# OO.js: Object-Oriented Design Suggestions for JavaScript Programs

Mohayeminul Islam
University of Alberta
mohayemin@ualberta.ca

## Abstract

JavaScript, the most used programming language on the planet, is frequently criticized for its rather confusing syntax and behaviour. Recent improvements in the language have mitigated many of such problems. One such improvement is the convenient class syntax which was missing prior to the language's 2015 release. With the improved class construct available, many developers may want to migrate their non-object-oriented code to an object-oriented version. To help this process, I present OO.js, a tool that suggests a set of object-oriented design of a given JavaScript program. It generates a call graph of the input program and applies agglomerative clustering to it. The clusterings are then scored based on standard object-oriented design metrics. OO.js is validated against 10 sample programs and has performed well for the small ones.

***Keywords:*** JavaScript, Object-Oriented Design, Call Graph, Clustering

## 1 Introduction

JavaScript, also known as just JS, was originally developed as a language for web browsers and is now the de-facto standard in this platform [11]. With the rise of Node.js[1], JavaScript has become a popular choice for server-side and desktop applications as well. On top of that, JavaScript is also widely used for cross-platform mobile applications. Due to availability in a wide range of platforms, 8th in a row, JavaScript has maintained its position as the most used programming language in the 2020 Developer Survey [2] conducted by the popular programming Q&A site Stack Overflow[2].

Stack Overflow has conducted this developer survey every year since 2011 asking developers about technology and other related questions[3]. About 65 thousand people from 180 countries participated in the year 2020 survey. Despite being the most used language, 41% of JavaScript's current users do not want to continue using JavaScript the next year.

[1]Node.js is an open-source, cross-platform, back-end, JavaScript runtime environment that executes JavaScript code outside a web browser.
[2]As of 6 December 2020, Stack Overflow has 14 million users who asked 21 million questions, https://stackexchange.com/sites
[3]https://insights.stackoverflow.com/survey/

*CMPUT 500, Fall 2020, University of Alberta*
2020.

I tried to understand why developers may not want to continue using JavaScript. Table 1 shows the most popular general purpose programming language according to the 2020 survey. We can see that most of the languages in the list support OO programming, making OO the most used programming paradigm. JavaScript supports OO programming as well, although its objects are prototype-based as opposed to conventional class-based objects.

```javascript
// Animal constructor
function Animal(name, energy) {

    this.name = name
    this.energy = energy
}

// Member function of Animal
Animal.prototype.eat = function (amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
}

// Another member function of Animal
Animal.prototype.play = function (length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
}

Animal.prototype.talk = function () {
}

// Cat constructor
function Cat(name, energy) {
    // Equivalent to Java's super call
    Animal.call(this, name, energy)
}

// Cat inherits Animal
Cat.prototype = Object.create(Animal.prototype)

// Override talk
Cat.prototype.talk = function () {
    console.log('meow')
}

new Cat("kitty", 100).talk()
// prints "meow"
```

**Listing 1.** Example of ES5 class

Despite being very powerful, many consider JavaScript's language constructs rather confusing compared to other languages [11]. As shown in Listing 1, the original syntax for creating a class-like structure in JavaScript is very unconventional. Including inheritance adds additional complexity to the syntax. Silva, Ramos, Bergel and Anquetil conducted an empirical study in 2015 on 50 popular open-source JavaScript GitHub repositories[13]. They found that only 8% of the projects implement most of their data structures as classes. Another 2016 survey reports that only 47% of Node.js developers use some sort of custom objects in their applications[10].

```
class Animal {
    constructor(name, energy) {
        this.name = name
        this.energy = energy
    }
    eat(amount) {
        console.log(`${this.name} is eating.`)
        this.energy += amount
    }
    play(length) {
        console.log(`${this.name} is playing.`)
        this.energy -= length
    }
    talk() {
    }
}

class Cat extends Animal {
    constructor(name, energy) {
        super(name, energy)
    }
    talk() {
        console.log('meow')
    }
}

new Cat("kitty", 100).talk()
// prints "meow"
```

**Listing 2.** Example of ES6 class, supported since 2015

JavaScript started supporting convenient class syntax from ECMAScript (ES) version 6, released in the year 2015 [4]. Listing 2 shows a ES6 code that is equivalent of ES5 code of 1. Microsoft released TypeScript in 2012, which is a static typing super-set of JavaScript with the support of the class construct. TypeScript class has all features of ES6 classes and adds additional static typing features to that. With convenient class construct available in both ES6 and TypeScript, new JavaScript/TypeScript systems are likely to utilize OO features more extensively. What is important for my study is, developers may want to refactor [9] their existing JavaScript code to use OO features. At the least of leveraging OO features is building classes composing functions and data. An automated suggestion of possible class design of an existing JavaScript code can be helpful for developers.

In this paper, I present OO.js - a solution for suggesting potential OO designs of an existing JavaScript code. OO.js takes a JavaScript codebase as input, creates a call graph of the code, and applies agglomerative clustering to the call graph. Each level of the clustering is considered a potential OO class design. The system also evaluates the designs based on different OO metrics and assigns comparative ranks to the designs by combining all the metrics.

The tool is tested with 10 sample programs. The best designs identified the system are pretty good for small programs, however, results are not as good for larger programs. I have identified several scopes of improvements in the call graph generation, clustering and scoring steps keeping the overall design as it is. The open-source implementation of the technique is available on GitHub [5]. The repository contains code, input and results of the experiment, and the necessary documentation to reproduce the results.

## 2 Background

There are a few terminologies about which the readers need to be on the same page with the author to understand the paper clearly.

```
function A() {
    B()
    C()
    D()
}

function B() {
    D();
}

function C() {
    D();
}

function D() {
    G();
}

function e() {
    f();
    G();
}

function f() {
    G()
}

function G() {
}
```

**Listing 3.** A sample JavaScript Program

**Function vs Method.** The terms function and method are often used interchangeably, however, there is a subtle difference. A function is a piece of code that can be invoked by name. Whereas the method is a function that is associated with an object. It is therefore technically correct to call a

[4]ECMAScript is the specification of JavaScript, JavaScript versions are referenced using ES version

[5]OO.js GitHub repository: https://github.com/mohayemin/OO.js

method a function, but not the other way around. In this paper, we use the terms function and method as applicable.

**Call Graph.** The call graph is an intermediate representation of a program that shows the call information of the program. Figure 1 shows the call graph for the program in 3.

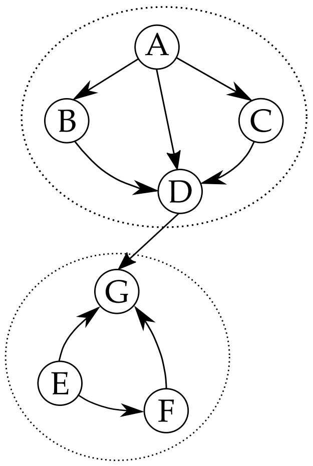

**Figure 1.** Call graph of the program in Listing 3. Functions within a dashed area are considered to be in the same class.

**Cluster and Clustering.** A *cluster* is a group of similar objects. *Clustering algorithms* are the algorithms to find clusters in a dataset. A set of clusters is called *clustering*. We use agglomerative clustering, a hierarchical or iterative clustering algorithm, in our technique which produces a clustering in each iteration.

**Object-Oriented Design Metrics.** There are several metrics to calculate the quality of an OO design. Two common and widely used metrics are the cohesion of class and the coupling between classes. The cohesion of class is a measure of how much relevant are the members of a class. On the other hand, the coupling between classes indicates the magnitude of interdependency between classes in a design. There are many ways to calculate the cohesion and the coupling available in the literature. Some of them are discussed in the related work section on page 6.

## 3 Methodology

OO.js takes a JavaScript code as input and passes it to a call graph generator. An aggromerative clustering module then generates a set of clusterings. Each of the clusterings are considered a potential OO design. These designs are then scored using cohesion and coupling metrics. Overview of the technique is shown in figure 2. The steps are ellaborated in the subsections below.

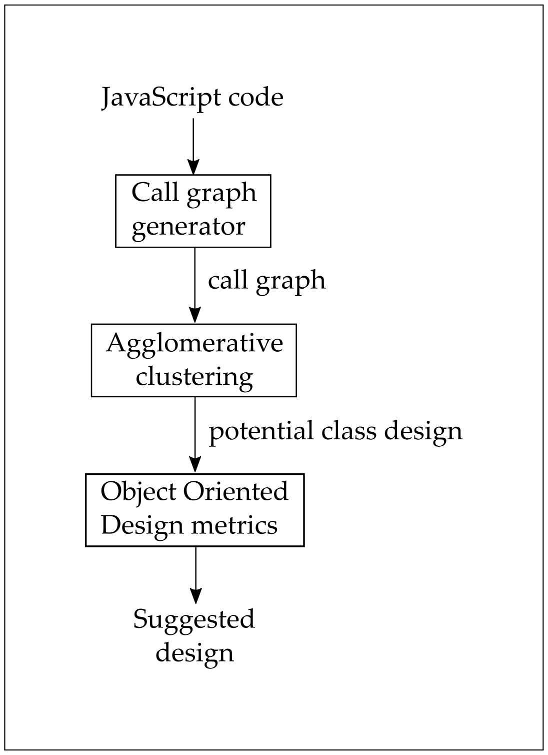

**Figure 2.** Methodology

### 3.1 Generating a call graph

Feldthaus, Schäfer, Sridharan, Dolby and Top developed a method of constructing an approximate call graph for JavaScript code [8]. OO.js uses js-callgraph, an open-source implementation of their technique [1]. js-callgraph can create a call graph from one or multiple files. It supports both Node.js CLI and in-program invocation. I have used the in-program version in OO.js code, and also used the CLI for experimentation.

js-callgraph has some limitations, two of which affects OO.js. First, the graph does not include data nodes. Therefore, a few OO metrics were not possible to calculate. Second, two functions with the same name in the same file cannot be distinguished. It was especially problematic for constructors and anonymous functions.

### 3.2 Clustering the call graph

I used agglomerative clustering on the call graph. Agglomerative clustering is a hierarchical clustering technique. In the beginning, it considers all the graph nodes to be in separate clusters. In each step, the closest cluster pair is merged into one cluster. The process goes on until all the nodes are merged into one cluster. OO.js considers the sets of clusters in each level of agglomerative clustering as a potential OO design, where each of the clusters is a class and the nodes in a cluster are the methods of the corresponding class.

| Language | Supports OO? | Usage | | Wants to continue | | Does not want to continue | |
|---|---|---|---|---|---|---|---|
| | | % | Rank | % | Rank | % | Rank |
| JavaScript | Yes | 67.70% | 1 | 58.30% | 10 | 41.70% | 10 |
| Python | Yes | 44.10% | 2 | 66.70% | 3 | 33.30% | 17 |
| Java | Yes | 40.20% | 3 | 44.10% | 13 | 55.90% | 7 |
| C# | Yes | 31.40% | 4 | 59.70% | 8 | 40.30% | 12 |
| PHP | Yes | 26.20% | 5 | 37.30% | 16 | 62.70% | 4 |
| TypeScript | Yes | 25.40% | 6 | 67.10% | 2 | 32.90% | 18 |
| C++ | Yes | 23.90% | 7 | 43.40% | 14 | 56.60% | 6 |
| C | No | 21.80% | 8 | 33.10% | 17 | 66.90% | 3 |
| Go | Yes | 8.80% | 9 | 62.30% | 5 | 37.70% | 15 |
| Kotlin | Yes | 7.80% | 10 | 62.90% | 4 | 37.10% | 16 |
| Ruby | Yes | 7.10% | 11 | 42.90% | 15 | 57.10% | 5 |
| Swift | Yes | 5.90% | 12 | 59.50% | 9 | 40.50% | 11 |
| Rust | No | 5.10% | 13 | 86.10% | 1 | 13.90% | 19 |
| Objective-C | Yes | 4.10% | 14 | 23.40% | 19 | 76.60% | 1 |
| Dart | Yes | 4.00% | 15 | 62.10% | 7 | 37.90% | 13 |
| Scala | Yes | 3.60% | 16 | 53.20% | 11 | 46.80% | 9 |
| Perl | Yes | 3.10% | 17 | 28.60% | 18 | 71.40% | 2 |
| Haskell | No | 2.10% | 18 | 51.70% | 12 | 48.30% | 8 |
| Julia | No | 0.90% | 19 | 62.20% | 6 | 37.80% | 14 |

**Table 1.** Most popular general purpose programming languages based on Stack Overflow Developer Survey 2020 [2]

The closeness between two clusters is determined by the Jaccard similarity of the neighbours of the two clusters. For two sets $A$ and $B$, Jaccard similarity is calculated as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

For a method (or node) $m$, the neighbourhood, $N(m)$, is the set of undirected neighbours of $m$, and $m$ itself:

$$N(m) = \{x : x \text{ calls } m\} \cup \{x : m \text{ calls } x\} \cup \{m\}, \quad \text{where } m \text{ is a method}$$

The neighbourhood of a class (or cluster) $c$, $N(c)$, is the union of neighbourhoods of the member methods.

Therefore closeness of two classes $c1$ and $c2$ is

$$C(c1, c2) = J(N(c1), N(c2))$$

## 3.3 Scoring class designs

Cohesion and coupling are used to calculate the score of a design. More specifically, a design with high cohesion of the classes and low coupling between the classes is considered good [6]. The cohesion and coupling values are normalized between 0 to 1 and the score of a design $d$, $S(d)$, is calculated by subtracting the coupling from the cohesion. The score, therefore, ranges between $-1$ and 1, lower value denoting poor design. OO.js does not try to make the score as a universal design quality metric. If the score of a potential design of one code is higher than the score of the potential design of another code, it does not necessarily mean that the design of the first code is better than the design of the second code. However, if the score of a potential design of a code is higher than the score of another potential design of the same code, OO.js consider the first design to be better than the second design.

There are many cohesion and coupling metrics available in the literature. The studied metrics are summarized in the related work section on page 6. Most of them rely on both method and data elements of a class. Unfortunately, the call graph generation technique OO.js uses can analyze only the functions/methods. As a result, I was limited to the metrics that depend only on methods.

**The cohesion metric.** OO.js uses Logical Relatedness of Methods (LORM) as the cohesion metric [7]. Etzkorn and Delugach developed this semantic cohesion metric which is calculated as

$$\text{LORM} = \frac{\text{total number of relations in the class}}{\text{total number of possible relations in a class}}$$

OO.js uses method calls as a notion of relations. That means the number of relations in the class is essentially the number of method pairs that have at least one call from one to another. Whereas the total number of possible relations in a class the total number of method pairs. Therefore, a class containing $n$ methods has a total of $n \times (n-1)/2$ possible relations.

The possible values of cohesion range between 0 and 1. Cohesion 0 means there are no calls between the methods. Cohesion 1 means there is at least one call between each method pair. A special case is a class with just one method for which cohesion is considered 1.

To evaluate the cohesion of a class design, OO.js uses a simple mean of the cohesions of the classes.

$$Coupling(d) = \sum_{i=1}^{n} \frac{c_i}{n}, \text{where } c_i \text{ is a class in design } d$$

**The coupling metric.** OO.js uses Coupling Between Object classes (CBO) defined by Chidamber and Kemerer [5] as coupling metric. CBO of a class is the number of other classes the class depends on. If a design has $n$ classes, the CBO of a class in that design can therefore be between 0 to $n - 1$.

The coupling of a design is the simple mean of coupling of the classes. Coupling measures are not in the range between 0 to 1, therefore, it needs to be normalized to be usable with cohesion. If the highest coupling of all designs are $H$ and the lowest is $L$, then normal coupling $N$ for a raw coupling $R$ is

$$N = \frac{(R - L)}{(H - L)}$$

This normalized coupling is considered the coupling of a design.

## 4 Experiment Setup and Results

The experiment is done on 10 JavaScript programs. The largest program has 36 functions. I could not find any benchmark dataset for the OO design of given methods. Therefore it was not possible to validate the results directly. However, 7 of the analyzed programs already use class signatures, therefore, I was able to validate them with themselves. For the other three programs, I created an expected class design myself and validated against it. Other than validating against an expected result, looking at the cohesion and coupling measures of the produced design also gives an idea of their quality. I rated the programs as very poor, poor, average, good and excellent depending on how well they match with the expected design. Very poor means the produced design has almost no similarity with the expected design, excellent on the other hand means that the produced design exactly matches the expected design.

The results are given in Table 3. Looking at the design metrics, it is clear that OO.js tries hard to create a decoupled program. 8 of the 10 suggested designs have 0 coupling measures. On the other hand, cohesion values are more diverse.

The two largest programs yield very poor results. Both the programs produce one big class with most of the methods in it, which is not in the case of the real code. Such designs typically have low coupling, however, for the code.js program, the coupling is considerably high. These two programs have their corresponding original design available. The other 8 programs have 10 or fewer methods in them. Only one of them is rated as poor, others are either Good or Excellent.

Overall, OO.js has performed well for small programs, but performance degraded considerably as the program size increased.

## 5 Limitations

I have found a few limitations in OO.js as described below.

**Missing data elements.** OO.js uses the js-callgraph library to generate the call graph. This library does not provide information about data elements. In the code in 4, *getValue* and *setValue* methods do not call each other, however, they are connected through the *value* variable. Our tool will show no relationship between the two methods. It will hamper both closeness and scoring calculations. If we considered the value variable in the call graph, Jaccard similarity between getValue and setValue would be calculated as follows:

$$\text{N(getValue)} = \{\text{getValue, value}\}$$

$$\text{N(setValue)} = \{\text{setValue, value}\}$$

$$\text{J(getValue, setValue)} = \frac{|\{\text{value}\}|}{|\{\text{getValue,setValue,value}\}|} = \frac{1}{3}$$

Unfortunately, OO.js will find no common elements in the neighbourhoods of getValue and setValue, therefore similarity will be 0.

```
let value = -1

function getValue() {
    return value
}

function setValue(newValue) {
    value = newValue
}
```

**Listing 4.** Code with program elements which are related, but OO.js cannot detect the relations

**Unresolved function name collision.** js-callgraph cannot differentiate methods with the same name in the same file. This is especially troublesome for constructors and anonymous methods which are named as "constructor" and "anon" respectively. The call information provided by the library does not include the line number of the caller function. OO.js could not solve this problem.

**Unexplored clustering algorithm.** To my knowledge, agglomerative clustering is the most appropriate clustering method for the scenario. However, other potential clustering algorithms are not thoroughly investigated, specially non-hierarchical ones. Other algorithms may potentially improve the results of OO.js.

**Unexplored OO design metrics.** LORM is used as the cohesion metric and calls between methods are used as a heuristic for conceptual relatedness between methods. This is a fairly simple heuristic and may not be the most appropriate one. I had not done extensive research on this research either.

| Program | Description | Validation Process |
|---|---|---|
| code.js | A custom program derived from OO.js original code | Matched with original code |
| json2Csv | A library to format a JSON as CSV | Matched with original code |
| JSON2CSVTransform.js | A part of json2csv library | Matched with original code |
| JSON2CSVBase.js | A part of json2csv library | Matched with original code |
| abcdefg-joint.js | A test program | Validated by author |
| abcdefg-disjoint.js | A test program | Validated by author |
| circular.js | A test where four methods call one another in a circular fation | Validated by author |
| utils.js | A part of json2csv library | Matched with original code |
| transforms | A part of json2csv library | Matched with original code |
| JSON2CSVParser.js | A part of json2csv library | Matched with original code |

**Table 2.** Analyzed Programs

| | Original program | | Result of best suggested design | | | | |
|---|---|---|---|---|---|---|---|
| Program | Program Size (no of methods) | Largest class size | Largest class size | Score* | Cohesion | Coupling | Quality |
| code.js | 35 | 10 | 31 | 0.25 | 0.82 | 0.57 | Very poor |
| json2Csv | 32 | 8 | 20 | 0.91 | 0.91 | 0.00 | Very poor |
| JSON2CSVTransform.js | 10 | 10 | 9 | 0.63 | 0.63 | 0.00 | Excellent |
| JSON2CSVBase.js | 9 | 9 | 3 | 1.00 | 1.00 | 0.00 | Poor |
| abcdefg-joint.js | 7 | 4 | 4 | 0.53 | 0.92 | 0.39 | Good |
| abcdefg-disjoint.js | 7 | 4 | 4 | 0.92 | 0.92 | 0.00 | Excellent |
| circular.js | 4 | 3 | 4 | 0.67 | 0.67 | 0.00 | Excellent |
| utils.js | 4 | 1 | 1 | 1.00 | 1.00 | 0.00 | Excellent |
| transforms | 4 | 1 | 1 | 1.00 | 1.00 | 0.00 | Good |
| JSON2CSVParser.js | 3 | 3 | 3 | 0.67 | 0.67 | 0.00 | Excellent |
| **Mean** | | | | **0.74** | **0.86** | **0.12** | |

**Table 3.** Experimental Results

* Score = (Cohesion - Coupling)

**Missing OO concepts.** OO.js only suggest what elements should be in a class, that means encapsulation. It cannot suggest anything about inheritance and dynamic polymorphism. Therefore, OO.js can only be used as a suggestive tool, but not a completely automated process to convert the input program.

### Directions for improvement

Based on the limitations discussed above, below are the scopes of improvements in OO.js.

- Replace js-callgraph with a more sophisticated call graph generator so that data elements are considered and the functions are more accurately identified. Possible options are discussed in the related work section on page 6.
- Study other clustering algorithms and replace agglomerative clustering with another more suitable method. Or at least cross-validate the top-scoring clusters generated by agglomerative clustering with other algorithms.
- Incorporate better cohesion and coupling algorithms. The algorithms used by OO.js are limited to the ones that depend only on functions. Using more advanced metrics may improve the results generated by OO.js.
- Score the designs with more OO metrics in addition to coupling and cohesion. The current implementation of OO.js can take an arbitrary number of OO design metrics to calculate the score of the designs. However, only LORM and CBO are implemented and used. Adding more OO metrics can potentially improve the quality of the scoring mechanism.

## 6 Related Work

The research related to OO.js are described in the subsections below.

### 6.1 Call graph generation for JavaScript

There are two primary ways to generate call graphs, static approach and dynamic approach.

**Static Approaches.** Since JavaScript is dynamically typed, it is not possible to precisely create a call graph for JavaScript code. However, several studies tried to generate approximate call graphs.

Static analysis is essential for sophisticated integrated development environments (IDE) features, however, static analysis is hindered due to the dynamic nature of JavaScript. Feldthaus, Schäfer, Sridharan, Dolby and Top identify call graph generation as the key problem to improve IDE services for JavaScript[8]. They developed a field-based flow analysis to construct an approximate call graph. Their approach is naturally unsound but highly accurate for large size real-world program. The original authors' implementation of the technique is not maintained, however, there is a third-party open-source implementation of their technique which OO.js uses.

Jensen, Møller and Thiemann developed a static analysis infrastructure that uses abstract interpretation to infer type information in JavaScript code[11]. The goal of the tool is not generating a call graph, however, the call graph is generated as a byproduct of the tool. Unfortunately, I was not able to find a suitable implementation of their technique.

code2flow[6] is a python tool that can generate call graphs for Python and JavaScript. The output is in DOT format, so it is possible to use the tool along with a DOT to JSON converted. However, the project is out of maintenance and does not support newer JavaScript syntaxes.

I discovered a few other possible JavaScript call graph generation techniques when my implementation already had some significant progress, therefore could not integrate. My implementation can be improved using one or more of these techniques. These tools are listed below.

The primary goal of Google Closure Compiler[7] is converting a developer-written JavaScript code into another JavaScript code that is the same in terms of functionality but improves in terms of compression and execution time. Closure compiler is also used as a linting tool for JavaScript. It does not directly produce a call graph. However, it can output the program information in various ways, from which it may be possible to extract a call graph.

T. J. Watson Libraries for Analysis (WALA)[8] by IBM is a popular infrastructure for static analysis of Java bytecode and JavaScript. The system is developed in Java, however, a JavaScript library is also available[9]. Unfortunately, the JavaScript library does not have a call graph generator.

The TypeScript[10] compiler has the potential to generate a call graph for JavaScript code . TypeScript compiler converts a TypeScript code into JavaScript. Since TypeScript is a strict superset of JavaScript, it can just handle JavaScript code.

[6]https://github.com/scottrogowski/code2flow
[7]https://developers.google.com/closure/compiler
[8]https://github.com/wala/WALA
[9]https://github.com/wala/JS_WALA
[10]https://www.typescriptlang.org/

**Dynamic Approaches.** Dynamic analysis can build more precise call graphs compared to static analysis. But that comes with the price of running the actual program for the analysis to be done. Dynamic call graph generation for JavaScript seems to be less researched compared to the static counterpart.

Toma and Islam[14] overrode JavaScript's function call mechanism to log function calls and generated call graphs. Instead of production code, they relied on automated test cases to execute the code. If the project has high test coverage, it will eventually invoke most of the public methods. However, the downside is, it is common to use mock or stub objects during testing. As such, the real call information will not be discovered. This approach requires a static call graph generator alongside for more accurate results.

## 6.2 Clustering algorithms

Cluster analysis is a well-researched field due to its wide range of applications. Clustering algorithms can be divided into several categories, hierarchical clustering being one of them. Hierarchical clustering is the technique where clusters are generated in a step-by-step manner. OO.js aims to *suggest* potential OO designs as opposed to finding the *correct* design. Hierarchical clustering gives a set of clusterings and each of the clusterings is considered a potential design. As this is possible only with hierarchical clustering approaches, this category of clustering was explored. There are two major approaches of hierarchical clustering: divisive and agglomerative [12].

The divisive approach starts with all elements in the same cluster. In each step, the largest cluster is split until all elements are on separate clusters. There are $2^n$ ways of splitting a cluster of size $n$, therefore it is impractical to use divisive clustering without a heuristic.

In agglomerative clustering, all elements are in separate clusters at the beginning. In each step, two closest clusters are merged into one cluster. The process continues until all clusters are merged in a single cluster.

## 6.3 OO Design Metrics

Extensive research is done for measuring the quality of OO designs. At a high level, cohesion and coupling are among most prominent metrics. Researchers have defined many implementations of both the high-level metrics.

Chidamber and Kemerer designed several metrics for OO design, including Coupling between object classes (CBO) [5]. CBO for a class is the number of other classes this class depends on.

Logical Relatedness of Methods (LORM) [7] calculates cohesion as the ratio of existing relations between methods in the class over the total number of possible relations in the class. This metric can be simplified by considering method call as the relation. Some metrics calculate the lack of cohesion instead of cohesion, which can be reversed to obtain cohesion. LCOM1 [4] calculates lack of cohesion as the number of method pairs that do not share attributes. TLCOM [3] is more accurate in the sense that it considers transitive use of attributes. For example, if method $m1$ calls method $m2$ and $m2$ uses an attribute $a$, TLCOM considers that $m1$ also uses attribute $a$.

## 7 Conclusion

JavaScript is the most used programming language on the planet, however, many of its users tend not to continue using the language in the future. Dispute being an Object-Oriented language, JavaScript introduced a convenient class syntax only recently. As a result, users may be more interested in using OO features of the language and possibly migrate their existing non-OO JavaScript code to utilize class constructs. To help automate the process, OO.js introduces a technique to suggest possible OO designs for a program.

OO.js works by constructing a call graph of a program, which is then passed to an agglomerative clustering module. Each of the clustering levels of the hierarchical clustering is considered a potential class design. These potential class designs are then assigned scores according to cohesion and coupling metrics. The users can choose the best scoring design, or look into other potential candidates and choose from there. The open-source implementation of the system is available on GitHub.

OO.js has performed quite well for the small-sized programs, however, improvement is needed for the larger programs. My understanding is that the high-level design of the technique is alright, however, improvements are required in the detailed implementation.

## References

[1] [n.d.]. *Field-based Call Graph Construction for JavaScript.* https://github.com/Persper/js-callgraph

[2] Accessed 6 December, 2020. *Stack Overflow Developer Survey 2020.* https://insights.stackoverflow.com/survey/2020

[3] Jehad Al Dallal. 2011. Transitive-based object-oriented lack-of-cohesion metric. *Procedia computer science* 3 (2011), 1581–1587.

[4] Shyam R Chidamber and Chris F Kemerer. 1991. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications.* 197–211.

[5] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[6] Johann Eder, Gerti Kappel, and Michael Schrefl. 1994. *Coupling and cohesion in object-oriented systems.* Technical Report. Citeseer.

[7] Letha Etzkorn and Harry Delugach. 2000. Towards a semantic metrics suite for object-oriented design. In *Proceedings. 34th International Conference on Technology of Object-Oriented Languages and Systems-TOOLS 34.* IEEE, 71–80.

[8] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 752–761.

[9] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[10] Munawar Hafiz, Samir Hasan, Zachary King, and Allen Wirfs-Brock. 2016. Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving JavaScript features. *Journal of Systems and Software* 121 (2016), 191–208.

[11] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium.* Springer, 238–255.

[12] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis.* Vol. 344. John Wiley & Sons.

[13] Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. 2015. Does JavaScript software embrace classes?. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 73–82.

[14] Tajkia Rahman Toma and Md Shariful Islam. 2014. An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application. In *2014 International Conference on Informatics, Electronics & Vision (ICIEV).* IEEE, 1–6.