JSClassFinder: A Tool to Detect Class-like Structures in JavaScript

Leonardo Humberto Silva, Daniel Hovadick, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, Anne Etien

JSClassFinder: A Tool to Detect Class-like Structures in JavaScript

Leonardo Humberto Silva[1], Daniel Hovadick[2], Marco Tulio Valente[2],
Alexandre Bergel[3],Nicolas Anquetil[4], Anne Etien[4]

[1]Department of Computing – Federal Institute of Northern Minas Gerais (IFNMG)
Salinas – MG – Brazil

[2]Department of Computer Science – Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

[3]Department of Computer Science – Pleiad Lab
University of Chile – Santiago – Chile

[4]RMoD Project-Team – INRIA Lille Nord Europe – France

leonardo.silva@ifnmg.edu.br, {dfelix,mtov}@dcc.ufmg.br,
abergel@dcc.uchile.cl, {nicolas.anquetil,anne.etien}@inria.fr

Abstract. *With the increasing usage of JavaScript in web applications, there is a great demand to write JavaScript code that is reliable and maintainable. To achieve these goals, classes can be emulated in the current JavaScript standard version. In this paper, we propose a reengineering tool to identify such class-like structures and to create an object-oriented model based on JavaScript source code. The tool has a parser that loads the AST (Abstract Syntax Tree) of a JavaScript application to model its structure. It is also integrated with the Moose platform to provide powerful visualization, e.g., UML diagram and Distribution Maps, and well-known metric values for software analysis. We also provide some examples with real JavaScript applications to evaluate the tool.*

Video: `http://youtu.be/FadYE_FDVM0`

1. Introduction

JavaScript is a loosely-typed dynamic language with first-class functions. It supports object-oriented, imperative, and functional programming styles. Behaviour reuse is performed by cloning existing objects that serve as prototypes [Guha et al. 2010]. ECMAScript [ecm 2011] is a scripting language, standardized by ECMA International, that forms the base for the JavaScript implementation. ECMAScript 5 (ES5) is the version currently supported by most browsers. Version 6 of the standard is planned to be officially released around mid 2015[1].

Due to the increasing usage of JavaScript in web applications, there is a great demand to write JavaScript code that is reliable and maintainable. In a recent empirical study, we found that many developers emulate object-oriented classes to implement parts of their systems [Silva et al. 2015]. However, to the best of our knowledge, none of the

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript, verified 05/18/2015

existing tools for software analysis of JavaScript systems identify object-oriented entities, such as classes, methods, and attributes.

In this paper, we propose and describe the JSClassFinder reengineering tool that identifies class-like structures and creates object-oriented models based on JavaScript source code. Although ES5 has no specific syntax for class declaration, JSClassFinder is able to identify structures that emulate classes in a system. Prototype-based relationships among such structures are also identified to infer inheritance. The resulting models are integrated with Moose[2], which is a platform for software and data analysis. The main features of the proposed tool are:

- Identification of class-like entities to build an object-oriented model of a system.
- Integration with a complete platform for software analysis.
- Graphical visualization of the retrieved class-like structures, using UML class diagrams, distribution maps, and tree view layouts.
- Automatic computation of widely known source code metrics, such as number of classes (NOC), number of methods (NOM), number of attributes (NOA), and depth of inheritance tree (DIT).

This tool paper is organized as follows. Section 2 describes JSClassFinder's architecture. Section 3 uses one toy example and two real applications to demonstrate the tool. Section 4 describes exceptions that are not covered by the tool. Section 5 presents related work and Section 6 concludes the paper.

2. JSClassFinder in a Nutshell

The execution of JSClassFinder is divided into two stages: preprocessing and visualization. The preprocessing is responsible for analyzing the AST of the source code, identifying class-like entities, and creating an object-oriented model that represents the code. In the visualization stage, the user can interact with the tool to visualize the model and to inspect all metrics and visualization features that the software analysis platform provides.

JSClassFinder is implemented in Pharo[3], which is a complete Smalltalk environment for developing and executing object-oriented code. Pharo also offers strong live programming features such as immediate object manipulation, live update, and hot recompilation. The system requirements to execute JSClassFinder are: (i) the AST of a JavaScript source code in JSON format; (ii) A Pharo image with JSClassFinder. A ready-to-use Pharo image is available at the JSClassFinder website[4]. Figure 1 shows the architecture of the JSClassFinder, which includes the following modules:

AST Parser: This module receives as an input the AST of a JavaScript application. Then it creates a JavaScript model as part of the preprocessing stage. Currently, we are using Esprima[5], a ECMAScript parser, to generate the AST in JSON format.

Class Detector: This module is responsible for identifying class-like entities in the JavaScript model. It is the last step of the preprocessing stage, when an object-oriented model of an application is created and made available to the user.

[2]http://www.moosetechnology.org/, verified 05/18/2015

[3]http://pharo.org/, verified 05/18/2015

[4]http://aserg.labsoft.dcc.ufmg.br/jsclasses/, verified 05/18/2015

[5]http://esprima.org/, verified 05/18/2015

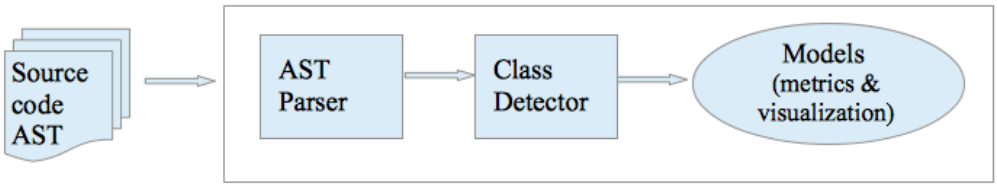

Figure 1. JSClassFinder's architecture

Models (metrics & visualization): This module provides visualizations for an user to interact with the tool and to "navigate" by the application's model. All information about classes, methods, attributes, and inheritance relationships is available. The main visualizations provided are: UML class diagram, distribution maps [Ducasse et al. 2006], and tree views. For the metrics, the tool provides the total number of classes (NOC) and, for each class: number of methods (NOM), number of attributes (NOA), number of children (subclasses) and depth of inheritance tree (DIT).

Figure 2 shows the main browser of JSClassFinder's user interface. In the top menu, the user can load a new JavaScript application or open one existing model, previously loaded. The only information required to load new applications are: (i) application's name and (ii) root directory where the JSON files with the target system's AST are located. After the preprocessing stage, the tool opens a new model inside a panel, where the user can navigate through class entities and select any graphical visualizations or metrics available. Section 3 presents some examples of what the user can do once they have a model created.

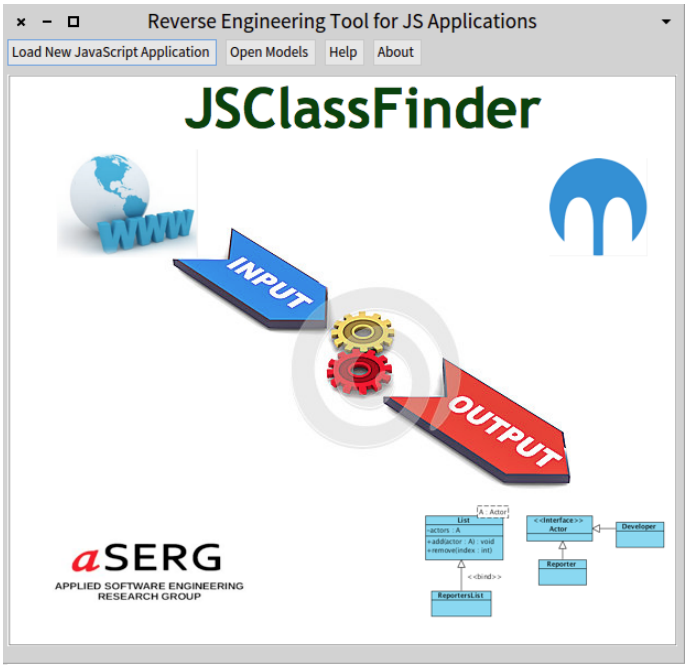

Figure 2. JSClassFinder initial user interface

2.1. Strategy for class detection

In this section, we describe the strategy used by JSClassFinder to represent the way classes are created in JavaScript and the way they acquire fields and methods. A detailed description is provided in a conference paper [Silva et al. 2015].

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 of type `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}`.

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

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

3. Examples

This section shows examples of usage of JSClassFinder to analyze one toy example and two real JavaScript applications extracted from GitHub.

3.1. Toy Example

This example includes two simple classes, `Mammal` and `Cat`, to illustrate how classes can be emulated in JavaScript. Listing 1 presents the function that defines the class `Mammal` (lines 1-3), which includes an attribute `name`. This class also has a method named `toString` (lines 4-6), represented by a function which is associated to the prototype of `Mammal`. Line 7 indicates that the class `Cat` (lines 9-11) inherits from the prototype of `Mammal`. The usage of variables `animal` and `myPet` demonstrate how the classes can be instantiated and used (lines 12-13).

Listing 1 represents one way of defining classes and instantiating objects. There are some variations and customized implementations, as we can find in [Flanagan 2011, Crockford 2008, Gama et al. 2012]. JSClassFinder supports different types of class implementation, as reported in [Silva et al. 2015].

3.2. Algorithms.js

`Algorithms.js`[6] is an open source project that offers traditional algorithms and data structures implemented in JavaScript. We analyzed version 0.8.1, which has 3,263 LOC.

[6]https://github.com/felipernb/algorithms.js/ verified 05/18/2015

```
function Mammal(name) {
  this.name=name;
}
Mammal.prototype.toString=function(){
  return '['+this.name+']';
}
Cat.prototype = Object.create(Mammal.prototype);  // Inheritance
...
function Cat(name) {
  this.name='"meow" ' + name;
}
var animal = new Mammal('Mr. Donalds');
var myPet = new Cat('Felix');
```

Listing 1. Class declaration and object instantiation

Figure 3 shows the class diagram of Algorithms.js, generated automatically by JSClassFinder, after the preprocessing stage. The classes represent the data structures that are supported by Algorithms.js, as we can check in the project's documentation webpage[7]. The main algorithms, e.g., Dijkstra, EulerPath, Quicksort, that the application offers in its API, are implemented as JavaScript global functions, not classes.

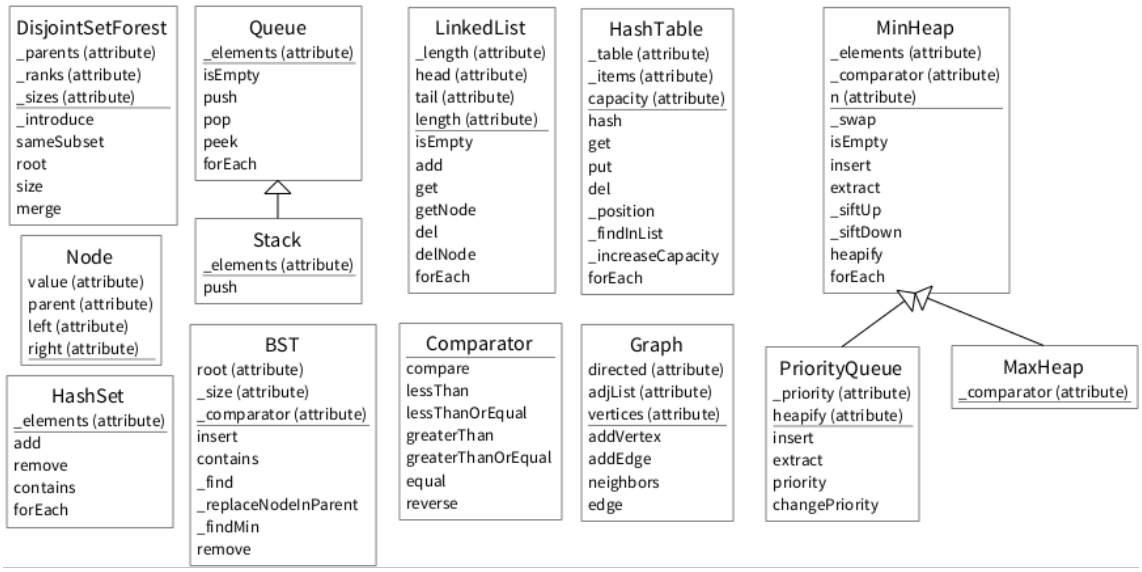

Figure 3. UML class diagram for Algorithms.js

3.3. PDF.js

PDF.js[8] is a Portable Document Format (PDF) viewer that is built with HTML5. It is a community-driven project supported by Mozilla Labs. We analyzed version 1.1.1, which has 57,359 LOC, 182 classes, 947 methods, and 876 attributes. Users can interact with the model to access all visualization features and metric values. It is also possible to use drill-down and drill-up operations when one entity is selected. For example, when the user performs a click on the number of classes ("All classes") metric, the panel will drill-down to show a list with all the classes. If the user performs a click on one of the classes,

[7] http://algorithmsjs.org/ verified 05/18/2015

[8] https://github.com/mozilla/pdf.js verified 05/18/2015

the tool will show all information related to the class, i.e., methods, attributes, subclasses, and superclasses. If the user performs a click on “All methods”, the panel will drill-down again to show a list with all the methods, etc. A menu with all visualization options and diagrams is shown when the user performs a right click on a given element.

JavaScript does not define language constructs for modules or packages. Therefore, a module can be a single file of JavaScript code that might contain a class definition, a set of related classes, a library of utility functions, or just a script of code to execute [Flanagan 2011]. JSClassFinder uses source code files as packages to allow the visualization of distribution maps per packages, like in the example shown in Figure 4. When a user selects the distribution map option, it is possible to choose which parameter will be exposed in the diagram. In this case, packages are represented by external rectangles and the small blue squares are classes. It is also possible to change the colors and to establish a valid range to be considered. For example, users can inform that the diagram should use red squares and consider only classes with more than five methods. This feature can be used, for example, to easily locate the biggest classes in a system.

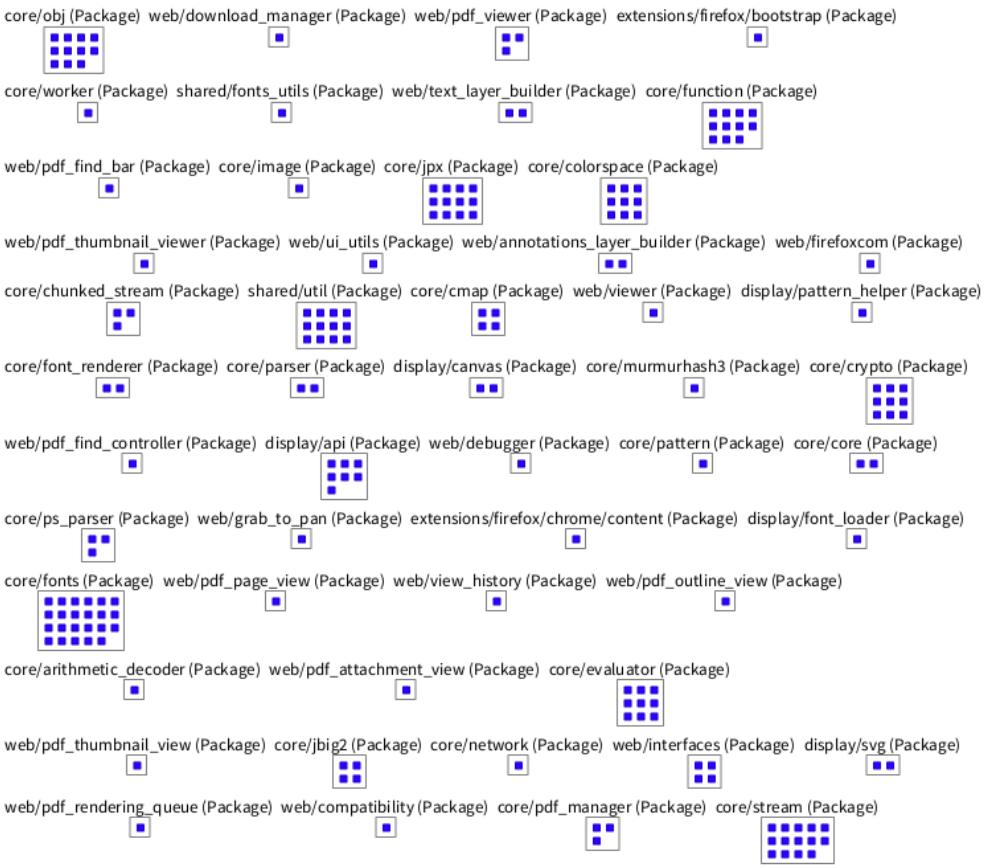

Figure 4. Distribution map for PDF.js (small squares are classes)

4. Limitations

As mentioned before, there are different ways to emulate classes and inheritance in JavaScript. The following specific cases are not covered by our current implementation:

Use of third-party libraries with customized object factories. For example, ClazzJS[9] is a portable JavaScript library for class-style OOP programming. It provides a DSL for defining classes. Our tool is not able to detect classes nor inheritance relationships implemented with this particular DSL. One idea to make JSClassFinder more robust against this kind of limitation is to gather the most common libraries that provide such service and implement specific strategies to identify them.

Use of singleton objects. Objects implemented directly, without using any class-like constructor functions, are not considered classes. Listing 2 shows the implementation of one singleton. Although this kind of object is not considered a class, it can be used to compose and clone other objects.

```
1  var person = {  firstName:"John",
2          lastName:"Doe",
3          birthDate: "01-01-2000",
4          getAge: function () { ... }
5        };
```

Listing 2. Singleton object example

Use of properties bound to variables that are functions. If a class constructor has a property that receives a variable, it is identified as an attribute, even if this variable holds a function. It occurs because JavaScript is a dynamic and loosely typed language, and JSClassFinder relies on static analysis.

5. Related Work

There is an increasing interest on JavaScript software engineering research. For example, JSNose is a tool for detecting code smells based on a combination of static and dynamic analysis [Fard and Mesbah 2013]. One of the code smells detected by JSNose, Refused Bequest, refers to subclasses that use only some of the methods and properties inherited from its parents. Differently, JSClassFinder provides models, visualizations, and metrics about the object-oriented portion of a JavaScript system, including inheritance relationships. Although JSClassFinder is not specifically designed for code smells detection, information provided by our tool can be used for this purpose.

Clematis [Alimadadi et al. 2014] and FireDetective [Zaidman et al. 2013] are tools for understanding event-based interactions, based on dynamic analysis. Their main goal is to reveal the control flow of events during the execution of JavaScript applications, and their interactions, in the form of behavioral models. In contrast, JSClassFinder aims the production of structural models.

ECMAScript definition, in its next version ES6 [ecm 2014], provides support for class definition. ES6 offers a proper syntax for creating classes and inheritance, similar to the syntax used in some traditional object-oriented languages, such as Java. Since ES6 uses specific keywords in the new syntax, it will be simple to adapt JSClassFinder once the new standard is released. Moreover, JSClassFinder could help on the migration of legacy JavaScript code to the new syntax supported by ECMAScript 6.

[9]https://github.com/alexpods/ClazzJS verified on 05/18/2015

6. Conclusions and Future Work

In this paper we proposed a reengineering tool that supports the identification of class-like structures and the creation of object-oriented models for JavaScript applications. The users do not need to have any prior knowledge about the structure of a system in order to build its model. It is possible to interact with the tool to obtain metric data and visual analysis about the object-oriented portion of JavaScript systems. As future work, we plan to extend JSClassFinder with three major features: (i) support for the new ECMAScript 6 standard, (ii) support to coupling information between classes, (iii) support to the computation of metric thresholds for JavaScript class-like structures [Oliveira et al. 2014].

JSClassFinder is publicly available at: **http://aserg.labsoft.dcc.ufmg.br/jsclasses/**.

Acknowledgment: This work was supported by FAPEMIG, CAPES and INRIA.

References

(2011). European association for standardizing information and communication systems (ECMA). ECMA-262: ECMAScript language specification. Edition 5.1.

(2014). European association for standardizing information and communication systems (ECMA). ECMAScript language specification, 6th edition, draft October.

Alimadadi, S., Sequeira, S., Mesbah, A., and Pattabiraman, K. (2014). Understanding JavaScript event-based interactions. In *International Conference on Software Engineering (ICSE)*, pages 367–377.

Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly.

Ducasse, S., Gîrba, T., and Kuhn, A. (2006). Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 203–212.

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

Flanagan, D. (2011). *JavaScript: The Definitive Guide*. O'Reilly.

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

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of JavaScript. In *24th European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150.

Oliveira, P., Valente, M. T., and Lima, F. (2014). Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263.

Silva, L. H., Ramos, M., Valente, M. T., Bergel, A., and Anquetil, N. (2015). Does JavaScript software embrace classes? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 73–82.

Zaidman, A., Matthijssen, N., Storey, M. D., and rie van Deursen (2013). Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218.