Detecting Function Constructors in JavaScript

Shahriar Rostami, Laleh Eshkevari, Davood Mazinanian and Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
Email: {s_rostam, l_mousa, d_mazina, tsantalis}@cse.concordia.ca

***Abstract*—Prior to the recent updates of the JavaScript language specifications, developers had to use custom solutions to *emulate* constructs such as classes, modules, and namespaces in JavaScript programs. This paper introduces JSDEODORANT, an automatic approach for detecting function constructors declared locally, under a namespace, or even in other modules. The comparison with the state-of-the-art tool, *JSClassFinder*, shows that while the precision of the tools is very similar (97% and 98%), the recall of JSDEODORANT (98%) is much higher than *JSClassFinder* (61%).**

I. INTRODUCTION

JavaScript is a dynamically-typed, interpreted, prototype-based, multi-paradigm scripting language, which is one of the three fundamental building blocks of web applications (along with HTML and CSS), and ranked as the most popular programming language on GitHub [1]. In addition to functional and imperative programming paradigms, JavaScript now supports object-orientation in the traditional sense, with the recent updates to its language specifications. Prior to these updates, JavaScript developers had to use custom solutions and patterns to *emulate* constructs for which there was no syntactical support in the language, *e.g.*, classes, modules, and namespaces. In a nutshell, this was done by exploiting *function* and *object literal* constructs; a practice which mostly leads to a code-base which is difficult to understand and maintain [2], [3]. With the rapid adoption of the language [4], tool support for developing and maintaining JavaScript applications has become vital [5]. There exist tools such as JSHint, JSLint, and ESLint, which solely focus on coding style and syntax errors. Unlike other programming languages (*e.g.*, Java and C++), existing JavaScript IDEs (*e.g.*, Eclipse, WebStorm, Komodo IDE and NetBeans), have a limited support for code navigation, refactoring, reverse engineering of the architectural design, and visualization.

Our long-term goal and research vision is to fill this gap, by improving the state-of-the-art techniques and tools for developing and maintaining JavaScript programs. As the first step towards this goal, we tackle the detection of *class emulations* in JavaScript programs. Identifying functions used for emulating the behavior of classes (*i.e.*, *function constructors*) enables developers to distinguish them from normal functions, providing a better understanding of the code and its organization. It can also facilitate the automatic migration of existing JavaScript code to the new version of ECMAScript with explicit support for class declaration.

While previous works have proposed different approaches for detecting class emulations in JavaScript [2], [6], they suffer from various limitations, including but not limited to: failing to consider all different styles a class might be declared [2], or not supporting modern ways developers use to modularize JavaScript applications (*e.g.*, using namespaces) [6]. This essentially means that current approaches may fail to identify a considerable amount of function constructors.

This paper introduces JSDEODORANT [15], an Eclipse plugin that combines and extends the previous techniques to detect functions emulating class behavior in JavaScript. JSDEODORANT is able to identify module dependencies and supports code navigation from the object instantiations to their class declaration. The main contributions of this work are:

- A technique for identifying class emulations at different levels of granularity: within files, namespaces, and modules.
- An automatically-generated (and thus, unbiased) oracle that enables replication, and tool comparison.
- A comparison of our technique with the state-of-the-art tool *JSClassFinder* [6].
- A quantitative and qualitative evaluation of the results to identify our current limitations and opportunities for future improvements.

II. BACKGROUND

Traditional programming languages use constructs (*e.g.*, classes, modules) for facilitating code reuse and encapsulation. Code reuse decreases redundancy in the code, and encapsulation reduces the risk of exposing implementation details. JavaScript developers tend to use different techniques to achieve code modularity. As mentioned, the lack of syntactical support for declaring classes, modules, and namespaces in JavaScript prior to the recent updates of the language specification, forces developers to employ custom solutions and patterns to emulate them. In the following sub-sections, we briefly discuss some of the solutions introduced and promoted in the JavaScript community [7].

A. Emulating classes

In JavaScript, functions are *first-class* entities, *i.e.*, a function can be passed as an argument to other functions, returned by other functions, or stored in a variable. JavaScript developers use functions to emulate the behavior of object-oriented *classes*. In this case, JavaScript functions are used as *constructors* and *methods* of classes. Thus, we differentiate between three types of roles in which a function can serve in JavaScript, following Gama *et al.* [2]:

- **Function constructors:** In JavaScript, it is possible to define functions and variables that *belong* to another function. Such

a function is called a *function constructor*, and corresponds to the constructor of an object-oriented class. Accordingly, functions and variables defined for this function constructor are the methods and attributes of that class, respectively. As we will see, there are alternative ways to define functions and variables that belong to a function constructor. In this paper, the terms *class* and *function constructor* are used interchangeably.

- **Methods:** We refer to a function that *belongs* to a function constructor as a *method*. A method can be declared within the body of a function constructor, or added to the prototype of the function constructor.
- **Functions:** Finally, all other routines (*i.e.*, normal functions) are considered as functions.

Figure 1 demonstrates two alternative ways for defining a function constructor. In Figure 1-A, the function constructor `Clazz` contains an attribute (*i.e.*, `foo`) and a method (*i.e.*, `bar`). Note the use of the keyword `this`, which refers to the object that is created when this class is instantiated (*i.e.*, when the function constructor is invoked).

Alternatively, Figure 1-B shows a case where the function constructor is an *anonymous* function, *i.e.*, the function constructor itself does not have a name. Instead, the variable `Clazz` is assigned with the function constructor. The method `bar` is added to the *prototype* of the variable `Clazz`. In JavaScript, each object has a prototype associated with it, which is the object containing the methods and attributes that will be shared by all objects instantiated from the same class. In other words, the method `bar` will exist in other objects of the type `Clazz`, because it belongs to to `Clazz`'s prototype. Line 1 of Figure 1-C shows the instantiation of a `Clazz` object, and lines 2-3 show how to access its members.

```
1 function Clazz(){
2   this.foo = 0;
3   this.bar=function(){
4     console.log(this.foo);
5   }
6 }
```
A

```
1 var Clazz = function() {
2   this.foo = 0;
3 }
4 Clazz.prototype.bar=function(){
5   console.log(this.foo);
6 }
```
B

```
1 var instance=
      new Clazz();
2 instance.foo=2;
3 instance.bar();
  // 2
```
C

Fig. 1. Alternative ways for defining function constructors

B. Emulating namespaces

Most of the traditional programming languages provide mechanisms for grouping semantically-related concepts (*e.g.*, classes and files). Examples are *packages* and *namespaces*, that provide a better organization for the code and also help in avoiding name collisions. In JavaScript, mimicking the behavior of a namespace can be done in different ways [7]. In Figure 2, a namespace (namely, `namespace`) that has a class inside it (`PublicClass`) is created in four different ways. The class contains an attribute, `myProperty`, and its constructor prints a message to the console.

In Figure 2-A, an *object literal* construct is used to define a namespace. In JavaScript, an object literal is denoted by an open and a close curly bracket, where the functions and variables defined inside them are essentially the methods and attributes of that object, respectively. Here, the `namespace` variable is assigned with an object literal that contains a function constructor, namely `PublicClass`.

Figure 2-B and C show the use of the *immediately-invoked function expression* (IIFE) construct for simulating a namespace declaration. IIFE refers to a function that is invoked right after it is declared. In Figure 2-B, the variable `template` is first initialized with an empty object, using the object literal construct. A function constructor (`PublicClass`) is added to `template`, which will be returned by the enclosing anonymous function. As the enclosing function is called right after it is declared, the value of `namespace` will be the object `template`. Similarly, in Figure 2-C, the IIFE returns an object literal which contains the variable `PublicClass`, where `PublicClass` is assigned with a variable containing the function constructor `classDeclaration`.

In the code snippet of Figure 2-D, an empty object named `namespace` is created first. Then, an anonymous function is created that assigns the function constructor `PublicClass` to the `this` object. Then, the `apply()` function is called on the anonymous function. In JavaScript, calling `apply()` on an object that references a function, results in invoking that function with the `this` object bound to the first argument of `apply()` . Consequently, in this example, the `this` object in the anonymous function is bound to `namespace`, and therefore, `PublicClass` will belong to `namespace`.

In all the examples, `PublicClass` can be instantiated in the same way, as depicted in Figure 2-E.

C. The "Module" pattern

JavaScript used to lack a mechanism for importing classes and functions from other JavaScript files. However, the JavaScript community has proposed different approaches for defining *modules* (*i.e.*, files that can be imported to other files). The two most popular module systems are *CommonJS* [8] and *AMD* [9]. The former is designed for server-side JavaScript, while the latter is used in client-side JavaScript.

For instance, in *CommonJS*, a JavaScript module can expose all or parts of its behavior as an API, by setting the `exports` property. In Figure 3-A, the module `csv.js` exports the variable `Writer` at line eight, referencing a function constructor.

Likewise, a JavaScript module can access other modules' APIs by invoking a call to the `require` function. The function accepts a *module identifier* as its argument, and returns the exported APIs of the imported module. The module identifier can be specified by the module's name, or the path/URL to the JavaScript file where the module is defined in, or a path/URL of a folder containing a set of modules (*i.e.*, a *package*). In Figure 3-B, `usage.js` imports features that are exposed in the module `csv.js` (line two). At line three, an instance of `Writer` is created by accessing the `csvModule`'s exposed function constructor.

III. Approach

The overview of our approach for detecting function constructors in JavaScript is illustrated in Figure 4. In the following paragraphs, we explain each of the steps in detail.

Step 1: Parsing JavaScript files and building model: In the first step, JSDeodorant uses Google Closure Compiler [10] to parse and extract the ASTs of the JavaScript files in

```
1 var namespace = {
2   PublicClass : function() {
3     console.log('I am a class');
4     this.myProperty = 0;
5   }
6 }
```
A

```
1 var namespace = (function() {
2   var template = {};
3   template.PublicClass = function() {
4     console.log('I am a class');
5     this.myProperty = 0;
6   }
7   return template;
8 })();
```
B

```
1 var namespace = (function() {
2   var classDeclaration = function() {
3     console.log('I am a class');
4     this.myProperty = 0;
5   }
6   return {
7     PublicClass : classDeclaration
8   };
9 })();
```
C

```
1 var namespace = {};
2 (function() {
3   this.PublicClass = function() {
4     console.log('I am a class');
5     this.myProperty = 0;
6   }
7 }).apply(namespace);
```
D

```
var theClassInstance = new namespace.PublicClass(); // I am a class
```
E

Fig. 2. Different ways to define a namespace in JavaScript

```
1 // csv.js
2 var Writer=function(filename) {
3   // create csv file ...
4 }
5 Writer.prototype.write=function(content){
6   // write the content to file
7 }
8 module.exports.Writer = Writer;
```
A

```
1 // usage.js
2 var csvModule=require('./csv');
3 var csvWriter=
      new csvModule.Writer('test.csv');
4 csvWriter.write('hello, world!');
```
B

Fig. 3. An example illustrating the export and import in JavaScript.

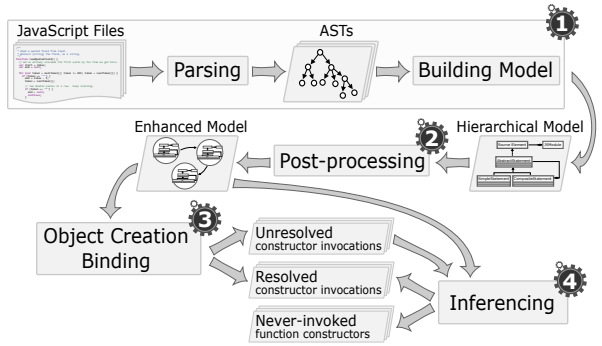

Fig. 4. Overview of the proposed approach

a given project. JSDEODORANT then traverses the generated ASTs to build a hierarchical model, that provides a higher level of abstraction, by capturing the nesting structure of the code at statement level.

Step 2: Post-processing: Once the model is built, JSDEODORANT performs a post-processing with the objective of identifying file dependencies, which results in an enhanced model with these dependencies resolved. Our model is built upon the *CommonJS* module system, as it is one of the most popular APIs used by developers. However, our approach to identify file dependencies is generic enough that can be easily extended to capture other dependency patterns, such as the AMD [9]. We identify and resolve the dependencies between JavaScript modules (*i.e.*, files) by analyzing the `require` and `exports` statements (as in Figure 3). Using the module identifier (*i.e.*, the argument of the `require` function), we build the dependencies between JavaScript modules. Moreover, if the dependencies are specified in the file `package.json` (*i.e.*, the file that contains information about the module and its dependencies in *CommonJS*), we parse it to identify the JavaScript files to be imported.

Step 3: Binding object creations: Next, JSDEODORANT performs a lightweight data-flow analysis on the enhanced model, produced in the previous step. The objective of this step is to bind each class instance creation to a function constructor, which results in two disjoint sets of resolved and unresolved function constructor invocations. JSDEODORANT traverses the model, and in each JavaScript module, it analyzes the identifier of the instance creation expressions (*i.e.*, expressions using the `new` operator). Each identifier can be either a *simple* or *composite* name.

Simple name identifier: If the identifier is a simple name (*e.g.*, `MockWindow` in `new MockWindow()`), JSDEODORANT first searches the current block to find the function declaration, where the function name matches the constructor invocation name. If no such declaration is found, it recursively searches the parent blocks until it finds a matching declaration, and marks the corresponding constructor invocation as resolved. If we get to the root of the JavaScript module and yet we are unable to identify a declaration, we search a list of predefined JavaScript functions for a match. The reason for consulting the predefined function list after our internal search (and not before) is that it is possible to redefine JavaScript native objects, although it's a risky practice [11].

Composite name identifier: Resolving composite identifiers addresses the cases where the function constructor is defined within a nested namespace or an object literal, internally or externally. That is, the declaration of the function constructor may be within the same file or elsewhere. Examples of composite names are given in Figure 2-E where the function constructor is defined in a namespace (`namespace.PublicClass`), and Figure 3-B (line three) where the class is defined in another module (`csvModule.Writer`). JSDEODORANT first splits the identifier name into its tokens (*e.g.*, [`namespace`, `PublicClass`] and then searches the current block for the leftmost token (*i.e.*, `namespace`). Statements of interest are variable assignments and property names in object literals. If it finds a match (line one in Figure 2-A), it tries to match the next token (*i.e.*, `PublicClass`) within the already matched context (*e.g.*, lines 1-6 in Figure 2-A) and this process continues until all tokens are matched.

Step 4: Inferring function constructors: If a constructor invocation cannot be resolved to a function constructor in the previous step, JSDEODORANT attempts to resolve it by using an inference mechanism. JSDEODORANT analyzes the body and prototype objects of functions; if a function defines properties or methods (either directly or through the `prototype` object), it is considered as a function constructor. Some of these identified function constructors can be matched with the unresolved constructor invocations from the previous step. The rest essentially correspond to function constructors that are nowhere instantiated in the code.

IV. EVALUATION

We designed an empirical study, with the goal of answering the following research questions:

- **RQ1:** *What is the accuracy of* JSDEODORANT *in detecting function constructors?* The goal is to assess how accurate and complete is the sample of detected function constructors. We measure precision and recall to answer this research question.
- **RQ2:** *To what extend does* JSDEODORANT *outperform the state-of-the-art tool in terms of precision and recall?* The goal is to compare the efficacy of JSDEODORANT and *JSClassFinder* in terms of precision and recall on the same dataset.

Building the dataset and oracle: To measure the precision and recall of JSDEODORANT/*JSClassFinder*, we need an oracle of function constructor declarations. We used three open-source JavaScript projects to *automatically* build the oracle (*i.e.*, to avoid any bias). There are two ways to achieve this:

Using JSDoc annotations: JavaScript developers can annotate function constructors using JSDoc's `@constructor` annotation [12]. In this case, function constructors can be automatically found by simply parsing JSDoc comments. We used the *closure-library* (a library used in Google products, such as Gmail and Maps), as it is JSDoc-annotated.

Using transpiled code: TypeScript [13] and CoffeeScript [14] are supersets of JavaScript that add the missing features (*e.g.*, syntax for class declaration) to JavaScript. The code written in their syntax is *transpiled* to vanilla JavaScript. By parsing and extracting class declarations in TypeScript/-CoffeeScript programs, class declarations can be found. The corresponding function constructors, resulting from transpiling TypeScript/CoffeeScript code, are then added to the oracle. We selected two GitHub-trending projects, *doppio* and *atom*, respectively written in TypeScript and CoffeeScript.

In Table I, we report the main characteristics of the programs used to build the oracle.

TABLE I
CHARACTERISTICS OF THE ANALYZED PROGRAMS.

Project	Version	Size (KLOC)	#JS Files	#Functions	#Function Constructors
closure-library	20160315	605	1,502	23535	946
doppio	rev:7229e7d	17.7	47	1977	154
atom	v1.7.0-beta5	34	116	3175	102

V. RESULTS

RQ1: *What is the accuracy of* JSDEODORANT *in detecting function constructors?*

As it is observed in Table II, JSDEODORANT can achieve a high precision and recall, for all three analyzed projects. We further qualitatively analyzed false positives and false negatives. For *closure-library*, 18 of the false positives are functions that are annotated with `@interface`. JSDEODORANT identified these as function constructors, since methods (with empty body) were added to their prototype. Other false positives in *closure-library* (83 cases) included functions defined in test files, without any JSDoc annotation. The first two authors of this paper independently inspected these functions; if the function was invoked as a constructor (with the `new` operator), or at least one property or method (non-empty body) was defined in its body (or added to its prototype), it was labeled as a true positive (TP). The final label was decided by unanimous voting, and a third opinion was sought in case of conflict. Out of 83 cases, 82 were labeled as TP, leaving only one case as FP. Our investigation showed that our approach is unable to identify function constructors that are not invoked, and at the same time, do not contain any property or method (*i.e.*, false negatives). Here we briefly discuss the main reasons behind the false negatives and ways to improve our technique.

TABLE II
JSDEODORANT PRECISION AND RECALL.

Program	Identified function constructors	TP	FP	FN	Precision	Recall
closure-library	1008	907	101	39	90%	96%
doppio	154	153	1	1	99%	99%
atom	106	101	5	1	95%	99%

Analyzing function's use: JSDEODORANT fails to identify two sets of non-invoked function constructors: 1) empty function constructors, and 2) function constructors with no method or property. We plan to analyze the use of these functions to improve our detection technique. For example, if a function is an operand of `instanceof`, or an argument of `Object.create()` function, it is most probably a function constructor. Another way is to infer inheritance relationships between objects. For example, Figure 5 shows an example in *closure-library* where the function constructor is not invoked in the program and it is not possible to infer from its body that the `goog.net.WebChannel.MessageEvent` is a function constructor. However, we could infer from the statement at line 183 that the function is involved in an `extends` relation with another class and thus it is a function constructor.

```
179 goog.net.WebChannel.MessageEvent = function() {
180   goog.net.WebChannel.MessageEvent.base(
181       this, 'constructor', goog.net.WebChannel.EventType.MESSAGE);
182 };
183 goog.inherits(goog.net.WebChannel.MessageEvent, goog.events.Event);
```

Fig. 5. Example of a function constructor JSDEODORANT failed to identify

Distinguishing classes from interfaces: JSDEODORANT fails to differentiate classes and interfaces. This can be mitigated by checking whether a class contains methods with an empty body, and it is not a superclass of other classes. In such a case, the detected class can be labeled as an interface.

Overall, we found that JSDEODORANT achieves high precision (95%) and recall (98%).

RQ2: *To what extend does* JSDEODORANT *outperform the state-of-the-art tool in terms of precision and recall?*

We run *JSClassFinder* on the same dataset we used for our first research question, summarizing the results in Table III. *JSClassFinder* crashed on two sub-directories of closure-library (`/closure/goog/crypt` and `/closure/goog/i18n`), due to *invalid input format* and *out of memory* exceptions, respectively. Thus, we excluded the function constructors (27 cases) in those two sub-directories, to calculate precision and recall in a fair manner. A manual analysis of the false positives was done similarly to **RQ1**, and out of 42 cases, 41 were labeled as TP. All false negatives were cases where function constructors were not invoked, which is one of the major limitations of *JSClassFinder*.

TABLE III
JSClassFinder RESULTS OF DETECTION.

Project	Oracle	Identified function constructors	TP	FP	FN	Precision	Recall
closure-library	919†	769	727	42	192	94%	79%
doppio	154	23	22	1	131	96%	14%
atom	102	95	95	0	7	100%	93%

† Function constructors in two sub-directories of closure-library are removed from the oracle.

To compare the performance of the tools, we enhanced our automatically-built oracle by adding those cases labeled as TP during our qualitative analysis, and removed the function constructors belonging to the sub-directories in which *JSClassFinder* crashed from the oracle and the results of JSDEODORANT. In Table IV, the accuracy measures for both tools are reported. As it is observed, *JSClassFinder* had very good precision and recall for *atom*. This is because all function constructors were invoked in *atom*, and thus, *JSClassFinder* identified all of them. While in the other two projects, it fails to identify function constructors that are not invoked.

TABLE IV
ACCURACY MEASURES FOR BOTH TOOLS

Project	*JSClassFinder*		JSDEODORANT	
	Precision	Recall	Precision	Recall
closure-library	99%	76%	98%	96%
doppio	96%	14%	99%	99%
atom	100%	93%	95%	99%
Average	**98%**	**61%**	**97%**	**98%**

Overall, we found that both tools have very high precision on average, while JSDEODORANT greatly outperforms *JSClassFinder* with respect to recall.

VI. THREATS TO VALIDITY

Construct validity: We only addressed module dependencies that comply with `CommonJS`; thus, we need to replicate this study with JavaScript projects that use other module systems, *e.g.*, `AMD`. The post-processing step can be easily extended to support other module systems. Since the core of the approach does not depend on any module system, we are confident that JSDEODORANT will perform similarly well.

In addition, our lightweight data-flow analysis only supports simple name aliasing, and thus, more complex cases of name aliasing (*e.g.*, variable name in a return of a function call) is left as future work.

External validity: While these preliminary results of analyzing three projects from different domains show that JSDEODORANT detects function constructors with a high accuracy, we acknowledge the need for examining a wider range of JavaScript projects. To enable the reproduction of our study, the oracle along with the detected classes for both tools are available on-line [15].

VII. RELATED WORK

Previous works [2], [6] proposed different techniques for identifying class definitions in JavaScript. Gama *et al.* proposed an approach to identify five different styles of class declarations, with the goal of normalizing them to one unique style [2]. However, their approach fails to identify cases where the class definition does not comply with their supported styles (*e.g.*, 13% of the classes in one of the programs analyzed in this paper does not comply with their styles). Silva *et al.* [6] proposed an automatic approach, *JSClassFinder*, which identifies class definitions by analyzing the instance creation statements (statements using the `new` operator). One of the limitations of *JSClassFinder* is the lack of support for classes defined in nested namespaces (composite names). Moreover, classes are matched by their names regardless of the file dependencies, which might lead to incorrect matches when different files contain classes with the same name. Finally, their approach fails to identify classes that are not instantiated.

VIII. CONCLUSION AND FUTURE WORK

This paper introduces JSDEODORANT, an Eclipse plugin to detect class emulations in JavaScript programs. JSDEODORANT analyzes class instance creation statements as well as the bodies of functions to detect function constructors. The results of our study on three open-source programs show that JSDEODORANT detects function constructors with high accuracy (on average, 95% precision and 98% recall). Moreover, a comparison with the state-of-the-art tool *JSClassFinder* showed that while both tools reach high precision (97% and 98%, respectively), the recall of JSDEODORANT (98%) is much higher than *JSClassFinder* (61%).

Future work will focus on tackling the limitations discussed in Section V, as well as evaluating JSDEODORANT on a wider range of JavaScript open-source programs. Moreover, we plan to perform an empirical study exploring the use of classes over the evolution of JavaScript projects. In the long run, we plan to enhance and advance the support for code smell detection and refactoring in JavaScript programs.

REFERENCES

[1] A. La. Language trends on github. [Online]. Available: https://github.com/blog/2047-language-trends-on-github

[2] 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*, 2012, pp. 79–83.

[3] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript Code Smells," in *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, 2013, pp. 116–125.

[4] 2015 Developer Survey. [Online]. Available: http://stackoverflow.com/research/developer-survey-2015

[5] A. Mesbah, "Software Analysis for the Web: Achievements and Prospects," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) – FoSE Track (invited)*, 2016.

[6] L. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil, "Does JavaScript software embrace classes?" in *Proceedings of the 22nd International Conference of Software Analysis, Evolution and Reengineering (SANER)*, 2015, pp. 73–82.

[7] A. Osmani, *Learning JavaScript Design Patterns - a JavaScript and jQuery Developer's Guide.* O'Reilly Media, 2012.

[8] CommonJS. [Online]. Available: http://www.CommonJS.org/

[9] Asynchronous Module Definition. [Online]. Available: https://github.com/amdjs/amdjs-api/blob/master/AMD.md

[10] Closure Compiler. [Online]. Available: https://developers.google.com/closure/compiler/

[11] Extending builtin natives. Evil or not? [Online]. Available: http://perfectionkills.com/extending-native-builtins/

[12] JSDOC. [Online]. Available: http://usejsdoc.org/

[13] TypeScript. [Online]. Available: https://www.typescriptlang.org/

[14] CoffeeScript. [Online]. Available: http://coffeescript.org/

[15] JSDeodorant. [Online]. Available: https://github.com/sshishe/jsdeodorant