Does JavaScript Software Embrace Classes?

Leonardo Humberto Silva
Federal Institute of
Northern Minas Gerais, Brazil
leonardo.silva@ifnmg.edu.br

Miguel Ramos,
Marco Tulio Valente
Federal University of
Minas Gerais, Brazil
{miguel.ramos,mtov}@dcc.ufmg.br

Alexandre Bergel
Department of Computer Science
DCC - Pleiad Lab
University of Chile
abergel@dcc.uchile.cl

Nicolas Anquetil
RMoD Project-Team
INRIA Lille Nord Europe
France
nicolas.anquetil@inria.fr

***Abstract*—JavaScript is the *de facto* programming language for the Web. It is used to implement mail clients, office applications, or IDEs, that can weight hundreds of thousands of lines of code. The language itself is prototype based, but to master the complexity of their application, practitioners commonly rely on informal class abstractions. This practice has never been the target of empirical research in JavaScript. Yet, understanding it is key to adequately tuning programming environments and structure libraries such that they are accessible to programmers. In this paper we report on a large and in-depth study to understand how class emulation is employed in JavaScript applications. We propose a strategy to statically detect class-based abstractions in the source code of JavaScript systems. We used this strategy in a dataset of 50 popular JavaScript applications available from GitHub. We found four types of JavaScript software: class-free (systems that do not make any usage of classes), class-aware (systems that use classes, but marginally), class-friendly (systems that make a relevant usage of classes), and class-oriented (systems that have most of their data structures implemented as classes). The systems in these categories represent, respectively, 26%, 36%, 30%, and 8% of the systems we studied.**

***Index Terms*—JavaScript; Class-based languages; Reverse Engineering.**

I. INTRODUCTION

JavaScript is the *de facto* programming language for the Web [1]. The language was initially designed in the mid-1990s to extend web pages with small executable code. Since then, its popularity and relevance has only grown [2], [3]. For example, JavaScript is now the most popular language at GitHub, considering new repositories created by language. It is also reported that the language is used by 97 out of the web's 100 most popular sites [4]. 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, IDEs, etc, which can reach hundreds of thousands lines of code[1].

Despite the complexity, size, and relevance of modern JavaScript software, only a few research effort has been carried on how developers effectively organize and manage large JavaScript software systems. Specifically, JavaScript is an imperative, and object-oriented language centered on prototypes, rather than a class-based language [1], [5], [6]. Despite not having explicit class constructions, the prototype-based object system of the language is flexible enough to support the implementation of mainstream class-based abstractions, including attributes, methods, constructors, inheritance hierarchies, etc. However, structuring a software around such abstractions is a design decision, which should be taken by JavaScript developers. In other words, the language is flexible enough to support different modularization paradigms, including procedural programming (*e.g.*, considering a system as a set of functions, like in C), modular programming (*e.g.*, a system is a set of modules that encapsulate data and operations, like in Modula-2), and class-based object-oriented programming (*e.g.*, a system is a set of classes, like in Java).

In this paper, we report on an empirical study conducted to shed light on how often JavaScript developers modularize their systems around abstractions that resemble object-oriented classes. We found that almost 40% of the systems we studied make a relevant usage of classes. Therefore, one might consider the adaptation to the JavaScript ecosystem of tools, concepts, and techniques widely used in class-based languages, such as reverse engineering techniques, IDEs with class-based views, bad smells detection tools, recommendation engines and techniques to detect violations and deviations in class-based architectures. Furthermore, the new standard version of JavaScript, named ECMAScript 6, will include syntactical support for classes [7]. Therefore, revealing how JavaScript developers currently emulate classes is a valuable information for those that plan to use classes in their systems, according to this new standard syntax.

The main contributions of our work are as follows:

- We document how prototypes in JavaScript are used 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 II). In this paper, we use the term classes to refer to such structures, since they have a very similar purpose as the native classes available in mainstream object-oriented languages. We also propose a strategy to statically identify classes in JavaScript code (Section III).
- We propose an open-source supporting tool, called JSCLASSFINDER, that practitioners can use to detect and inspect classes in JavaScript software. This tool is described in Section III-B
- We provide a thorough study on the usage of classes in a dataset of 50 popular JavaScript software available at GitHub (Section IV). This study aims to answer

[1]http://sohommajumder.wordpress.com/2013/06/05/gmail-has-biggest-collection-of-javascript-code-lines-in-the-world, verified 11/15/2014

the following research questions: (a) Do developers use classes in JavaScript applications? (b) Do developers use inheritance in JavaScript applications? (c) What is the size of JavaScript classes?

II. Classes in JavaScript

This section lists the different mechanisms to emulate classes in JavaScript. To identify these mechanisms we carefully conducted an informal survey on documents available on the web, including tutorials[2], blogs[3], and StackOverflow discussions[4]. We also surveyed a catalogue of five encapsulation styles for JavaScript proposed by Gama *et al.* [8] and JavaScript books targeting language practitioners [9], [10].

Basically, an *object* in JavaScript is a set of name-value pairs. Method and variable names are strings, called *properties*, and the values are any objects, including immediate values (*e.g.,* numbers, boolean) and functions. To implement *classes* in JavaScript — *i.e.*, data structures that resemble the class concept of mainstream object-oriented languages; the most common strategy is to use functions. Particularly, any function can be used as template for the creation of objects. In a function, the standard JavaScript keyword `this` is used to define properties that emulate attributes and methods. *Attributes* are properties associated with any object, except functions. *Methods* are properties associated with inner functions. The keyword `new` is used to create objects for a class.

To illustrate the definition of classes in JavaScript, we use a simple `Circle` class. Listing 1 presents the function that defines this class (lines 1-6), which includes a `radius` attribute and a `getArea` method.

```
1 function Circle (radius) {    // function -> class
2   this.radius= radius;     // property -> attribute
3   this.getArea= function () {// function -> method
4     return (3.14 * this.radius * this.radius);
5   }
6 }
7 // Circle instance -> object
8 var myCircle = new Circle (10);
```

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 OOP is assimilated to the prototype link between an object and its prototype. To evaluate an expression like `obj.p`, the runtime 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 reaches an empty object. 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 an `Object`. Therefore, a chain of prototype links usually ends at `Object`.

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript, verified 11/15/2014

[3]http://javascript.crockford.com/prototypal.html, verified 11/15/2014

[4]http://stackoverflow.com/questions/387707/whats-the-best-way-to-define-a-class-in-javascript, verified 11/15/2014

By manipulating the `prototype` property, we can define a method whose implementation is shared by all object instances. It is also possible to define properties 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 and a `getCircumference` method. It is worth noting that `getCircumference` is not a method attached to the class (as a `static` method in Java). It has for example access to the `this` variable, whose value is not determined using lexical scoping rules, but instead using the caller object.

```
// prototype property -> static attribute
Circle.prototype.pi = 3.14;

// function -> method
Circle.prototype.getCircumference= function () {
   return (2 * this.pi * this.radius);
}
```

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

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

```
function Circle2D (x, y) {  // class Circle2D
  this.x = x;
  this.y= y;
}

// Circle2D is a subclass of Circle
Circle2D.prototype = new Circle(10);

// Circle2D extends Circle with new methods
Circle2D.prototype.getX = function () {
  return (x);
}
Circle2D.prototype.getY = function () {
  return (y);
}
```

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

III. Detecting Classes in JavaScript

In this section, we describe a strategy to statically detect classes in JavaScript source code (Section III-A). Section III-B describes the tool we implemented to detect classes in JavaScript using the proposed strategy. We also report limitations of this strategy, mainly due to the dynamic behavior of JavaScript (Section III-C).

TABLE I
CLASS-BASED LANGUAGES VS JAVASCRIPT

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

A. Strategy to Detect Classes

To detect classes, we reused with minimal adaptations a simple grammar for JavaScript, originally proposed by Anderson *et al.* [11] to represent the way objects are created in JavaScript and the way objects acquire fields and methods. This grammar is as follows:

```
Program  ::=  FuncDecl*
FunDecl  ::=  function f() { Exp }
Exp      ::=  new f()
              this.a= Exp;
              this.a= function { Exp }
              f.prototype.a= Exp;
              f.prototype.a= function { Exp }
              f.prototype= new f();
              Object.create(f.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 to create objects and to 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 program P, must respect the following conditions:

- P must have a function with name `C`.
- P must include at least one expression `new C()` or `Object.create(C.prototype)`.
- For each $a \in \mathcal{A}$, the function `C` must include an assignment `this.a = Exp` or P must include an assignment `C.prototype.a = Exp`.
- For each $m \in \mathcal{M}$, function `C` must include an assignment `this.m = function {Exp}` or P must include an assignment `C.prototype.m = function {Exp}`.

Functions that partially match these conditions are not classified as classes. For example, if a function matches all conditions, except the one that requires a `new` expression in the program, it is not listed as a class.

In the JavaScript examples of Section II we have two classes:

- `(Circle,{radius,pi},{getArea,getCircumference})`
- `(Circle2D, {x, y}, {getX, getY})`

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)`.

In the examples of Section II, `Circle2D` is a subclass of `Circle`.

B. Tool Support

We implemented a tool, called JSCLASSFINDER, for identifying classes in JavaScript programs. As illustrated in Figure 1, this tool works in two steps. In the first step, Esprima[5]—a widely used JavaScript Parser—is used to generate a full abstract syntax tree, in JSON format. In the second step, we implemented an application that supports the strategies described in Section III-A to detect classes in a JavaScript AST in the JSON format. JSCLASSFINDER also collects the following basic object oriented metrics: Number of Attributes (NOA), Number of Methods (NOM), Depth of Inheritance Tree (DIT), and Number of Children (NOC) [12].

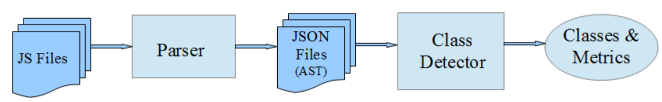


Fig. 1. JSClassFinder

C. Limitations

The proposed strategy requires each class `C` to have at least one corresponding `new C` expression in the program. This is important because any function in JavaScript has access to `this` and can use it to add properties to the calling object or to the global context. For example, consider the following code:

```
function f(x) {
   this.x= x;
}
f(10);
```

The call to `f` does not have a target object. In this case, the result is to add a property `x` in the object that represents the global context in JavaScript programs. Therefore, although `f` resembles a constructor function, it is not in fact used as a template to create objects, since the program does not include a `new`. For this reason, it is not classified as a class, according to our definition. On the other hand, classes designed to be instantiated by client applications, as would be the case of the public interface of APIs, are not detected by the proposed strategy. In this case, the call to `new` is typically made by the clients.

We also acknowledge that there is not one single strategy to emulate classes in JavaScript. For example, it is possible to create "singleton" objects directly, without using any class-like construction, as in this example:

[5]http://esprima.org, verified 11/15/2014

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

In addition, there are numerous JavaScript frameworks, like Prototype[6] and AngularJS[7], 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. Recognizing other ways to mimic classes could be the goal of some future work.

Moreover, there are abstractions related to classes 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 4, 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 considered as private, because they can only be accessed by the "private function" `f2`. However, we decided to not classify `f2` as a private method, mainly because it cannot access the `this` object, nor can it be directly called from the public methods of the class.

```
function f1 () {          // outer function -> class
  var x;                       // local variable
  function f2 () {             // inner function
    // can access "x"
    // cannot access "this"
  }
}
```

Listing 4. Using closures to implement "private" inner functions

In JavaScript, it is also 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 defined in its class. It is also possible to modify the prototype chains dynamically. When we detect multiple assignments on the prototype link of a class `A`, a warning is raised reporting the alternative superclasses of `A`. Finally, the behavior of a program can be modified dynamically, using the `eval` operator [13], [14]. However, we do not consider the impact of `eval`'s in the strategy described in Section III-A. For example, we do not account for classes entirely or partially created by means of `eval`.

IV. Study

The goal of this study is to evaluate whether the strategy we propose (Section III-A) is able to detect classes in real JavaScript software. The result of our study targets at JavaScript developers interested in understanding and evolving the classes used in their systems. The context we defined for this study consists of 50 popular JavaScript systems, available at GitHub.

[6]http://prototypejs.org, verified 11/15/2014
[7]https://angularjs.org, verified 11/15/2014

In this study, we answer the following research questions:

- RQ #1: Do developers use classes in JavaScript applications?
- RQ #2: Do developers use inheritance in JavaScript applications?
- RQ #3: What is the size of JavaScript classes, in terms of number of methods and attributes? (Are the results similar when comparing to traditional OO languages?)

In the following, we first describe the process we followed to select JavaScript software from GitHub and to clean up the downloaded code (Section IV-A). Next, we present and discuss answers for the proposed research questions (Sections IV-B to IV-D). Finally, we discuss threats to validity (Section IV-E).

A. Data Extraction

The JavaScript systems considered in this study are available at GitHub. We selected systems ranked with at least 1,000 stars at GitHub, whose sole language is JavaScript, that have at least 150 commits and are not forks of other projects. This search was performed on June, 2014 and resulted in 50 systems. After the check out of each system, we automatically inspected the source code to remove the following files: compacted files used in production to reduce network bandwidth consumption (which have the extension $*$`.min.js`), copyright files (`copyright.js`), documentation files (located in directories called `doc` or `docs`), and files belonging to third party libraries (located in directories `thirdparty` or `node_modules`). We did not discard test files and examples, because these files usually include `new` expressions, which are primordial for the success of the strategy proposed to detect classes, as described in Section III-A.

The selected systems are presented in Table II, including their version, a brief description, size (line of code), number of files, and number of functions. Although we did not discard test files and examples, they are not counted when computing the size metrics in Table II. 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 194,159 LOC and 574 files with `.js` extension. The smallest system (`masonry`) has 197 LOC and a single file. The average size is 13,846 LOC (standard deviation 33,720 LOC) and 58.7 files (standard deviation 121.1 files). The median is 2,462 LOC and 16 files. We also found systems with hundreds of functions in a single JavaScript file. For example, `reveal.js` is a system with a single file and 105 functions.

After downloading the systems and cleaning up the code, we executed the JSClassFinder tool to extract class information and metrics on each system.

B. Do developers use classes in JavaScript applications?

Table II presents the total number of classes detected by JSClassFinder in the selected systems. We found classes in 37 out of 50 systems (74%). The system with the largest number of classes is `pdf.js` (144 classes), followed by `ace` (133 classes), `three.js` (106 classes), and `brackets` (101

TABLE II
JAVASCRIPT SYSTEMS (ORDERED ON THE CUR COLUMN, SEE DESCRIPTION IN ACCOMPANYING TEXT)

System	Version	Description	LOC	# Files	# Func	# Class	# Meth	# Attr	CUR	SCUR	DOCR
masonry	3.1.5	Cascading grid layout library	197	1	10	0	0	0	0.00	-	-
randomColor	0.1.1	Color generator	361	1	17	0	0	0	0.00	-	-
respond	1.4.2	Polyfill for CSS3 media queries	460	3	15	0	0	0	0.00	-	-
clumsy-bird	0.1.0	Flappy Bird Game	628	7	1	0	0	0	0.00	-	-
deck.js	1.1.0	Modern HTML Presentations	732	1	22	0	0	0	0.00	-	-
impress.js	0.5.3	Presentation framework	769	1	23	0	0	0	0.00	-	-
async	0.9.0	Async utilities	1,117	1	75	0	0	0	0.00	-	-
turn.js	3.0.0	Page flip effect for HTML5	1,914	1	18	0	0	0	0.00	-	-
zepto	1.1.3	Minimalist jQuery API	2,456	17	149	0	0	0	0.00	-	-
jade	1.0.2	Template engine for Node	4,051	28	41	0	0	0	0.00	-	-
select2	3.4.8	Replacement for select boxes	4,132	45	44	0	0	0	0.00	-	-
jQueryFileUp	9.5.7	File upload widget	4,442	15	49	0	0	0	0.00	-	-
semantic-UI	0.18.0	UI component framework	11,951	19	25	0	0	0	0.00	-	-
wysihtml5	0.3.0	Rich text editor	5,913	69	107	2	0	3	0.02	0.00	0.50
underscore	1.6.0	Functional programming helpers	1,390	1	91	2	1	1	0.03	0.00	0.00
paper.js	0.9.18	Vector graphics framework	25,859	67	143	2	2	0	0.03	0.00	0.00
intro.js	0.9.0	Templates for introductions	1,026	1	24	1	0	2	0.04	-	1.00
timelineJS	2.25.0	Visualization chart	18,237	89	213	10	0	7	0.05	0.00	0.40
jasmine	2.0.0	JavaScript testing framework	2,956	48	239	7	8	13	0.06	0.17	0.57
reveal.js	2.6.2	HTML presentation framework	3,375	1	105	3	6	11	0.09	0.00	0.67
floraJS	1.0.0	Simulation of natural systems	3,325	26	104	4	6	8	0.10	0.00	0.50
express	4.4.1	Minimalist framework for Node	2,942	11	84	3	7	11	0.12	0.00	0.33
numbers.js	0.4.0	Mathematics library for Node	2,454	10	119	1	14	4	0.13	-	0.00
typeahead.js	0.10.2	Auto-complete library	2,468	19	95	12	0	48	0.13	0.00	1.00
video.js	4.6.1	HTML5 video library	7,939	38	432	5	53	6	0.13	0.00	0.40
sails	0.10.0	MVC framework for Node	13,053	98	154	9	15	54	0.16	0.00	0.22
ionic	1.0.0	HTML5 mobile framework	14,376	90	283	15	31	26	0.16	0.14	0.27
chart.js	0.2.0	HTML5 charts library	1,417	1	34	6	0	0	0.18	0.00	0.00
grunt	0.4.5	JavaScript task runner	1,932	11	94	1	16	8	0.18	-	0.00
angular.js	1.3.0	Web application framework	79,753	539	705	40	86	74	0.18	0.03	0.43
ghost	0.4.2	Blogging platform	15,048	122	205	12	32	17	0.21	0.18	0.08
brackets	0.41.0	Source code editor	122,971	403	2,723	101	638	452	0.27	0.15	0.34
backbone	1.1.2	Data structure for web apps	1,681	2	17	3	2	0	0.29	0.00	0.00
skrollr	0.6.25	Scrolling library	1,764	1	44	1	12	0	0.30	-	0.00
leaflet	0.7.0	Library for interactive maps	8,389	71	63	9	10	19	0.30	0.00	0.33
ace	1.0.0	Source code editor	194,159	574	3,176	133	810	535	0.30	0.14	0.53
gulp	3.7.0	Streaming build system	282	4	9	1	2	4	0.33	-	1,00
three.js	0.0.67	JavaScript 3D library	37,102	164	609	106	120	497	0.37	0.00	0.74
pdf.js	0.8.0	Web PDF reader	48,090	41	531	144	58	574	0.38	0.19	0.81
bower	1.3.5	Package manager	8,194	53	306	13	112	34	0.41	0.00	0.08
algorithms.js	0.2.0	Data structures & algorithms	1,594	29	82	7	28	13	0.43	0.33	0.14
mustache.js	0.8.2	Logic-less template syntax	571	1	27	3	9	7	0.44	0.00	0.33
parallax	2.1.3	Motion detector for devices	1,007	3	57	1	24	40	0.44	-	1,00
less.js	1.7.0	CSS pre-processor	10,578	48	202	52	42	190	0.47	0.02	0.88
2048	-	Number puzzle game	873	10	66	4	33	13	0.56	0.00	0.50
pixiJS	1.5.3	Rendering engine	13,896	72	361	61	152	267	0.59	0.00	0.69
isomer	0.2.4	Isometric graphics library	770	71	47	7	29	25	0.77	0.00	0.43
slick	1.3.6	Carousel visualization engine	1,684	1	64	1	50	0	0.80	-	0.00
fastclick	1.0.2	Library to remove click delays	798	1	22	1	18	9	0.86	-	0.00
socket.io	1.0.4	Realtime application framework	1,223	4	49	4	44	36	0.98	0.00	0.25

classes). `Semantic-UI` is the largest system (11,951 LOC) that does not have classes, at least as formalized in Section III-A. Figure 2 shows two classes detected in the `pixiJS` system [8]. These classes represent an `Ellipse` (with four attributes and three methods) and a `JSONLoader` (with four attributes and four methods).

Systems in Table II are classified in ascending value of what we called *Class Usage Ratio (CUR)*, which is defined as:

$$CUR = \frac{\#\ methods + \#\ classes}{\#\ functions}$$

[8]We omitted the full code due to paper's size constraints.

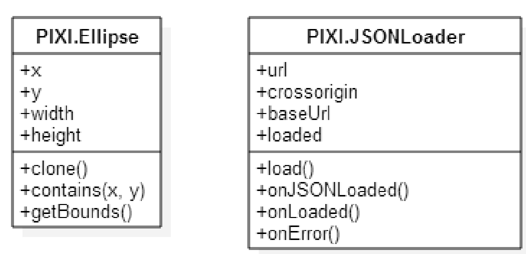


Fig. 2. Examples of classes detected in `pixiJS`

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.

Figure 3 shows the distribution of the CUR values, considering all 50 systems (on the left and systems with CUR greater than zero on the right). On the left, there are systems with very small CUR values. The first quartile is 0.005 (lower bound of the black box within the "violin") and 13 systems have CUR equal to zero (the width of the "violin" indicates the number of the distribution of the systems for a given CUR value). The median for all systems (white dot at the heart of the violin) is 0.15 and the third quartile is 0.36 (upper bound of the black box). We also found one almost fully class-oriented system, `socket.io`, with CUR equal to 0.98.

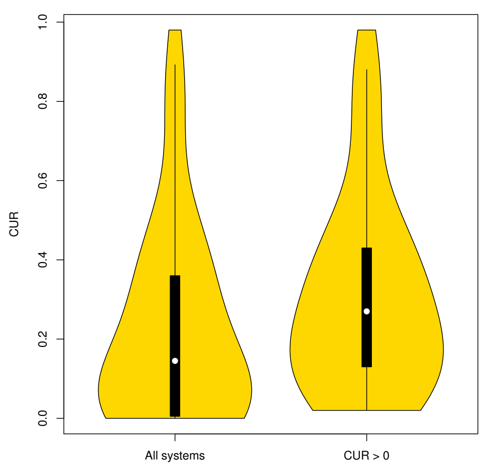


Fig. 3. Class Usage Ratio (CUR) distribution

There is a significant change in the CUR distribution when we only consider the systems with CUR greater than zero, as represented in the violin plot on the right. One could consider that these systems are those that chose to use classes with the convention that our tool is looking for. Other systems might use other conventions, or no class abstraction at all. The first quartile of CUR is now 0.13, the median is 0.27, and the third quartile is 0.43. In other words, 26% of the systems do not use classes at all or are using an abstraction other than the one we are looking for. This might be a deliberate design decision of their developers. On the other hand, in the remaining systems, the emulation of class represents on the median 27% of the functions.

Figure 4 shows scatterplots with size metrics on the x-axis in a logarithmic scale and CUR on the y-axis. We also computed the Spearman's rank correlation coefficient between CUR and the following system size metrics: LOC, number of files, and number of functions. The intention is to clarify the effect of the system's size on the usage of class-based structures. The results are presented in Table III. We found a very weak correlation for LOC (ρ=0.11, p-value=0.45), and number of files (ρ=0.22, p-value=0.13), and slightly better for number of functions (ρ=0.29, p-value=0.04).

TABLE III
SPEARMAN CORRELATION BETWEEN CUR AND SIZE METRICS

	LOC	# Files	# Func
Spearman	0.110	0.217	0.295
p-value	0.446	0.130	0.037

In summary, we observed four main groups of systems:

- *Class-free systems*: 13 systems that do not use classes at all ($CUR = 0$).
- *Class-aware systems*: 18 systems that use classes, but marginally ($CUR \leq 0.21$).
- *Class-friendly systems*: 15 systems where classes represent a common data structure ($0.21 < CUR < 0.59$)
- *Class-oriented systems*: four systems that have most of their structures organized as classes ($CUR \geq 0.77$).

We used the Kruskal-Wallis test to check if the LOC distributions in these groups are equal. For this test, we merged the class-friendly and class-oriented groups into a single one, because the latter has only four systems. The test resulted in a p-value of 0.03, 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 different (1,117; 3,350; and 1,684; respectively). It is not clear however if this difference (statistically significant on our experimental corpus) is relevant in practice. More research is necessary to clarify this point.

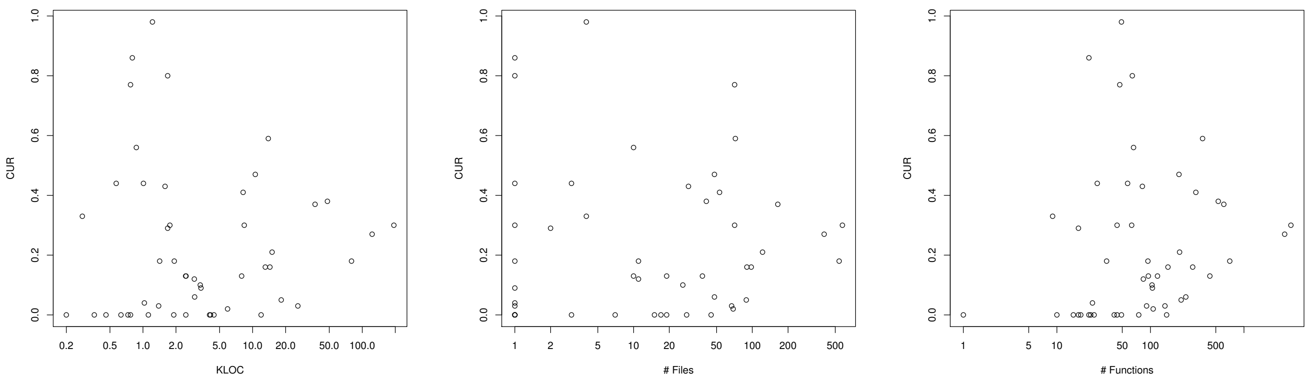
C. Do developers use inheritance in JavaScript applications?

To evaluate the usage of inheritance, we propose a metric called *Subclass Usage Ratio (SCUR)*, defined as:

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

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

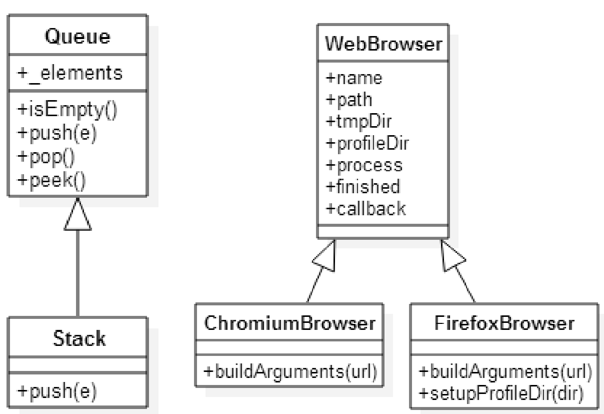
As shown in Table II, the use of prototype-based inheritance is rare in JavaScript systems. First, we counted 29 systems (58%) having two or more classes, i.e., systems where it is possible to detect the use of inheritance. However, in 20 of such systems (69%), we did not find subclasses ($SCUR = 0$). The system with the highest use of inheritance is `algorithms.js` ($SCUR = 0.33$). As an example, in this system we found a class `Stack` defined as a subclass of `Queue`, as represented in the class diagram of Figure 5a. In this case, `Stack` inherits three methods from `Queue` (`isEmpty()`, `pop()`, and `peek()`) and redefines one method (`push()`). Figure 5b shows an example



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

Fig. 4. Size metrics vs Class Usage Ratio (CUR)

detected in `pdf.js`, including a `WebBrowser` class and two subclasses, `ChromiumBrowser` and `FirefoxBrowser`.



(a) `algorithms.js` (b) `pdf.js`

Fig. 5. Examples of inheritance

D. What is the size of JavaScript classes in terms of number of methods and attributes?

To answer this question, we initially investigated seven systems with at least 40 classes: `pdf.js`, `ace`, `three.js`, `brackets`, `pixiJS`, `less.js`, and `angular.js`. These systems have a number of classes that are compatible with medium-sized systems in class-based languages. Figure 6 shows the quantile functions for the Number of Attributes (NOA) and Number of Methods (NOM) values in each system. In this figure, the black lines represent the systems. The x-axis represents the quantiles and the y-axis represents the upper 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 values. This means that p% of the classes in the system in question have at most k attributes. Moreover, Figure 6 includes a red line that represents the whole population of classes found in our dataset (787 classes, detected in 37 systems).

Regarding NOA, the quantile functions reveal that the vast majority of the classes have at most, 11 attributes. Specifically, the 90th percentile range from five attributes (`angular.js`) to 11 attributes (`brackets` and `three.js`). Regarding NOM, the vast majority of the classes have less than 13 methods. In this case, the 90th percentile ranges from one method (`pdf.js`) to 13 methods (`brackets`). In the whole population of classes, the results are similar. For NOA, the 90th percentile is eight; for NOM, it is seven.

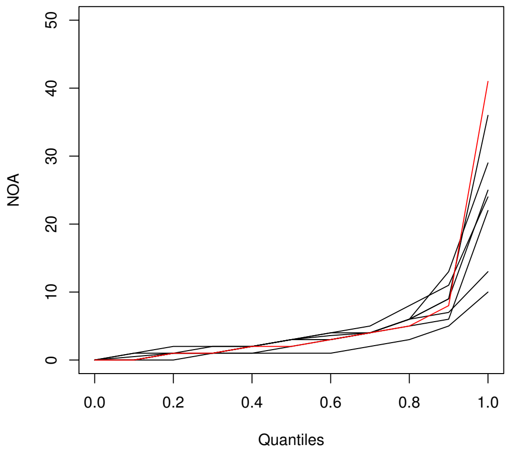
When generating the quantile functions, we found a very large class in the `ace` system, with 164 attributes and 503 methods. By inspecting its source code, we discovered that this class is a PHP parser, automatically generated by a parser generator tool. For this reason, we removed it from the quantile functions in Figure 6. Otherwise, it would require the presentation of very high values in the y-axis. This finding shows that class-emulation is also a design decision followed by the developers of code generation tools.

Figure 6 shows that the NOA and NOM values tend to present a right-skewed (or heavy-tailed) behavior, meaning that while the bulk of the distribution occurs for fairly small classes there is a small number of large classes with NOA and NOM measures much higher than the typical value. This results in a long tail to the right, if the metric values are presented in a histogram. In fact, this heavy-tailed behavior is normally observed in source code metrics [15]–[17].

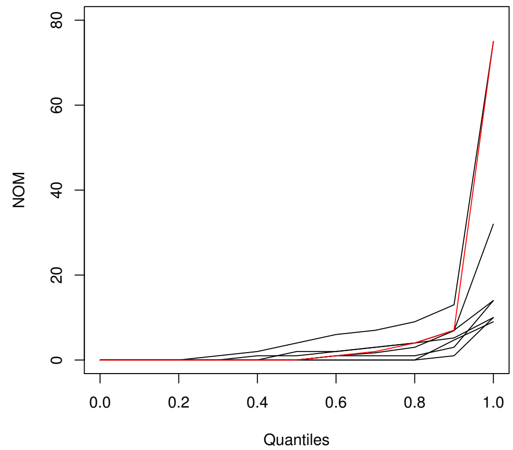
In Table II, 16 out of 37 systems with classes have a total number of attributes (*# Attr* column) greater than the total number of methods (*# Meth* column). This contrasts to the common shape of classes in class-based languages, when classes have usually more methods than attributes [18]. To understand this phenomenon, we propose a metric called *Data-Oriented Class Ratio (DOCR)* defined as follows:

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

where $Classes$ is the set of all classes in a given system. *DOCR*



(a) NOA



(b) NOM

Fig. 6. Quantile functions

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 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.

Table II presents the *DOCR* values for each system and Figure 7 shows the *DOCR* distribution using a violin plot. The median DOCR value is 0.34, which is a high measure. For example, metric thresholds for Java suggest that classes should have at most 8 attributes and 16 methods, i.e., they recommend two methods per attribute for a typical class [19]. On the other hand, half of the JavaScript systems we studied have more than 34% of classes with more attributes than methods. We did not inspect the code in detail to explain this fact, but we hypothesize that it is due to JavaScript developers placing less importance on encapsulation. For example, getters and setters are rare in JavaScript.

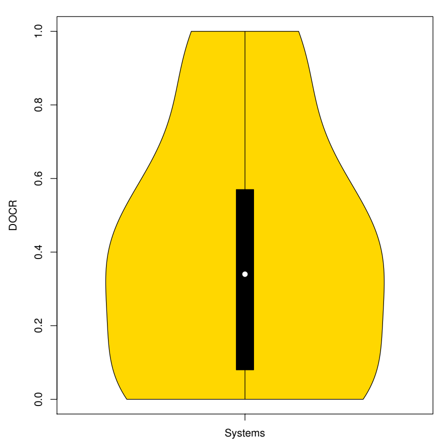


Fig. 7. Data-Oriented Classes Ratio (DOCR) distribution

E. Threats to Validity

First, the proposed strategy to detect classes do not handle the whole spectrum of class styles supported by JavaScript and also by third-party frameworks, as discussed in Section III-C. However, we cover the style that closely resembles the syntax and semantics of classes, attributes, and methods in class-based languages. Second, as usual, our dataset 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. Third, the proposed strategy depends on `new` expressions to detect classes. Therefore, it misses important classes of APIs, libraries, and frameworks, which are designed to be instantiated by clients. To mitigate this issue, we did not remove test files and files with examples from our dataset, when searching for classes. In a manual inspection, we did not find classes in such files. However, if they exist, they are counted in our study.

V. Related Work

Studies: Richards et al. [13] conducted 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. It is usually considered a best practice for example when loading scripts or data asynchronously. After this first study, restricted to `eval`'s, the authors conducted a second study, when they investigated a broad range of JavaScript dynamic features [4]. They concluded 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 classes 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.

Tools: Gama et al. [8] identified five styles for implementing methods in JavaScript: inside/outside constructor functions using anonymous/non-anonymous functions and using prototypes. Their main goal was 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.

Fard and Mesbah [20] proposed 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 JSNode, 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 [21], [22], 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.

There is also a variety of tools and techniques for analyzing, improving, and understanding JavaScript code, including tools to prevent security attacks [23]–[25], to understand event-based interactions [26], [27], and to support refactorings [28], [29].

ECMAScript 6: ECMAScript is the standard definition of JavaScript [1]. ECMAScript 6 [7] is the next version of this standard, which is currently in *frozen state* and it is planned to be officially released in early 2015[9]. Interestingly, a syntactical support to classes is included in this new release. For example, it will support 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);
}
```

[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla, verified 11/15/2014

The strategy proposed in this paper straightforwardly detects this previous code as a `Circle` class, with a `radius` attribute and a `getArea` method, as specified in ECMAScript 6. Therefore, by revealing how JavaScript developers emulate classes in the current version of the language we can help developers that plan to use syntactical classes in their systems, according to the new ECMAScript standard. The proposed strategy and the JSCLASSFINDER tool can also support a new variety of tools, aiming to translate "old JavaScript class styles" to ECMAScript 6 syntax.

CoffeeScript is another language that aims to expose the "good parts of JavaScript" by only changing the language's syntax[10]. CoffeeScript compiles one-to-one into JavaScript code. As ECMAScript 6, the language includes class-related keywords, like `class`, `constructor`, `extends`, etc.

VI. CONCLUSION

We conclude by presenting our findings (Section VI-A) and the practical implications of this study (Section VI-B).

A. Findings

This paper provides the first 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 proposed a strategy to statically detect class emulation in JavaScript and the JSCLASSFINDER tool, that supports this strategy. We used JSCLASSFINDER on a corpus of 50 popular JavaScript applications, with different sizes and from multiple domains. 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, 26%, 36%, 30%, and 8% of the systems we studied.

Second, 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 nine systems making use of 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" [30]. Second, prototype-based inheritance is more complex than the usual implementation of inheritance available in mainstream class-based object-oriented languages.

Fourth, JavaScript classes tend to have a similar number of attributes as for example Java classes. We found that most

[10]http://coffeescript.org, verified 11/15/2014

JavaScript classes have 8 attributes or less, which is close to thresholds previously proposed for Java [19]. On the other hand, JavaScript classes tend to have less methods than Java ones. We found that most classes in JavaScript have less than seven methods, when Java classes usually have less than 16 methods. We hypothesize that this predominance of attributes over methods is due to the lack of encapsulation mechanisms in JavaScript. For example, getters and setters are almost non-existent in JavaScript software.

B. Practical Implications and Future Work

Javascript is a class-free, prototype-based language and will probably always keep this status. However, almost 40% of the systems we studied make a relevant usage of classes ($CUR \geq 0.30$). In fact, this usage may increase in the future, as ECMAScript 6 includes syntax for classes. 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 JavaScrip classes; (e) recommendation engines to suggest best practices; (f) techniques to detect violations and deviations in the class-based architecture of JavaScript systems; (g) tools to migrate to ECMAScript 6.

We also plan to: (i) compare our findings with projects written in languages that support OO natively; (ii) investigate strategies used by JavaScript frameworks to implement classes; (iii) measure precision and recall of our strategy to detect classes.

Our tools and data set are freely available at:
`http://aserg.labsoft.dcc.ufmg.br/JSClasses`

Acknowledgments

This research is supported by CNPq, CAPES, FAPEMIG and FONDECYT (project 1120094, Chile). We thank Renato Cerro for his feedback on the writing.

References

[1] "European association for standardizing information and communication systems (ECMA). ECMA-262: ECMAScript language specification. edition 5.1," 2011.
[2] H. Kienle, "It's about time to take JavaScript (more) seriously," *IEEE Software*, vol. 27, no. 3, pp. 60–62, May 2010.
[3] A. Nederlof, A. Mesbah, and A. van Deursen, "Software engineering for the web: the state of the practice," in *36th International Conference on Software Engineering (ICSE), Companion Proceedings*, 2014, pp. 4–13.
[4] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 1–12.
[5] A. H. Borning, "Classes versus prototypes in object-oriented languages," in *ACM Fall Joint Computer Conference*, 1986, pp. 36–40.
[6] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *24th European Conference on Object-Oriented Programming (ECOOP)*, 2010, pp. 126–150.
[7] "European association for standardizing information and communication systems (ECMA). ECMAScript language specification, 6th edition, draft october, 2014."
[8] W. Gama, M. Alalfi, J. Cordy, and T. Dean, "Normalizing object-oriented class styles in JavaScript," in *14th IEEE International Symposium on Web Systems Evolution (WSE)*, Sept 2012, pp. 79–83.
[9] D. Crockford, *JavaScript: The Good Parts*. O'Reilly, 2008.
[10] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly, 2011.
[11] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for JavaScript," in *19th European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 428–452.
[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[13] G. Richards, C. Hammer, B. Burg, and J. 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.
[14] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval begone!: Semi-automated removal of eval from JavaScript programs," in *27th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012, pp. 607–620.
[15] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the Shape of Java Software," in *21st Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006, pp. 397–412.
[16] P. Louridas, D. Spinellis, and V. Vlachos, "Power Laws in Software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, pp. 1–26, 2008.
[17] R. Wheeldon and S. Counsell, "Power Law Distributions in Class Relationships," in *International Working Conference on Source Code Analysis and Manipulation*, 2003, pp. 45–54.
[18] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class Corpus: A compiled version of the Qualitas Corpus," *Software Engineering Notes*, vol. 38, no. 5, pp. 1–4, 2013.
[19] P. Oliveira, M. T. Valente, and F. Lima, "Extracting relative thresholds for source code metrics," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 254–263.
[20] A. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript code smells," in *13th Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 116–125.
[21] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
[22] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.
[23] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Network and Distributed System Security Symposium (NDSS)*, 2007.
[24] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for Ajax intrusion detection," in *18th International Conference on World Wide Web (WWW)*, 2009, pp. 561–570.
[25] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *34th Symposium on Principles of Programming Languages (POPL)*, 2007, pp. 237–249.
[26] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 367–377.
[27] A. Zaidman, N. Matthijssen, M. D. Storey, and rie van Deursen, "Understanding Ajax applications by connecting client and server-side execution traces," *Empirical Software Engineering*, vol. 18, no. 2, pp. 181–218, 2013.
[28] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip, "Refactoring towards the good parts of JavaScript," in *26th Conference on Object-Oriented Programming (OOPSLA), Companion Proceedings*, 2011, pp. 189–190.
[29] ——, "Tool-supported refactoring for JavaScript," in *26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011, pp. 119–138.
[30] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.