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, JClassFinder, shows that while the precision of the tools is very similar (97% and 98%), the recall of JSDeodorant (98%) is much higher than JClassFinder (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 JClassFinder [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 `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.

```
function Clazz() {
  var foo = 0;
  constructor(this);
}

Clazz.prototype.bar = function() {
  this.bar = 0;
}
```

```
var instance = new Clazz();
instance.bar();
```

In the code snippet of Figure 2-D, an empty object named `Object` is assigned with a variable containing 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
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 JSEDeodorant 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 extent does JSEDeodorant outperform the state-of-the-art tool in terms of precision and recall? The goal is to compare the efficacy of JSEDeodorant and JSClassFinder in terms of precision and recall on the same dataset.

**Building the dataset and oracle**: To measure the precision and recall of JSEDeodorant/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, are then added to the oracle.

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

**TABLE I**

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>Size (KLOC)</th>
<th>JS Files</th>
<th>#Functions</th>
<th>#Function Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure-library</td>
<td>20160315</td>
<td>235</td>
<td>505</td>
<td>35</td>
<td>82</td>
</tr>
<tr>
<td>doppio</td>
<td>rev:7229738</td>
<td>154</td>
<td>38</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>atom</td>
<td>v1.7.8beta5</td>
<td>235</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**V. RESULTS**

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

As it is observed in Table II, JSEDeodorant 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`. JSEDeodorant 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.

**Analyzing function’s use**: JSEDeodorant 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.

**Distinguishing classes from interfaces**: JSEDeodorant 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 JSEDeodorant achieves high precision (95%) and recall (98%).

**RQ2**: To what extent does JSEDeodorant 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.
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>
<thead>
<tr>
<th>Project</th>
<th>Identified function constructors</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure-library</td>
<td>9191</td>
<td>769</td>
<td>722</td>
<td>1</td>
<td>94%</td>
<td>79%</td>
</tr>
<tr>
<td>doppio</td>
<td>154</td>
<td>23</td>
<td>22</td>
<td>1</td>
<td>96%</td>
<td>14%</td>
</tr>
<tr>
<td>atom</td>
<td>102</td>
<td>95</td>
<td>95</td>
<td>0</td>
<td>100%</td>
<td>93%</td>
</tr>
</tbody>
</table>

Table III: RESULTS OF DETECTION.

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. [2] proposed an approach to identify five different styles of class declarations, with the goal of normalizing them to one unique style. 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


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>
<thead>
<tr>
<th>Project</th>
<th>JSClassFinder</th>
<th>JSDEODORANT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>closure-library</td>
<td>99%</td>
<td>76%</td>
</tr>
<tr>
<td>doppio</td>
<td>96%</td>
<td>14%</td>
</tr>
<tr>
<td>atom</td>
<td>100%</td>
<td>95%</td>
</tr>
<tr>
<td>Average</td>
<td>98%</td>
<td>61%</td>
</tr>
</tbody>
</table>

Table IV: ACCURACY MEASURES FOR BOTH TOOLS