Identifying Classes in Legacy JavaScript Code

Leonardo Silva, Marco Valente, Alexandre Bergel, Nicolas Anquetil, Anne Etien

Identifying Classes in Legacy JavaScript Code

Leonardo Humberto Silva[1], Marco Tulio Valente[1], Alexandre Bergel[2], Nicolas Anquetil[3], Anne Etien[3]

[1]Department of Computer Science, UFMG, Brazil
[2]Department of Computer Science, DCC - Pleiad Lab, University of Chile
[3]RMod Project Team, INRIA Lille Nord Europe, France
{leonardosilva,mtov}@dcc.ufmg.br, abergel@dcc.uchile.cl, {nicolas.anquetil,anne.etien}@inria.fr

Abstract

JavaScript is the most popular programming language for the Web. Although the language is prototype-based, developers can emulate class-based abstractions in JavaScript to master the increasing complexity of their applications. Identifying classes in legacy JavaScript code can support these developers at least in the following activities: (i) program comprehension; (ii) migration to the new JavaScript syntax that supports classes; and (iii) implementation of supporting tools, including IDEs with class-based views and reverse engineering tools. In this paper, we propose a strategy to detect class-based abstractions in the source code of legacy JavaScript systems. We report on a large and in-depth study to understand how class emulation is employed, using a dataset of 918 JavaScript applications available on GitHub. We found that almost 70% of the JavaScript systems we study make some usage of classes. We also performed a field study with the main developers of 60 popular JavaScript systems in order to validate our findings. The overall results range from 97% to 100% for precision, from 70% to 89% for recall, and from 82% to 94% for F-score.

1 Introduction

JavaScript is the most popular programming language for the Web. The language was initially designed in the mid-1990s to extend web pages with small executable code. Since then, its popularity and relevance only grew [1, 2, 3]. JavaScript is now the most popular language on GitHub, including newly created repositories. Richards et al. [4] also reported that the language is used by 97 out of the web's 100 most popular sites. Concomitantly with its increasing popularity, the size and complexity of JavaScript software is in steady growth. The language is now used to implement mail clients, office applications, and IDEs, which can reach hundreds of thousands of lines of code.[1]

[1]http://sohommajumder.wordpress.com/2013/06/05/gmail-has-biggest-collection-of-javascript-code-lines-in-the-world

JavaScript is an imperative and object-oriented language centered on prototypes, rather than a class-based language [5, 6, 7]. Recently, the new standard version of the language, named ECMAScript 6 (ES6), included syntactical support for classes [8]. In this new version, it is possible to implement classes using a syntax very similar to the one provided by mainstream class-based object-oriented languages, like Java and C++. However, there is a large codebase of legacy JavaScript source code, i.e., code implemented in versions prior to the ECMAScript 6 standard. To mention an example, GitHub has currently over three million active repositories whose main language is JavaScript[2], most of them implemented in ECMAScript 5 (ES5). In this legacy code, developers can emulate class-based abstractions, i.e., data structures including attributes, methods, constructors, inheritance, etc, using the prototype-based object system of the language, which is part of JavaScript since its first version.

In a previous paper, we presented a set of heuristics followed by an empirical study to analyze the prevalence of class-based structures in legacy JavaScript code [9]. This empirical study was conducted on 50 popular JavaScript systems, all implemented according to ES5. The results indicated that: (i) class-based constructs are present in 74% of the studied systems; (ii) there is no correlation between code size and the number of class-like structures; and (iii) emulating inheritance through prototype chaining is not common. In this paper, we extend this previous work as follows:

- We conduct a new study and increase our dataset from 50 to 918 systems. We use an external library called Linguist to allow the extraction of a large dataset from GitHub, ignoring binary or third-party files, and suppressing files generated automatically.
- We perform a field study with 60 professional JavaScript developers to evaluate the accuracy of the proposed strategy to detect class-like structures in legacy JavaScript code.
- We measure precision, recall, and F-score for the identification of classes, methods, and attributes. The overall results range from 97% to 100% for precision, from 70% to 89% for recall, and from 82% to 94% for F-score.
- We investigate if JavaScript developers intend to use the new support for classes that comes with ES6.

The main objective of this work is to propose, implement, and evaluate a set of heuristics to identify class-based structures, and their dependencies, in legacy JavaScript code. Identifying classes in legacy JavaScript code is important for two major reasons. Firstly, it can support developers to migrate their legacy code to ES6, manually or by using tools that rely on the heuristics proposed in this paper. Secondly, it opens the possibility to implement a variety of analysis tools for legacy JavaScript code, including IDEs with class-based views, bad smells detection tools, reverse engineering tools, and techniques to detect violations and deviations in class-based architectures.

The main contributions of our work are as follows:

[2] http://githut.info/

- We document how prototypes are used in JavaScript to support the implementation of structures including both data and code and that are further used as a template for the creation of objects (Section 2). We use the term *classes* to refer to such structures, since they have a very similar purpose as the native classes from mainstream object-oriented languages.
- We propose a strategy to statically identify *classes* in JavaScript code (Section 3). We also propose an open source supporting tool, called JSClassFinder, that practitioners can use to detect and inspect *classes* in legacy JavaScript software.
- We provide a thorough study on the usage of *classes* in a dataset of 918 JavaScript systems available on GitHub (Section 4). This study aims to answer the following research questions: (RQ #1) Do developers emulate classes in legacy JavaScript applications? (RQ #2) Do developers emulate subclasses in legacy JavaScript applications? (RQ #3) Is there a relation between the size of a JavaScript application and the number of class-like structures? (RQ #4) What is the shape of the classes emulated in legacy JavaScript code? By "shape of a class" we mean how it is organized in terms of the number of attributes and methods.
- We report the results of a field study with 60 professional JavaScript developers (Section 5). We rely on these developers to validate our findings and our strategy to detect *classes*. This study aims to answer the following research questions: (RQ #5) How accurate is our strategy to detect *classes*? (RQ #6) Do developers intend to use the new support for *classes* that comes with ECMAScript 6?

The remainder of this paper is organized as follows. Section 2 provides a background on how *classes* are emulated in legacy JavaScript code using functions and prototypes. Section 3 introduces our strategy and tool to identify *classes* in JavaScript. Section 4 describes the research questions that guide this work, along with the dataset, metrics, and methodology used in our studies. We show and discuss answers to the proposed research questions in Section 5. We discuss the implications of our results and future research opportunities in Section 6. Threats to validity are exposed in Section 7 and related work is discussed in Section 8. We conclude by summarizing our findings in Section 9.

2 Classes in JavaScript

In this section, we discuss how classes can be emulated in legacy JavaScript code (Subsection 2.1). We also describe the syntax proposed in ECMAScript 6 to support classes (Subsection 2.2).

2.1 Class Emulation in Legacy JavaScript Code

This section describes the different mechanisms to emulate classes in legacy JavaScript. To identify these mechanisms we conducted an informal survey on documents available on the web, including

tutorials[3], blogs[4], and StackOverflow discussions[5]. We surveyed a catalogue of five encapsulation styles for JavaScript proposed by Gama *et al.* [10] and JavaScript books targeting language practitioners [11, 12]. We also interviewed the developer of a real JavaScript project to tune our tool and strategy. This developer is the leader of the open source project `select2`[6] (a customizable replacement for select boxes).

An *object* in JavaScript is a set of name-value pairs. Methods and variables are called *properties*, and their values can be any objects, including immediate values (*e.g.,* numbers, boolean) and functions. To implement *classes* in JavaScript, prior to ECMAScript 6 standard, the most common strategy is to use functions. Particularly, any function can be used as a template for the creation of objects. When a function is used as a class constructor, its `this` is bound to the new object being constructed. Variables linked to `this` are used to define properties that emulate attributes and methods. If a property is an inner function, then it represents a *method*, otherwise, it is an *attribute*. The operator `new` and the method `Object.create(...)` are usually used to instantiate *classes*.

To illustrate the definition of *classes* in legacy JavaScript code, we use a simple `Circle` *class*. Listing 1 presents the function that defines this *class* (lines 1-8), which includes two attributes (`radius` and `color`) and two methods (`getArea` and `setColor`). Functions used to define methods can be implemented inside the body of the *class* constructor, like `getArea` (lines 4-6), or outside, like `setColor` (lines 9-11). An instance of the *class* `Circle` is created with the keyword `new` (line 13).

```
1  function Circle (radius, color) {  // function -> class
2    this.radius = radius;    // property -> attribute
3    this.color = color;    // property -> attribute
4    this.getArea = function() { // function -> method
5      return (3.14 * this.radius * this.radius);
6    }
7    this.setColor = setColor; // function -> method
8  }
9  function setColor(c) {// function
10   this.color = c;  // property -> attribute
11 }
12 // Circle instance -> object
13 var myCircle = new Circle (10, 0x0000FF);  // 0x0000FF = Blue
```

Listing 1: *Class* declaration and object instantiation

Each object in JavaScript has an implicit `prototype` property that refers to another object. The instance link between an object and its class in mainstream object-oriented languages is assimilated to the prototype link between an object and its prototype in JavaScript. To evaluate an expression

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

[4] http://javascript.crockford.com/prototypal.html

[5] http://stackoverflow.com/questions/387707/whats-the-best-way-to-define-a-class-in-javascript

[6] https://select2.github.io/

like `obj.p`, the runtime system starts searching for property `p` in `obj`, then in `obj.prototype`, then in `obj.prototype.prototype`, and so on until it finds the desired property or the search fails. When an object is created using `new C` its `prototype` is set to the `prototype` of the function `C`, which by default is defined as pointing to `Object` (the global base object in JavaScript). Therefore, a chain of prototype links usually ends at `Object`.

By manipulating the `prototype` property, we can define methods whose implementations are shared by all object instances. It is also possible to define attributes shared by all objects of a given *class*, akin to static attributes in class-based languages. In Listing 2, `Circle` includes a `pi` *static attribute* (line 2) and a `getCircumference` method (lines 5-7). It is worth noting that `getCircumference` is not attached to the *class* (as a `static` method in Java). It has for example access to the object `this`, whose value is not determined using lexical scoping rules, but instead using the caller object.

```
1 // prototype property -> static attribute
2 Circle.prototype.pi = 3.14;
3
4 // function -> method
5 Circle.prototype.getCircumference= function () {
6   return (2 * this.pi * this.radius);
7 }
```

Listing 2: Using `prototype` to define methods and static attributes

Prototypes are also used to introduce inheritance hierarchies [13, 14]. In JavaScript, we can consider that a *class* `C2` is a *subclass* of `C1` if `C2`'s prototype refers to `C1`'s prototype or to an instance of `C1`. For example, Listing 3 shows a *class* `Circle2D` that extends `Circle` with its position in a Cartesian plane.

```
1 function Circle2D (x, y) {  // class Circle2D
2   this.x = x;
3   this.y = y;
4 }
5
6 // Circle2D is a subclass of Circle
7 Circle2D.prototype = new Circle(10, 0x0000FF);
8
9 // Circle2D extends Circle with new methods
10 Circle2D.prototype.getX = function () {
11   return (this.x);
12 }
13 Circle2D.prototype.getY = function () {
14   return (this.y);
15 }
```

Listing 3: Implementing *subclasses*

Alternatively, the *subclass* may refer directly to the prototype of the *superclass*, which is possible using the `Object.create()` method. This method creates a new object with the specified prototype object, as illustrated by the following code:

```
Circle2D.prototype = Object.create(Circle.prototype)
```

Table 1 summarizes the mechanisms presented in this section to map class-based object-oriented abstractions to JavaScript abstractions.

Table 1: Class-based languages vs JavaScript

Class-based languages	JavaScript
Class	Function
Attribute	Field property
Method	Inner function property
Static attribute	Prototype property
Inheritance	Prototype chaining

2.2 ECMAScript 6 Classes

ECMAScript is the standard definition of JavaScript [5]. ECMAScript 6 (ES6) [8] is the latest version of this standard, which was released in 2015[7]. Interestingly, a syntactical support to classes is included in this last release. For example, ES6 supports the following class definition:

```
class Circle {
  constructor (radius) {
    this.radius = x;
  }
  getArea() {
    return (3.14 * this.radius * this.radius);
  }
}
```

However, this support to classes does not impact the semantics of the language, which remains prototype-based. For example, the previous class is equivalent to the following code:

```
function Circle (radius) {
  this.radius = radius;
}
Circle.prototype.getArea = function () {
  return (3.14 * this.radius * this.radius);
}
```

The emulation strategies discussed in the previous section straightforwardly detects this code as a `Circle` class, with a `radius` attribute and a `getArea` method. Therefore, identifying class-

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla

like structures in legacy JavaScript code can, for example, motivate developers to migrate such structures to syntax-based classes, according to the ES6 standard.

3 Detecting Classes in Legacy JavaScript

In this section, we describe our strategy to statically detect *classes* in legacy JavaScript source code (Subsection 3.1). Subsection 3.2 describes the tool we implemented for this purpose. We also report limitations of this strategy, mainly due to the dynamic nature of JavaScript (Subsection 3.3).

3.1 Strategy to Detect Classes

To detect *classes*, we reuse with minimal adaptations a simple grammar, originally proposed by Anderson *et al.* [15] to represent how objects are created in JavaScript and how objects acquire fields and methods. This grammar is as follows:

Program	::=	*FuncDecl**
FunDecl	::=	`function` *Id*`()` `{` *Exp* `}`
Exp	::=	`new` *Id*`();` |
		`Object.create(`*Id*`.prototype);` |
		`this.`*Id* `=` *Exp*`;` |
		`this.`*Id* `=` `function` `{` *Exp* `}` |
		Id`.prototype.`*Id* `=` *Exp*`;` |
		Id`.prototype.`*Id* `=` `function` `{` *Exp* `}` |
		Id`.prototype` `=` `new` *Id*`();` |
		Id`.prototype` `=` `Object.create(`*Id*`.prototype);`

This grammar assumes that a program is composed of functions, and that a function's body is an expression. The expressions of interest are the ones that create objects and add properties to functions via `this` or `prototype`.

Definition #1: A *class* is a tuple $(C, \mathcal{A}, \mathcal{M})$, where C is the *class* name, $\mathcal{A} = \{a_1, a_2, \ldots, a_p\}$ are the attributes defined by the *class*, and $\mathcal{M} = \{m_1, m_2, \ldots, m_q\}$ are the methods. Moreover, a *class* $(C, \mathcal{A}, \mathcal{M})$, defined in a JavaScript program P, must respect the following conditions:

- P must have a function with name C.
- For each attribute $a \in \mathcal{A}$, the *class* constructor or one of its methods must include an assignment `this.a` = `Exp` or P must include an assignment `C.prototype.a` = `Exp`.
- For each method $m \in \mathcal{M}$, function C must include an assignment `this.m` = `function` **{**`Exp`**}** or P must include an assignment `C.prototype.m` = `function` **{**`Exp`**}**.

However, when functions matching *Definition #1* are implemented in the same lexical scope, as functions `Circle` and `setColor` in Listing 1, we must distinguish those that are *class* constructors from those that are methods. To achieve that, we do not consider as a *class* constructor a function that: (i) has no inner functions bound to `this`, (ii) does not participate in inheritance relationships defined using prototypes, and (iii) is never instantiated with neither `new` nor `Object.create`. In Listing 1, function `setColor` does not have inner functions bound to `this` nor inheritance relationships and it is never instantiated. Therefore, it is not considered a function constructor, but a method of *class* `Circle`.

Definition #2: Assuming that $(C1, \mathcal{A}1, \mathcal{M}1)$ and $(C2, \mathcal{A}2, \mathcal{M}2)$ are *classes* in a program P, we define that $C2$ is a *subclass* of $C1$ if one of the following conditions holds:

- P includes an assignment `C2.prototype = new C1()`.
- P includes an assignment `C2.prototype = Object.create(C1.prototype)`.

3.2 Tool Support

We implemented a tool, called JSClassFinder [16], for identifying *classes* in legacy JavaScript programs. As illustrated in Figure 1, this tool works in two steps. In the first step, Esprima[8]—a widely used JavaScript Parser—is used to generate a full abstract syntax tree (AST), in JSON format. In the second step, the "Class Detector" module is responsible for identifying *classes* in the JavaScript AST and producing an object-oriented model of the source code.

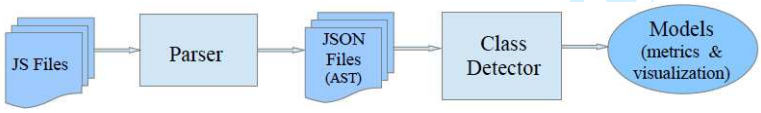


Figure 1: JSClassFinder's architecture

The models generated by JSClassFinder are integrated with Moose[9], which is a platform for software and data analysis [17]. This platform provides visualizations to interact with the tool and to "navigate" the application's model. All information about classes, methods, attributes, and inheritance relationships is available. Users can interact with a Moose model to access all visualization features and metric values. This model also allows the use of drill-down and drill-up operations when an entity is selected. The visualization options include UML class diagrams [18], distribution maps [19], and tree views.

[8] http://esprima.org
[9] http://www.moosetechnology.org/

It is possible for a user to customize the diagrams and to choose which elements to expose. For example, Figure 2 shows a distribution map for the system JADE. In this visualization, *classes* are represented by external rectangles, the small blue squares are methods, and the links between *classes* represent inheritance relationships. It is also possible to show a similar diagram where the external squares are JavaScript files and the internal squares are *classes*.

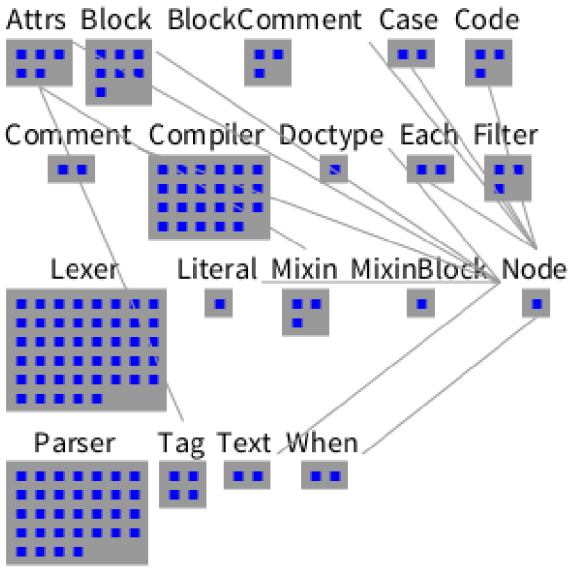


Figure 2: Example of distribution map for system JADE, generated by JSClassFinder

JSClassFinder also collects the following metrics: Number of Attributes (NOA), Number of Methods (NOM), Depth of Inheritance Tree (DIT), and Number of Children (NOC) [20].

3.3 Limitations

We acknowledge that there is not a single strategy to emulate classes in JavaScript. For example, it is possible to create "singleton" objects directly, without using any class-like constructions, as in Listing 4. Even though we do not consider such objects as classes, we chose to follow the definition presented in [21], in which Booch et al. state that classes and objects are tightly interwoven, but there are important differences between them ("a class is a set of objects that share a common structure, common behavior, and common semantics", "a single object is simply an instance of a class", page 93).

```
var myCircle = {
  radius: 10,
  pi: 3.14,
  getArea: function () { ... }
}
```

Listing 4: Example of "singleton" object

In addition, there are various JavaScript frameworks, like Prototype[10] and ClazzJS[11], that support their own style for implementing class-like abstractions. For this reason, we do not struggle to cover the whole spectrum of alternatives to implement *classes*. Instead, we consider only the strategy closest to the syntax and semantics of class-based languages and that ES6 code can be directly translated to (as discussed in Subsection 2.2).

Moreover, there are object-oriented abstractions that are more difficult to emulate in JavaScript, like abstract classes and interfaces. Encapsulation is another concept that does not have a straightforward mapping to JavaScript. A common workaround to simulate private members in JavaScript is by using local variables and closures. As shown in Listing 5, an inner function `f2` in JavaScript has access to the variables of its outer function `f1`, even after `f1` returns. Therefore, local variables declared in `f1` can be viewed as private, because they can only be accessed by the "private function" `f2`. However, we do not classify `f2` as a private method, mainly because it cannot be accessed from the object `this`, nor can it be directly called from the public methods associated to the prototype of `f1`.

```
1 function f1 () {      // outer function
2   var x;                 // local variable
3   function f2 () {       // inner function
4     // can access "x"
5     // cannot be called outside "f1"
6   }
7 }
```

Listing 5: Using closures to implement "private" inner functions

In JavaScript, it is possible to remove properties from objects dynamically, *e.g.*, by calling `delete myCircle.radius`. Therefore at runtime, an object can have less attributes than the ones initially defined. It is also possible to modify the prototype chains dynamically, which would mean modifying the "inheritance" links. Finally, the behavior of a program can also be dynamically modified using the `eval` operator [22, 23]. However, we do not consider the impact of `eval`'s in the strategy described in Subsection 3.1. For example, we do not account for *classes* entirely or partially created by means of `eval`.

Still due to the dynamic nature of JavaScript, if a *class* has a property that receives the return of a function call, this property is classified as an attribute, even if this call returns another function. Listing 6 shows an example in which the property `this.x` (line 6) is classified as an attribute, instead of a method, because the language is loosely typed and we do not evaluate the results of function calls.

[10] http://prototypejs.org
[11] https://github.com/alexpods/ClazzJS

```
function getF () {
  // getF() returns another function
  return function () {...};
}
function f1 () {    // class constructor
  this.x = getF();  // property x
  ...
}
```

Listing 6: Property that receives a function as the return of a function call

4 Evaluation Design

In this section, we describe the methodology we use to evaluate and to validate the strategy proposed to detect classes in legacy JavaScript code. We first present the questions that motivate our research (Subsection 4.1). Next, we describe the process we follow to select JavaScript repositories on GitHub and to carry out the necessary clean up of the downloaded code (Subsection 4.2). The metrics we use in our evaluation are described in Subsection 4.3. Finally, we report the design of a field study with JavaScript developers in Subsection 4.4.

4.1 Research Questions

Our main goal is to evaluate the strategy we propose to detect class-like abstractions in legacy JavaScript software. To achieve this goal, we pose the following research questions:

- RQ #1: Do developers emulate classes in legacy JavaScript applications?
- RQ #2: Do developers emulate subclasses in legacy JavaScript applications?
- RQ #3: Is there a relation between the size of a JavaScript application and the number of class-like structures?
- RQ #4: What is the shape of the classes emulated in legacy JavaScript code?
- RQ #5: How accurate is our strategy to detect classes?
- RQ #6: Do developers intend to use the new support for classes that comes with ECMAScript6?

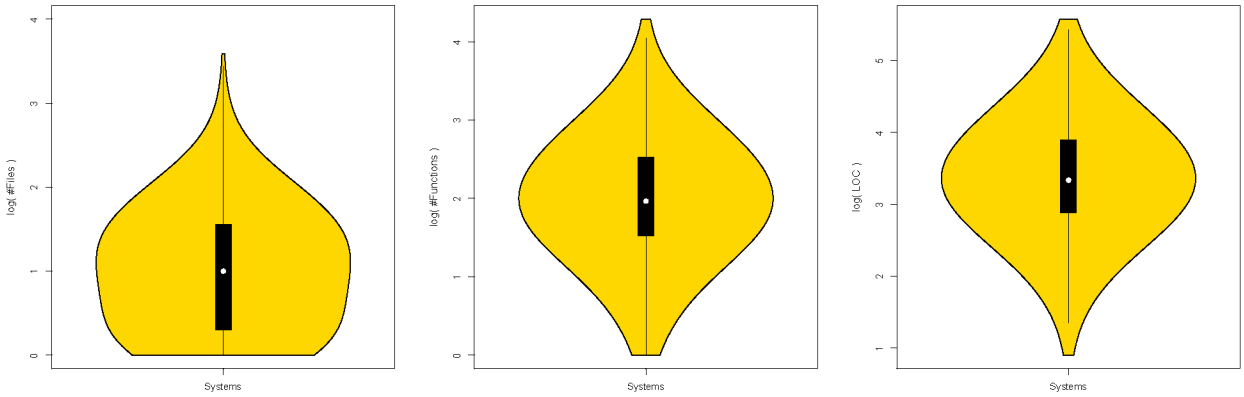
With RQ #1, we check if the emulation of classes is a common practice in legacy JavaScript applications. RQ #2 checks the usage of prototype-based inheritance. With RQ #3, we verify if the number of JavaScript *classes* in a system is related to its size, measured in lines of code. With RQ #4, we analyze the shape of JavaScript *classes* regarding the relation between the number of *attributes* and the number of *methods*. With RQ #5, we evaluate the accuracy of the proposed

approach to identify class-like structures. With RQ #6, we verify if developers intend to use the concrete syntax to define classes provided by ES6.

4.2 Dataset

Our dataset includes the last version of the top 1,000 JavaScript repositories on GitHub, according to the number of stars. This selection was performed in July, 2015. After cloning the repositories, we used an external library called Linguist[12] to clean up the source code files. Linguist is used by GitHub to ignore binary, third-party, and automatically generated files when computing statistics on the programming languages used by a repository. After running Linguist, we also performed a custom-made script to remove tests, examples, documentation, and configuration files. More specifically, this script removes the following files: `gulpfile.js`, `gruntfile.js`, `package.js`, `*thirdparty.js`, `*_test.js`, `*_tests.js`, `test.js`, `tests.js`, `license.js`; and the following folders: `test`, `tests`, `examples`, `example`, `build`, `dist`, `spec`, `demos`, `demo`, `minify`, `release`, `releases`, `docs`, `bin`, `test-*`, and `testing`.

After this clean up process, 82 systems were not exploitable because they did not contain any significant contributions, i.e., they remained with no source code files. Therefore, the final dataset was composed of 918 systems. Figure 3 shows the distribution of number of files, number of functions, and lines of code (LOC) in logarithm scale (base 10). The width of the "violin plot" indicates the number of systems for a given value. The largest system (`gaia`) has 375,615 LOC and 1,650 files with `.js` extension. The smallest system (`jswiki`) has 8 LOC and a single file. The average size is 8,778 LOC (standard deviation 21,801 LOC) and 41 files (standard deviation 163 files). The median (white dot at the center of the "violin") is 2,170 LOC and 10 files.



(a) # Files (b) # Functions (c) LOC

Figure 3: Dataset size distributions (log scales)

[12] https://github.com/github/linguist

4.3 Metrics

In the following we describe the metrics we use to answer the first four research questions proposed in Subsection 4.1.

4.3.1 Class Density (CD)

To measure the amount of source code related to the emulation of *classes* (as defined in Subsection 3.1) we use *Class Density (CD)*, which is defined as:

$$CD = \frac{\#\ function\ methods + \#\ classes}{\#\ functions}$$

This metric is the ratio of functions in a program that are related to the implementation of *classes*, i.e., that are methods or that are *classes* themselves. It ranges between 0 (system with no functions related to *classes*) to 1 (a fully class-oriented system, where all functions are used to support *classes*). The denominator includes all functions in a JavaScript program. We use the number of functions to implement methods (*function methods*) instead of the number of methods because, in JavaScript, it is possible to share the same function to implement multiple methods. Listing 7 shows an example found in the system SLICK, where a function body is shared by two methods. In this example, the `Slick` *class* provides two methods (`getCurrent` and `slickCurrentSlide`) that perform the same action when called. Therefore, the number of methods is equal to two, but the number of *function methods* is one.

```
1 Slick.prototype.getCurrent =
2   Slick.prototype.slickCurrentSlide = function() {
3     var _ = this;
4     return _.currentSlide;
5   };
```

Listing 7: Methods sharing the same body in system SLICK

We used CD to classify the systems in four main groups:

- *Class-free systems*: systems that do not use *classes* at all ($CD = 0$).
- *Class-aware systems*: systems that use *classes*, but marginally ($0 < CD \leq 0.25$).
- *Class-friendly systems*: systems with an important usage of *classes* ($0.25 < CD \leq 0.75$)
- *Class-oriented systems*: systems where most structures are *classes* ($CD > 0.75$).

4.3.2 Subclass Density (SCD)

To evaluate the usage of inheritance, we propose the metric *Subclass Density (SCD)*, defined as:

$$SCD = \frac{|\{C \in Classes \mid DIT(C) \geq 2\}|}{|Classes| - 1}$$

where *Classes* is the set of all *classes* in a given system and *DIT* is the *Depth of Inheritance Tree*. *Classes* with $DIT = 1$ only inherit from the common base *class* (`Object`). *SCD* ranges from 0 (system that does not make use of inheritance) to 1 (system where all *classes* inherit from another *class*, except the *class* that is the root of the *class* hierarchy). *SCD* is only defined for systems that have at least two *classes*.

4.3.3 Data-Oriented Class Ratio (DOCR)

In a preliminary analysis, we noticed many *classes* having more attributes than methods. This contrasts to the common shape of classes in class-based languages, when classes usually have more methods than attributes [24]. To better understand the members of JavaScript *classes*, we propose a metric called *Data-Oriented Class Ratio (DOCR)*, defined as follows:

$$DOCR = \frac{|\{C \in Classes \mid NOA(C) > NOM(C)\}|}{|Classes|}$$

where *Classes* is the set of all *classes* in a system. *DOCR* ranges from 0 (system where all *classes* have more methods than attributes or both measures are equal) to 1 (system where all *classes* are data-oriented *classes*, i.e., their number of attributes is greater than the number of methods). *DOCR* is only defined for systems that have at least one *class*.

4.4 Field Study Design

To validate our strategy for detecting *classes*, we perform a field study with the developers of 60 JavaScript applications, including 50 systems from our previous work [9], and 10 new systems. These systems have at least 1,000 stars on GitHub, 150 commits, and are not forks of other projects. After checking out each system, we cleaned up the source code to remove unnecessary files, as we did for the dataset described in Subsection 4.2.

The systems considered in the field study are presented in Table 2, including their version, a brief description, size (in lines of code), number of files, and number of functions. The selection includes well-known and widely used JavaScript systems, from different domains, covering frameworks (*e.g.*, ANGULAR.JS and JASMINE), editors (*e.g.*, BRACKETS), browser plug-ins (*e.g.*, PDF.JS), games (*e.g.*, 2048 and CLUMSY-BIRD), etc. The largest system (ACE) has 140,023 LOC and 594 files with `.js` extension. The smallest system (MASONRY) has 208 LOC and a single file. The average

size is 12,870 LOC (standard deviation 25,961 LOC) and 56 files (standard deviation 101 files). The median size is 3,363 LOC and 13 files.

This field study was conducted between March and June, 2015. For each system, we performed the following steps:

1. We downloaded the latest version on GitHub and cleaned up the source code;
2. We executed the parser (Esprima) to generate the AST;
3. We executed JSClassFinder to identify class-like structures and to build a class diagram;
4. We used the information available on GitHub to identify the main developers of each system. For systems supported by a team of developers the developer selected was the one with the highest number of commits in the previous three months. We then sent an email to the application's main developer with the class diagram attached and asked him to validate the detected *classes*. Figure 4 shows the class diagram sent to the developer of ALGORITHMS.JS. This diagram includes 14 *classes* representing common data structures, such as `Stack`, `LinkedList`, `Graph`, `HashTable`, etc.
5. We analyzed and categorized the developer's responses.

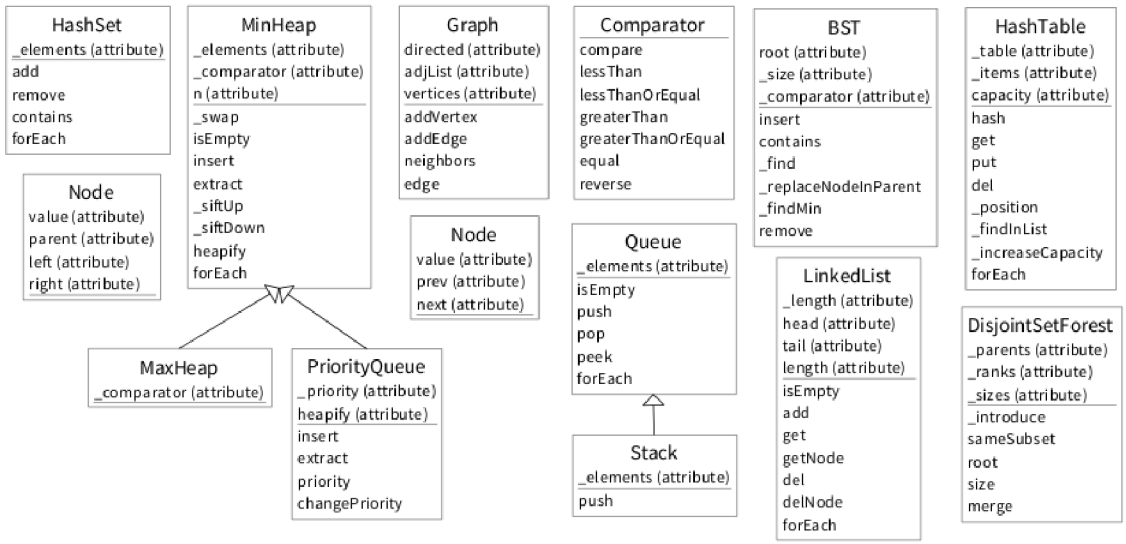


Figure 4: Class diagram for ALGORITHMS.JS, generated by JSClassFinder

Table 2: JavaScript systems (ordered by the CD column, see description in accompanying text). SCD can only be computed for systems with 2 or more classes. DOCR can only be computed for systems with at least one class.

System	Version	Description	LOC	#Files	#Func	#Class	#Meth	#Attr	CD	SCD	DOCR
masonry	3.2.3	Cascading grid layout library	208	1	10	0	0	0	0.00	-	-
randomColor	0.2.0	Color generator	373	1	16	0	0	0	0.00	-	-
respond	1.4.2	Polyfill for CSS3 queries	460	3	15	0	0	0	0.00	-	-
resume	-	Resume creator	460	1	19	0	0	0	0.00	-	-
clumsy-bird	-	Flappy Bird Game	672	7	36	0	0	0	0.00	-	-
impress.js	0.5.3	Presentation framework	769	1	24	0	0	0	0.00	-	-
jquery-pjax	1.9.3	Plugin to handle Ajax requests	913	1	33	0	0	0	0.00	-	-
async	1.1.0	Async utilities	1,114	1	100	0	0	0	0.00	-	-
modernizr	2.8.3	HTML5 and CSS3 detector	1,382	1	69	0	0	0	0.00	-	-
deck.js	1.1.0	Modern HTML Presentations	1,473	6	51	0	0	0	0.00	-	-
zepto.js	1.1.6	Minimalist jQuery API	2,497	17	233	0	0	0	0.00	-	-
photoSwipe	4.0.7	Image gallery	4,401	9	185	0	0	0	0.00	-	-
semantic-UI	1.12.3	UI component framework	18,369	23	1,191	0	0	0	0.00	-	-
jQueryFileUp	9.9.3	File upload widget	4,011	14	179	1	1	3	0.01	-	1.00
leaflet	0.7.3	Library for interactive maps	8,711	75	677	4	0	7	0.01	0.00	1.00
backbone	1.1.2	Data structure for web apps	1,681	2	115	1	1	0	0.02	-	0.00
chart.js	1.0.2	HTML5 charts library	3,463	6	189	2	2	5	0.02	0.00	0.50
turn.js	4.0.0	Page flip effect for HTML5	6,916	5	267	3	3	6	0.02	0.00	1.00
react	0.13.2	Library for building UI	16,654	143	608	7	8	17	0.02	0.00	0.57
meteor	1.1.0.2	Development platform	41,195	72	1,378	15	12	14	0.02	0.21	0.20
underscore	1.8.2	Functional helpers	1,531	1	123	1	5	1	0.03	-	0.00
jasmine	2.2.1	JavaScript testing framework	7,749	62	892	3	8	11	0.03	0.00	0.67
paper.js	0.9.22	Vector graphics framework	26,039	65	1,071	30	10	115	0.04	0.00	0.90
typeahead.js	0.10.5	Auto-complete library	2,576	19	233	11	1	72	0.05	0.00	1.00
d3	3.5.5	Visualization library	13,079	268	1,259	19	45	41	0.05	0.22	0.58
wysihtml5	0.3.0	Rich text editor	5,913	69	343	2	17	8	0.06	0.00	0.00
sails	0.11.0	MVC framework for Node	12,724	101	425	8	23	40	0.07	0.00	0.25
ionic	1.0.0.4	HTML5 mobile framework	19,322	103	492	8	26	21	0.07	0.29	0.50
jquery	2.1.4	jQuery JavaScript library	7,736	79	330	6	25	31	0.09	0.00	0.50
ghost	0.6.2	Blogging platform	15,290	142	659	15	47	44	0.09	0.00	0.27
timelineJS	2.35.6	Visualization chart	18,371	93	896	12	69	11	0.09	0.00	0.08
express	4.12.3	Minimalist framework	3,590	11	131	3	12	14	0.11	0.00	0.67
reveal.js	3.0.0	HTML presentation framework	5,811	16	242	5	22	18	0.11	0.00	0.40
video.js	4.12.5	HTML5 video library	9,823	46	586	6	63	17	0.11	0.00	0.50
three.js	0.0.71	JavaScript 3D library	39,449	202	1,266	99	48	544	0.12	0.00	0.92
numbers.js	-	Mathematics library for Node	2,965	10	132	2	16	4	0.14	0.00	0.00
polymer	0.5.5	Library for building web apps	11,849	1	763	22	103	68	0.16	0.00	0.41
grunt	0.4.5	JavaScript task runner	1,932	11	103	1	16	8	0.17	-	0.00
skrollr	0.6.29	Scrolling library	1,772	1	58	1	12	0	0.22	-	0.00
ace	1.1.9	Source code editor	140,023	594	4,337	291	673	785	0.22	0.01	0.46
mousetrap	1.5.3	Library for handling shortcuts	1,281	5	46	1	10	0	0.24	-	0.00
hammer.js	2.0.4	Handle multi-touch gestures	2,348	19	124	6	33	25	0.31	0.00	0.33
brackets	1.3.0	Source code editor	130,770	392	4,298	173	1,239	750	0.33	0.09	0.31
angular.js	1.4.0.1	Web application framework	49,220	191	981	61	276	171	0.34	0.03	0.21
intro.js	1.0.0	Templates for introductions	1,255	1	42	1	14	2	0.36	-	0.00
algorithms	0.8.1	Data structures & algorithms	3,263	58	165	14	59	32	0.44	0.23	0.21
pdf.js	1.1.1	Web PDF reader	57,359	88	2,277	181	895	795	0.47	0.11	0.44
bower	1.4.1	Package manager	8,464	60	304	15	143	97	0.51	0.00	0.40
mustache.js	2.0.0	Logic-less template syntax	594	1	33	3	15	7	0.55	0.00	0.33
less.js	2.3.1	CSS pre-processor	12,045	99	707	64	327	278	0.55	0.21	0.34
gulp	3.8.11	Streaming build system	99	3	5	1	2	6	0.60	-	1.00
fastclick	1.0.6	Library to remove click delays	841	1	23	1	16	10	0.74	-	0.00
pixiJS	3.0.2	Rendering engine	21,024	113	703	87	453	546	0.76	0.33	0.46
isomer	0.2.4	Isometric graphics library	770	7	47	7	31	27	0.81	0.00	0.57
2048	-	Number puzzle game	873	10	76	7	62	29	0.91	0.00	0.14
slick	1.5.2	Carousel visualization engine	2,300	1	81	1	86	0	0.93	-	0.00
floraJS	3.1.1	Simulation of natural systems	2,942	20	86	18	62	315	0.93	0.00	0.94
parallax	2.1.3	Motion detector for devices	1,007	3	57	2	56	75	0.95	0.00	1.00
jade	1.9.2	Template engine for Node	11,427	27	169	19	142	73	0.95	0.83	0.26
socket.io	1.3.5	Realtime app framework	1,297	4	57	4	58	46	1.00	0.00	0.00

In the mails to the developers, we asked two questions:

- Do you agree that the classes in the attached class diagram are correct?
- Do you intend to use the new support for classes that comes with ECMAScript 6? Why?

The developers had to answer the questions and point out their reasons. The first question aims to evaluate the accuracy of our approach to detect class-like structures (RQ #5). The second question aims to measure the interest in a concrete syntax to implement classes in JavaScript (RQ #6). In the cases where, after one month, an answer was not received, a gentle reminder was sent. For the systems where we did not find any *classes*, we also sent emails requesting the developers to confirm that they really do not emulate *classes* in their systems.

We sent 60 emails and received 33 answers, which represents a response ratio of 55%. Out of the 33 answers, 29 were obtained after a first round, and the other four after sending a gentle reminder.

We had three answers that could not be properly classified in our study. The first came from a developer who said he agreed with our findings but he was not totally sure. In the second case, the developer sent a web link which contains the API documentation of his application, and he recommended us to validate the *classes* ourselves. In the last case, the developer just stated that we should never use classes. Therefore, after discarding these cases, we have 30 valid answers.

Figure 5 shows the distribution of the valid answers per group of systems according to the class density (CD values). The distribution indicates that our field study includes systems in all four main groups: *class-free* (4 answers), *class-aware* (15 answers), *class-friendly* (7 answers) and *class-oriented* (4 answers).

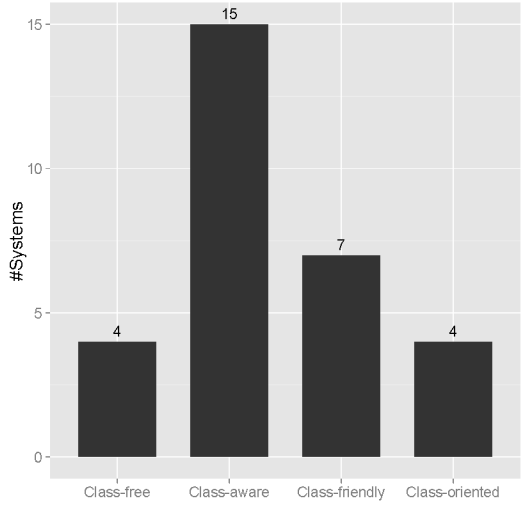


Figure 5: Number of answers, per group, from developers that agree with our findings

Finally, we use developers' answers to measure precision, recall, and F-score for the *classes*, methods, and attributes identified by our tool. These measures are calculated as follows:

$$Precision\ (P) = \frac{TP}{TP + FP}$$

$$Recall\ (R) = \frac{TP}{TP + FN}$$

$$F\text{-}score\ (F_1) = 2 \times \frac{P \times R}{P + R}$$

where *TP* represents the true positives, *FP* the false positives, and *FN* the false negatives. For *classes*, *TP* is the number of class-like structures correctly identified by our tool, *FP* is the number of class-like structures erroneously identified, and *FN* is the number of existing class-like structures that are not identified. F-score is the harmonic mean of precision and recall. For methods and attributes, the measures are defined in a similar way, but searching for method-like and attribute-like structures, respectively.

5 Results

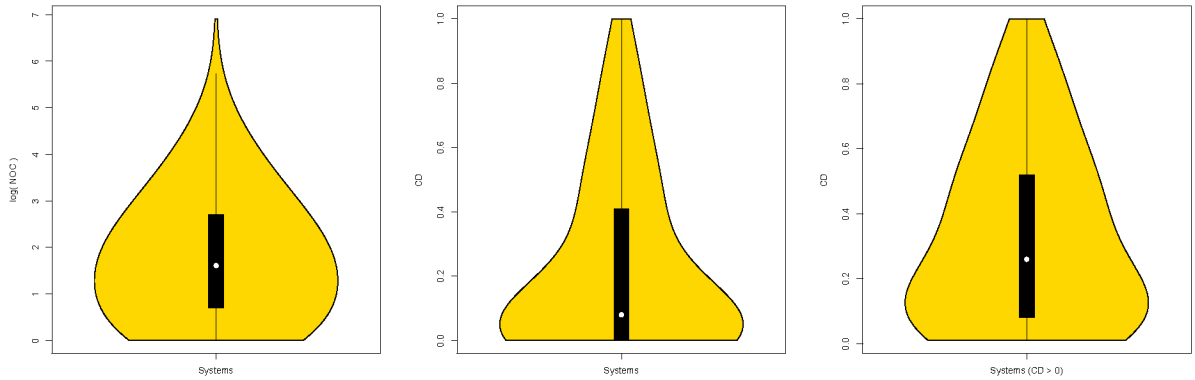
In this section, we present the answers to the six proposed research questions.

5.1 Do developers emulate classes in legacy JavaScript applications?

We found *classes* in 623 out of 918 systems (68%). The system with the largest number of *classes* is GAIA (1,001 *classes*), followed by NODEINSPECTOR (330 *classes*), and BABYLON.JS (294 *classes*). MATHJAX is the largest system (122,683 LOC) that does not have *classes*. Figure 6(a) shows the distribution of the number of *classes* for the systems that have at least one *class*. The first quartile is two (lower bound of the black box within the "violin") with 135 systems having only one *class*. The median is 5 and the third quartile is 15 (upper bound of the black box). Listing 8 shows an example of a *class* `Color`, detected in the system THREE.JS. We omit part of the code for the sake of readability.

```
1  THREE.Color = function ( color ) {  // Constructor
2    ...
3    return this.set( color )
4  };
5  THREE.Color.prototype = {
6    r: 1, g: 1, b: 1,  // Attributes
7    // Methods
8    setRGB: function ( r, g, b ) { ... },
9    ...
10 }
```

Listing 8: Example of class in THREE.JS



(a) # Classes (log scale) (b) CD (All Systems) (c) CD

Figure 6: Metric distributions. Results in (a) and (c) are reported only for systems with at least one class.

Figure 6(b) shows the distribution of the CD values. We found that 295 systems have CD equal to zero. In other words, 32% of the systems do not use *classes* at all or are using an abstraction other than the one we are looking for. The median is 0.08 and the third quartile is 0.41. We also found seven fully class-oriented systems (CD = 1). Table 3 shows the ten systems with the highest values of CD.

Table 3: Top-10 systems with highest CD values

System	CD	#Class	LOC
SKEUOCARD	1.00	8	1,685
RAINYDAY.JS	1.00	5	1,005
SIDE-COMMENTS	1.00	3	523
ZOOM.JS	1.00	2	229
STEADY.JS	1.00	1	215
TMI	1.00	1	203
LAYZR.JS	1.00	1	164
SOCKET.IO	0.97	4	1,350
CLNDR	0.97	1	1,197
SLAP	0.97	11	938

Figure 6(c) shows the CD distribution when we only consider the systems with CD greater than zero. The first quartile is 0.08, the median is 0.26, and the third quartile is 0.52. In other words, the emulation of *classes* represents on the median 26% of the functions, for the systems that include at least one *class*.

Figure 7 shows the number of systems in each of the four proposed groups (*class-free*, *class-aware*, *class-friendly*, and *class-oriented systems*) according to the CD values. The largest group is the *class-aware* (34%), in which systems use *classes* but they correspond to less than 25% of the

implemented functions. *Class-oriented* is the smallest group, in which the systems use more than 75% of their functions to emulate *classes*.

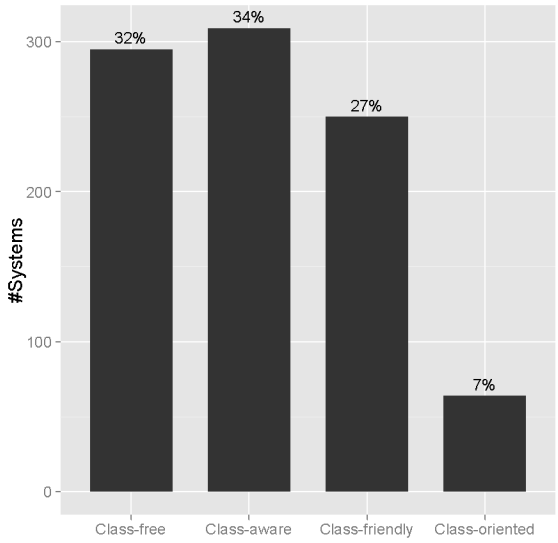


Figure 7: Class Density (CD) groups

Summary: We found classes in 623 out of 918 systems (68%). Therefore, 32% of the systems are class-free. Moreover, class-aware and class-friendly systems correspond to 34% and 27% of the systems; 7% of the systems are class-oriented systems.

5.2 Do developers emulate subclasses in legacy JavaScript applications?

As shown in Figure 8, the use of prototype-based inheritance is rare in JavaScript systems. First, we counted 499 systems (54%) having two or more *classes*, i.e., systems where it is possible to detect the use of inheritance. However, in 429 of such systems (86%), we did not find any *subclasses* ($SCD = 0$). The system with the highest use of inheritance is PROGRESSBAR.JS ($SCD = 0.8$). Figure 9 shows the class diagram for this system. As can be seen, the `Shape` *class* has four *subclasses*: `Circle`, `Line`, `SemiCircle`, and `Square`.

Summary: We found subclasses in only 70 out of 918 systems (8%).

5.3 Is there a relation between the size of a JavaScript application and the number of class-like structures?

Figure 10 shows scatterplots with size metrics on the x-axis in a logarithmic scale and CD on the y-axis. We also computed the Spearman's rank correlation coefficient between CD and the following size metrics: KLOC, number of files, and number of functions. The results are presented in Table 4.

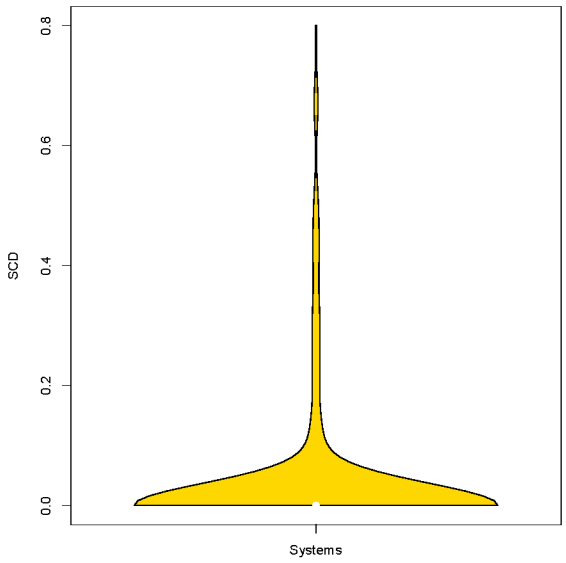


Figure 8: Subclass Density (SCD) distribution

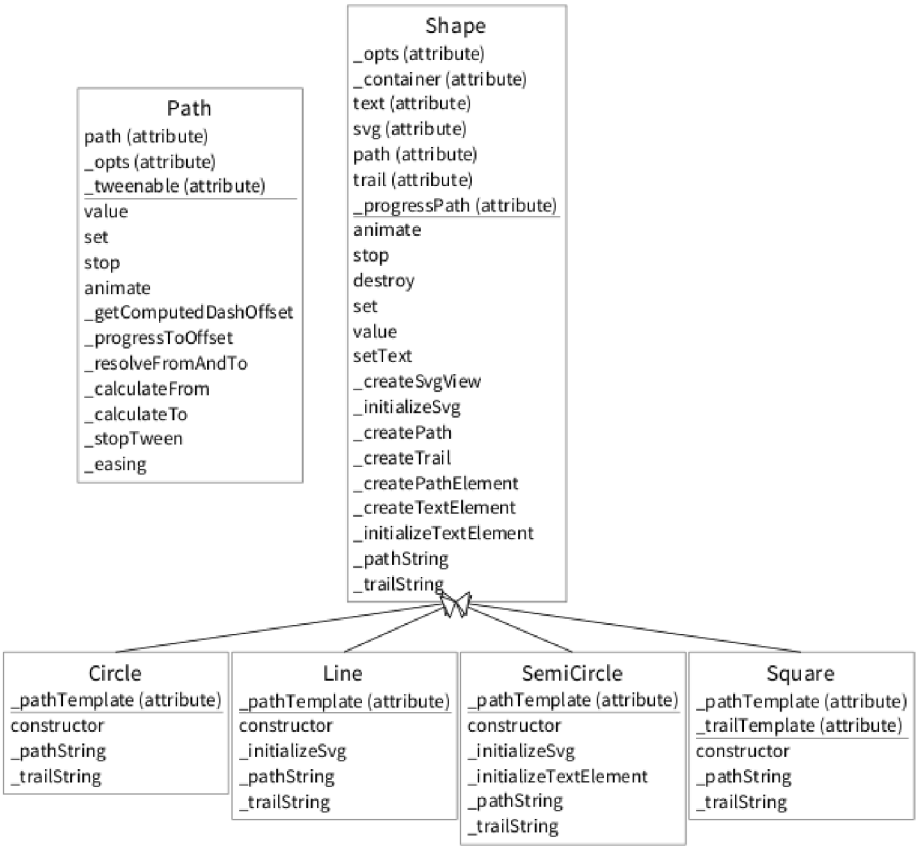
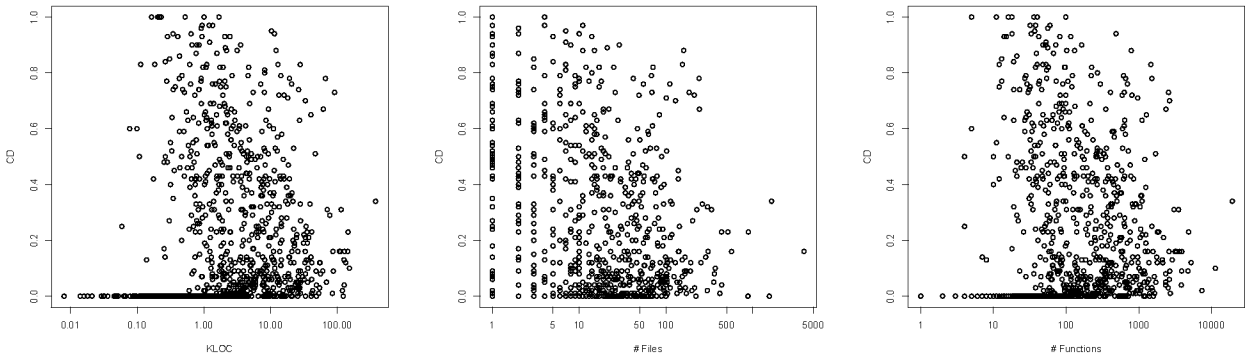


Figure 9: Inheritance in system PROGRESSBAR.JS

We found a weak correlation for KLOC (ρ=0.250), number of files (ρ=0.178), and for number of functions (ρ=0.289). For example, there are systems with similar sizes having both low and high class densities. ALOHA-EDITOR is an example of a system with a considerable size (69 KLOC) and low class density (CD = 0.05). By contrast, END-TO-END is also a large system (67 KLOC) but with a high class density (CD = 0.78).



(a) KLOC vs CD (b) # Files vs CD (c) # Func vs CD

Figure 10: Size metrics vs Class Density (CD)

Table 4: Correlation between CD and size metrics

	KLOC	# Files	# Func
Spearman	0.250	0.178	0.289
p-value	1.407e-14	6.216e-08	< 2.2e-16

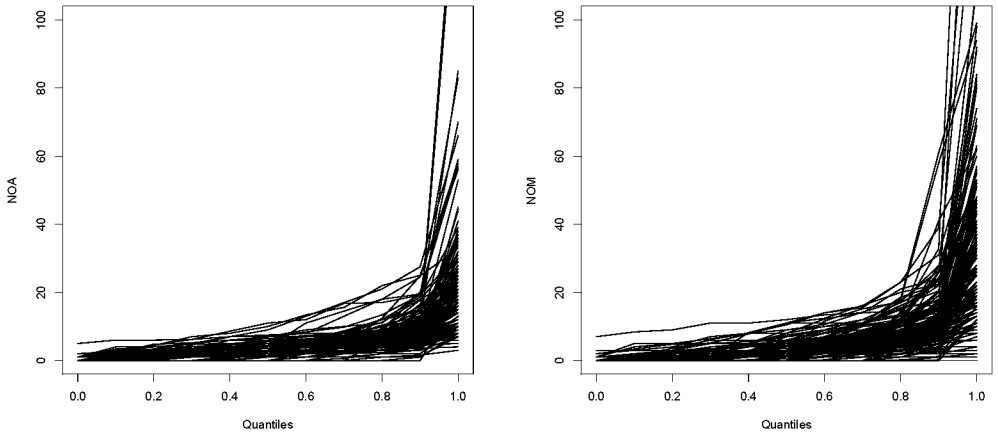
We also used the Kruskal-Wallis test to check if the LOC distributions in all four groups (*class-free*, *class-aware*, *class-friendly*, and *class-oriented systems*) are equal. The test resulted in a p-value < 2.2e-16, leading us to reject the null hypothesis (the groups have systems with equal size), at a 5% significance level. In fact, the median measures of each tested group are quite different (690; 5,667; 2,578; and 1,150; respectively).

Summary: We found no correlation between the size of a JavaScript application and the number of class-like structures.

5.4 What is the shape of the classes emulated in legacy JavaScript code?

To verify the shape of JavaScript *classes*, regarding the number of methods and attributes, we focus on systems that have the number of *classes* greater than or equal to 15 (which represents the 3rd quartile of this distribution). Figure 11 shows the quantile functions for the Number of Attributes (NOA) and Number of Methods (NOM) in such systems. The x-axis represents the quantiles and

the y-axis represents the metric values for the *classes* in a given quantile. For example, suppose the value of a quantile p (x-axis) is k (y-axis), for NOA. This means that p% of the *classes* in this system have at most k attributes. As can be observed, the curves representing the systems have a right-skewed (or heavy-tailed) behavior. In fact, this behavior is normally observed in source code metrics [25, 26, 27].



(a) NOA (b) NOM

Figure 11: Quantile functions

Regarding NOA, the quantile functions reveal that the vast majority of the *classes* have at most 28 attributes (90th percentile). Regarding NOM, the vast majority of the *classes* have less than 61 methods (90th percentile). To compare NOA and NOM measures, Figure 12 shows the *DOCR* distribution using a violin plot. The median DOCR value is 0.39, which is a high measure when compared to other languages. For example, metric thresholds for Java suggest that classes should have at most 8 attributes and 16 methods [28]. By contrast, half of the JavaScript systems we studied have more than 39% of their *classes* with more attributes than methods. We hypothesize that it is due to JavaScript developers placing less importance on encapsulation. For example, *getters* and *setters* are rare in JavaScript.

Summary: Classes in JavaScript have usually less than 28 attributes and 61 methods (90th percentile measures). It is also common to have data-oriented classes, i.e., classes with more attributes than methods. In half of the systems, we have at least 39% of such classes.

5.5 How accurate is our strategy to detect classes?

As described in Section 4, we measure accuracy using precision, recall, and F-score. Table 5 summarizes the results according to the developers' answers. The developers of 21 out of 30 systems (70%)

Figure 12: Data-Oriented Class Ratio (DOCR) distribution

fully agreed that the class diagrams we provided, based only on the code base of their systems, correctly model the *classes* they implemented. Therefore, precision, recall, and F-score for these systems are equal to 100%. The following two comments are examples of answers we received for such systems:

"Yes, everything looks like it actually is in the code base." (Developer of system LESS.JS*)*

"I do in fact agree with your findings on classes/methods/attributes. In building numbers.js I did have OOP in mind." (Developer of system NUMBERS.JS*)*

5.5.1 Precision

We achieve a precision of 100% in 28 out of 30 systems for *classes*; in all 30 systems for methods; and in 29 systems for attributes. In the following paragraphs we discuss the false positives we detected for *classes* and attributes.

False positives for classes. The developers of systems ACE and ANGULAR.JS pointed out that our strategy incorrectly identified some entities as *classes*. In both cases, the false positives are due to a limitation regarding JavaScript scoping rules. Listing 9 shows an example for ANGULAR.JS. In this example, we have a `MessageFormatParser` *class*, with a method `startStringAtMatch` (lines 4-6). Since there is also a function `match` in the global scope (line 1) our tool initially classifies `stringQuote` as a method (line 5). However, due to the scoping rules of JavaScript, this property is initialized with the formal parameter of `startStringAtMatch`, which is also named `match`. Moreover, `match` always receives a non-function value and therefore it should have been classified as an attribute. A similar issue happens in ACE.

Table 5: Precision, Recall, and F-Score results

Systems	Precision (%)			Recall (%)			F-Score (%)		
	Classes	Methods	Attributes	Classes	Methods	Attributes	Classes	Methods	Attributes
ACE	93	100	100	100	100	100	96	100	100
ALGORITHMS.JS	100	100	100	100	100	100	100	100	100
ANGULAR.JS	92	100	87	100	93	100	96	96	93
BOWER	100	100	100	100	100	100	100	100	100
CLUMSY-BIRD	100	100	100	0	0	0	0	0	0
D3	100	100	100	83	48	79	91	65	88
EXPRESS	100	100	100	60	36	56	75	53	72
INTRO.JS	100	100	100	100	100	100	100	100	100
JADE	100	100	100	100	100	100	100	100	100
JASMINE	100	100	100	7	5	24	13	10	39
JQUERY	100	100	100	100	100	100	100	100	100
JQUERYFILEUP	100	100	100	100	100	100	100	100	100
LEAFLET	100	100	100	9	0	4	17	0	8
LESS.JS	100	100	100	100	100	100	100	100	100
MASONRY	100	100	100	100	100	100	100	100	100
MODERNIZR	100	100	100	100	100	100	100	100	100
MOUSETRAP	100	100	100	100	100	100	100	100	100
MUSTACHE.JS	100	100	100	100	100	100	100	100	100
NUMBERS.JS	100	100	100	100	100	100	100	100	100
PAPER.JS	100	100	100	100	3	59	100	6	74
PDF.JS	100	100	100	100	100	100	100	100	100
PIXIJS	100	100	100	100	100	100	100	100	100
RANDOMCOLOR	100	100	100	100	100	100	100	100	100
SAILS	100	100	100	100	100	100	100	100	100
SKROLLR	100	100	100	100	100	100	100	100	100
SLICK	100	100	100	100	100	100	100	100	100
SOCKET.IO	100	100	100	100	100	100	100	100	100
THREE.JS	100	100	100	100	100	100	100	100	100
UNDERSCORE	100	100	100	100	100	100	100	100	100
VIDEO.JS	100	100	100	11	15	16	20	26	28
Mean	99.5	100	99.57	85.67	80	84.6	86.93	81.87	86.73

False positives for attributes. We have two situations in which methods are indeed identified as attributes in the system ANGULAR.JS. Listing 10 shows part of the implementation for the *class* `JQLite`. Our strategy correctly classifies the property `ready` (line 2) as a method, but it is not able to do the same with the property `splice` (line 3). The function `[].splice` is not recognized as a function because its implementation is not part of the source code of ANGULAR.JS (it is a JavaScript native function from `Array` object). Currently, our implementation does not recognize as methods functions that are initialized with JavaScript built-in functions.

Listing 11 shows another example of a property that is not identified as a method in ANGULAR.JS, as we can see in the following comment:

"$get is marked as attribute a lot, it should always be a method." (Developer of ANGULAR.JS*)*

In this case, the property `$get` receives an array that contains a function in its second element. Although the developer considers that this property is a method, our approach identifies it as an array and therefore classifies it as an attribute.

```
function match() {...};

MessageFormatParser.prototype.startStringAtMatch =
  function startStringAtMatch(match) {
    this.stringQuote = match;
    ...
  };
```

Listing 9: Example of method incorrectly identified as a class in ANGULAR.JS

```
JQLite.prototype = {
  ready: function(fn) {...},
  splice: [].splice,
  ...
};
```

Listing 10: Example of missing method (line 3 - system ANGULAR.JS)

5.5.2 Recall

We achieve a recall of 100% in 24 out of 30 systems for *classes*; in 22 systems for methods; and in 23 systems for attributes. In the following paragraphs we discuss the false negatives we detected for *classes*, methods, and attributes.

False negatives for classes. Six developers pointed out at least one missing *class* in their systems. In the case of the system CLUMSY-BIRD, the base *class* constructors are not available in the GitHub repository. The application imports an external file, which contains these base *classes*.[13] The import statement is placed directly in the main HTML file. For this reason, we were not able to detect *classes* in this system.

As a second case, EXPRESS' developer stated that our tool missed two *classes*, as shown in the following answer excerpt:

"So I have taken a look at the UML diagram you attached to the email and they do look mostly right. The main thing missing is there is also an Application class and a Router class, to round out a total of five main classes. The three you have there do look right, though." (Developer of system EXPRESS*)*

According to our strategy, `Application` and `Router` are not *classes*. `Application` is implemented as a singleton object, and we do not identify such structures as *classes*, as commented in Subsection 3.3. `Router` is not a *class* because its methods and attributes are not directly bound to

[13] http://cdn.jsdelivr.net/melonjs/2.0.2/melonJS.js

```
this.$get = ['$window', function(\$window) {...}];
```

Listing 11: Example of an array that contains a function (system ANGULAR.JS)

`this` nor `prototype`. Instead, the constructor function uses `___proto___` (an accessor property), as we can see in Listing 12 (line 5). In fact, `___proto___` is a special name used by Mozilla's JavaScript implementation to expose the internal `prototype` of the object through which it is accessed. However, the use of `___proto___` has been discouraged[14], mostly because it is not supported by other browsers.

In the four remaining systems (D3, JASMINE, VIDEO.JS, and LEAFLET), the causes for missing *classes* are related to the use of external frameworks and libraries that provide their own style for implementing class-like abstractions. The following comments are examples of answers in this category:

"The classes you found are only a small part of Leaflet classes. This is because Leaflet uses its own class utility: https://github.com/Leaflet/Leaflet/blob/master/src/core/Class.js" (Developer of system LEAFLET*)*

"From a pure Object Orientation point of view, I would probably call almost every file inside 'src / core' in the jasmine repo its own class (minus a few like 'util.js' and 'base.js' at least), which is more like 45 classes." (Developer of system JASMINE*)*

```
1 var proto = module.exports = function() {
2   function router() {
3     ...
4   }
5   router.__proto__ = proto;
6   router.params = {};
7   router.stack = [];
8   ...
9 };
10 proto.param = function param(name, fn) {...};
11 proto.handle = function() {...};
12 ...
```

Listing 12: Example of function `router` which is not detected as a class in system EXPRESS

False negatives for methods and attributes. In all six systems with missing *classes* we also have, as consequence, missing methods and attributes. Besides these cases, developers of other two systems pointed out missing methods. In the first case, for system ANGULAR.JS, our approach identified some methods as attributes, as discussed in the previous subsection (precision). In the second case, PAPER.JS's developers use a customized implementation that allows our approach to identify the *classes*, but not the methods. Listing 13 illustrates this issue for the *class* `Line`. In this case, the association between the constructor `Line` (line 3) and the methods `getPoint()`, `getVector()`, etc (lines 9-11) is built by using a project-specific function called `Base.extend` (line 1). The usage of this function hides the methods and some attributes from our tool.

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto

```
var Line = Base.extend({
  _class: 'Line',
  initialize: function Line(arg0, arg1, ...) {
    // Attributes
    this._px = arg0;
    this._py = arg1;
    ...
  },
  getPoint: function() {...},
  getVector: function() {...},
  ...
}
```

Listing 13: Class implementation for system PAPER.JS which uses a project-specific function (`Base.extend`) to implement classes

5.5.3 F-Score

Table 5 also reports the F-score results. The measures are equal to 100% in 22 out of 30 systems for *classes*, methods, and also for attributes. In the remaining systems, the measures range from 0% (CLUMSY-BIRD) to 96% (ACE and ANGULAR.JS) for *classes*, from 0% (CLUMSY-BIRD and LEAFLET) to 96% (ANGULAR.JS) for methods, and from 0% (CLUMSY-BIRD) to 93% (ANGULAR.JS) for attributes. The system CLUMSY-BIRD has F-score equal to zero because it uses base *class* constructors that are not available in its source code repository, as discussed in Subsection 5.5.2.

5.5.4 Overall results

Figure 13 presents the results for precision, recall, and F-score considering the whole population of *classes*, methods, and attributes, independently from system. The overall measurements range from 97% (*classes*) to 100% (methods) for precision, from 70% (methods) to 89% (attributes) for recall, and from 82% (methods) to 94% (attributes) for F-score.

5.6 Do developers intend to use the new support for classes that comes with ECMAScript 6?

Table 6 summarizes the answers for this question. Nineteen developers (58%) answered that they intend to use the new syntax. Two of them declared to have plans to migrate their systems to the new syntax, while the others stated that they intend to use it only when implementing new features and projects, as stated in the following answer:

"I'm quite confident that ES6 will make for a more robust codebase. And I think the most interesting point is that it can be applied progressively. We don't have to make a massive rewrite. Any new code we add can be ES6, and then we can slowly rewrite old code to be ES6 as well." (Developer of system SOCKET.IO*)*

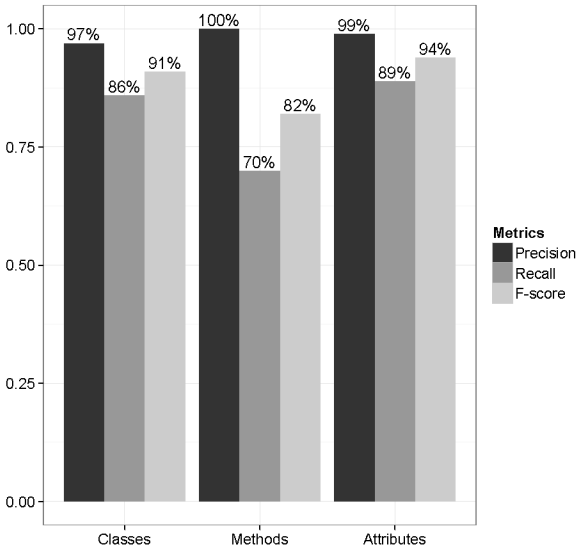


Figure 13: Overall results for precision, recall, and F-score

Table 6: Intention to use ES6 classes

Type of answer	#	%
Yes	19	58
No	12	36
Did not know	2	6

Twelve developers (36%) declared they do not intend to use ES6 syntax for classes, because they have to keep compatibility with legacy code, as stated in the following answer:

"For us right now it makes more sense to use CJS patterns and integrate with existing module ecosystems. When the ES6 penetration and support is higher, maybe we will switch." (Developer of system PIXIJS*)*

Summary: 58% of JavaScript developers intend to use the ES6 syntax for classes, but mostly for new features and projects.

6 Discussion

6.1 Heuristics Limitations and Improvement Opportunities

Based on the evaluation presented in Subsection 5.5, we summarize possible improvements to our heuristics in order to avoid both false positives and false negatives.

A false positive may occur when there are different identifiers with equal names, in different scopes. For example, when an identifier is declared with the same name of a global function. In this

case, when the identifier is assigned to a class property, we can improve our heuristics by checking if this identifier corresponds to a variable or parameter that is valid in the same scope. This way, the property can be classified as an attribute, instead of being wrongly classified as a method. We also acknowledge that, during program's execution, identifiers can receive a function as a value, transforming the class property into a method. However, this is the case of dynamically modified features, and our approach identifies class structures statically. The same understanding can be applied to class properties assigned to functions that are not part of the application, i.e., functions that belong to the JavaScript API or to external libraries and frameworks.

To reduce the chances of false negatives, we can modify the heuristics to also recognize the syntax with `__proto__`, used by Mozilla's JavaScript implementation, to expose the internal `prototype` of objects. Even though, as mentioned in Subsection 5.5.2, the use of this syntax has been discouraged by Mozilla. Moreover, we can also review our heuristics regarding singletons. Since not every singleton object is a class, further research is needed to precisely determine which ones indeed represent classes in a legacy JavaScript system. For the other false negatives pointed in Subsection 5.5, the base class constructors implemented in external files (e.g., in libraries and frameworks) cannot be statically identified as classes because their source code is not part of the system under analysis.

6.2 Practical Implications

Almost 70% of the systems we studied use *classes* ($CD > 0$). In fact, this usage may increase in the future because many developers intend to use the new ECMAScript 6 syntax for classes, as shown in our field study (Subsection 5.6). Therefore, we might consider the adaptation to the JavaScript ecosystem of many tools, concepts, and techniques widely used in class-based languages, like: (a) metrics to measure class properties like coupling, cohesion, complexity, etc; (b) reverse engineering techniques and tools to extract class and other diagrams from source code; (c) IDEs that include class-based views, like class browsers; (d) tools to detect bad smells in JavaScript classes; (e) recommendation engines to suggest best object-oriented programming practices; (f) techniques to detect violations and deviations in the class-based architecture of JavaScript systems; (g) tools to migrate to ECMAScript 6.

7 Threats to Validity

This section presents threats to validity according to the guidelines proposed by Wohlin et al. [29]. These threats are organized in three categories, addressing internal, external, and construct validity.

Internal Validity. In the field study, to address RQ #5, we recognize three internal threats. First, we consider that the developers correctly evaluated all elements we provided in the class diagrams of their systems. We acknowledge this activity is error-prone. However, we asked the main developers of each system, who are probably the most qualified people to conduct such evaluation. Second,

since some developers did not provide the names of all classes that represent false negatives in their systems, the first author of this study performed a manual verification in the related source code files in order to identify the remaining structures. The third internal threat is related to the non classification of singletons as classes, as mentioned in Subsection 3.3. In fact, in our field study some of the interviewed developers considered that singletons are classes.

External Validity. To address the first four research questions, we used a dataset of 918 JavaScript systems. For research questions RQ #5 and RQ #6, which involved contacting developers, we used a dataset of 60 JavaScript systems. As a threat, our datasets, both obtained from GitHub repository, might not represent the whole population of JavaScript systems. But, at least, we selected a representative number of popular and well-known systems, of different sizes and covering various domains.

Construct Validity. We use the library Linguist and a custom-made script, as described in Subsection 4.2, to remove unnecessary files from our dataset. We assume that this clean up process does not remove any source code files that could be used to implement *classes*.

8 Related Work

Richards et al. [22] conduct a large-scale study on the use of `eval` in JavaScript, based on a corpus of more than 10,000 popular web sites. They report that `eval` is popular and not necessarily harmful, although its use can be replaced with equivalent and safer code or language extensions in most scenarios. Moreover, it is usually considered a good practice to use `eval` when loading scripts or data asynchronously. After this first study, restricted to `eval`'s, the authors conduct a second study on a broad range of JavaScript dynamic features [4]. They conclude for example that libraries often change the prototype links dynamically, but such changes are restricted to built-in types, like `Object` and `Array`, and changes in user-created types are more rare. The authors also report that most JavaScript programs do not delete attributes from objects dynamically. To some extent, these findings support the feasibility of using heuristics to extract class-like structures statically from JavaScript code, as proposed in this paper.

Gama et al. [10] identify five styles for implementing methods in JavaScript: inside/outside constructor functions using anonymous/non-anonymous functions and using prototypes. Their main goal is to implement an automated approach to normalizing JavaScript code to a single consistent object-oriented style. They claim that mixing styles in the same code may hinder program comprehension and make maintenance more difficult. The strategy proposed in this paper covers the five styles proposed by the authors. Additionally, we also detect attributes and inheritance.

Feldthaus et al. [30, 31] describe a methodology for implementing automated refactorings on a nearly complete subset of the JavaScript language (ECMAScript 5). The authors specify and implement three refactorings: *rename property*, *extract module*, and *encapsulate property*. The

rename property is similar to the refactoring *rename field* for typed languages. The main difference is that while fields in Java, for example, are statically declared within class definitions, properties in JavaScript are associated with dynamically created objects and are themselves dynamically created after first write. The goal of the refactoring *extract module* is to use anonymous functions to make global functions become local. These anonymous functions will then return object literals with properties through which the previous global functions can be invoked. The *encapsulate property* refactoring can be used to encapsulate state by making a field private and redirecting access to that field via newly introduced getter and setter methods. It targets constructor functions that emulate *classes* in JavaScript. To determine if a function works as a constructor, they look for functions that initialize an object when invoked, like those that are invoked with `new` or `Object.create()`.

Fard and Mesbah [32] propose a set of 13 JavaScript code smells, including generic smells (*e.g.*, long functions and dead code) and smells specific to JavaScript (*e.g.*, creating closures in loops and accessing `this` in closures). They also describe a tool, called JSNose, for detecting code smells based on a combination of static and dynamic analysis. Among the proposed patterns, only Refused Bequest is directly related to class-emulation in JavaScript. In fact, this smell was originally proposed to class-based languages [33, 34], to refer to subclasses that do not use or override many elements from their superclasses. Interestingly, our strategy to detect *classes* opens the possibility to detect other well-known class-based code smells in JavaScript, like Feature Envy, Large Class, Shotgun Surgery, Divergent Change, etc.

Nicolay et al. [35] present an abstract machine for a core JavaScript-like language that tracks write side-effects in JavaScript functions to detect their purity. A function is considered pure if it does not generate observable side-effects. Since *classes* and methods, detected by our strategy, are functions in JavaScript, it is possible to extend the concept of purity to such class-like structures in order to improve program understanding and maintenance.

Nguyen et al. [36] use a static-analysis-based mining method to mine JavaScript usage patterns in web applications. They introduce JSModel, a graph representation for JavaScript code, and JSMiner, a tool that mines inter-procedural and data-oriented JavaScript usage patterns. Although they do not consider class-like structures in their work, the different strategies for class emulation can be considered usage patterns in JavaScript.

There is also a variety of tools and techniques for analyzing, improving, and understanding JavaScript code, including tools to prevent security attacks [37, 38, 39], and to understand event-based interactions [40, 41, 42, 43]. CoffeeScript[15] is another language that aims to expose the "good parts of JavaScript" by only changing the language's syntax [44, 45]. CoffeeScript compiles one-to-one into JavaScript code. As ECMAScript 6, the language includes class-related keywords, like `class`, `constructor`, `extends`, etc.

[15] http://coffeescript.org

9 Conclusion

This paper provides a large-scale study on the usage of class-based structures in JavaScript, a language that is used nowadays to implement complex single-page applications for the Web. We propose a strategy to statically detect *class* emulation in JavaScript and the JSClassFinder tool, that supports this strategy. We use JSClassFinder on a corpus of 918 popular JavaScript applications, with different sizes and from multiple domains, in order to describe the usage of class-like structures in legacy JavaScript systems. We perform a field study with JavaScript developers to evaluate the accuracy of our strategy and tool.

We summarize our findings as follows. First, there are essentially four types of JavaScript software, regarding the usage of *classes*: *class-free* (systems that do not make any usage of *classes*), *class-aware* (systems that use *classes* marginally), *class-friendly* (systems that make relevant usage of *classes*), and *class-oriented* (systems that have the vast majority of their data structures implemented as *classes*). The systems in these categories represent, respectively, 32%, 34%, 27%, and 7% of the systems we studied. Precision, recall and F-score measures indicate that our tool is able to identify the *classes*, methods, and attributes in JavaScript systems. The overall results range from 97% to 100% for precision, from 70% to 89% for recall, and from 82% to 94% for F-score.

Second, we found that there is no significant relation between size and *class* usage. Therefore, we cannot conclude that the larger the system, the greater the usage of *classes*, at least in proportional terms. For this reason, we hypothesize that the background and experience of the systems' developers have more impact on the decision to design a system around *classes*, than its size.

Third, prototype-based inheritance is not popular in JavaScript. We counted only 70 out of 918 systems (8%) using inheritance. We hypothesize that there are two main reasons for this. First, even in class-based languages there are strong positions against inheritance, and a common recommendation is to "favor object composition over class inheritance" [46, 47]. Second, prototype-based inheritance is more complex than the usual implementation of inheritance available in mainstream class-based object-oriented languages.

Fourth, *classes* in JavaScript have usually less than 28 attributes and 61 methods (90th percentile measures). It is also common to have data-oriented *classes*, i.e., *classes* with more attributes than methods. In half of the systems, we have at least 39% of such *classes*.

Fifth, 58% of JavaScript developers answered our field study saying they intend to use the ES6 new syntax for class emulation, but usually only for new features and projects.

As future work, we plan to adapt our approach to be able to: (a) measure other class properties, like coupling, cohesion, and complexity; (b) extract class dependencies and other diagrams from source code; (c) identify bad smells in JavaScript classes; (d) recommend best object-oriented programming practices for JavaScript; (e) detect violations and deviations in the class-based architecture of JavaScript systems; (f) support developers that intend to migrate their legacy code to use ECMAScript 6 classes.

All our data and toolset are publicly available at `https://github.com/aserg-ufmg/JSClassFinder`.

Acknowledgments

The authors would like to thank CNPq, CAPES and FAPEMIG. This research is partially supported by STICAmSud project 14STIC-02 and FONDECYT 1160575.

References

[1] H.M. Kienle. It's about time to take JavaScript (more) seriously. *IEEE Software*, 27(3):60–62, May 2010.

[2] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. JavaScript errors in the wild: An empirical study. In *22nd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109. IEEE Computer Society, 2011.

[3] Alex Nederlof, Ali Mesbah, and Arie van Deursen. Software engineering for the web: the state of the practice. In *36th International Conference on Software Engineering (ICSE), Companion Proceedings*, pages 4–13, 2014.

[4] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[5] European Association for Standardizing Information and Communication Systems (ECMA). ECMA-262: ECMAScript Language Specification. edition 5.1, 2011.

[6] A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM Fall Joint Computer Conference*, pages 36–40, 1986.

[7] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *24th European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, 2010.

[8] European Association for Standardizing Information and Communication Systems (ECMA). ECMAScript Language Specification, 6th edition, Draft October, 2014.

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

[10] W. Gama, M.H. Alalfi, J.R. Cordy, and T.R. Dean. Normalizing object-oriented class styles in JavaScript. In *14th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 79–83, Sept 2012.

[11] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[12] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2011.

[13] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *2nd Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 227–242. ACM, 1987.

[14] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 36–40. IEEE Computer Society Press, 1986.

[15] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP)*, pages 428–452, 2005.

[16] Leonardo Humberto Silva, Daniel Hovadick, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. JSClassFinder: A Tool to Detect Class-like Structures in JavaScript. In *6th Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Demonstration Track*, pages 113–120, 2015.

[17] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: An agile reengineering environment. In *10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, ESEC/FSE-13, pages 1–10, New York, NY, USA, 2005. ACM.

[18] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003.

[19] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 203–212, 2006.

[20] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[21] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., 2004.

[22] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *25th European Conference on Object-oriented Programming (ECOOP)*, 2011.

[23] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: Semi-automated removal of eval from JavaScript programs. In *27th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620, 2012.

[24] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[25] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software. In *21st Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 397–412. ACM, 2006.

[26] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power Laws in Software. *ACM Transactions on Software Engineering and Methodology*, 18:1–26, 2008.

[27] Richard Wheeldon and Steve Counsell. Power Law Distributions in Class Relationships. In *International Working Conference on Source Code Analysis and Manipulation*, pages 45–54, 2003.

[28] Paloma Oliveira, Marco Tulio Valente, and Fernando Lima. Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263, 2014.

[29] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[30] Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, and Frank Tip. Refactoring towards the good parts of JavaScript. In *26th Conference on Object-Oriented Programming (OOPSLA), Companion Proceedings*, pages 189–190, 2011.

[31] Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schafer, and Frank Tip. Tool-supported refactoring for JavaScript. In *26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 119–138, 2011.

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

[33] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[34] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.

[35] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting function purity in JavaScript. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 101–110, 2015.

[36] Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *36th International Conference on Software Engineering (ICSE)*, pages 791–802, 2014.

[37] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[38] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *18th International Conference on World Wide Web (WWW)*, pages 561–570, 2009.

[39] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *34th Symposium on Principles of Programming Languages (POPL)*, pages 237–249, 2007.

[40] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *International Conference on Software Engineering (ICSE)*, pages 367–377, 2014.

[41] Andy Zaidman, Nick Matthijssen, Margaret-Anne D. Storey, and rie van Deursen. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218, 2013.

[42] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37, pages 321–345, 2015.

[43] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.

[44] Mark Bates. *Programming in CoffeeScript*. Addison-Wesley Professional, 1st edition, 2012.

[45] Alex MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, Inc., 2012.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[47] S. Stefanov. *JavaScript Patterns*. O'Reilly Media, 2010.