IDENTIFICATION OF JAVASCRIPT FUNCTION CONSTRUCTORS USING STATIC SOURCE CODE ANALYSIS

Shahriar Rostami Dovom

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

August 2016

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Shahriar Rostami Dovom**

Entitled: **Identification of JavaScript Function Constructors Using Static Source Code Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Lata Narayanan _____ Chair

Joey Paquet _____ Examiner

Weiyi Shang _____ Examiner

Nikolaos Tsantalis _____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Amir Asif, PhD, PEng, Dean
Faculty of Engineering and Computer Science

Abstract

Identification of JavaScript Function Constructors Using Static Source Code Analysis

Shahriar Rostami Dovom

Software maintenance and comprehension constitute a considerable portion of the required effort for software development, and thus, myriad number of studies have proposed approaches for improving maintainability of software systems. However, the majority of these studies have examined software systems written in traditional programming languages, such as Java and C++. While the ubiquity of web has resulted to JavaScript to be extensively adopted by developers, studies that investigate maintainability issues in JavaScript are scarce.

Prior to the recent updates on the JavaScript language specifications, developers had to use custom solutions to emulate classes, modules, and namespaces in JavaScript programs; consequently, detecting classes in JavaScript programs is non-trivial due to the flexibility of JavaScripts syntax. To improve the maintenance and comprehension of JavaScript programs, we design and implement *JSDeodorant*, an automatic approach for detecting Function Constructors (i.e., the emulation of Object-Oriented classes) in JavaScript programs. These function constructors can be declared locally, under a namespace, or in other modules. The comparison with the state-of-the-art tool, JSClassFinder, shows that, while the precision of the tools are very similar (97% and 98%, respectively for *JSDeodorant* and JSClassFinder), the recall of *JSDeodorant* (98%) is much higher than JSClassFinder (61%).

Finally, we conduct an empirical study to compare the extent to which JavaScript programs in different domains (websites, server-side programs written in NodeJS, and libraries) adopt Object-Oriented classes. Our study shows that classes are more frequent in websites than NodeJS programs. Also, NodeJS projects have fewer classes compared to libraries.

Acknowledgments

First I want to express my sincere gratitude to my advisor Dr. Nikolaos Tsantalis for his guidance all the way through the research. His patience, profound knowledge and motivation make me able to improve my skills, to tackle difficult obstacles and empower my willing to get this research done.

Besides my advisor, I would like to thank my thesis examiners, Dr. Joe Paquet and Dr. Weiyi Shang for their valuable time to read my thesis and for their invaluable comments. Other faculty members of the Department of Computer Science and Software Engieering, specially Dr. Emad Shihab and Dr. Peter C. Rigby, for providing the necessary guidance.

I would like to thank Dr. Laleh M. Eshkevari for her valuable contribution in my thesis project. My special thank to my lab-mate and colleague Davood Mazinanian, for his continuous help in my research and his great contribution for an Eclipse plugin written to help developers easily use *JSDeodorant* analysis engine.

I would like to thank FQRNT, the Faculty of Engineering and Computer Science and Financial Aid and Award office at Concordia University for their generous financial support of this project. I want to express my thankfulness to the staff members of our university for keeping our environment clean, safe and engaging.

Last but not least, I am very thankful to my parents, for their love, sacrifices and their generous financial helps during my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript is the language of the web browsers [Cro08]. The language evolved from being in just client-side to make HTML elements more interactive. Nowadays, JavaScript is used for server-side programming [AMP16a], and also is adopted by emerging environments, across a wide range of Cloud Computing to Internet of Things [Pat]. It is the most used language on Github for three consecutive years since 2013 [Aly].

JavaScript is built on some good and bad ideas [Cro08]. The language created in ten days by Brendan Eich in May 1995. The language was designed under conflicting forces of managers at NetScape to make the language like Java, to keep it simple for beginners and to have control on everything in Netscape browser [Her12]. One of the issues with JavaScript is its global variable programming model. Everything in JavaScript is global, unless some actions are taken in order to reduce global footprints. The more global variable a program has, the more chances of bad interactions (collisions) with other applications or libraries [Cro08].

Crockford suggests using single global variable, which can be used by other variables or objects as a container while other practitioners suggest different techniques to leverage modularity in JavaScript such as closures.

This chapter introduces the concept of object-orientation in JavaScript and motivation of this thesis as well as the contribution and approach.

1.1 Software Maintenance and Comprehension

Maintenance is an important part of a software life cycle. Following software release, there is a need for improvements such as performance boost, fixing new defects or changes for adaptability to a changed environment. Preserving software integrity while making changes to software is the objective of maintenance [SBF14]. McKee [McK84] claims two thirds of a programmer activity attributed for maintenance of existing parts. An estimation of 50% of total life cycle cost is associated with the maintenance part [vV08].

Various techniques have been introduced to be used in software maintenance, including *program comprehension*, *re-engineering*, *reverse engineering*, *migration* and *retirement* [SBF14]. Different comprehension tools address different levels of a problem. Program slicers, static analyzers, dynamic analyzers and dependency analyzers are among the well known tools used by practitioners.

Spending considerable time on program comprehension (including reading source code and documentation) to be able to apply changes by programmers, makes tool makers and researchers to design code browsers and code visualizers to alleviate the difficulties of understanding source code. There are still lots of gaps that should be addressed and with *JSDeodorant*, we are trying to help developers better understand JavaScript programs to overcome barriers of comprehension in the language.

We designed and implemented a developer aided tool, *JSDeodorant*, which presents a state of the art static source code analysis which aim at identifying class-like structures in JavaScript that developers use to improve re-usability and utilizing object oriented practices adapted from known programming languages such as Java and C++.

1.2 Object Oriented Programming in JavaScript

In early nineties, object-oriented programming found its way to industry. It was a combination of different ideas which were previously conceived and accepted by the software community. However this time it was in the form of books, courses and programming languages which were built on top of this phenomena. Many developers think of object-oriented as the key to most of the problems, and if a system does not obey accepted principles in object orientation, it would be considered inferior [Hav14].

Object oriented programming is a paradigm which an *object* as the main entity, is a representative of data and behavior known as attributes and methods, respectively [KK11]. Mozilla Developer Network [Moza] defines OOP as a mean of abstraction over the real world. Modularity, polymorphism and encapsulation are predecessor which all of them can be found in OOP. Objects can send and receive messages to each other and process different data by themselves. OOP promotes understandability and maintainability if it is used properly. Class-based programming is one of the most known styles of object-oriented programming.

JavaScript, on the other hand, is a prototype based language, but to overcome the complexity of implementation in different scenarios and to increase the reusability of different components, developers rely on non-standard JavaScript's class abstraction [SRV+15].

Following object oriented definitions quoted from Mozilla Developer Network [Moza] and we will use these terms throughout this thesis:

- **Namespace**:

  A container which lets developers bundle all functionality under a unique, application-specific name.

- **Class**:

  Defines the object's characteristics. A class is a template definition of an object's properties and methods.

- **Object**:

  An instance of a class.

- **Property**:

  An object characteristic, such as color.

- **Method**:

  An object capability, such as walk. It is a subroutine or function associated with a class.

- **Constructor**:

  A method called at the moment when an object is instantiated. It usually has the same name as the class containing constructor.

- **Inheritance**:

  A class can inherit characteristics from another class.

- **Encapsulation**:

  A method of bundling the data and methods that use the data.

- **Abstraction**:

  A technique in computer science to hide the complexity of a given component using methods, attributes, inheritance and etc.

- **Polymorphism**:

  Poly means "many" and morphism means "forms". Different classes might define the same method or property (implementing an interface), but leaving the implementation details to concrete types. Knowing that which of the implementations will be called is deferred to the runtime.

There are different ways that programmers leverage modularity in the source code. One of the fine-grained and well understood techniques for achieving modularity is using object oriented *classes*. Besides classes, another type of OOP could be of prototype-based programming. Prototype style does not used classes, however it is using existing prototype objects to achieve the desired behavior by decorating them for reusability purpose.

Every JavaScript object has a prototype, which again is an object. Properties and methods of an object will be inherited by children objects. Despite the fact that JavaScript is a prototype-based language, it has *indirect* support for classes (No syntactical support for class, but there are different emulation techniques for classes). However, lacking of a syntactical support for declaring a class or defining a module in JavaScript prior to the recent update on the language specification[1] [Ecm15], makes developers employ different strategies to emulate classes and modules.

1.3 Motivation

There is an imminent growth to JavaScript popularity [AMP16b]. This growth makes JavaScript a programming language that can implement a sophisticated business application or to be used

[1] ECMA-262 Edition 6, approved on June 17, 2015 officially adds support for JavaScript class and module.

for creating an interactive user interface. Further development and maintenance of such programs require good tool support for assisting developers better understand source code [vMVH97].

Silva et. al. [SRV+15] developed a tool, JSClassFinder, to identify class-like structures in JavaScript. Based on their findings they divided JavaScript programs into four types of programs regarding class-like structures: *class-free* (systems with no class at all), *class-aware* (programs that use classes marginally), *class-friendly* (programs with relevant usage of classes) and *class-oriented* (systems with major usage of classes). However they did not evaluate JSClassFinder accuracy in terms of precision and recall. Thus we extend this work by addressing its current limitations and build an oracle to enable the evaluation of accuracy for their tool against ours. The most important limitations of their work are as following:

- The do not support finding classes that are in nested hierarchies such as namespace
- Without instantiation, their tool is not capable of finding function constructors, it means JSClassFinder cannot infer classes.

Another aspect of software comprehension, is providing a good tool support in IDE to assist programmers. There is a great possibility to improve existing tool support in IDEs. Limited support by Eclipse and its JavaScript source code analysis tool (JSDT) for navigation and outlining JavaScript programs, makes *JSDeodorant* plugin a valuable asset for JavaScript developers.

A tool with a known accuracy can be employed to facilitate the automatic migration of existing JavaScript code to the new version of ECMAScript with explicit support for class declaration. This thesis aims at automating the process of source code analysis with integrating it to Eclipse by giving the user a power of navigating between different modules and also depicting UML class diagrams.

1.4 Contributions

This thesis introduces *JSDeodorant*, an Eclipse plugin that combines and extends the previous techniques to detect functions emulating class behavior in JavaScript. *JSDeodorant* is also 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 by other tools, and tool comparison. We also did a quantitative and qualitative evaluation of the results to identify our current limitations and opportunities for future improvements.
- A comparison of our technique with the state-of-the-art tool *JSClassFinder* [SRV+15].

Chapter 2

Background

Traditional programming languages use constructs (*e.g.*, classes, modules) for improving code reusability and encapsulation. Code reuse decreases redundancy in the code (known as clones), and encapsulation reduces the risk of exposing implementation details. JavaScript developers tend to use different techniques to achieve code modularity.

As mentioned, lack of syntactical support for declaring classes, modules, and namespaces in JavaScript prior to the recent updates on the language specification, forces developers to employ different strategies to emulate them. In the following sub-sections, we briefly discuss some of the strategies introduced and promoted in the JavaScript community [Osm12, Cro08, Stab, Tru]. It should be noted that due to the inherent flexibility of JavaScript language, developers are able to follow different strategies for emulating the mentioned constructs, which makes the maintenance of JavaScript programs a non-trivial task.

2.1 Emulating Classes

In JavaScript, functions are *first-class* entities of type Object [1], *i.e.*, a function can be passed as an argument to other functions, returned by other functions, or stored as a value in a variable. Moreover, functions can have methods and properties themselves [Mozb].

JavaScript developers use functions to emulate the behavior of object-oriented *classes*. In this

[1] In JavaScript, all values, except primitive values are inherited from Object.

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.* [GACD12b]:

- **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 thesis, 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.

Thus the concept of *class* provided by major object oriented languages can be achieved by functions in JavaScript. Use of object literals in JavaScript is another way to provide a semi-class like behavior when developers require singleton classes. Definition 1 and 2 bellow explain the function and singleton based constructs respectively.

Definition 1 (Non-singleton classes: Function Constructor)
A simple function declaration can be a function constructor if it is preceded by *new* keyword in the call site. Then a new empty object of type *function* will be created and the *this* keyword inside the function bounds to the new object being constructed [Ecm11] [Mozc]. Two common approaches for declaring function constructors are listed in Figure 1.A and 1.B.

For declaring properties and methods, developers use the *this* keyword or use function constructor's prototype (prototype is usually used for defining methods). As shown in Figure 1, property *foo* and method *bar* are defined in the class body. Unlike conventional object oriented languages which provides implicit *this* scope, a JavaScript class must refer to its own properties and methods using *this* keyword explicitly. JavaScript does not provide access modifier for class members, so everything is always public and accessible in global scope, unless some actions are taken to make

```
function TheClass() {
  this.foo = 0;
  this.bar = function() {
    console.log(this.foo);
  }
}
```
A

```
var TheClass = function() {
  this.foo = 0;
  this.bar = function() {
    console.log(this.foo);
  }
}
```
B

```
var theInstance = new TheClass();
theInstance.foo = 2;
theInstance.bar(); // 2
```
C

Figure 1: Function constructor can be a function declaration (A) or an anonymous function expression assigned to a variable (B), the instantiation remains same for both cases (C)

them private using some namespacing techniques, which we will explore in Section 2.2 where we are discussing about immediately invoked function expressions (IIFE).

Figure 2 shows an equivalent example of a JavaScript class where the method assigns to the prototype of class, instead of beings assigned to *this* within in the body of function constructor.

```
function TheClass() {
  this.foo = 0;
}
TheClass.prototype.bar = function() {
    console.log(this.foo);
}
```
A

```
var TheClass = function() {
  this.foo = 0;
}
TheClass.prototype.bar = function() {
    console.log(this.foo);
}
```
B

```
var theInstance = new TheClass();
theInstance.foo = 2;
theInstance.bar(); // 2
```
C

Figure 2: Function constructor where the method is assigned to prototype rather than *this* keyword within the class body

It is very important to note that function *bar* is still accessing fields and other methods of a class *i.e.*,foo using the keyword *this*, despite the fact that the definition of methods is not located within the function constructor's body. In Section 6 we will show how important it is to support such a style of definition, because developers are using this style in many cases.

Noteworthy, declaring methods outside the function constructor's has many benefits including but not limited to [Staa]:

- **Ease of modification method behavior** Changing the method behavior in the run-time is just as simple as changing it once – that means the developer is not obligated to change multiple places to have unified behavior among instances of class.

- **Performance** Putting a method inside the the constructor leads to poorer performance (*i.e.*, Figure 1). So, a method needs to be created whenever an instance of a class is created, but if the method has been created on top of the prototype chain (*i.e.*, Figure 2), it will be *inherited* by each instance rather than being created multiple times.

Definition 2 (Singleton classes)

There are various techniques to emulate singleton classes. We will explore two different, yet popular ways of declaring singleton classes. The first style uses built-in JavaScript object literals to define a class. Figure 3.A shows how a singleton class can be defined by object literal.

In JavaScript, the object literal is a list of key-value pairs where each value can be of any data type such as object literal, function, or array literal. It 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. One of the major differences between object literal and function declaration style (*i.e.*, 3.B) is, the *new* keyword is not required for creating an instance of the object literal. In contrast, we can pass arguments to a function constructor while object literal style does not provide this functionality.

```
1 var singletonClass ={
2   foo: 0,
3   printIt: function(){
4     console.log(this.foo);
5   }
6 }
```

A

```
1 var singletonClass = new function(){
2   this.foo = 0;
3   this.printIt = function(){
4     console.log(this.foo);
5   };
6 }
```

B

```
1  singletonClass.foo = 5;
2  singletonClass.printIt(); // 5
```

C

Figure 3: An instance will be created immediately without the need of *new* keyword.

Another way of having a singleton class is using a combination of *new* and *function* keywords. If the *new* keyword precedes an anonymous function expression, the function constructor is formed

with a specific behavior: *new* keyword is going to be invoked once (singleton), to create an empty object. Figure 3.B shows an implementation example of this technique.

Taking the fact into account that these variations may be combined with many other techniques (namespacing) to achieve better encapsulation makes the identification of these structures a non-trivial source code analysis task. Different namespacing techniques suggested by practitioners in industry and above-mentioned emulation patterns may affect the precision of a static source code analysis tool for finding classes in JavaScript codes. In Section 2.2, we explore various namespacing techniques posed to developers for achieving a higher level of modularity.

2.2 JavaScript Namespace Emulation

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 [Osm12].

An example of a JavaScript program containing a variable and a function declared in global namespace shown in Figure 4.

```
1 // global scope
2 var myGlobalVariable = 1;

3 function sayHello(){
4   console.log(myGlobalVariable);
5 }
6 sayHello();
```

Figure 4: variable *myGlobalVar* declared in the global scope.

Both variable *myGlobalVariable* and function *sayHello* are in the global scope of such a program. Having that said, now let us assume we have two different files, *i.e.*, Sales.js and Orders.js with different functionalities. These two files provide a method (in the global scope) to return a list of sales and orders respectively. Figure 5 shows these two files containing a function with an identical name.

```
// Sales.js
var getList = function() {
 // do something here
}
```
A

```
// Orders.js
var getList = function() {
 // do something here
}
```
B

Figure 5: function with name *getList* exists in both files which returns appropriate result for corresponding file.

In the production code, when the two files *Sales.js* and *Orders.js* are included in the HTML file (Figure 6), invocation to *getList()* will lead to a name collision. Interpreter fails to understand by *getList()* which one of the definitions should be resolved. Hence name collision occurs.

```
<!DOCTYPE html>
<html>
<body>

<script src="Sales.js"></script>
<script src="Orders.js"></script>

</body>
</html>
```

Figure 6: adding external JavaScript files into a single HTML file in production code

There is also a best practice in client-side code deployment, which recommends concatenation of all JavaScript source files into a single one, to make it more suitable for the browser to reduce the number of HTTP requests over network (Round-trip). In those cases, JavaScript is more error prone in terms of variable or generally, identifier collisions.

Previous releases (prior to version 1.6) of JavaScript do not provide direct (syntactical) support for namespacing, and thus to avoid conflicts developers combine different language constructs to achieve namespace emulation.

In the following two sub-sections we briefly describe two common approaches used to avoid name conflicts.

Approach 1- Nested object literals:

The first and easiest way is to use nested object literals. Figure 7 shows how to form a nested object literal with a class declaration in its deepest level. To create an instance of a class, the call

site should provide a full path to the function declaration (function constructor).

```
var namespace = {
  module: {
    subModule: {
      TheClass: function() {
        this.foo = 0;
        this.printIt = function() {
          console.log(this.foo);
        }
      },
      AnotherClass: function() {
        // Body of another class
      }
    }
  }
};

var theInstance = new namespace.module.subModule.TheClass();
theInstance.foo = 7;
theInstance.printIt();
```

Figure 7: Nested object literals form a namespace for enclosing class, *TheClass*.

A composite identifier follows the *new* keyword to access *TheClass* for instantiation and prevents it to be exposed to the global scope. Now, only *namespace* is a global object and other nested objects including *module*, *subModule*, *TheClass*, and *AnotherClass* are wrapped in *namespace*. This is one of the simplest ways to define namespace and is widely used in open-source JavaScript programs.

Approach 2- Immediately-invoked Function Expressions (IIFE): IIFE refers to a function that is invoked right after it is declared. The self-invoked function only runs once and all variables and functions declared within the body of such namespace are by default private unless some actions are taken to expose them outside [Osm12], which we will explain in details. An example of IIFE is shown in Figure 8. The first pair of parentheses *(function() { ... })* transforms the code into an expression, and the second pair of parentheses *(function() { ... }) ()* calls the function that results from that expression [Stac].

As shown in Figure 8, the function *foo* is not exposed to the outside of IIFE. The only differences between the following techniques is how to expose members within IIFE to the outside.

```
(function(){
    // all your code here
    var foo = function() {};
    window.onload = foo;
    // ...
})();
// foo is unreachable here (it's undefined)
```

Figure 8: Immediately Invoked Function Expression

Expose members using return statement: To expose private members of an IIFE, developers use the return statement to return an object containing variables and functions. In Figure

```
var namespace = (function() {
  var template = {};
  template.myProperty = 0;
  template.PublicClass = function() {
    console.log('I am a class');
  }
  return template;
})();
```

A

```
var namespace = (function() {
  var classDeclaration = function() {
    console.log('I am a class');
  }
  return {
    myProperty : 0,
    PublicClass : classDeclaration
  };
})();
```

B

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

C

Figure 9: Immediately Invoked Function Expression With a Return Statement

9.A, the function constructor *PublicClass* is assigned as a property of an empty object with name *template* and then that object is returned by the *return* statement. As a result of the evaluation of this IIFE, the *rvalue* of the return statement (*i.e.*, return foo; where foo is the rvalue) within the function will be stored into the variable with name *namespace* which is in a global scope.

To create an instance of *PublicClass*, one should provide the qualified name as *namespace.PublicClass* like the instantiation in Figure 9.C.

If the *namespace* object is already exist in the global scope, there should be a work-around to check and use existing object rather than creating a new object. Expressions such as `var namespace = namespace || {};` would be a help to prevent overriding existing objects.

In case developer needs to verify if an object exists or not (above-mentioned expression), it

should be convenient for a developer to be able to augment an existing object, rather than creating new object for namespace.

For such scenarios, it is suitable to pass the namespace object to IIFE and assign methods and properties directly to that object. This way neither *return* statement nor creation of an empty object is required. Figure 10.A shows a case where a parameter can be passed to IIFE, and used as a container for methods and properties.

```
var namespace = {};
(function(ns) {
  ns.myProperty = 0;
  ns.PublicClass = function() {
    console.log('I am a class');
  }
})(namespace);
```

A

```
var namespace = {};
(function() {
  this.myProperty = 0;
  this.PublicClass = function() {
    console.log('I am a class');
  }
}).apply(namespace);
```

B

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

C

Figure 10: Immediately invoked function expression add methods and properties to a parameter (A) and to keyword this (B) which is equivalent

Another approach instead of using *ns* within the body of the IIFE in Figure 10.A, would be the use of built-in function *apply* to assign the *this* keyword to the passed argument. In JavaScript, calling `apply()` on an object that references a function, results to invoking that function, and the `this` object belonging to the called function is bound to the `apply()`'s first argument. Consequently, in Figure 10.B, the `this` object in the anonymous function is bound to `namespace`, and therefore, `PublicClass` will belong to `namespace`. The instantiation remains same for both cases A and B as shown in Figure 10.C.

2.3 JavaScript Module Pattern

JavaScript used to lack a mechanism for importing classes and functions from other JavaScript files. However, JavaScript community has proposed different approaches for defining JavaScript *modules* (*i.e.*, files that can be imported to other files). The two most popular module systems are

CommonJS [Com] and *AMD* [2] [AMD]. The former is designed to be more suitable for server-side JavaScript, while the latter is used mostly client-side JavaScript. There is also a more domain specific style developed by Google Closure Library [Clob] team, which has been used widely in their library.

CommonJS:

In *CommonJS*, a JavaScript module can expose all or parts of its behavior as an API, by setting the `exports` property.

```
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

Figure 11: An example illustrating the export and import in CommonJS style.

In Figure 11-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 11-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.

Asynchronous Module Definition (AMD):

Module pattern proposes a module pattern that supports asynchronous loading for both the module and its dependencies [Stae]. Like *CommonJS* which has *module.exports* and *require*, the two most important idea for AMD style is *define* method for exposing module definitions and *require* method for handling dependencies. Figure 12 shows the syntax of define and require which is slightly different from CommonJS style but signifies similar semantic.

[2] Asynchronous Module Definition

```
1  // package/lib is a dependency we require
2  define(["package/lib"], function (lib) {
3      // behavior for our module
4      function foo() {
5          lib.log("hello world!");
6      }
7      // export (expose) foo to other
8      // modules as foobar
9      return {
10         foobar: foo
11     }
12 });
```

A

```
1 // Somewhere else the module can be
2 // used with:
  require(["package/myModule"],
3       function(myModule) {
4     myModule.foobar();
5 });
```

B

Figure 12: An Example Illustrating the Export and Import in AMD Style.

As illustrated in the code comments, in Figure 12.A we have the definition of module, which itself, requires another module with name *lib* which is located in relative path *package* folder. In line 4, we have the definition of function *foo* but it is not yet exposed to the outside. Line 9 shows how developer can export private members within the *define* function by using the *return* statement which in this example is function *foo*.

In Figure 12.B another module tries to import above-mentioned module with the use of *require* function. A parameter of the *require* function will be injected by the interpreter with the corresponding definition (*i.e.*, object containing foo). It should be noted that the imported object is not the function *foo*, it is rather an object containing *foo* which is wrapped by the created object literal in Figure 12.A following *return* statement in line 9.

Closure Library Modules:

Similarly, for exposing behavior or part of the module, a function call to *goog.provide* is necessary. The argument to this module should be identical with another or other module's import (to be specific: *goog.require*) argument. In Figure 13 line 1, it is specified that this file is exposing a module with name *goog.async.run* and in line 3 and 4 it is dictated that this module requires two different modules with names *goog.async.WorkQueue and goog.async.nextTick*.

In line 6, it should be noted the module assigns the corresponding function declaration to a similarly named identifier, which previously (line 1) specified as the provide's argument. When a module manifests it requires this module, the function declared in line 6 is exposed to call-site and is responsible for execution in the run-time.

```
goog.provide('goog.async.run');

goog.require('goog.async.WorkQueue');
goog.require('goog.async.nextTick');

goog.async.run = function(callback, opt_context) {
 // ...
};
```

Figure 13: An Example Illustrating the Export and Import Using Closure Library Style.

Chapter 3

Literature Review

Despite the fact that JavaScript is used widely both for client-side and server-side programming, there seems to be remarkably little other work related to JavaScript source code analysis. The following nine papers give us information about how they analyze source code.

3.1 Does JavaScript Software Embrace Classes? [SRV+15]

Silva proposed an automatic approach, *JSClassFinder*, that identifies class definitions by analyzing the instance creation statements (statements using the `new` operator). They define function constructor as a function where the invocation consists of an instantiation and they conduct an empirical study on 50 popular JavaScript applications obtained from Github with the aim at answering whether JavaScript developers use classes or not and to see if they are using inheritance.

To the best of our knowledge, their research is the most relevant to ours but one of the limitation 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, leading to incorrect matches when different files contain classes with the same name, and finally their approach fails to identify classes that are not instantiated.

JSClassFinder follows static source code analysis technique, which is applied through a search for matching names between the operand of *new* expression and function declarations (known as function constructors). They extract Github projects that ranked with at least 1000 stars and have

at least more than 150 commits. Then they remove all minified[1] files, copyright and documentation files with *.js* extensions.

They found class declarations in 37 out of 50 systems (74%). For the systems containing classes, 34% of them have more attributes than methods (those classes are more like data classes), and they speculate it is more likely JavaScript developers are placing less importance on encapsulation (*i.e.*, getters and setters are rare in JavaScript)

It is concluded that there are four types of systems in JavaScript. *class-free* systems that do not make any usage of classes, *class-aware* systems that make rare use of classes, *class-friendly* systems were use classes relevantly, and finally *class-oriented* systems that have vast majority of uses in terms of class like structures. They also concluded that there is no significant co-relation between size of the corpus and class usage. They hypothesize that developers' background and experience impacts the usage of classes rather than the project's size.

3.2 Normalizing Object-Oriented Class Styles in JavaScript [GACD12a]

The intention of this work is to increase stylistic consistency and improve maintainability of object oriented JavaScript applications by normalizing the representation of classes into a single model. Authors discuss that JavaScript is a prototype-based language and lacking of an explicit class declaration to define fields and methods has led to a plethora of different class representation techniques.

They list three of the different styles that developers may use to mimic class declaration in JavaScript despite the fact that they are almost identical semantically and this difference can lead to stylistic inconsistency issues. The first style is closest to the conventional object oriented languages. The function *Class* acts as both class definition and the constructor.

The second style uses the function prototype object which, prototype of constructor function is an anonymous that is stored in the variable *Class*. The initialization moved to the code following constructor call.

The third one is similar to the above mentioned samples as it uses *Class* variable and anonymous function but assigned the method m to the prototype of the object. This way all objects created

[1]The process of removing unnecessary characters from code with preserving semantic is called minification

using the *Class* constructor function will share the m across the program.

They use the TXL Programming Language developed to support source transformation with an existing grammar for parsing JavaScript to pattern match source code. They merge each and every JavaScript file into one file with snippet of automatic comments to identify the original file boundaries. They recognize various object class styles and mark them for transformation to a preferred style. According to the survey on several tutorials, books and a large set of over 70 production JavaScript applications, *Prototype Lambda* is selected as the preferred style. After they have identified all of the classes and method functions in source codes, normalization transformation is carried out by another TXL program for changing them to the *Prototype Lambda* style.

They found real world applications are using different styles of those described in books and tutorials, that are more elegant and robust.

They found some applications were already using the target pattern, *prototype lambda*, thus transformation would not apply on them. Interestingly the other applications has four different patterns, that makes the project hard to maintain and inconsistent in style.

From this study it was found out that there are many different styles for class declaration that has been used interchangeably in real world applications. It can also concluded that developers are adopting some object oriented principles in this language like classes, encapsulation and inheritance(using prototypes) despite the fact that there is no direct language support for them. We adopt their technique to infer classes in JavaScript without the need for instantiation.

3.3 JSNOSE: Detecting JavaScript Code Smells [FM13]

JSNOSE tool combines static and dynamic analysis on JavaScript to gather adequate information to detect code smells. Authors define code smells as indication of potential comprehension and maintenance issues in JavaScript. Detecting code smells manually is tediously time consuming and error-prone. Lacking of a sophisticated smell detection tool for JavaScript, authors provide a tool to overcome the highly dynamic nature of the language.

Smell detection process of many object oriented languages is dependent on identifying objects, classes and functions inside the code and unlike languages such as Java or C++, identification of such language elements is not straightforward in JavaScript. Besides theoretical contributions they made in this paper, one of the valuable impacts of their work is the implementation of research into

an open source tool, which aims to find code smells. The tool is evaluated through an empirical study over 11 real JavaScript web applications. The contributions of this tool results in decreasing longterm development cost and increasing the chance of project success by helping developers improve maintainability of the code.

Among different code smells mentioned in both object oriented languages and JavaScript resources, they choose 13 specific code smells: 7 of them are existing prevalent object oriented smells adapted to JavaScript and the rest of them are specific JavaScript smells. They use a metric based approach, which is generic in almost all object oriented languages.

Consequently the threshold values are selected based on studies performed in other object-oriented languages. By performing static analysis and traversing abstract syntax tree (AST) they gather information about objects, properties, inheritance relations, functions, and code blocks. Their approach for using static analysis is very useful specifically for local objects and variables within functions, because they are not available in the global object during execution. Dynamic analysis is used in order to monitor object creations or updates at runtime and also to find unused/dead code for measurement purpose by code instrumentation.

The result indicates that lazy object, long method/function, closure smells, coupling between JavaScript and HTML and excessive global variables are the most prevalent code smells.

Their work indicates that source code analysis on JavaScript is not as easy as major object oriented languages. It provides some key features of other programming languages, but the way it behaves diverged from well-known object oriented languages. An example of this difference is prototype chains which somehow resembles inheritance in other languages with an exception that in JavaScript, programmer can create and modify existing object prototype at runtime. That means redecoration of object hierarchy during execution.

Unlike their approach, we can only rely on information gathered based on static analysis. We think it is possible to augment JavaScript weakly-typed system to infer unknown types without the need for code instrumentation and other types of analysis such as dynamic and symbolic execution. The limitation of their work is that they need to gather information during execution which cannot help programmers to find code smells during development phase.

3.4 An Analysis of the Dynamic Behavior of JavaScript Programs [RLBV10]

In this paper they did an empirical study on a large corpus of widely-used JavaScript programs to measure how and why the dynamic features of the language are used by programmers. They also measure the degree of dynamism in such programs. Moreover they made a good comparison with assumptions that are generally made within JavaScript communities in both industry and academia.

Due the magnitude of growth of using JavaScript in the real world web applications and also gradual increase in using JavaScript as a general purpose language for developing tools and libraries (running on server-side such as Node.js) it has taken academic attention. They characterized and produce behavioral data of JavaScript program by code instrumentation to obtain run-time traces of 103 web sites.

The main motivation behind their work is to assess feasibility of the static type system to apply type checking on existing JavaScript programs. They argue that existing techniques generally rely on minor simplifications, for instance one approach make the following assumption: "Usually, no further properties are defined after the initialization and the type of them rarely change" or Google's V8 engines is reported to optimistically *associate classes* to objects on the assumption that their shape will not change too much.

Their instrumentation tool integrates with Safari, browser from Apple, to capture traces consisting of DOM, AJax and purely JavaScript code that is stored in a database for following steps. Traces were compressed and store on disk without noticeable impact on live interactions in programs. Traces were stored in a database for further several static analyses using Mozilla Rhino JavaScript parser. The good point in their research is that they handle *eval* function calls as it is similar to load new source file.

Results of their study classified into two major branches. Firstly, static analysis is done to find following metrics:

- Corpus size
- Instruction Mix
- Prototype Chains

- Object Kinds

And the second part lay down in measuring program dynamism for the following metrics:

- Call site dynamism
- Function Variability
- Uses of *eval*
- Object protocol dynamism
- Constructor polymorphism
- Constructor prototype modification
- Changes to the prototype chains
- Object lifetimes
- The effects of JavaScript libraries

Based on the results, they evaluate assumptions that are made in the JavaScript community and argue that most of them are not true in real-world projects.

What we can infer from the results to make sure about our approach regarding static analysis is as following: the prototype hierarchy is invariant. Moreover, properties are added at object initialization and also they are rarely deleted. In most languages, *variadic* functions are rare, and based on this study nearly 10% of functions are *variadic* in JavaScript. It means that declared function signatures are indicative of types. Call-site dynamism is low which means that the level of polymorphic call sites are only 19% and monomorphic call sites are dominant in almost all projects.

With respect to the results of this study we can eliminate dynamic analysis and just improve statically analyzed precision of our research. With comparison with A. M. Fard and A. Mesbah's [FM13] approach, we can find out that there is no explicit need to have dynamic analysis for monitoring all object creations and modifications during runtime because they are not that much prevalent in real world applications.

3.5 Dont Call Us, Well Call You: Characterizing Callbacks in JavaScript [GMB15]

Authors of this paper try to understand callback usage in real-world JavaScript programs. Callbacks are one of the core successful features of this language and this feature owes its existence in JavaScript to the way that it defines functions. Functions are first-class objects which make it able to be saved in a variable, to be passed as an argument to other functions and so on. They studied 138 corpus of JavaScript programs dividing into two major category: 1) Server-side codes known as Node.js projects and 2) Client-side code which runs in the browser. The result of this research helps to show the language designers on how developers are using callbacks and tool builders to find good solutions to be able to get rid of excessive and nested callbacks known as callback hell.

JavaScript callback is very important because it is playing an important role. For example it is used to service multiple concurrent client requests on server-side. On the client-side, responsiveness of user interface controls are feasible because event-model system can be implemented using callback mechanism. Usage of callbacks is important because JavaScript comprehension and maintainability is correlated to the usage of callbacks. They study how often are callbacks used and in what extent callbacks are prevalent in client or server-side codes.

They design and implement a static analysis tool to identify different styles of callback patterns used by JavaScript developers. They built a tool on top of existing tools to generate AST, walk through it and a type inference engine to find parameter types to distinguish simple parameters from parameters that accept a function. Out of 138 open source projects, 86 of them were NPM Node's packages, 16 web applications, 16 game engines, 8 client-side framework and 6 visualization libraries.

For over 5 million lines of code in total, they found out of 10 functions, one of them accept a function as a parameter. Over 43% of callbacks were anonymous and majority of callbacks are nested to other callbacks. Most of the nested callbacks appear in client-side code with 72% than server-side with 55%. Error-first protocol was among the most popular techniques to overcome difficulties related to callbacks. The other two major solutions are Async.js and Promises.

They found that over half of all callbacks are named (not anonymous). They concluded that all JavaScript program that handle and response to events are using callbacks. These programs suffer from excessive callbacks which reduce the comprehensibility and maintainability of source code.

3.6 Detecting Inconsistencies in JavaScript MVC Applications [OPM15]

Maintaining consistency among emerging MVC [2] frameworks in JavaScript platform makes authors of this paper to design and implement a tool, named AUREBESH that automatically detects inconsistencies between Model, View and Controller of such applications. In their previous work [OBPM13] they found majority of JavaScript bugs are DOM-related [3] ones. DOM related bugs are related to the interaction between HTML, CSS and JavaScript. Bugs that MVC frameworks are claiming that this pattern would help developers reduce the complexity of working with DOM in different scenarios, and such bugs are more often result from developer's incomplete understanding of the relationship between JavaScript and DOM.

Despite the fact that MVC frameworks are created to solve such problems, they are still vulnerable to DOM-JavaScript interaction bugs. They further explain that these frameworks rely on use of identifiers to connect model objects, controller methods and view's field accesses to model. The main problem arises because JavaScript is loosely typed. When developer change the name of a method or generally an identifier, consistencies will not throw mismatch or undefined compiled-timed exceptions. To tackle such a problem they create a static analysis tool to identify variables and methods in these different components (model, view and controller) and then they try to infer the type of those identifiers to find inconsistencies.

They opt for static analysis in this work. And throughout their work, they have a working example inspired by one of these popular MVC frameworks (AngularJS), however they claim that their technique can be extended to all MVC frameworks. They extract model variables and controller functions and then they apply type inference techniques to find the type assigned to a model variable in the model or controller by inspecting the right-hand side of assignment expressions. So their tool is capable of identifying both identifier(name) and type inconsistencies.

They did an empirical study on 11 JavaScript applications and in total their tool found a total of 15 error messages. Of these 15 errors, 13 of them were identifier inconsistencies and 2 type inconsistencies. They categories following patterns for faults:

- Identifier defined elsewhere (found in 7 cases)

[2] Model, view and controller
[3] Document object model

- Incorrect identifier (found in 5 cases)
- Boolean assigned a string (found in 2 cases)
- Identifier name not updated (found in 1 case)

They only found one false positive due to a misleading (corner case) alias in one of the applications. Their tool indicates their approach is very accurate in finding inconsistency in JavaScript MVC framework applications. They reach to an overall 96.1% recall and a precision of 100%. The tool is useful in detecting bugs, which help researchers of this work to find 15 real-world bugs in application built on top of AngularJS, which is a popular JavaScript MVC framework.

3.7 Development Nature Matters: An Empirical Study of Code Clones in JavaScript Applications [CRK16]

The authors mainly investigate:

- Clone properties in different languages (Java and JavaScript) and application domains (JavaScript stand-alone projects(JSproj) and JavaScript web applications(JSweb)).
- Correlation between clone properties and software metrics and comparing this relation in different languages and application domains.
- Identifying development practices in Java, JSweb, and JSproj.

One of the active research areas in software is code clones [TMK15]. Researchers have been studied this subject for almost two decades. While clones are investigated from different perspectives the authors addressed new limitations in code clone studies. Firstly, almost all clone studies have been done on statically typed languages. Since the nature of statically and dynamically languages are different, developers might clone code differently. Secondly, there is no study that compares the relationship existing between clone properties and software metrics in different languages or application domains.

Thirdly, a limited number of studies on web applications have included qualitative analysis along with quantitative once. Fourthly, few studies compared clone between statically and dynamically

typed languages. Lastly, clone coding studies in dynamic languages do not consider different application domains and in spite of the popularity of Java script, studies on JavaScript clones are limited.

They developed a clone detector specifically for JavaScript. This tool is inspired by Deckard [DEC], so they can use the same configuration for both Java and JavaScript. Then, they used the following code metrics to compare clones in different languages and domains:

- Cloning locality
- Average and maximum lines of cloned code
- Clone coverage
- Files associated with clones
- Function-level clones

The results show that different features offered by programming languages affect code duplication patterns. For example, as JavaScript has no support for method overloading, the number of method level clone for JavaScript is smaller in comparison to Java. It also reveals that a considerable amount of JSweb projects have clones and web developers duplicate code intentionally. In addition, clone properties in the same language might not be the same for different domains.

3.8 JavaScript Errors in the Wild: An Empirical Study [JPZ11]

The authors of this paper intended to study JavaScript errors in web applications and understanding the source of these errors. They also plan to find design guidelines for developing process.

Specifically, they want to investigate:

- Errors in JavaScript applications and find common characteristics of these errors among web applications.
- The correlation between the speed of interaction between the user and the rate of JavaScript errors.
- Non-deterministic errors in JavaScript web applications.

- Correlation between using JavaScript static and dynamic features and the number of errors.
- Correlation between frameworks in developing JavaScript code for web applications and a number of errors.

JavaScript is the most popular language for developing web application client-side functionalities. More than 95 percent of today's websites contains thousands of lines of JavaScript codes. In spite of Javascript popularity, there is no study on characterizing errors related to JavaScript in web applications. Web applications might undergo serious malfunctioning due to errors in JavaScript codes.

In order to investigate the errors in web applications, the authors create test cases. Each test case contains fifteen test cases based on interaction with web applications. The cases run with different speed and multiple times on fifty web applications from Alexa [4] Top 100.

The results of this empirical study show that:

- Almost all JavaScript web applications face with error.
- They found these errors mainly fall into four categories: Null Exception, Permission Denied, Syntax Error, and Undefined Symbol.
- The speed of interaction affects the number of errors.
- Most of JavaScript web applications errors are non-deterministic.
- Correlation between frameworks in developing JavaScript code for web applications and a number of errors.
- There is a correlation between some of the dynamic and static features of the languages.
- Increasing the number of frameworks increases the number of errors in web applications.

This study showed that errors occur in almost all websites with JavaScript client-side code. JavaScript developers need to consider the static and dynamic features that they use in their code and to avoid the error-prone features. Since some errors happen by changing the speed of the interaction with the websites, this application should be tested in different interaction speed.

[4] http://www.alexa.com

3.9 JavaScript: The Used Parts [GHWB14]

The authors of the paper believe that JavaScript language designers change the language features without knowing how developers apply these features. So, finding the most used part of the language to help language designers, tool builders, browser vendors, and researchers is the main concern of this paper. This paper also investigates the reason of choosing some features of the language by the developers. Finally, it shows that how developers adopt new added features and problem prone features.

Websites ranked on the Alexa are popular but they cannot be a comprehensive representation of all JavaScript codes in all web applications. So in this paper, they gather a diverse corpus of source codes to include different developers code from different applications and backgrounds. They collect more than a million unique scripts that fall into five categories: Node.js applications, Firefox add-ons, Alexa pages, browsed pages(by using Win Web Crawler), and JS libraries.

They used a combination of static and dynamic analysis to collect information from the scripts. They instrumented Spider Monkey and it is used for both static and dynamic analysis of browsed and Alexa pages and JS libraries. Node.js applications were analyzed with a instrumented V8 engine. Firefox add-ons were the only ones that is analyzed manually.

The presented results show that:

- Developers tend to apply some problematic features of the language like *for ... in* and block level declarations (readability of the code is one of the reason that some developers tend to do that).
- There are some misunderstanding and browser support that prevents developers to use some of the new language features. For example, the strict mode is only used one percent in the corpus.
- Node.js developers try to benefit from object-oriented features of JavaScript while functional programming is more popular in web applications.

Chapter 4

Approach

The process of analyzing JavaScript source code and extracting sophisticated structures (*i.e.*, class emulations, namespaces, and module patterns) requires more than just a source code parser or an Abstract Syntax Tree (AST) visitor.

JSDeodorant is a tool developed at Concordia University to analyze JavaScript source code and it employs the best practices learned by a former tool *JDeodorant* developed and designed in Concordia Software Refactoring Lab as Java Refactoring recommendation tool [FTC07] [TCC08]. *JSDeodorant* involves a multi-step analysis to create its own abstraction, which makes it much easier to perform different software analysis techniques (*e.g.*, control and data flow analysis).

The overall view of our approach for detecting function constructors in JavaScript is illustrated in Figure 14.

In the following, each of the steps is discussed in detail.

4.1 Step 1: Parsing JavaScript Files and Building Model

In the first step, *JSDeodorant* uses the Google Closure Compiler [cloa] to parse and extract ASTs of JavaScript files of the given project. We use the Closure Compiler tool as it is a well established project on GitHub with a large community of developers to maintain and support it (179 contributors, 61 releases since 2009). As it is shown in Figure 14, the input is a set of JavaScript files (at least one JavaScript file), which are parsed to obtain their ASTs.

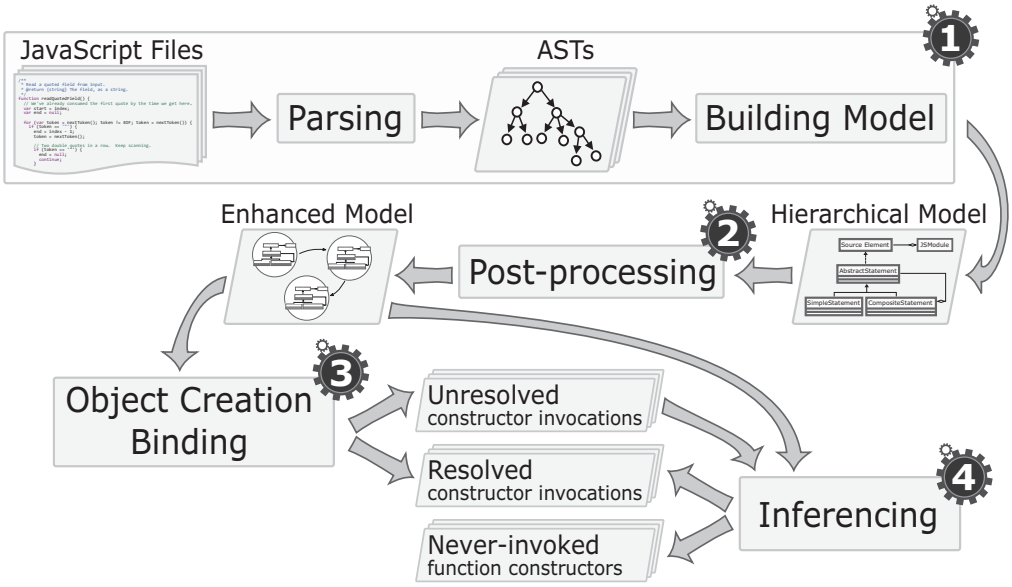

Figure 14: Overview of the Proposed Approach

JSDeodorant then traverses the generated ASTs to build a hierarchical abstract model, which provides a higher level abstraction than AST, by capturing the nesting structure of code at the statement level.

4.1.1 Hierarchical Model

We are using the Composite design pattern to eliminate the complexity of distinguishing between a leaf node and predicate node (if, for, while, etc.). The benefits of the composite model is that, it can treat complex and primitive objects uniformly, and thus it is possible to manipulate a single instance of the object in the same way as it is to manipulate a group of them. The structure of a simple composite is shown in Figure 15.

JSDeodorant has a hierarchical data structure keeping elements including statements that are important components of the source code's semantics, where changing those statements would likely result in different execution behaviors. This means that we are not specifically interested in trivial statements such as $a = a + b$, however we keep them in a hierarchical model and then we can retrieve them when it is necessary for our source code analysis purpose.

A list of important elements that we are keeping in our hierarchical model is as follows:

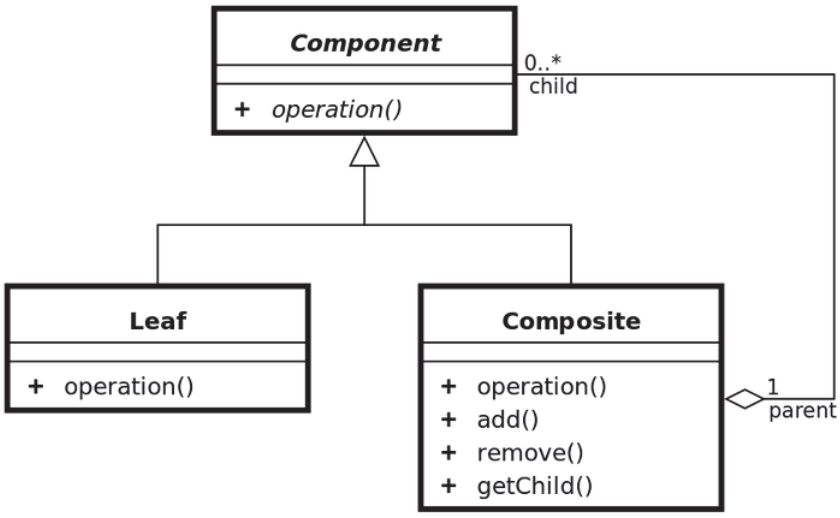

Figure 15: Composite Design Pattern

- Creations (*i.e.*,new foo())
- Function invocations (*i.e.*,bar())
- Function declarations (*i.e.*,function baz() { ... })
- Object literals (*i.e.*,var obj = myVar :20, myFunction : function() { ... })
- Assignment statements (*i.e.*,a = b;)
- Variable declarations (*i.e.*, var a = b;)

Capturing above expressions and statements is achieved by our abstraction model and is only implemented in *AbstractFunctionFragment* class which is an abstract class where AbstractExpression and AbstractStatement are inherited from.

When we are creating an instance of *AbstractExpression* or *AbstractStatement*, in the constructor of these two classes there are various method invocations (these methods belong to parent class, which is *AbstractFunctionFragment*) to test, extract and store necessary information (detailed list of elements listed above).

During the construction of our model, we create a `Program` object to represent the program (at file level) which we are going to analyze. A `Program` holds a list of `SourceElement` objects

that can be either `AbstractStatement` or `FunctionDeclaration`. Class *Program* implements the `SourceContainer` interface, which is the representation of a container. Other classes that implements `SourceContainer` are `CompositeStatement`, *FunctionDeclarationExpression* and *ObjectLiteralExpression* since they can contain expressions or statements.

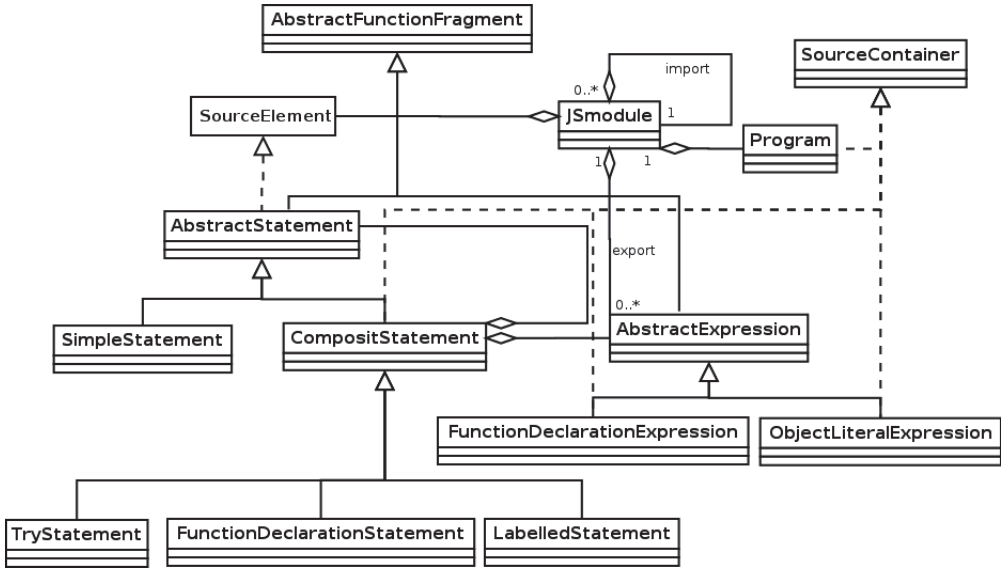

Figure 16: An Overview of JSDeodorant Architecture

`AbstractStatement` and `AbstractExpression` classes are representing AST nodes that can be children of a composite structure.

Figure 16 shows the relationships between `Program`, `CompositeStatement` and `AbstractStatement` classes. It should be noted that at one higher level of abstraction, there is a class named "module" which contains a program and a list of other *modules* as dependencies. For example if the code contains a *require* statement, to include another JavaScript file, then the analyzed version of that module (means the parsed and abstracted model of that file) will be added as a dependency.

JavaScript programs can have statements and function declarations in the root. This is different from object-oriented programming languages, in which programs have to start with a *Class* as the root element of source code.

For this reason we modeled the program which ought to have two kinds of source elements. The first one is function declarations, the most common way to achieve modularity in JavaScript programs.

The other source element that can be in the root of program is an `AbstractStatement`. A list of these source elements is kept in `Program` class in our model for future analysis.

We have a utility class to extract expressions or statements from a source container at any nesting level. To be able to extract the desired information we need a visitor to check every element type of the AST. Class `FunctionDeclaration` represents the structure of a *function* node in the AST. The hierarchy of composite statements is also shown in Figure 16. A composite statement can be a *TryStatement*, a *FunctionDeclarationStatement* or a *LabelledStatement*.

For example, for a `for` block (*i.e.*, a composite statement that contains other composite or simple statements), the model includes only high-level information about the `for` loop statement (but *JSDeodorant* does not have corresponding *For* class in our model, because it is not important for our analysis purpose) and the nested statements, which is necessary for detecting patterns we are looking for.

4.2 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.

The reason for post-processing is that we have to query the existing model for expressions that are specific to the import/export mechanism depending on the chosen module system. Otherwise, we have to deal with low-level AST nodes (*i.e.*, visit all expressions containing binary operator where the right operand is *require* function call, and the left operand is a variable declaration and the operator is *equal* operator).

Due to our abstract model, we can query all *assignment* expressions in the post-processing phase where the assignment's right operand is a function call with the identifier *require*. This is much more convenient than dealing directly with AST nodes.

Moreover, when we find a *require* function call, we have to resolve its argument as a path to another module, and in that module, we have to find the appropriate *exports* or *module.exports* to be able to match these two modules mutually.

Our model is built upon the *CommonJS* module system, as it is one of the most popular APIs used by developers. The *Closure Library Modules* is also supported.

However, our approach to identify file dependencies is generic enough that it can be easily

extended to capture other dependency patterns, such as AMD.

For dependencies that cannot be resolved with the `require/exports` pattern, *JSDeodorant* tries to resolve them by reading the `package.json` file which is specified by the *CommonJS* standard, and contains information about module's main file, and dependencies to other modules. This way, *JSDeodorant* tries all the possible approaches to make sure that the dependencies between files are resolved in a proper way.

In terms of design and implementation, it requires for each module system, a class is being created and make it to inherit from *PackageExporter* and *PackageImporter* interfaces for extracting export and import statements correspondingly.

4.3 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 into two disjoint sets of resolved and unresolved function constructor invocations. *JSDeodorant* traverses the model, and in each JavaScript module, it analyzes the identifier name in the instance creation expressions (*i.e.*, expressions using the `new` operator). Each identifier name can be either a *simple* or *composite* name.

4.3.1 Simple Name Identifier:

For simple identifier names (*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 reach the root of the JavaScript module and we are unable to identify a declaration, we search a list of predefined JavaScript functions for a match [javb].

The reason for consulting the predefined list after our internal search is that, it is possible to redefine JavaScript native objects, although it's a risky practice [Stad] [per] [java].

4.3.2 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 Figures 9 and 10 where the function constructor is defined in a namespace (`namespace.PublicClass`), and Figure 11-A (line 2) where the class is defined in another module (`Writer.write`).

JSDeodorant first splits the identifier name into its tokens (*e.g.*, <`namespace, PublicClass`> for `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 *JSDeodorant* finds a match (line 1 in Figure 9-A), it tries to match the next token (*i.e.*, `PublicClass`) within the already matched context (*e.g.*, lines 1-7 in Figure 9-A) and this process continues until all tokens are matched.

4.4 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 corresponds to the function constructors that are nowhere invoked (instantiated) in the code.

This technique is useful for those JavaScript projects that are designed and built for specific purposes which they do not have instantiation of the code along with the definition such as libraries or frameworks.

The detailed algorithm of our technique is written in a pseudo code style in Algorithm 1.

Algorithm 1 Function constructor detection algorithm

```
1: cont_list=loadListofConstrucotrs()
2: for fd in functionDeclarations do
3:     if ! (fd in cont_list) then
4:         if HASPROPERTY(fd) OR HASMETHOD(fd) then
               cont_list.add(fd)
5:         end if
6:     end if
7: end for

8: procedure hasProperty(fd)
9:     assignments=fd.getAllAssignment()
10:    for a in assignments do
11:        if a.getLHS() instanceof FieldAccess AND !(a.getRHS() instanceof FunctionDeclaration) then
               return TRUE
12:        end if
13:    end for
14: end procedure

15: procedure hasMethod(fd)
16:    assignments=fd.getAllAssignment()
17:    parent= fd.getParent()
18:    for a in assignments do
19:        if a.getLHS() instanceof FieldAccess AND a.getRHS() instanceof FunctionDeclaration then
               return TRUE
20:        end if
21:    end for
22:    parent_assignments=parent.getAllAssignment()
23:    for pa in parent_assignments do
24:        if Prototypeof(fd).contains(pa.getLHS()) AND pa.getLHS() instanceof FunctionDeclaration then
               return TRUE
25:        end if
26:    end for
27: end procedure
```

Chapter 5

Evaluation of Accuracy

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

- **Precision and Recall:** *What is the performance 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.
- **Comparison with the state-of-the-art tool:** *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.

5.1 Oracle

To measure the precision and recall of *JSDeodorant* and *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 [JSDb]. In this case, function constructors can be automatically found by simply parsing JSDoc comments. We used *closure-library* (a large library used in Google products, such as Gmail and Maps), as it is JSDoc-annotated project. Unfortunately, *JSClassFinder* threw Out of Memory error for *closure-library*, so we had to manually

execute it folder-by-folder.

Using transpiled code: TypeScript [typ] and CoffeeScript [cof] 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.

These projects are medium-sized, to prevent *JSClassFinder* from crashing.

In Table 1, we have reported the main characteristics of the programs used to build the oracle.

Table 1: Characteristics of the Analyzed Programs.

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

This section reports the results of our quantitative and qualitative analysis of the three programs with respect to the two research questions formulated above.

5.2 Precision and Recall

To find out about the efficacy of these two tools, we calculated the precision and recall using the following formulas. TP stands for True Positive: is a function that is detected by the tool as function constructor and is in the oracle too. FP stands for False Positive: is a function that is detected by the tool as function constructor, but it is not in the oracle. FN stands for False Negative: is a function that is NOT detected by the tool as function constructor, but it is in the oracle) :

$$Precision = \frac{|TP|}{|TP| + |FP|} \quad \text{and} \quad Recall = \frac{|TP|}{|TP| + |FN|}$$

Table 2 illustrates the results of *JSDeodorant*.

As it is observed, *JSDeodorant* can achieve a high precision and recall, for all three analyzed projects.

Table 2: *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%

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. We manually inspected these functions and marked them as true positive (TP) or false positive (FP). The decision to label the detected functions as TP or FP was based on the use and definition of the functions.

That is, 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 TP. The final label was decided by unanimous voting, and the third vote was sought in case of conflict. Out of 83 cases, 82 were labeled as TP, leaving only one case as FP.

As explained in Section 4, *JSDeodorant* identifies functions that have at least one property, or method as function constructors. Moreover, those constructor invocations that *JSDeodorant* binds to a valid declaration are also identified as function constructors. The false negatives were those cases where the body of the function did not match our definition of function constructor, and they were not instantiated.

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 ways to improve our technique.

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 17 shows an example in *closure-library* where the function

```
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);
```

Figure 17: Example of a Function Constructor *JSDeodorant* Failed to Identify

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: *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%) on the selected dataset.

5.3 Comparison with the state-of-the-art tool

We run *JSClassFinder* on the same dataset we used for our first research question, summarizing the results in Table 3. *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 false positives was done similarly to Section 5.2, and out of 42 cases, 41 were labeled as TP. By manually checking 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. Moreover, to do a fair comparison, we removed function constructors belonging to sub-directories in which *JSClassFinder* crashed from the oracle and the results of *JSDeodorant*.

Table 3: *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.

In Table 4, the accuracy measures for both tools are reported. As it is observed, *JSClassFinder* performed well on both accuracy measures for *atom*. This is because all function constructors were invoked in *atom*, and thus, *JSClassFinder* identified them all.

However in the other two projects, it fails to identify function constructors that are not invoked.

Table 4: Accuracy Measures for Both Tools

	JSClassFinder		***JSDeodorant***	
Project	**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%**

Here we want to report the main problems we faced when working with JSClassFinder:

- Running JSClassFinder on JavaScript programs is not straightforward. This tool requires to first manually extract ASTs from JavaScript files using a third-party JavaScript parser and store it in JSON [1] format. The JSON files are then used as an input to JSClassFinder. This extra step makes JSClassFinder less usable and difficult to integrate in Integrated Development Environments (IDEs).

- JSClassFinder is matching *new* expressions with function names "blindly". Even if you have two independent files that are not connected using CommonsJS or AMD import/export statements, the tool will still search for a matching between these two files. We think the process of finding function constructor should be more sophisticated than a simple name or AST node matching and that's why we choose to implement CommonJS and ClosureLibrary styles for connecting files that are linked to each other.

[1] JavaScript Object Notation

```
1 function ArrayIterator(array) {
2   this.array_ = array;
3   this.current_ = 0;
4 }
5 goog.inherits(ArrayIterator, goog.iter.Iterator);
```

Figure 18: Example of a Function in the Test Code That Was Not Annotated, but Added to the Oracle as a Function Constructor.

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

5.4 Threats to Validity

Construct validity: We only addressed module dependencies that comply with *CommonJS* and *Closure-Library style*; 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.

In the manual investigation phase for augmenting the oracle the independent evaluators might both misinterpreted the intention of the developer who wrote the code. For example, in Closure Library project, we found a function located in *closure-library-v20160315 closure goog iter iter_test.js* line 23, which does not contain any @Constructor annotation. Figure 18 shows the piece of code that is considered as a class based on the votes of two independent investigators. It might be possible that the developer's intention is not to use this function as a function constructor, but taking into account the inherits call that follows, it seems that ArrayIterator extends Iterator by adding to additional fields.

External validity:

While these preliminary results of analyzing three projects from different domains show that *JSDeodorant* detects function constructors with a high accuracy compare to JSClassFinder tool, and the result of analysis on three JavaScript projects confirms existence of classes in different JavaScript projects, 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 [jsda].

Chapter 6

Empirical Study

In Chapter 5 we evaluated the efficacy of *JSDeodorant* in detecting function constructors in JavaScript programs. *JSDeodorant* achieves relatively high precision and recall. To gain a better understanding how JavaScript developers adopt classes in their programs, we conduct and empirical study by running *JSDeodorant* against a dataset of JavaScript projects from different categories. This chapter presents the targeted JavaScript programs, the study setup and the results of the study.

6.1 Study Setup and Results

We collect a dataset consisting of three major categories: NodeJS projects, websites and libraries. We think breaking down JavaScript projects into these three categories shows the difference of the adoption of class usage among three major domain that JavaScript is used. Moreover other researchers also propose this categorization based on the domain of programs in previous works [GMB15] [CRK16].

For NodeJS projects, our selection was based on the NPMJS website [1] which ranks Node packages by the number of downloads. For the JavaScript files on the Websites, we ran Crawljax [MvDL12], a headless crawling engine, to collect JavaScript files. Crawljax emulates the behaviour of users by firing user events, e.g., clicking on elements in web pages, while capturing JavaScript files in different stages. Finally, for libraries, we collected seven popular JavaScript libraries, based

[1] www.npmjs.com/

on the number of stars of the corresponding repositories on GitHub.

Note that, for libraries and NodeJS projects, we excluded all the minified files (i.e., files that were minified using specific programs that decrease file size by removing whitespace characters, renaming variables, etc.), to avoid the detection of duplicated classes in the original and minified version. We also choose to run *JSDeodorant* against Silva et.al [SRV+15] dataset. All the projects they collected are considered as libraries, thus we put them in the libraries category.

Table 5, 6 and 7 shows the characteristics of the projects under analysis, including their project name, analyzed version, size (in KLOC), number of files, number of functions along with the number of identified function constructors in different JavaScript programs. The last two columns shows the number how many number of methods defined within the body (Method Inside - MI) of function constructor and the number of methods assigned to the prototype object (Method Outside - MO).

6.2 Discussion

As it is shown in Table 5, there are 25 programs with no classes which forms 52% of all node projects. It is shown that less.js has a relatively high number of classes compared to other projects with total of 77 classes. Looking further, the number of methods assigned to the prototype object (MO) for less.js is 55 compared to 22 methods defined inside the body of function constructors. We think less.js adopts a good level of OOP and also it follows best practices promoted by the community such as defining methods to the prototype [PRO].

In Table 6 it is shown that only 10% of the websites do not have any usage of classes.. There is one website (*facebook.com*) with a relatively high number of classes (312).

Interestingly Table 7 shows that 10% of all examined libraries have no usage of classes..

As it can be observed, from an eyeball analysis of the results there is a difference between the number of classes for projects within different categories. We observe that NodeJS projects have fewer number of classes compared to websites.

To statistically confirm this observation, we first normalize the number of classes with the number of files for each of the projects, in order to conduct a fair comparison among the categories. As we need to select an appropriate statistical test for the comparison, we first used the Shapiro-Walk test to assess whether the distribution of the data for these normalized values is normal or

not. The results of the test *rejected* the null hypothesis that the data is normally distributed (p-value <2.179e-10 , 1.88e-08 and 7.973e-12 respectively for NodeJS projects, websites and libraries). Thus, we use the Wilcoxon rank sum test (also called the MannWhitney U test), a non-parametric statistical hypothesis test for which the data under test does not need to be normally distributed, to find out if there is any statistically significant difference between the number of classes in these categories. The results show that NodeJS projects have fewer classes compared to website projects (p-value <6.78e-07). We cannot conclude if there is a difference between libraries and websites, since the p-value is 0.5034. Moreover, we confirm that NodeJS projects have less number of classes compared to libraries (p-value <1.963e-07).

When comparing the number of classes between the categories, we observed NodeJS projects have fewer classes compared to websites and libraries but we cannot compare the level of class adoption between comparing libraries and websites.

We conjecture that NodeJS projects have adopted classes to a lesser extent, because developers dominantly follow a different reuse approach when developing NodeJS applications. These applications are mainly constructed by importing and connecting small pieces of code from external packages, which are mostly written by other developers. In this case, as the code base is rather small, and the imported code is already organized using some packaging system (and therefore, only exposing what is required by the clients), the probability of name collisions would be low, making the use of classes an overkill for the maintainability. Moreover, the asynchronous (i.e., non-blocking) nature and event-driven architecture of NodeJS make it a great runtime environment for specific types of software which, instinctively, might not need classes for encapsulating data and behaviour.

On the other hand, we observe that websites are using more classes, and this can bear various reasons. One possible reason is that name collision is more probable in websites JavaScript files. This is because web pages usually need to import several JavaScript files. These files can be imported, in the required order, using the <script>tag in the HTML code. Alternatively, the developer might want to minify all these files and combine them to a single JavaScript file, with the goal of improving the speed of loading the web pages (i.e., by minimizing the download time,

and also the number of HTTP requests). In both cases, there is a high chance that identifier collision happens in these JavaScript files. Furthermore, in contrast to NodeJS projects, the size of JavaScript files in websites can be relatively high, and it calls for better organization of code.

Like other programming languages, JavaScript libraries can serve multiple goals, including but not limited to: promoting re-usability, saving developers effort by preventing wheel re-invention, or providing a higher abstraction level by hiding low-level details that are difficult for developers to deal with (for example, a JavaScript library that provides the necessary functionality for working with network streams). Object-Oriented classes can definitely facilitate reaching all these goals, e.g., by encapsulating the functionality and exposing re-usable interfaces to the clients.

6.3 Threats to Validity

External validity: In this study we selected a diverse set of projects from different application domains, varying in size and development nature, including server-side, client-side and library JavaScript projects. Though we cannot generalize our findings, to mitigate the threat to external validity (generalizability) we selected programs with different size and nature.

As mentioned, we used Crawljax to collect JavaScript files from the websites in the dataset. However, Crawlajx may miss to find all JavaScript files from websites. This is mainly because Crawljax performs dynamic analysis, and like any other dynamic analysis approach, it needs to be fed with proper input and execution scenarios to achieve acceptable results. This means that, one needs to adequately configure Crawljax for each of the websites under analysis, so that it can explore all possible DOM states of the website to collect all the JavaScript files referenced in different DOM states; a task which is cumbersome and impractical in general.

Table 5: Number of Function Constructors in JavaScript NodeJS Program.

Project	Version	Size (KLOC)	#Files	#Functions	#Classes	#MI	#MO
express	v5.0.0-alpha.2	16.7	150	2224	8	2	6
forever	v0.9.2	2.6	21	138	0	0	0
less.js	v2.5.3	25.06	162	871	77	22	55
node-browserify	v9.0.8	5.96	378	688	2	1	1
npm	v3.3.6	34.14	360	2917	35	27	8
pm2	v0.5.7	22.29	224	2117	11	2	9
statsd	v0.7.2	4.19	31	434	11	5	6
yo	v1.4.8	1.54	21	142	0	0	0
abbrev	v1.0.9	1	2	4	0	0	0
argparse	v1.0.7	4	32	158	5	1	4
async	v2.1.0	0.69	2	104	1	1	0
balanced-match	v0.4.2	1	3	3	0	0	0
brace-expansion	v1.1.6	1	2	12	0	0	0
colors	v1.1.2	0.5	5	29	0	0	0
contact-map	v0.0.1	0.5	3	5	0	0	0
console-browserify	v1.1.0	0.2	3	21	0	0	0
core-util-is	v1.0.2	0.1	2	15	0	0	0
date-now	v1.0.1	0.04	3	7	0	0	0
dateformat	v1.0.12	0.16	2	0	0	0	0
debug	v2.2.0	0.28	3	17	1	1	0
domelementtype	v1.3.0	0.01	15	1	0	0	0
domhandler	v2.3.0	0.35	4	18	1	0	1
domutils	v1.5.1	0.69	12	85	0	0	0
entities	v1.1.1	0.32	5	53	0	0	0
eventemitter2	v2.1.0	0.57	2	0	0	0	0
exit	v0.1.2	0.25	5	28	0	0	0
faye-websocket	v0.11.0	1.7	19	195	10	4	6
findup-sync	v0.4.2	1.7	19	7	0	0	0
gaze	v0.1.1	0.5	2	42	1	0	1
getobject	v0.1.0	0.15	3	9	0	0	0
glob	v7.0.5	1.3	12	118	1	0	1
globule	v1.0.0	0.76	5	45	0	0	0
graceful-fs	v4.1.5	0.64	3	78	4	4	0
grunt	v1.0.1	2.21	13	136	1	0	1
hooker	v0.2.3	0.94	6	61	6	6	0
htmlparser2	v3.9.1	1.83	16	137	7	0	7
iconv-lite	v0.4.13	1.2	15	43	0	0	0
js-yaml	v3.6.1	3.5	33	99	8	6	2
lru-cache	v4.0.1	1.1	5	89	3	2	1
minimatch	v3.0.2	1.78	6	54	4	2	2
nopt	v3.0.6	0.6	3	17	0	0	0
noptify	v0.0.3	0.57	10	50	0	1	0
qs	v6.2.1	15.3	9	632	25	14	11
readable-stream	v2.1.4	1.7	10	99	9	5	4
rimraf	v2.5.4	0.28	4	22	0	0	0
shelljs	v0.7.3	2.26	30	84	0	0	0
sigmund	v1.0.1	0.34	3	28	0	0	0
tiny-lr	v0.0.5	1.76	8	164	10	1	9

Column MI stands for number of Methods defined Inside the body of function constructor.
Column MO stands for number of Methods defined Outside of the body of function constructor (to prototype object).

Table 6: Number of Function Constructors in JavaScript Websites.

Project	Version	Size (KLOC)	#Files	#Functions	#Classes	#MI	#MO
google.com	May 16, 2016	0.26	10	723	128	78	50
facebook.com	May 16, 2016	2.37	61	9696	312	168	144
baidu.com	May 16, 2016	0.69	30	2302	54	44	10
yahoo.com	May 16, 2016	0.08	8	190	15	7	8
amazon.com	May 16, 2016	4.42	171	5677	124	116	8
twitter.com	May 16, 2016	5.05	20	3802	147	127	20
vimeo.com	May 16, 2016	0.29	14	9	0	0	0
tumblr.com	May 16, 2016	1.48	81	3,100	124	105	19
slideshare.net	May 16, 2016	0.14	13	113	11	5	6
snapdeal.com	May 16, 2016	0.78	13	1039	33	27	6
soundcloud.com	May 16, 2016	0.26	10	70	4	4	0
paypal.com	May 16, 2016	7.5	23	454	30	20	10
microsoft.com	May 16, 2016	1.24	24	163	9	9	0
linkedin.com	May 16, 2016	1.24	8	1289	66	46	20
indeed.com	May 16, 2016	0.6	22	1160	144	82	62
imgur.com	May 16, 2016	0.85	53	198	12	10	2
hp.com	May 16, 2016	5.56	45	2732	106	76	30
groupon.com	May 16, 2016	0.3	35	111	8	8	0
github.io	May 16, 2016	0.27	5	89	7	4	3
icloud.com	May 16, 2016	0.15	13	22	0	0	0
imdb.com	May 16, 2016	0.92	182	244	10	2	8
livedoor.jp	May 16, 2016	0.42	18	103	4	4	0
foxnews.com	May 16, 2016	2.08	52	882	58	34	24
instagram.com	May 16, 2016	0.39	20	942	58	39	19
ifeng.com	May 16, 2016	7.12	237	1045	28	28	0
hulu.com	May 16, 2016	1.43	19	1263	27	7	0
paytm.com	May 16, 2016	1.06	80	214	11	10	1
pixiv.net	May 16, 2016	0.63	54	1516	16	16	0
orange.fr	May 16, 2016	3.46	31	429	19	18	1
putlocker.is	May 16, 2016	0.21	8	849	20	19	1
taboola.com	May 16, 2016	3.1	26	436	16	16	0
wikihow.com	May 16, 2016	0.35	26	491	12	8	4
wix.com	May 16, 2016	2.02	18	168	8	8	0
wordpress.org	May 16, 2016	0.11	9	91	0	0	0
walmart.com	May 16, 2016	3.06	84	551	62	28	34
wikia.com	May 16, 2016	3.22	120	857	20	19	1
yelp.com	May 16, 2016	1.03	24	70	0	0	0
zendesk.com	May 16, 2016	1.47	29	622	19	13	6
zillow.com	May 16, 2016	1.83	55	458	18	18	0

Table 7: Number of Function Constructors in JavaScript Libraries.

Project	Version	Size (KLOC)	#Files	#Functions	#Classes	#MI	#MO
ace	v1.1.3	8.22	50	544	5	5	0
angular	v1.5.7	287.6	1091	9896	109	75	34
backbone	v1.3.0	26.47	21	1344	18	15	3
ember	v2.7.0-beta.2	97.57	680	4542	32	27	5
jquery	3.0.0-rc1	59.29	166	3334	45	41	4
pdf	v1.5.188	81.01	166	3717	306	277	29
underscore	1.8.3	10.41	15	799	25	21	4
masonry	3.1.5	197	1	10	4	2	2
randomColor	0.1.1	361	1	17	0	0	0
respond	1.4.2	460	3	15	2	2	0
clumsy-bird	0.1.0	628	7	1	0	0	0
deck.js	1.1.0	732	1	22	26	5	21
impress.js	0.5.3	769	1	23	0	0	0
async	0.9.0	1.1	1	75	4	2	2
turn.js	3.0.0	1.9	1	18	0	0	0
zepto	1.1.3	2.4	17	149	19	18	1
jade	1.0.2	4	28	41	3	2	1
select2	3.4.8	4.1	45	44	0	0	0
jQueryFileUp	9.5.7	4.4	15	49	4	1	3
semantic-UI	0.18.0	11.9	19	25	46	42	4
wysihtml5	0.3.0	5.9	69	107	119	86	33
paper.js	0.9.18	25.8	67	143	28	24	4
intro.js	0.9.0	1	1	24	2	2	0
timelineJS	2.25.0	18.2	89	213	25	25	0
jasmine	2.0.0	2.9	48	239	66	44	22
reveal.js	2.6.2	3.3	1	105	14	10	4
flora.js	1.0.0	3.3	26	104	247	27	220
number.js	0.4.0	2.4	10	119	7	6	1
typehead.js	0.10.2	2.4	19	95	15	15	0
video.js	4.6.1	7.9	38	432	6	5	1
sails	0.10.0	13	98	154	10	5	5
ionic	1.0.0	14	90	283	257	207	50
chart.js	0.2.0	1.4	1	34	32	31	1
grunt	0.4.5	1.9	11	94	1	0	1
ghost	0.4.2	15	122	205	27	14	13
skrollr	0.6.25	1.7	1	44	12	10	2
leaflet	0.7.0	8.3	71	63	8	6	2
gulp	3.7.0	2	4	9	1	0	1
three.js	0.0.67	37	164	609	750	439	311
bower	1.3.5	8.1	53	306	37	19	18
algorithm.js	0.2.0	1.5	29	82	12	2	10
mustache.js	0.8.2	5	1	27	3	0	3
parallax	2.1.3	1	3	57	12	8	4
2048	-	8	10	66	7	1	6
pixiJS	1.5.3	13.8	72	361	185	125	60
isomer	0.2.4	7	71	47	8	1	7
slick	1.3.6	1.6	1	64	1	0	1
fastclick	1.0.2	7	1	22	1	0	1
socket.io	1.0.4	1.2	4	49	4	0	4

Chapter 7

Tool Demonstration

To be able to run *JSDeodorant*, the machine is required to have at least JDK 7 installed on the machine with an Eclipse instance that has Gradle plugin installed on it or Gradle installed on the machine accessible through terminal. So it is possible to resolve dependencies with *gradle build* command to install JAR dependencies without the need for Gradle plugin on Eclipse.

This tool also comes with an Eclipse plugin, which itself is able to analyze JavaScript projects. In the first part, we will explore command-line mode to see how to generate CSV files as the outputs and console logs for experimental purpose. In the second part, we will show how to use the Eclipse plugin to analyze JavaScript files.

7.1 CLI Mode

To be able to use *JSDeodorant* in CLI mode (without Eclipse), you should run *gradle assembly* in the *core* folder of *JSDeodorant* to build the appropriate JAR file in the target folder. Then, you can run the tool with the following command:

```
java -jar target/jsdeodorant-0.0-SNAPSHOT-jar-with-dependencies.jar -help
```

to show the switches that you can pass to the tool.

Here is the list of switches you can pass to the command-line runner:

- `-class_analysis` : Advanced static analysis to match function definitions with function calls (call-site)
- `-function-analysis` : Advanced function analysis to match class definitions with initialization (call-site)
- `-calculate-cyclomatic` : Enable calculation of cyclomatic complexity
- `-js` : The JavaScript filenames separated by space
- `-directory-path` : Directory path for JavaScript project
- `-analyze-lbClasses` : Analyze libraries to find class usage in them
- `-builtin-libraries` : List of libraries located somewhere on the system such as Node's built-in libraries i.e. Error or Util
- `-disable-log` : Enable logging mechanism
- `-externs` : List of externs files to use in the compilation
- `-libraries` : List of libraries to distinguish between production/test codes.
- `-module-analysis` : Enable module analysis for CommonJS or Closure Library style packaging
- `-package-system` : Select the package system including CommonJS and Closure Library
- `-output-csv` : Generate a CSV file containing analysis info
- `-output-db` : Put analysis info into a Postgres DB
- `-name` : Project name
- `-version` : Project version
- `-psqlServer` : Postgres server name
- `-psqlPort` : Postgres port
- `-psqlDbName` : Postgres database name

- `-psqlUser` : Postgres username
- `-psqlPassword` : Postgres password

An example of a working set of switches for project Closure Library is:

```
java -jar target/jsdeodorant-0.0-SNAPSHOT-jar-with-dependencies.jar -output-csv -class-analysis
 -module-analysis -package-system ‘‘ClosureLibrary’’ -analyze-lbClasses -directory-path
 ‘‘/Users/Shahriar/Documents/workspace/era/dataset/closure-library-v20160315’’
-name ‘‘closure-library’’
```

After running this command, *log/classes* and *log/functions* folders should contain output CSV files generated during the analysis.

For instance, *log/classes/class-declarations.csv* file contains rows corresponding to the number of class declarations on the system that we ran the analysis.

- Class name
- File path where class declared
- Is Predefined JavaScript class?
- Class offset
- Has new expression
- Has inferred? {yes/no}
- Constructor lines of codes
- Total class lines of codes (If methods are assigned to the prototype, then number of lines of codes should contain method body)
- Constructor lines of codes
- Has namespace? {yes/no}
- Number of methods
- Number of attributes

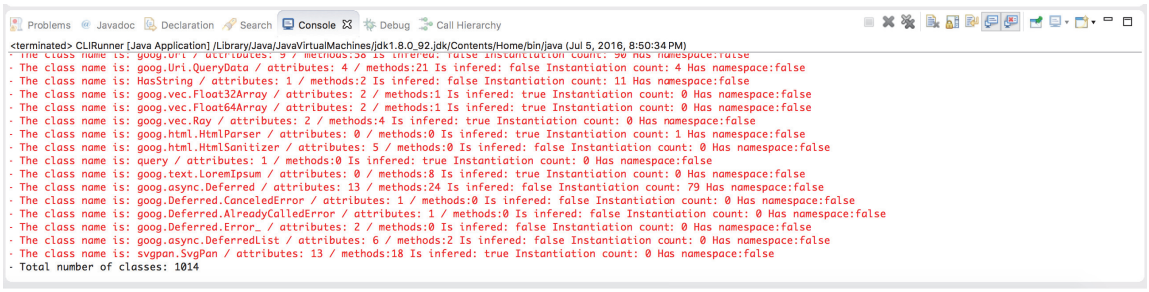

Figure 19: Analysis Result in The Console

- Is declaration in library?
- Is aliased?
- Number of instantiation

If the user prefer to inspect results of analysis in the Eclipse or terminal, there is a short and minimal information presented in console (standard output stream). Figure 19 shows few lines of results in Eclipse console window.

There is a more general overview of the analyzed projects which is located under *log/aggregate/modules.csv*. This file contains information about a JavaScript files, number of dependencies and number of export statements.

7.2 Eclipse Plugin

To be able to import Eclipse Plugin into the workspace, you have to navigate to plugin root folder and run *gradle buildAndCopyLibs*. This way gradle would build *JSDeodorant*'s core component and the main JAR file and its dependencies will be copied to plugin target folder to resolve plugin's dependencies.

Then run the Eclipse plugin as an *Eclipse Application* through the *Run as* menu in Eclipse. Figure 20 depicts the steps to run the plugin.

In the new Eclipse instance that is started, user has to create a JavaScript project to be able to import a folder containing JavaScript files. Then by selecting on the desired folder from *Project Explorer* and pressing the *i* button in the *Modules View*, a tree consisting of modules, classes,

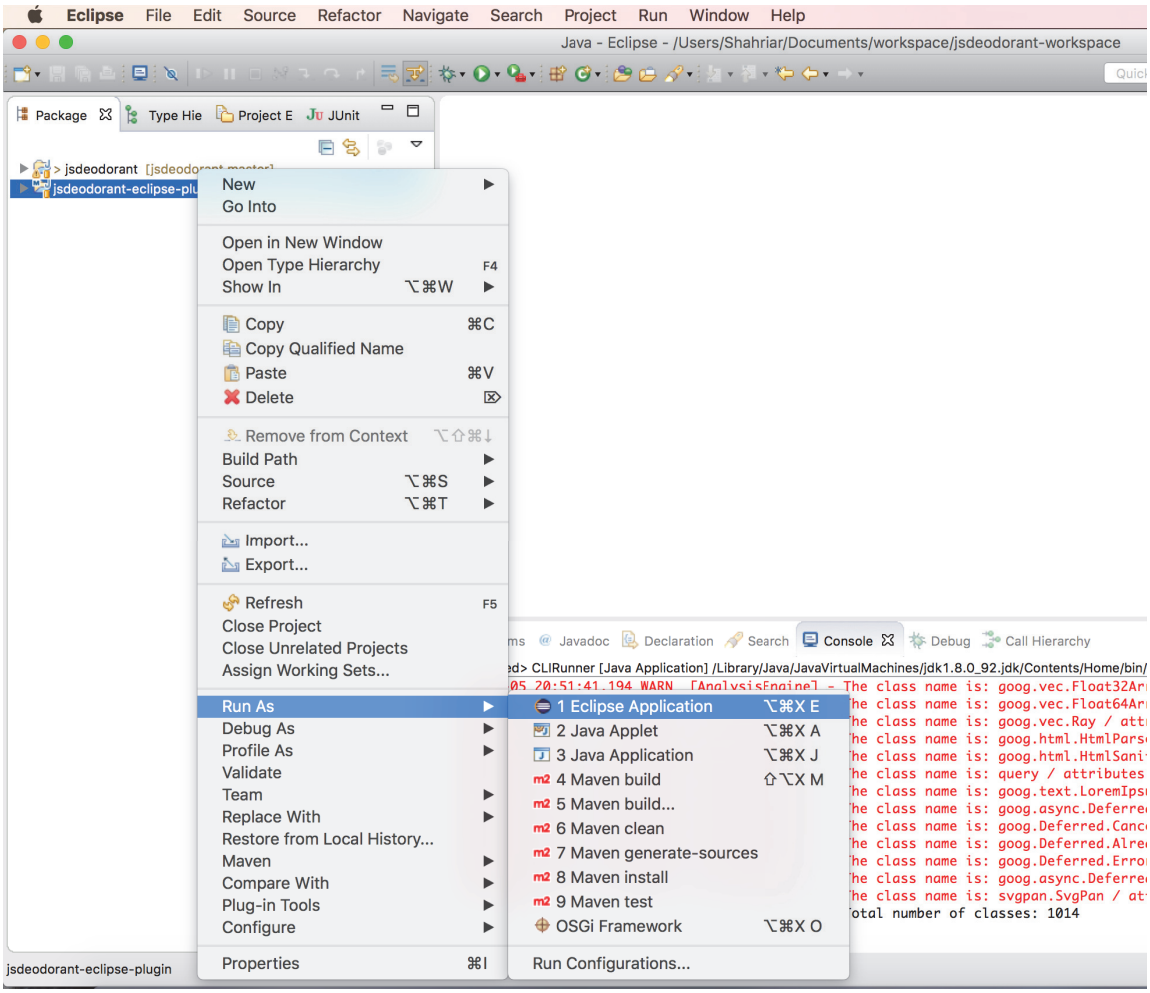

Figure 20: Run Eclipse Plugin

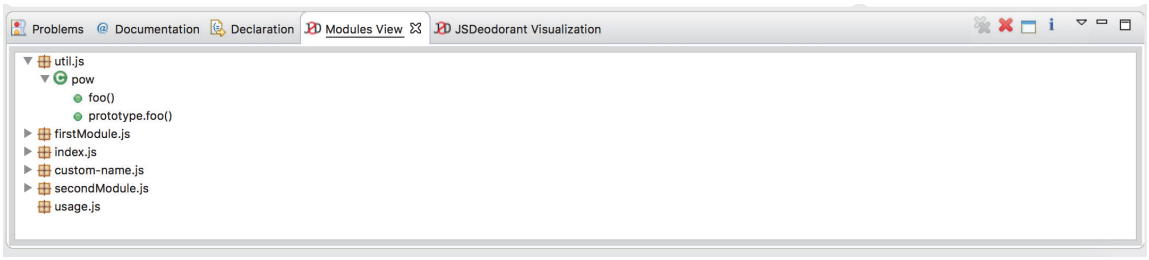

Figure 21: Modules View Window

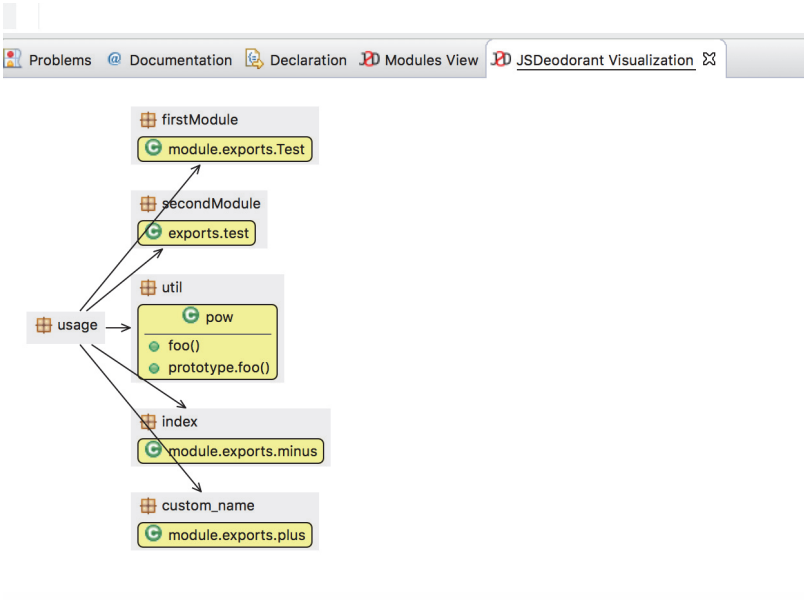

Figure 22: *JSDeodorant* Module Visualization

methods and attributes will be shown. Figure 21 illustrates the tree generated by *JSDeodorant* analysis.

To view the dependencies of a module, user can right-click on one of the modules in the *Modules View* and click on the *Show module dependencies* to get an overview of the dependencies. A module visualization will be depicted containing the module's dependencies and a class diagram of each module that this module is dependent on. Figure 22 shows an example of *JSDeodorant* module visualization.

In Figure 22 we can observe that there is a *usage* module which is dependent to many other modules (*firstModule*, *secondModule*, *util*, *index* and *custom_name*). This dependency is achieved

by requiring these modules by *usage* module. The diagram shows the export statements that each module has as well as method names.

Chapter 8

Conclusion and Future Work

This work paves the way for an ultimate research goal, that seeks to improve software comprehension and improve tooling for software maintenance for JavaScript programs. *JSDeodorant* implements key elements of a comprehensive infrastructure for applying static source code analysis on JavaScript source code. With this infrastructure, we were able to implement an approach for detecting *Function Constructors*, i.e., functions that are used for emulating Object-Oriented classes in JavaScript programs.

With an oracle comprising of three medium-size JavaScript projects, we measured the precision and recall of *JSDeodorant* in detecting function constructors, and compared it with JSClassFinder, the state-of-the-art tool that has been introduced for the same purpose. We found out that *JSDeodorant* reaches up to 97% precision and 98% recall on average, outperforming JSClassFinder, which can reach 98% precision and only 61% recall.

Nonetheless, we think there is still room for improving the accuracy of this tool by reducing false negatives and false positives.

- **Eliminating false negatives**
  - Using data-flow analysis can help us to find aliased function constructors. For example, in Figure 23, we have shown a function constructor, namely `Foo`, declared in line thee nested in a namespace. In line thirteen, we have called a function `anotherFunction` with passing the namespace. And in line ten, we are creating an instance of class through the

parameter. But to find what `param.foo` is referring to, we need a more in-depth data-flow analysis. In this example, `param.Foo` is an alias of `namespace.innerNamespace`.

```
1  var namespace={
2  innerNamespace : {
3       Foo: function() {
4       console.log('object literal way');
5     }
6 }
7 };
8
9 var anotherFunction = function(param){
10  var newInstance = new param.Foo();
11 }
12
13 anotherFunction(namespace.innerNamespace);
```

Figure 23: Aliasing

- Implementing a better approach for detecting namespaces or nested hierarchical scopes.

- **Eliminating false positives**

  - Avoid finding functions that are not actually function constructors. For instance, we believe that it is possible to improve our inference mechanism, by adding a constraint to differentiate an interface emulation from a class emulation, when interface has all its methods with empty body. With this improvement, we can eliminate 18 false positive cases we found for Closure Library.

Detecting and studying function constructors is only one of the possible studies that was facilitated through the infrastructure that *JSDeodorant* provides, and there are countless number of other conceivable studies that the research community can conduct. As an example, with the existing API of *JSDeodorant*, we can easily mine large JavaScript repositories to capture the extent of adoption of JavaScript design patterns. To the best of our knowledge, *JSDeodorant* already supports all the major emulation patterns for class-like structures, known namespace patterns, and one of the major module patterns, namely CommonJS. This definitely enables studies that require fine-grained static analysis requiring information for the mentioned constructs.

We are also planning to conduct an empirical study to have a better understanding of the evolution of JavaScript projects. To do so, we have collected a corpus of JavaScript projects from

different domains. In that study we will explore the usage of namespace, class, module patterns, object literals, and inheritance over the life-time of the projects. From this perspective we may investigate the hypothesis that whether, over time, the use of object-oriented practices is improving in JavaScript projects.

Moreover, we can use *JSDeodorant* in order to find code smells in JavaScript programs. These code smells can be related to the usage of classes in JavaScript (e.g., God Class, Data Class and Feature Envy). One of the features that will be added in the next release of *JSDeodorant* is the support for inheritance. Consequently, detecting inheritance-related code smells (such as Refused Bequest) becomes viable using *JSDeodorant*.

The current version of *JSDeodorant* supports illustrating UML class diagrams from existing JavaScript programs; however, by supporting inheritance in the tool, *JSDeodorant* might become very helpful for reverse engineering purposes. JavaScript developers can feed *JSDeodorant* with the source code and reverse engineer it to draw an overall view of the system with UML diagrams, leading to saving the enormous amount of time that might be wasted, e.g., when a new developer tries to grasp the architecture of an existing JavaScript project.

Finally, being able to transform existing JavaScript code to a better form is the ultimate goal of this research. Refactoring code smells (e.g., duplicated code) would be very useful, because JavaScript community is extensively suffering from lacking good tooling in this area.

Bibliography

[Aly] Alyson La. Language trends on github. https://github.com/blog/2047-language-trends-on-github. Online; Accessed: 2016-03-04.

[AMD] Asynchronous module definition. https://github.com/amdjs/amdjs-api/blob/master/AMD.md. Online; Accessed: 2016-02-24.

[AMP16a] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, page 11 pages. ACM, 2016.

[AMP16b] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, page 11 pages. ACM, 2016.

[cloa] Closure Compiler. https://developers.google.com/closure/compiler/. Online; Accessed: 2016-02-24.

[Clob] Closure Library Namespace. https://developers.google.com/closure/library/docs/introduction. Online; Accessed: 2016-02-24.

[cof] CoffeeScript. http://coffeescript.org/. Online; Accessed: 2016-07-02.

[Com] CommonJS. http://www.CommonJS.org/. Online; Accessed: 2016-02-24.

[CRK16] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in javascript applications. volume 21, pages 517–564, Hingham, MA, USA, April 2016. Kluwer Academic Publishers.

[Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 1st edition, 5 2008.

[DEC]

[Ecm11] Ecma International. *ECMAScript 2011 Language Specification*, 6 2011.

[Ecm15] Ecma International. *ECMAScript 2015 Language Specification*, 6 2015.

[FM13] AM. Fard and A Mesbah. Jsnose: Detecting javascript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125, Sept 2013.

[FTC07] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, pages 519–520, 2007.

[GACD12a] W. Gama, M.H. Alalfi, J.R. Cordy, and T.R. Dean. Normalizing object-oriented class styles in javascript. In *Web Systems Evolution (WSE), 2012 14th IEEE International Symposium on*, pages 79–83, Sept 2012.

[GACD12b] Widd Gama, Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Normalizing object-oriented class styles in JavaScript. In *14th IEEE International Symposium on Web Systems Evolution, WSE*, 2012.

[GHWB14] S. Gude, M. Hafiz, and A. Wirfs-Brock. Javascript: The used parts. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 466–475, July 2014.

[GMB15] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.

[Hav14] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press, 2 edition, 12 2014.

[Her12] David Herman. *Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript (Effective Software Development Series)*. Addison-Wesley Professional, 1 edition, 12 2012.

[java] Extending JavaScript Natives. `https://javascriptweblog.wordpress.com/2011/12/05/extending-javascript-natives/`. Online; Accessed: 2016-02-24.

[javb] JavaScript Built-in Functions. `http://www.tutorialspoint.com/javascript/javascript_builtin_functions.htm`. Online; Accessed: 2016-02-24.

[JPZ11] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn. Javascript errors in the wild: An empirical study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 100–109, Nov 2011.

[jsda] JSDeodorant. `https://github.com/sshishe/jsdeodorant`. Online; Accessed: 2016-07-02.

[JSDb] JSDOC. `http://usejsdoc.org/`. Online; Accessed: 2016-07-02.

[KK11] Eugene Kindler and Ivan Krivy. Object-oriented simulation of systems with sophisticated control. *International Journal of General Systems*, 40(3):313–343, 2011.

[McK84] James R. McKee. Maintenance as a function of design. In *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*, AFIPS '84, pages 187–193, New York, NY, USA, 1984. ACM.

[Moza] Mozilla Developer Network. Introduction to object-oriented javascript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript`. Online; Accessed: 2016-03-15.

[Mozb] Mozilla Developer Network. Javascript functions. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions`. Online; Accessed: 2016-02-24.

[Mozc] Mozilla Developer Network. Javascript this keyword. `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this`. Online; Accessed: 2016-02-26.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[OBPM13] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side javascript bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64, Oct 2013.

[OPM15] Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015.

[Osm12] Addy Osmani. *Learning JavaScript Design Patterns - a JavaScript and jQuery Developer's Guide.* O'Reilly Media, 2012.

[Pat] Patrick Catanzariti. Why javascript and the internet of things? `http://www.sitepoint.com/javascript-internet-things/`. Online; Accessed: 2016-03-04.

[per] Extending builtin natives. Evil or not? `http://perfectionkills.com/extending-native-builtins`. Online; Accessed: 2016-02-24.

[PRO] Use of 'prototype' vs. 'this' in javascript? `http://stackoverflow.com/questions/310870/use-of-prototype-vs-this-in-javascript`. Online; Accessed: 2016-08-06.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[SBF14] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[SRV+15] Leonardo Silva, Miguel Ramos, Marco Tulio Valente, Nicolas Anquetil, and Alexandre Bergel. Does Javascript software embrace classes? In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 73–82, 2015.

[Staa] Stackoverflow. Advantages of using prototype, vs defining methods straight in the constructor? `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this`. Online; Accessed: 2016-06-02.

[Stab] Stackoverflow. How do I declare a namespace in JavaScript? http://stackoverflow.com/questions/881515/how-do-i-declare-a-namespace-in-javascript. Online; Accessed: 2016-02-24.

[Stac] Stackoverflow. What is the (function())() construct in javascript? http://stackoverflow.com/questions/8228281/what-is-the-function-construct-in-javascript. Online; Accessed: 2016-06-04.

[Stad] Stackoverflow. Why is extending native objects a bad practice? http://stackoverflow.com/questions/14034180/why-is-extending-native-objects-a-bad-practice. Online; Accessed: 2016-02-24.

[Stae] Stackoverflow. Writing modular JS. https://addyosmani.com/writing-modular-js/. Online; Accessed: 2016-07-02.

[TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.

[TMK15] Nikolaos Tsantalis, Davood Mazinanian, and Giri P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, Nov 2015.

[Tru] Javascript Namespaces and Modules. https://www.kenneth-truyers.net/2013/04/27/javascript-namespaces-and-modules/. Online; Accessed: 2016-02-24.

[typ] TypeScript. https://www.typescriptlang.org/. Online; Accessed: 2016-07-02.

[vMVH97] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, 1997.

[vV08] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley, 3 edition, 6 2008.