a qualidade de sistemas JavaScript. Neste trabalho, apresentam-se os resultados da coleta e análise de 15 métricas de software para um conjunto de 50 sistemas JavaScript.*

1. Introdução

JavaScript é uma linguagem de programação que atualmente está ganhando grande popularidade já que, graças à evolução que apresentou nos últimos anos e a melhorias no seu desempenho, tornou-se peça fundamental no desenvolvimento de aplicações web modernas, ricas em conteúdo, interativas e funcionais [Cantelon et al. 2014]. Além disso, a fim de levar todas as características de JavaScript para outros ambientes, surgiram plataformas como Node.js que permitiram expandir o escopo de JavaScript para escrever aplicações que fazem uso intensivo de dados do lado do servidor, contribuindo para popularidade crescente da linguagem.

Dado o grande número de sistemas atualmente desenvolvidos em JavaScript, é natural investigar técnicas e métodos para escrever código de qualidade nessa linguagem, fazendo uso de todos os princípios da Engenharia de Software. JavaScript possui um conjunto de características como sua natureza dinâmica e assíncrona, sua orientação a protótipos, sua flexibilidade, entre outras. Essas características geram novos desafios no desenvolvimento desses sistemas e fazem com que alguns conceitos tenham que ser reconsiderados ou adaptados. Finalmente, JavaScript é uma linguagem relativamente nova, sendo que ainda existem poucos estudos que procuram avaliar e aprimorar as práticas de Engenharia de Software adotadas em projetos nessa linguagem.

Particularmente, um dos recursos mais importantes em Engenharia de Software para a análise de sistemas são métricas de software. Métricas fornecem uma grande variedade de informações, incluindo aspectos de qualidade, complexidade, manutenibilidade, tamanho, etc. Logo, métricas podem ser muito úteis no estudo das tendências e do estado da arte de sistemas JavaScript.

Este trabalho analisa um conjunto de métricas calculadas para uma coleção de 50 sistemas JavaScript cujo código-fonte foi pesquisado no sistema de controle de versões GitHub. O objetivo principal é que a análise desse conjunto de dados permita-nos obter informações sobre a forma, o tamanho e a complexidade dos sistemas JavaScript, bem como analisar quais tipos de métricas estão disponíveis para esses sistemas.

O restante deste artigo está organizado conforme descrito a seguir. Na Seção 2, apresenta-se a metodologia utilizada para realizar a coleta de métricas, incluindo uma descrição da ferramenta utilizada. Na Seção 3, é apresentada a análise de resultados para três grupos de métricas. Na Seção 4, descreve-se brevemente alguns trabalhos relacionados. Na Seção 5, são apresentadas as conclusões finais do artigo.

2. Metodologia

As métricas consideradas neste trabalho foram obtidas para um conjunto de 50 sistemas JavaScript escolhidos manualmente desde o sistema de controle de versão GitHub. A ferramenta utilizada para efetuar as medidas foi *escomplex*[1]. Essa ferramenta foi escolhida depois de realizar uma busca das ferramentas atualmente disponíveis para esse propósito e perceber que *escomplex* era a melhor opção por oferecer o maior número de métricas e ser a base de medição utilizada por outras ferramentas semelhantes.

2.1. Escomplex

Escomplex é uma ferramenta cujo objetivo é calcular métricas de software relacionadas principalmente com a complexidade de sistemas JavaScript. *Escomplex* faz a medição para cada função de um arquivo JS e também oferece um conjunto de medidas para o arquivo inteiro. A fim de obter essas medidas, *escomplex* gera uma Árvore Sintática Abstrata (AST) do código por meio de *esprima* (conhecido *parser* JavaScript) e posteriormente caminha nessa árvore para calcular as medidas estaticamente. As métricas oferecidas por *escomplex* são as seguintes:

- **Número de módulos:** Cada arquivo JS do sistema é considerado um módulo.
- **Linhas de código:** Contagem das linhas de código físicas (número de linhas en um módulo ou função) e lógicas (número de declarações imperativas).
- **Número de parâmetros:** É o número de parâmetros obtido da assinatura de cada função. É uma medida estática e, por isso, não são levadas em consideração as funções que dependem do parâmetro *arguments*.
- **Complexidade ciclomática (CC):** É a contagem do número de caminhos linearmente independentes no grafo de fluxo de controle do programa. [McCabe 1976]
- **Densidade de complexidade ciclomática:** Proposta como uma modificação para a complexidade ciclomática, esta métrica expressa a CC como uma porcentagem do número de linhas de código lógicas [Gill and Kemerer 1991].

[1] `https://github.com/philbooth/escomplex`

- **Métricas de complexidade de Halstead:** Conjunto de métricas que são calculadas com base no número de operadores distintos, o número de operandos distintos, o número total de operadores e o número total de operandos em cada função. Estas métricas são utilizadas para avaliar a complexidade do sistema.
- **Índice de manutenção:** Métrica de complexidade medida numa escala logarítmica, calculada a partir das linhas lógicas de código, a complexidade ciclomática e o esforço Halstead [Kuipers and Visser 2007].

2.2. Coleta de Métricas

Para obter os dados de medição apresentados neste trabalho foi necessário baixar o código de 50 sistemas JavaScript e desenvolver um programa encarregado de percorrer cada um desses sistemas, realizar as medidas por meio da ferramenta *escomplex* e construir uma tabela onde foram registradas todas essas medidas.

GitHub, o sistema de controle de versões, foi usado para pesquisar os sistemas utilizados neste trabalho e para baixar o código-fonte de cada um deles. Esse procedimento foi feito manualmente já que ainda é difícil encontrar um padrão utilizado globalmente nos sistemas JavaScript para a organização do código e, portanto, existe um alto risco de incluir código indesejado se o procedimento for feito automaticamente. Além disso, escolheram-se sistemas com uma alta popularidade no GitHub (maior número de *stars*), um número representativo de *commits* (pelo menos 150) e que não são uma cópia (*fork*) de outros projetos existentes.

A Figura 1 descreve os passos que foram seguidos a fim de obter as medições para cada sistema.

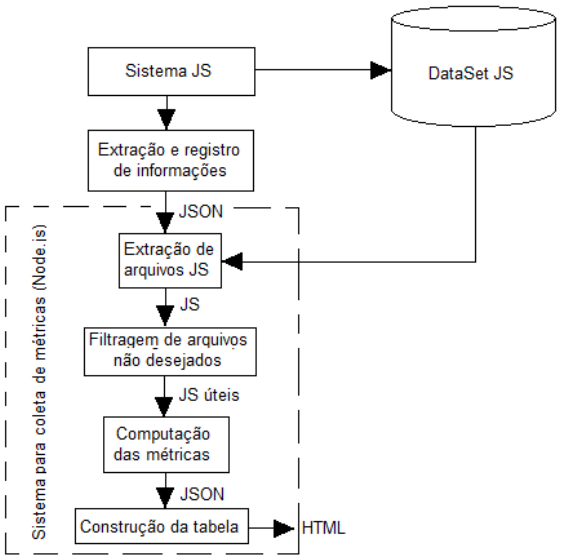

Figura 1. Procedimentos para coleta de métricas

O primeiro passo foi criar manualmente um arquivo JSON onde se encontra um vetor cujos elementos são objetos que representam cada sistema JS e armazenam informações como o nome, a versão e a localização da pasta onde fica o código principal do projeto. Esse arquivo é então usado como um parâmetro de entrada para o bloco en-

carregado do processamento de cada sistema JS e da geração da tabela com as medidas para cada um deles.

O programa desenvolvido a fim de explorar cada sistema e gerar o conjunto final de dados, foi escrito em JavaScript, utilizando *Node.js*. Inicialmente, o programa extrai recursivamente todos os arquivos *.js* que compõem o sistema descartando apenas os arquivos e as pastas que não atendem às condições de um filtro proposto para evitar o processamento dos arquivos que contêm código que não pertence ao núcleo dos sistemas. Nesse último passo são excluídos os arquivos de produção (*.min.js*), arquivos de direitos autorais (*copyright.js*), pastas de documentação (*doc*, *docs*), pastas de testes (*test*), pastas de exemplos (*example*, *examples*) e pastas que incluem dependências de terceiros (*thirdparty* ou *node_modules*).

Posteriormente, fazendo uso da ferramenta *escomplex*, foram realizadas as medidas para cada arquivo JavaScript que atendeu ao filtro e contaram-se o número de arquivos que geraram erros durante esse processo (esses erros geralmente são devido a problemas de sintaxe no arquivo que está sendo processado). Por fim, nos casos de algumas métricas, calcula-se uma medida para o sistema inteiro. Por exemplo, o número de linhas de código do sistema é calculado como a soma do número de linhas de código de cada módulo; por outro lado, a complexidade ciclomática do sistema é calculada como a média desta medida para cada módulo com seu respectivo desvio padrão.

Por fim, o último módulo se encarrega de percorrer o vetor onde se encontram todos os sistemas e da geração de uma tabela no formato HTML, onde são sumarizadas as medidas para cada sistema.

3. Resultados

A tabela com os resultados de todas as medições realizadas para cada sistema JS, além do seu nome e a versão utilizada para a medição, estão disponíveis no seguinte link: `http://aserg.labsoft.dcc.ufmg.br/qualitas.js`. A fim de analisar os resultados obtidos, usou-se o programa R.

3.1. Métricas de Tamanho

A Figura 2 permite visualizar a distribuição da medida do tamanho (linhas de código) dos sistemas JS medidos.

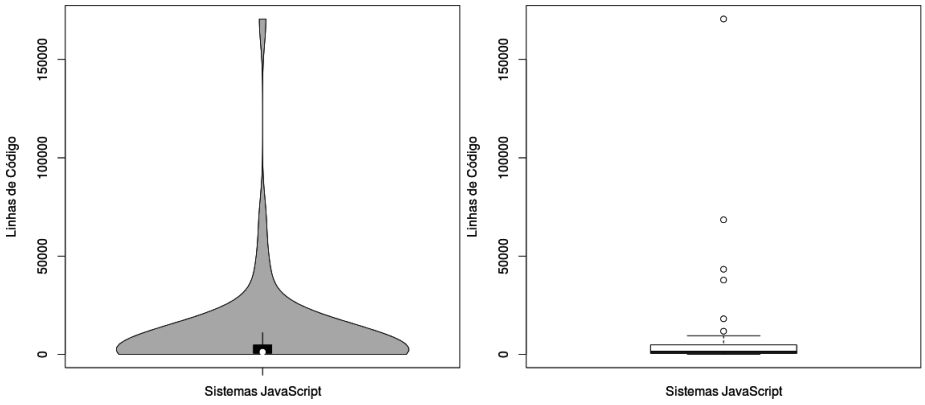

Figura 2. Linhas de cógido lógicas

Essa distribuição mostra que a maioria dos sistemas de JS considerados neste trabalho possui um tamanho relativamente pequeno. Depois de verificar os quartis calculados para esse conjunto de dados, pode-se concluir que 75% dos sistemas têm um número de linhas de código menor ou igual a 4,85 KLOC e que o tamanho de 50% das amostras não excede a 1,3 KLOC. No *boxplot* da Figura 2, também pode-se notar que algumas medidas têm valores muito discrepantes correspondentes a seis amostras com número de linhas de código que vão desde 11,82 KLOC até 170,58 KLOC.

Resumo dos Resultados: Embora a maior parte dos sistemas medidos neste trabalho tenha um tamanho pequeno, é possível afirmar que existem também sistemas JS cujo tamanho é comparável ao de sistemas desenvolvidos em outras linguagens de programação (Java, C + +, etc) e que, portanto, JavaScript está sendo usado para desenvolver programas que vão além da manipulação do DOM do lado do cliente para aspectos de apresentação.

3.2. Métricas de Organização Modular

Usando os dados coletados neste trabalho, é possível vislumbrar algumas tendências existentes na organização e separação do código dos sistemas JavaScript. Hoje em dia existem vários padrões que descrevem como o código escrito em JavaScript pode ser organizado. Entre eles, o padrão de módulos é um dos mais populares. Os gráficos da Figura 3 apresentam a distribuição do número de módulos e o número de funções por módulo dos sistemas considerados no presente trabalho.

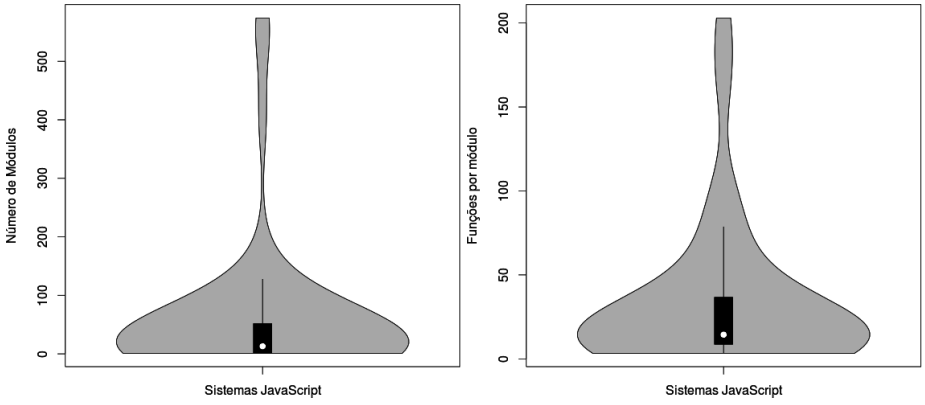

Figura 3. Módulos e funções por módulos

Um dos aspectos que chama atenção nos gráficos da Figura 3 é que o primeiro quartil da medida é igual a 1. Isso significa que pelo menos 25% dos sistemas são implementados em um único arquivo (módulo). No entanto, também é possível observar que metade dos sistemas apresentam mais de 13 módulos. Além disso, os três maiores sistemas possuem 403, 539 e 574 módulos.

Ao realizar uma análise isolada do número de linhas de código dos sistemas que foram implementados em um único módulo, encontrou-se que 75% desses sistemas têm menos de 0.93 KLOC e que nenhum deles possui mais de 1.47 KLOC. No que se refere àqueles sistemas com mais de 13 módulos, 75% desses sistemas têm mais de 2.28 KLOC. Após o cálculo do coeficiente de Spearman, cujo resultado foi de 0.84, determinou-se que existe uma correlação forte entre o número de módulos e o número de linhas de código de cada sistema.

Além disso, no que diz respeito ao número de funções por módulo, os resultados mostram que 50% dos sistemas têm uma média menor do que 14,4 funções por módulo. Por outro lado, 25% dos sistemas têm mais do que 36,6 funções por módulo, o que a princípio é um número alto. Após a realização de uma verificação manual na tabela de resultados, foi possível confirmar que a maioria dos sistemas incluídos nesse 25% corresponde a sistemas que foram implementados em um único módulo.

Resumo dos Resultados: Os resultados mostram que, quando se trata de sistemas pequenos, é muito provável que esses sejam implementados em um único arquivo JavaScript. No entanto, à medida que aumenta o tamanho dos sistemas, o código tende a ter uma melhor modularização.

3.3. Métricas de Complexidade

Os gráficos da Figura 4 mostram as distribuições das medidas de complexidade ciclomática e índice de manutenibilidade.

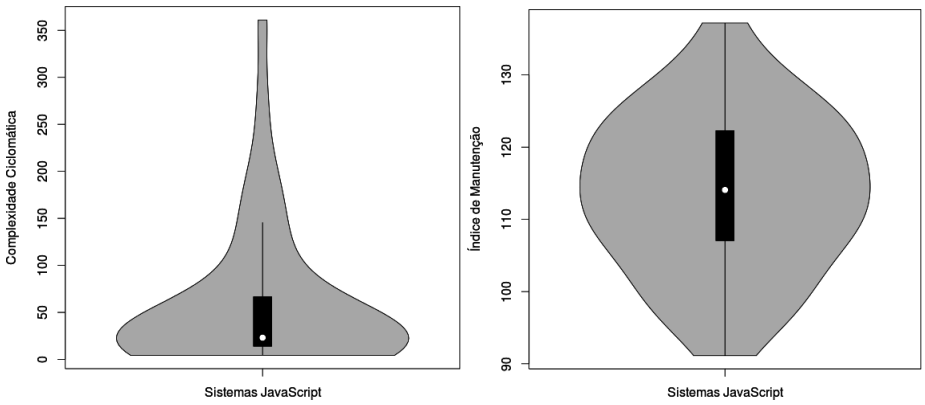

Figura 4. Complexidade ciclomática e Índice de manutenção

O terceiro quartil da medida de complexidade mostra que 75% dos sistemas considerados neste trabalho apresentam uma complexidade ciclomática menor que 66,69. Levando em consideração que, segundo Alves et al. [Alves et al. 2010], apenas métodos com medidas de complexidade acima de 14 deveriam ser considerados de alto risco para sistemas Java, e que uma classe típica em Java possui cerca de 10 métodos (o que deixa esse limite em 140 por classe), podemos concluir que complexidades ciclomáticas médias menores que 66,69 não representam uma medida alta. Somente três dos sistemas avaliados apresentaram medidas maiores do que 200.

Por outro lado, no que se refere a medida de índice de manutenibilidade, todos os sistemas apresentam um valor maior que 90 para essa medida. Assim, considerando os limites apresentados em [Coleman et al. 1994], todos os sistemas medidos apresentaram uma alta manutenibilidade ($\geq$ 85). Isso não faz muito sentido já que índices de manutenibilidade mais baixos deveriam ter sido obtidos pelo menos para aqueles sistemas que apresentam uma altíssima complexidade ciclomática ou uma grande quantidade de funções por módulo. Uma crítica semelhante ao índice de manutenibilidade é feita em [Fard and Mesbah 2013], dado que não se encontrou uma relação entre a má qualidade de código (*code smells*) e o índice de manutenibilidade de cada sistema. Outra discussão sobre problemas com esta métrica é apresentada em [Kuipers and Visser 2007].

Resumo dos Resultados: Para a grande maioria dos sistemas JavaScript analisados, as medidas de complexidade ciclomática são adequadas, pelo menos quando contrastadas com aquelas normalmente aceitas em sistemas Java. Por outro lado, no caso do índice de manutenibilidade, os resultados são menos conclusivos, reforçando as críticas a essa métrica encontradas na literatura.

4. Trabalhos Relacionados

Atualmente, existe um pequeno grupo de ferramentas usadas para medições de software em sistemas JavaScript. A maioria dessas ferramentas, como *escomplex*, estão focadas na análise de complexidade. Plato[2] é uma das ferramentas mais populares, pois faz uso de complexityReport.js[3] e JShint[4] para calcular métricas e gerar um relatório completo via gráficos interativos, que permitem comparações entre as medidas de cada módulo.

JSNose [Fard and Mesbah 2013] apresenta uma técnica para detecção de *code smells* que combina análise estática e dinâmica, a fim de encontrar padrões no código que possam resultar em problemas de compreensão e manutenção. Adicionalmente, são propostas algumas métricas alternativas para sistemas JavaScript, mas apenas é mostrado o resultado de cinco medidas feitas para 11 sistemas, fazendo uso da ferramenta complexityReport.js. Finalmente, uma análise da correlação entre o número de *code smells* detectados e as medições efetuadas em todos os sistemas é realizada.

5. Conclusão

Estudos similares aos reportados neste artigo existem para outras linguagens, notadamente Java [Baxter et al. 2006] e [Terra et al. 2013]. No entanto, no melhor de nosso conhecimento, este artigo reporta o mais extenso estudo realizado até o momento com o objetivo de caracterizar o comportamento estático de sistemas JavaScript, uma das linguagens mais populares atualmente. Como uma segunda contribuição, temos a disponibilização pública de um *dataset* de sistemas nessa linguagem, inspirado em *datasets* largamente utilizados em estudos empíricos envolvendo linguagens estáticas, como Java [Terra et al. 2013] e [Tempero et al. 2010].

Após a análise dos dados foram encontrados os seguintes resultados principais: (a) apesar de existirem ainda pequenos sistemas JavaScript, já existem também sistemas cujo grande tamanho sugere que são programas que vão além de pequenos *scripts* utilizados para manipulação do DOM; (b) no que se refere a modularidade, encontrou-se que aqueles sistemas pequenos são implementados inteiramente em um só módulo (arquivo) e que, somente aqueles de maior tamanho apresentam uma melhor distribuição do código fazendo uso de um maior número de módulos; (c) com relação às medidas de complexidade ciclomática, mostrou-se que a grande maioria dos sistemas JavaScript possuem valores adequados para essa medida quando comparados com valores comumente aceitos e praticados em sistemas Java.

Trabalhos futuros podem incluir um número maior de sistemas, adição de mais métricas utilizando ferramentas de medição de software alternativas, incluindo métricas que considerem também o comportamento dinâmico de sistemas JavaScript. Pretendemos

[2] `https://www.npmjs.org/package/plato`
[3] `https://www.npmjs.org/package/complexity-report`
[4] `http://jshint.com/`

também investigar valores de referência (*thresholds*) para essas métricas, usando a técnica proposta por Oliveira et al. [Oliveira et al. 2014]. Por fim, pretendemos trabalhar em uma versão histórica do dataset proposto, com dados de evolução dos sistemas JavaScript, a exemplo do que ocorre com datasets já disponibilizados para Java [Couto et al. 2013].

Agradecimentos

Esta pesquisa foi apoiada pelo PEC-PG - CNPq e FAPEMIG

Referências

Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the shape of Java software. In *21th Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 397–412.

Cantelon, M., Holowaychuk, T., Rajlich, N., and Harter, M. (2014). *Node. js in Action*. Manning Publications.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Couto, C., Maffort, C., Garcia, R., and Valente, M. T. (2013). COMETS: A dataset for empirical research on software evolution using source code metrics and time series analysis. *ACM SIGSOFT Software Engineering Notes*, pages 1–3.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Gill, G. K. and Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288.

Kuipers, T. and Visser, J. (2007). Maintainability index revisited–position paper. In *Special Session on System Quality and Maintainability (SQM 2007) of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Oliveira, P., Valente, M. T., and Lima, F. (2014). Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 336–345.

Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, pages 1–4.