JSDeodorant: Class-awareness for JavaScript programs

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

***Abstract*—Until the recent updates to JavaScript specifications, adding syntactical support for class and namespace declaration, developers used custom solutions to emulate modular decomposition (*e.g.*, classes and namespaces) and other object-oriented constructs, such as interfaces, and inheritance relationships. However, the lack of standards for several years led to a large variation and diversity of custom solutions for emulating object-oriented constructs, making maintenance and comprehension activities rather difficult in JavaScript projects developed based on the previous language specifications. In this paper, we present JSDEODORANT, an Eclipse plug-in that enables class-aware maintenance and comprehension for JavaScript programs. (https://youtu.be/k4U2LwkL6JU)**

I. INTRODUCTION

JavaScript is a dynamically-typed language, which supports procedural, functional, and prototypical object-oriented programming paradigms. Although having all these characteristics in one language makes it undoubtedly powerful, JavaScript programs are challenging to develop, debug, and maintain. This could be partly because, despite its extensive popularity, tool support for JavaScript is not comparable to that of *traditional* languages, *e.g.*, Java or C++ [1]. Popular tools such as JSHint, JSLint, and ESLint, aid JavaScript developers only in detecting coding style issues and syntax errors. The state-of-the-art IDEs provide limited support for code development and maintenance. For instance, the Eclipse JavaScript Development Tools (JSDT) offer limited code navigation, syntax highlighting, code completion, and trivial quick-fixes. JetBrains WebStorm, one of the most well-known commercial IDEs designed specially for JavaScript, additionally supports some basic refactorings, such as Extract and Inline function.

However, there are still several opportunities for further enhancements in tool support. One such example is the automatic detection of code fragments where developers *emulate* object-oriented constructs in JavaScript projects. Indeed, prior to the recent updates on the language's specifications, developers had to use custom solutions [2] to emulate modular decomposition constructs (*e.g.*, classes and namespaces), as well as native OOP constructs, such as interfaces and inheritance relations. This lack of standards led to a large variety of patterns and practices, each one offering different advantages. As a matter of fact, in a previous work [3], we documented 4 popular patterns for emulating class declarations, 5 patterns for emulating namespaces, and 3 module patterns for importing/exporting classes and functions, which can all be used in combination. Although the latest JavaScript specifications provide syntactical support for class and namespace declarations, there is still a large number of programs developed and maintained in the older versions of JavaScript, where object-oriented constructs are inevitably emulated. Moreover, not all practitioners are convinced to migrate their code to the latest ECMAScript specifications (*i.e.*, ES6). Indisputably, maintaining a program without support for recognizing its building blocks (modules, namespaces, classes, and interfaces) is extremely challenging.

In a recent work, Silva *et al.* [4] introduced JSClassFinder, which detects class and inheritance emulations. However, it suffers from several limitations related to scalability, accuracy and usability. We discussed these limitations in our previous work [3], and introduced JSDEODORANT, an automatic approach for detecting function constructors (*i.e.*, class declarations) with high precision and recall. In this work, we extend JSDEODORANT, by adding support for the identification of inheritance relations, and implement an Eclipse plug-in that brings several useful features, including class-aware code navigation and visualization. We also provide a comprehensive comparison of class-awareness related features among JSDEODORANT, Eclipse JSDT, JetBrains WebStorm, and JSClassFinder.

II. IMPLEMENTATION

JSDEODORANT has been developed as an IDE-independent tool suite for analyzing JavaScript projects. In its core, it stores a hierarchical object model representing the structure of a given JavaScript project, built from the Abstract Syntax Trees (AST) of the JavaScript files. The model obscures redundant AST details, yet it is detailed enough to allow implementing various analysis algorithms on top of it. At the highest level, the model captures JavaScript module dependency information. A module consists of a set of related JavaScript classes. JSDEODORANT supports two common *de facto* standards for defining modules in JavaScript (CommmonJS and Google Closure Library) [3], and it is extensible enough to support other standards (*e.g.*, AMD). In a nutshell, to identify function constructors (hereafter, classes), JSDEODORANT first analyzes object creation expressions (*i.e.*, usages of the `new` keyword) and performs a lightweight data flow analysis to bind each class instantiation expression to its corresponding class declaration. This approach labels functions as classes when there is an explicit class instantiation. JSDEODORANT then employs an inference mechanism to identify classes which are not

instantiated in the project. For that, JSDEODORANT evaluates the bodies of the remaining unlabeled function declarations, in addition to their *prototype* objects to infer whether a function actually mimics a class [3].

Identifying class hierarchies. The lack of syntactical support for object-oriented constructs forces JavaScript developers to emulate inheritance relations between classes using several different patterns. JSDEODORANT currently detects the three following popular patterns:

1) Initializing the prototype of the subtype with the prototype of its super type:

```
subType.prototype = Object.create(superType.prototype);
```

2) Using user-defined functions, such as `extends()` or `inherits()`, which internally implement the first pattern:

```
extends(subType, superType);
inherits(subType, superType);
```

3) Calling the constructor of the super type within the body of the subtype, using the `call` or `apply` functions. In JavaScript, these functions allow changing the context of a function, *i.e.*, the binding of the `this` object inside the body of the function on which the `call` or `apply` is called. In the following code snippet, for instance, the `superType` function is called, and `this` points to the `subType`'s context:

```
function subType(arg1, agr2, ..){
  superType.call(this, arg1, arg2, .. );
  // some code
}
```

III. TOOL FEATURES

A. File Outline

In JavaScript, attributes and methods can be declared in the body of an emulated class declaration, or added to its prototype (see the example below).

```
helma.Color = function(R, G, B) {
    var value = null;
    // some code
    this.getName = function() {
        return helma.Color.COLORVALUES[value];
    };
    // some code
}
helma.Color.prototype.fromName = function(name) {
    // some code
};
```

The advantages of adding a method to the prototype (`fromName` in the example) are: 1) it is stored in the memory once and shared among all instances of the class, and 2) it is possible to override the method in the derived class, and still invoke the method of the base class. However, methods defined in the body of a class have full access to the private members of the class, *i.e.*, local variables and functions (*e.g.*, function `getName` in the example has access to `value`). Thus, depending on the situation, the developer may favor one style over the other. To identify class members, JSDEODORANT evaluates the assignment statements within and outside the body of the class. The identified class members are displayed in the Modules View, which outlines the structure of a JavaScript module (see Figure 1).

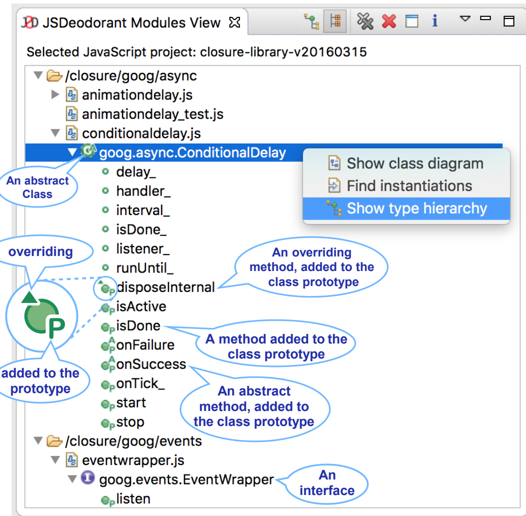

Fig. 1. JSDEODORANT file outline structure.

As mentioned, interfaces, abstract classes/methods, inheritance, and polymorphism are also emulated in JavaScript. An abstract method, for instance, is emulated by providing an empty method, or a method that logs or throws a "not implemented" error message [5], so that the subclasses are forced to implement it. JSDEODORANT recognizes abstract, overridden, and overriding methods, as shown in Figure 1. Class/interface members are decorated with descriptive icons.

Maintenance Scenario: Consider a developer who needs to perform a maintenance task on `conditionaldelay.js` (Figure 1). She can see that the file contains a class, *i.e.*, `goog.async.ConditionalDelay`. She can infer that this is a derived class (from the overriding icon) with all its methods added to the class's prototype. The Modules View enables her to have an abstract view of the JavaScript module and easily navigate to the source code of the class/interface or its members, by double clicking on the program elements.

B. Type Hierarchy

As mentioned, JavaScript developers apply various patterns to emulate inheritance (Section II), and JSDEODORANT supports three of the most common ones. The developer can view the type hierarchy of the class easily by a) right clicking on the class in the Modules View and selecting "Show type hierarchy" from the pop-up menu (Figure 1), or b) by hovering on the class name in the editor and selecting "JSDeodorant: open type hierarchy" from the pop-up menu (Figure 2). In both cases, the type hierarchy will be presented to the developer in the Modules View (Figure 3).

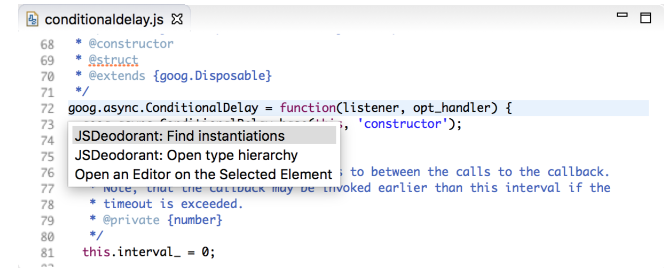

Fig. 2. JSDEODORANT type hierarchy from the editor view.

Maintenance Scenario: In the previous example, let us assume that after inferring from the file outline that the class

ConditionalDelay has a supertype, the developer decides to further inspect the type hierarchy of this particular class. By using the aforementioned navigation mechanisms, she can find the supertype class goog.Disposable, and navigate to its declaration by double-clicking on the supertype. Moreover, the Type Hierarchy View (Figure 3) facilitates the inspection of its sibling subtypes, and viewing all type declarations (classes and interfaces) in a tree structure for the entire JavaScript project. Finally, this view can assist the developer to capture the depth of the inheritance tree for a particular class.

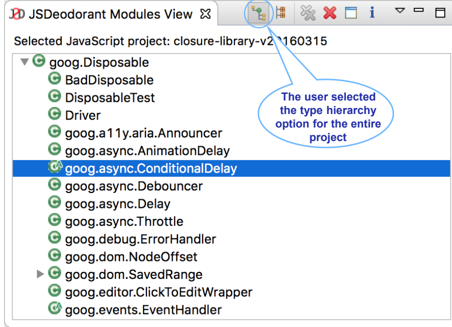

Fig. 3. JSDEODORANT type hierarchy view.

C. *Class-aware Code Navigation*

In [3] we presented our algorithm to detect class emulations in JavaScript programs. JSDEODORANT identifies and binds the class instance creation expressions to the class declaration, and thus enables class-aware code navigation from a class declaration to its instances and vice versa. Using JSDEODORANT, the developer can easily find the instantiations by a) hovering over the class name in the editor and selecting "JSDeodorant: find instantiations" from the pop-up menu (Figure 2), or b) right-clicking on the class in the Modules View and selecting "Find instantiations" from the pop-up menu (Figure 1). In both cases, JSDEODORANT provides the results in the Instantiations View (Figure 4), where the developer can double click on any of the expressions to navigate to the corresponding source code fragment. Class-aware code navigation is also available for navigating from an instantiation expression to the class declaration (Figure 5).

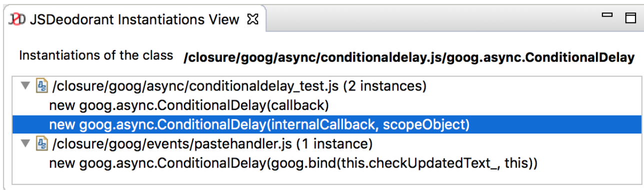

Fig. 4. JSDEODORANT class instantiation results.

Maintenance Scenario: Let us assume that the developer has to add (or remove) a parameter to the function constructor of class ConditionalDelay. Before modifying the function constructor's signature, she needs to identify all instantiations of this particular class in the project to perform the necessary changes in order to add (or remove) the argument corresponding to the parameter.

D. *Visualization*

JSDEODORANT visualizes the module dependencies, as well as the UML class diagram for a selected class. Module

Fig. 5. Navigation from an instantiation expression to the class declaration.

dependencies are visualized in a similar manner to UML package diagrams to provide an architectural view of the JavaScript project. The class diagram illustrates the class structure (attributes, methods) and inheritance relations. The developer can view the class diagram by right clicking on a class name in the Modules View and selecting "Show Class Diagram" from the pop-up menu. Figure 6 shows an example of the class diagram for class Shape. The developer can navigate to the source code where the elements are declared by double clicking on a class, attribute, or method. Moreover, the developer can easily navigate to the Type Hierarchy or Instantiations views by clicking on the corresponding buttons.

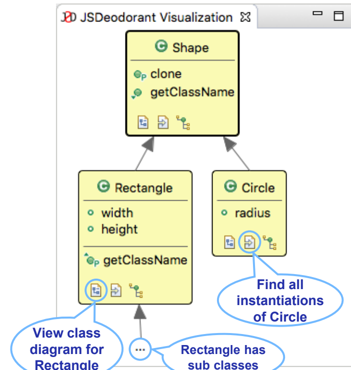

Fig. 6. JSDEODORANT Class Diagram View.

IV. VALIDATION

A. *Accuracy*

We evaluated JSDEODORANT's accuracy in the detection of inheritance relations in two open-source JavaScript projects. We selected the projects shown in Table I, since both use JSDoc annotations, allowing to build an unbiased oracle by finding all @extends JSDoc tags.

TABLE I
ACCURACY IN THE DETECTION OF INHERITANCE RELATIONS.

Program	Size (KLOC)	Identified relations	TP	FP	FN	Precision	Recall
closure-library-v20160315	605.3	643	643	0	120	100%	97%
Helma-1.7.0	31.5	29	29	0	2	100%	93%

Note that some of the classes declared in the test files of these projects were not annotated, and thus are not in the oracle, while JSDEODORANT identified the inheritance relations for them. Two authors of this paper independently inspected the source code and labeled the detected relations as true positives (TP) when they could infer an actual inheritance relation. The final labels were determined unanimously, and a third opinion was sought in case of conflicts. For the false negatives (FN), we observed that although some classes were annotated with @extends tags, the developers did not implement the inheritance relation in the code. We refer the reader to [3] for the accuracy of class detection.

B. Tool comparison

Table II summarizes a comparison of the class-awareness features offered by JSDEODORANT, Eclipse JSDT, WebStorm, and JSClassFinder.

TABLE II
TOOL COMPARISON.

Tool	File outline	Type hierarchy	Visualization	Class-to-instances navigation	Instance-to-class navigation
JSDT	◓	○	○	◓	◓
WebStorm	◓	◓	○	◓	◓
JSClassFinder	○	◓	◓	○	○
JSDEODORANT	●	●	◓	●	●

File outline: In WebStorm, the Structure View outlines the program elements in a JavaScript source file, and decorates them with icons (*e.g.*, `c` for classes, `f` for fields, and `m` for methods). However, WebStorm heavily depends on the presence of JSDoc annotations for detecting code constructs, and does not identify class emulations. For example, when we removed the `@constructor` tag from the JSDoc of a function constructor, the annotation in the structure view changed from `c` to λ, denoting a normal function. Moreover, the Structure View does not differentiate classes from interfaces, even when the `@interface` annotation is present. JSDT also provides the Outline View for JavaScript files; however, it does not recognize class emulations at all; thus, the program elements are only annotated as variables or functions. There is no outline view for JavaScript files in JSClassFinder, instead, it provides a window where the user can see a list of the detected classes and their members. JSDEODORANT, on the other hand, annotates elements in its Module View based on their functionality and regardless of the presence of JSDoc annotations. It differentiates classes from interfaces, and also identifies and annotates abstract classes/methods, as well as overridden/overriding methods.

Type hierarchy: The extraction of module dependencies is vital for the detection of type hierarchies, since the declarations of the classes participating in a hierarchy are not always located in the same file. WebStorm identifies type hierarchies based on the initialization of the subtype's prototype with its supertype's prototype (*i.e.*, the first pattern in Section II). As mentioned before, it also relies on the JSDoc tag `@extends` for detecting type hierarchies. It is important to note that, since WebStorm does not recognize module dependencies, it identifies a list of candidate supertypes (or subtypes) when multiple classes with the same name exist in the project. JSDT does not detect class emulations, and therefore it does not identify type hierarchies. Similar to WebStorm, JSClassFinder supports only the first pattern of inheritance. Moreover, it fails to correctly identity inheritance in the presence of duplicate class names and module dependencies. JSDEODORANT identifies type hierarchies based on the three patterns discussed in Section II. Because it identifies module dependencies, as well as namespaces, unlike WebStorm, it identifies the exact super/sub type even in the presence of duplicate class names.

Visualization: To the best of our knowledge, WebStorm and JSDT do not provide any visualization for JavaScript projects. JSClassFinder reverse-engineers the entire JavaScript project to a class diagram where the generalization relations are displayed. However, as mentioned, it fails to identify class hierarchies in the presence of module dependencies. JSDEODORANT visualizes the module dependencies as well as class diagrams. However, our class diagram visualization is now limited to a single class only. The class diagram illustrates the class hierarchy if the selected class has a supertype or subtypes. Therefore, reverse-engineering the whole project in a single class diagram is left for future work.

Class-aware code navigation: WebStorm and JSDT both support code navigation. That is, from an invocation expression, it is possible to navigate to the declaration of the function. For a function declaration, both tools identify its invocations in a list, where the developer can easily navigate to the actual statement where the function is called. Since class emulation is achieved via the definition of a function, both tools support class-aware code navigation. However, the identification in both tools is based on name matching, that is, in the case of uncertainty due to name duplication, a list of potential candidates will be presented to the developer. While JSClassFinder does not support code navigation at all, JSDEODORANT supports navigating from a class to its instances and vice versa, and utilizes module dependencies to report exact matches for the identified references or declarations.

V. CONCLUSIONS AND FUTURE WORK

In this demonstration, we presented JSDEODORANT, an Eclipse plug-in that assists JavaScript developers in program maintenance and comprehension. It provides an architectural view of the entire project via its module dependency visualization, and a detailed view of the OOP constructs and program elements inside a file, at a glance. One of the advantages of JSDEODORANT is that it recognizes classes, interfaces, and inheritance relations being emulated via functions and object literals, regardless of the presence of JSDoc annotations. It also supports class-aware code navigation, as well as class diagram visualization.

As future work, we plan to enhance JSDEODORANT code navigation by resolving function and method calls in order to build call graphs and enhance further code comprehension. Finally, this work provides the foundations for adding code smell detection and refactoring support in the future.

REFERENCES

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

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

[3] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis, "Detecting Function Constructors in JavaScript," in *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[4] L. H. Silva, D. Hovadick, M. T. Valente, A. Bergel, N. Anquetil, and A. Etien, "JSClassFinder: A Tool to Detect Class-like Structures in JavaScript," *Computing Research Repository (CoRR)*, vol. abs/1602.05891, 2016.

[5] Best Practices for Abstract functions in JavaScript? [Online]. Available: http://stackoverflow.com/questions/7477453/best-practices-for-abstract-functions-in-javascript