IDENTIFICAÇÃO DE CLASSES EM SISTEMAS LEGADOS JAVASCRIPT

LEONARDO HUMBERTO GUIMARÃES SILVA

IDENTIFICAÇÃO DE CLASSES EM SISTEMAS LEGADOS JAVASCRIPT

Tese apresentada ao Programa de Pós--Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Alexandre Bergel

Belo Horizonte
Setembro de 2017

LEONARDO HUMBERTO GUIMARÃES SILVA

IDENTIFYING CLASSES IN LEGACY JAVASCRIPT CODE

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: Alexandre Bergel

Belo Horizonte

September 2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Silva, Leonardo Humberto Guimarães.

S586i Identifying classes in legacy javaScript code. / Leonardo Humberto Guimarães Silva. – Belo Horizonte, 2017.
xxxiii, 171 f.: il.; 29 cm.

Tese (doutorado) – Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Alexandre Bergel

1. Computação - Teses. 2. JavaScript (Linguagem de programação de computador). 3. Engenharia reversa. 4. Compreensão de programas. 5. Reengenharia. I. Orientador. II. Coorientador III. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Identifying classes in legacy javaScript code

LEONARDO HUMBERTO GUIMARÃES SILVA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. MARCO TULIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. ALEXANDRE BERGEL - Coorientador
Departamento de Ciência da Computação - University of Chile

Eduardo Figueiredo

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Mariza Andrade da Silva Bigonha
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

[illegible]

PROF. PAULO HENRIQUE MONTEIRO BORBA
Departamento de Informática - UFPE

Belo Horizonte, 15 de setembro de 2017.

Acknowledgments

I would like to express my gratitude to everyone who walked with me through the path way on this PhD thesis. This work would have been impossible without their support, encouragement, and guidance. I thank God for having all these people in my life.

A special thanks for my mother Helenita, my wife Cristiane, and my daughter Isadora, who gave me the emotional support I needed.

I would like to thank my advisor Prof. Marco Túlio de Oliveira Valente and my co-advisor Prof. Alexandre Bergel for the lessons, attention, availability, and patience.

I thank Prof. Nicolas Anquetil for giving me the opportunity to work under his supervision in France.

I thank the members of the ASERG research group for their friendship and technical collaboration.

I thank the Department of Computer Science at UFMG for their constant support.

Finally, I thank the remaining members of my thesis committee: Prof. Paulo Borba, Prof. Eduardo Figueiredo, Prof. Marcelo Maia, and Prof. Mariza Bigonha, for participating in my defense and for their insightful comments.

"The greatest danger for most of us is not that our aim is too high and we miss it, but that it is too low and we reach it."
(Michelangelo)

Resumo

JavaScript é a linguagem de programação mais popular para a Web. Embora a linguagem seja baseada em protótipos, desenvolvedores JavaScript muitas vezes emulam classes para diminuir a crescente complexidade de suas aplicações. Identificar estruturas semelhantes a classes pode auxiliar estes desenvolvedores nas seguintes atividades: (i) compreensão de programas; (ii) migração de código para a nova sintaxe de classes introduzida na versão 6 de ECMAScript (ES6); e (iii) implementação de ferramentas de apoio, incluindo IDEs com visões baseadas em classes e ferramentas de engenharia reversa. Nesta tese, nós definimos, implementamos e avaliamos um conjunto de heurísticas para identificar estruturas que emulam classes, e suas dependências, em sistemas JavaScript legados, isto é, implementados em versões anteriores a ES6. Desenvolvemos um amplo estudo, utilizando um *dataset* de 918 aplicações JavaScript, disponíveis no GitHub, para entender como a emulação de classes é empregada. Encontramos evidências de que estruturas emulando classes estão presentes em quase 70% dos sistemas estudados. Realizamos um estudo com 60 desenvolvedores para avaliar nossa estratégia e ferramenta. Os resultados indicam que nossa ferramenta é capaz de identificar classes emuladas em sistemas legados JavaScript. Avaliamos também como o uso de um verificador estático pode inferir tipos que correspondem a referências para classes. Realizamos um estudo com duas aplicações legadas para avaliar a precisão na identificação de dependências entre classes. Conseguimos atingir precisão de 100% em ambos os sistemas, e o *recall* varia de 80% a 86% para dependências em geral e de 85% a 96% para associações. Além disso, apresentamos um conjunto de regras para migrar estruturas que emulam classes para usar a nova sintaxe introduzida em ES6. Em nosso estudo, detalhamos casos que permitem migração automática (as partes boas), casos que exigem intervenção manual e *ad-hoc* (as partes ruins) e casos que não podem ser migrados devido a limitações de ES6 (as partes feias). Finalmente, apresentamos razões que podem levar desenvolvedores a adiar ou rejeitar a adoção de classes ES6, com base no *feedback* recebido após a submissão de *pull-requests* sugerindo a migração.

Palavras-chave: JavaScript; Compreensão de programas; Engenharia reversa.

Abstract

JavaScript is the most popular programming language for the Web. Although the language is prototype-based, developers often emulate class-based abstractions in JavaScript to master the increasing complexity of their applications. Identifying structures similar to classes in JavaScript code can support these developers in the following activities: (i) program comprehension; (ii) migration to the new JavaScript syntax that supports classes, introduced by ECMAScript 6 (ES6); and (iii) implementation of supporting tools, including IDEs with class-based views and reverse engineering tools. In this thesis, we define, implement, and evaluate a set of heuristics to identify class-like structures, and their dependencies, in legacy JavaScript code, *i.e.*, code implemented in versions prior to ES6. We report on a large and in-depth study to understand how class emulation is employed, using a dataset of 918 JavaScript applications available on GitHub. We found evidence that structures emulating classes are present in almost 70% of the studied systems. We perform a field study with 60 developers to evaluate the accuracy of our strategy and tool. The results indicate that our tool is able to identify class-like structures in legacy JavaScript systems. We also demonstrate how to use a static type-checker to infer types that correspond to class references. We perform a study with two open-source applications aiming to measure the accuracy of the proposed approach to identify class-to-class dependencies. We achieve precision of 100% in both systems, and the values of recall ranges from 80% to 86% for dependencies in general and from 85% to 96% for associations. Moreover, we present a set of rules to migrate class-like structures to use the new ES6 class syntax. In our study, we detail cases that are straightforward to migrate (the good parts), cases that require manual and ad-hoc migration (the bad parts), and cases that cannot be migrated due to limitations and restrictions of ES6 (the ugly parts). Finally, we present a set of reasons that can lead developers to postpone or reject the adoption of ES6 classes, based on the feedback received after submitting pull requests suggesting the migration.

Keywords: JavaScript; Program comprehension; Reverse engineering.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter, we start by presenting the context of this thesis (Section 1.1) followed by a brief introduction about class emulation in JavaScript (Section 1.2). Next, we state our problem and motivation (Section 1.3). We discuss our objectives, contributions, and an overview of a technique proposed for identifying classes in legacy JavaScript applications in Section 1.4. Then, we present the publications derived from this thesis (Section 1.5). Finally, we present the thesis outline (Section 1.6).

1.1 Context

JavaScript is the most popular programming language for the Web. The language was initially designed in the mid-1990s to extend Web pages with small executable code. Since then, its popularity and relevance have only grown [Nederlof et al., 2014, Kienle, 2010]. For example, Richards et al. [2010] report that the language is used by 97 out of the web's 100 most popular sites. JavaScript is also the most popular language on GitHub, including newly created repositories. Figure 1.1 shows the number of GitHub repositories, per language, with more than 1,000 stars, as of May 2017. In this bar plot, we can see that JavaScript has almost three times the number of repositories than the second language, which is Java. To mention another example, among the top-2,500 most popular systems on GitHub, according to their number of stars, 34.2% are implemented in JavaScript [Borges et al., 2016]. Concomitantly with its increasing popularity, the size and complexity of JavaScript software is in steady growth. The language is now used to implement mail clients, office applications, and IDEs, which can easily reach hundreds of thousands of lines of code.[1]

[1]http://sohommajumder.wordpress.com/2013/06/05/gmail-has-biggest-collection-of-javascript-code-lines-in-the-world

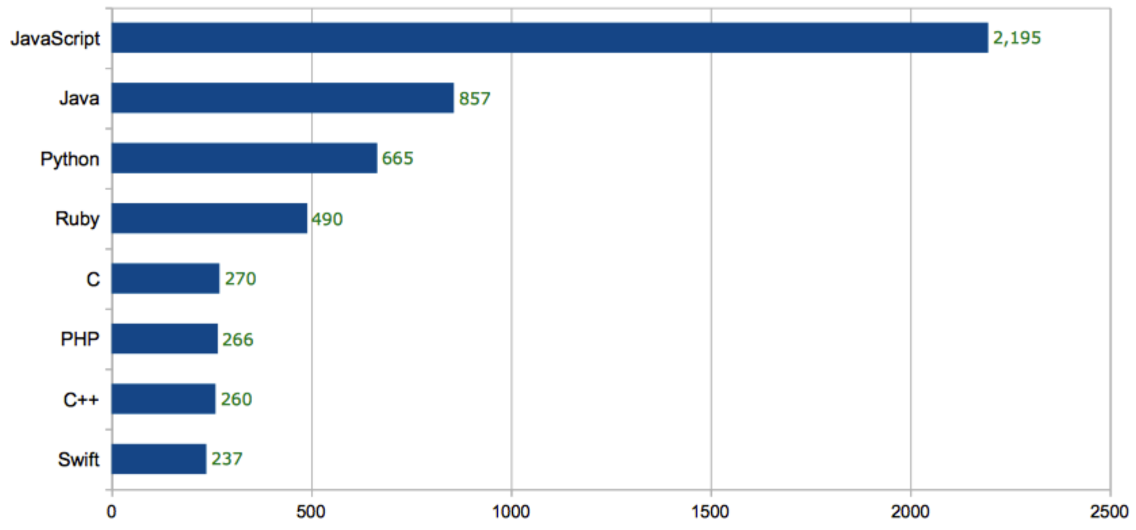


Figure 1.1: GitHub repositories, per language, with more than 1,000 stars (http://gittrends.io)

One of the most interesting developments gaining popularity in the JavaScript space is `Node.js`. This framework is a complete server-side JavaScript environment for developing high-performance and concurrent programs. `Node.js` has become one of the main elements of the "JavaScript everywhere" paradigm [Pereira, 2016], allowing web application development to unify around a single programming language, rather than relying on another language for writing the server-side of an application.

As a result of JavaScript's increasing popularity, there is a growing demand for solutions that assist programmers in their daily tasks, such as techniques for program comprehension and maintenance [Madsen et al., 2015, Nguyen et al., 2014], smells detection and localization [Fard and Mesbah, 2013, Artzi et al., 2012], refactoring [Gama et al., 2012, Feldthaus et al., 2011a,b], and preventing security attacks [Guha et al., 2009, Vogt et al., 2007, Yu et al., 2007].

1.2 JavaScript in a Nutshell

JavaScript is an imperative and object-based language centered on prototypes, rather than a class-based language [Guha et al., 2010, Crockford, 2008]. However, developers can emulate class-based abstractions, *i.e.*, data structures including attributes, methods, class constructors, and inheritance, using the prototype-based object system of the language, which is part of JavaScript since its first version [Flanagan, 2011]. Functions and prototypes can be used in JavaScript to support the implementation of structures including both data and code and that are further used as a template for the creation of objects. In this thesis, we use the term classes to refer to such structures, since they have a similar purpose as the native classes of mainstream object-oriented languages.

Listing 1.1 shows part of the code used to implement a *Linked List* in the system ALGORITHMS.JS.[2] This code is implemented according to version 5 of ECMAScript (ES5), which is a scripting-language specification that defines the syntax of JavaScript language [ECMA-International]. We can notice a function that emulates the class constructor (lines 2-8), the initialization of attributes (lines 4-6), and functions used to implement the list operations (lines 10-13), linked to the `prototype` property. In fact, this example shows one possible implementation for a class in JavaScript; there are other variations, *e.g.*, using anonymous/non-anonymous functions, defining properties inside/outside the constructor function, with/without using the prototype.

```
 1 // function -> class
 2 function LinkedList() {
 3   // properties -> attributes
 4   this._length = 0;
 5   this.head = null;
 6   this.tail = null;
 7   ...
 8 }
 9 // functions -> methods
10 LinkedList.prototype.isEmpty = function () { ... };
11 LinkedList.prototype.add = function (n, index) { ... };
12 LinkedList.prototype.del = function (index) { ... };
13 LinkedList.prototype.forEach = function (fn) { ... };
```

Listing 1.1: Example of class emulated in ALGORITHMS.JS

Probably motivated by the widespread use of class emulations in JavaScript, the newer standard version of the language, named ECMAScript 6 (ES6), includes syntactical support for classes [ECMA-International, 2015]. In this new language version, it is possible to implement classes using a syntax very similar to the one provided by class-based object-oriented languages, such as Java and C++. Listing 1.2 shows the same example of Listing 1.1, but using the new syntax for classes. We can see the keywords `class` (line 1) and `constructor` (line 2), and the organization of attributes (lines 4-6) and methods (lines 10-13) in the body of class `LinkedList`. However, a recent study shows that JavaScript developers are not fully aware of the changes introduced in ES6, and very few are using the new class syntax [Hafiz et al., 2016].

[2]https://github.com/felipernb/algorithms.js

```
class LinkedList {
  constructor() {
    // attributes
    this._length = 0;
    this.head = null;
    this.tail = null;
    ...
  }
  // methods
  isEmpty () { ... }
  add (n, index) { ... }
  del (index) { ... }
  forEach (fn) { ... }
}
```

Listing 1.2: Example of class using the new syntax provided by ES6

Therefore, we currently have a large codebase of *legacy* JavaScript source code, *i.e.*, applications that do not use the syntactic support for classes that comes with ES6. To mention an example, GitHub has currently over three million active repositories whose main language is JavaScript.[3]

In this thesis, we employ the term *legacy* to refer to JavaScript applications implemented in versions prior to ECMAScript 6.

1.3 Motivation and Problem

In this section, we highlight two challenges related to comprehension and maintenance of legacy JavaScript applications: (i) recognition of class-like structures; and (ii) detection of class-to-class dependencies in JavaScript.

Recognition of class-like structures in legacy JavaScript

Developers of mainstream object-oriented languages have the support of established reengineering techniques and design patterns to implement and maintain classes [Demeyer et al., 2009, Booch et al., 2004, Fowler, 2003]. On the other hand, JavaScript developers cannot yet benefit from such techniques to reduce the complexity of the class-like structures in their systems. Undoubtedly, maintaining a program without support for recognizing its building blocks (modules, namespaces, classes, and reusable patterns) is a challenging task.

[3]http://githut.info/

Indeed, in an empirical study presented in Chapter 3, we found evidence that structures emulating classes are present in almost 70% of the systems in a dataset of 918 legacy JavaScript systems [Silva et al., 2017]. We highlight that identifying classes in legacy JavaScript code is important for two major reasons. Firstly, it can support developers to migrate their code to ES6. Secondly, it opens the possibility to implement a variety of analysis tools for legacy JavaScript code, including IDEs with class-based views, bad smells detection tools, reverse engineering tools, and techniques to detect violations and deviations in class-based architectures.

Detection of class-to-class dependencies in JavaScript

Besides the identification of class-like structures, another challenge consists in detecting the dependencies between such structures. These dependencies form the basis to provide, for example, class diagrams for JavaScript applications. Accurately identifying dependencies between software components is essential in software maintenance tasks [Sangal et al., 2005a, Laval et al., 2009]. In a statically typed language (*e.g.,* Java), dependencies are expressed in the type of a program structure (*e.g.,* method definitions, variables, class references). Dependencies are therefore explicitly declared in the source code, in most cases. However, dynamically-typed languages, including JavaScript, do not offer type information which significantly raises the difficulty to extract dependencies.

In order to identify class-to-class dependencies in legacy JavaScript code, we can rely on type inference algorithms, which are usually based on static code analysis. Indeed, some type inferencer tools have been proposed for JavaScript [Hackett and Guo, 2012, Anderson et al., 2005]. Oddly, no attempt has been made to evaluate the accuracy of such tools to retrieve dependencies in legacy JavaScript programs, as far as we know.

1.4 Objectives and Contributions

JavaScript developers frequently emulate classes to master the complexity of their legacy systems, although the language is not class-based (at least, until recently). In order to address the lack of proper tools and solutions to support these developers to maintain their systems, **this thesis has two major objectives**:

1. to propose, implement, and evaluate a set of heuristics to identify class-based structures, and their dependencies, in legacy JavaScript code; and

2. to propose, implement, and evaluate a set of rules to migrate class-like structures from ES5 to ES6.

The contributions achieved with this thesis can be summarized as follows:

- **A set of heuristics to identify classes in legacy JavaScript code.** We propose, implement, and evaluate a set of heuristics to identify class-based structures in legacy JavaScript code. Using these heuristics, which are specified in Chapter 3, we also provide a thorough study on the usage of classes in a dataset of 918 JavaScript systems available on GitHub. This study aims to answer the following research questions: (RQ #1) "Do developers emulate classes in legacy JavaScript applications?", (RQ #2) "Do developers emulate subclasses in legacy JavaScript applications?", (RQ #3) "Is there a relation between the size of a JavaScript application and the number of class-like structures?", (RQ #4) "What is the number of attributes and methods of the classes emulated in legacy JavaScript code?". We also report a field study with 60 JavaScript developers to validate our findings and heuristics to detect classes.

- **An open-source supporting tool**, called JSClassFinder [Silva et al., 2015a], that practitioners can use **to detect and inspect classes in legacy JavaScript code**.

- **An evaluation of a static type inferencer tool to detect class-to-class dependencies** in legacy JavaScript. In Chapter 4, we show how to use Flow[4], a static type-checker for JavaScript, to infer types that correspond to class references. We perform a study with two open-source applications aiming to answer the following research questions: (RQ #1) "What is the accuracy of Flow in detecting class-to-class dependencies?", (RQ #2) "Can we improve the accuracy of Flow by expanding `require` statements?".

- **A set of rules to migrate class-like structures from ES5 to ES6**. We investigate the feasibility of rejuvenating legacy JavaScript code and, therefore, to increase the chances of code reuse in the language. Specifically, in Chapter 5, we propose a set of migration rules based on the way that classes are emulated in ES5. We also describe an experiment on migrating eight real-world JavaScript systems to the native syntax for classes provided by ES6. We first use JSClassFinder to identify class like structures in the selected systems. Then we apply the proposed rules to convert these classes to use the new syntax. In our study, we document:

[4]`https://flow.org/`

(a) cases that are straightforward to migrate (the good parts); (b) cases that require manual and ad-hoc migration (the bad parts); and (c) cases that cannot be migrated due to limitations and restrictions of ES6 (the ugly parts).

- **An open-source supporting tool**, called JSClassRefactor[5], that can be used **to migrate legacy code to the new syntax for classes that comes with ES6**. This tool allows to refactor legacy code according to the proposed migration rules and to warn developers about the cases that require manual intervention.

- **We document a set of reasons that can lead developers to postpone/reject the adoption of ES6 classes**. We collect the perceptions of the developers of eight JavaScript systems about migrating their code to the new syntax for classes.

1.5 Publications

Part of the work contained in this thesis is derived from the following publications, presented in chronological order:

- Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, Anne Etien. Identifying Classes in Legacy JavaScript Code. *Journal of Software: Evolution and Process*, 2017, pages 1-37. (Chapter 3)

- Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel. Refactoring Legacy JavaScript Code to Use Classes: The Good, The Bad and The Ugly. *In 16th International Conference on Software Reuse (ICSR)*, 2017, pages 155–171. (Chapter 5)

- Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel. Statically Identifying Class Dependencies In Legacy JavaScript Systems: First Results. *In 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Early Research Track, 2017, pages 427–431. (Chapter 4)

- Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Nicolas Anquetil, Alexandre Bergel. Does Javascript Software Embrace Classes? *In 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 73–82. (Chapter 3)

[5]`https://github.com/leonardo-silva/JSClassRefactor`

- Leonardo Humberto Silva, Daniel Felix, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, Anne Etien. JSClassFinder: A Tool to Detect Class-like Structures in JavaScript. *In 6th Brazilian Conference on Software: Theory and Practice (CBSoft Tools Track)*, 2015, pp. 113–120. (Chapter 3 - Section 3.1.2)

1.6 Thesis Outline

This thesis is structured in the following chapters:

- Chapter 2 describes background information on JavaScript and other scripting languages, dynamic analysis, class identification, refactoring support, code smells, and JavaScript patterns. We also discuss work on type inference for JavaScript.
- Chapter 3 proposes a set of heuristics to analyze class-like structures in JavaScript systems. We provide a thorough study on the usage of classes in a dataset of 918 JavaScript systems available on GitHub. We also report a field study with 60 JavaScript developers to validate our findings and heuristics to detect classes.
- Chapter 4 describes an evaluation on the usage of Flow, a JavaScript type-checker, to identify class-to-class dependencies in legacy applications.
- Chapter 5 investigates the challenges related to the migration of class-like structures from ES5 to the new syntax for classes provided by ES6. We propose a set of rules to perform the migration of legacy code. We document the limitations of these rules, *i.e.*, a set of cases where manual adjusts are required to migrate the code. We also document the limitations of the new syntax for classes provided by ES6, *i.e.*, the cases where it is not possible to migrate the code. Finally, we present a set of reasons that can lead developers to postpone/reject the adoption of ES6 classes.
- Chapter 6 concludes this thesis and outlines future work ideas.

Chapter 2

Background

In this chapter, we present background information related to this PhD thesis. First, we present central concepts about the JavaScript language (Section 2.1) highlighting the emulation of classes in JavaScript legacy code (Section 2.1.1). We discuss studies related to dynamic evaluation (Section 2.2), class identification and refactoring support (Section 2.3), code smells and JavaScript patterns (Section 2.4), and dynamic and static analysis for JavaScript systems (Section 2.5), which are related to our work. Moreover, we present a brief review on type inference for JavaScript (Section 2.6), topic that has a central role in Chapter 4. Finally, we conclude this chapter with general remarks on the discussed topics (Section 2.7).

2.1 JavaScript Overview

JavaScript is part of the triad of technologies used by most web developers: HTML to specify the content of web pages, CSS to specify the presentation, and JavaScript to specify the behavior of such pages. The language was originally created in 1995, by Brendan Eich, and first released with Netscape 2, early in 1996. Several months later, Netscape submitted JavaScript to ECMA International, a European standards organization, which resulted in the first edition of the ECMAScript standard. The standard received a significant update as ECMAScript edition 3 in 1999. The fourth edition was abandoned, due to political differences concerning language complexity. Many parts of the fourth edition formed the basis for ECMAScript edition 5, published in December of 2009, and for the 6th edition, published in June of 2015 [ECMA-International, 2015].

JavaScript is a loosely-typed dynamic programming language where everything is an object. Each object contains a set of properties that represent data (other objects) and operations (function objects). These properties are always public. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. The language has a class-free object system in which objects inherit properties directly from other objects [Stefanov, 2010]. In other words, JavaScript does not have native support for classes.

JavaScript typically relies on a runtime environment (*e.g.*, a Web browser) to provide objects and methods by which scripts can interact with, providing behavior to HTML pages. This is the most commonly used mechanism to provide dynamic interaction in web applications, as well as asynchronous interactions with the server [Powell, 2008]. Client-side JavaScript code can usually be embedded within HTML documents in four different ways:

- Inline, between a pair of < `script` > and < `/script` > tags;
- From an external file specified by the `src` attribute of a < `script` > tag;
- In an HTML event handler attribute, such as `onclick` or `onmouseover`;
- In a URL that uses the special `javascript` protocol.

The subsections that follow explain in more details the different styles for class emulation in JavaScript (Section 2.1.1), the new syntax from class emulation that comes with ECMAScript 6 (Section 2.1.2), and scripting languages that work on top of JavaScript (Section 2.1.3) .

2.1.1 Class Emulation and Prototypes

This section describes different mechanisms to emulate classes in legacy JavaScript code. To identify these mechanisms we conducted an informal survey on documents available on the web, including tutorials[1], blogs[2], and StackOverflow discussions[3]. We surveyed a catalogue of five encapsulation styles for JavaScript proposed by Gama et al. [2012] and JavaScript books targeting language practitioners [Crockford, 2008, Flanagan, 2011]. We also surveyed the developer of a real JavaScript project to tune our tool and strategy. The application SELECT2[4] was used for this purpose.

[1]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript`

[2]`http://javascript.crockford.com/prototypal.html`

[3]`http://stackoverflow.com/questions/387707/whats-the-best-way-to-define-a-class-in-javascript`

[4]`https://select2.github.io/`

Basically, an *object* in JavaScript is a set of name-value pairs. Methods and variables are called *properties* and their values can be any objects, including immediate values (*e.g.*, numbers, boolean) and functions. To implement *classes* in JavaScript, *i.e.*, data structures that resemble the class concept of mainstream object-oriented languages, the most common strategy is to use functions. Particularly, any function can be used as template for the creation of objects. When a function is used as a class constructor, its `this` is bound to the new object being constructed. Variables linked to `this` are used to define properties that emulate attributes and methods. If a property is an inner function, then it represents a *method*, otherwise, it is an *attribute*. The commands `new` and `Object.create()` are used to instantiate classes.

To illustrate the definition of classes in JavaScript, we use a simple `Circle` class. Listing 2.1 presents the function that defines this class (lines 1-8), which includes two attributes (`radius` and `color`) and two methods (`getArea` and `setColor`). Functions used to define methods can be implemented directly inside the body of the class constructor, like method `getArea` (lines 4-6), or outside, like method `setColor` (lines 9-11). An instance of the class `Circle` is created with the command `new` (line 13).

```
 1 function Circle (radius, color) {  // function -> class
 2   this.radius = radius;    // property -> attribute
 3   this.color = color;    // property -> attribute
 4   this.getArea = function() { // function -> method
 5     return (3.14 * this.radius * this.radius);
 6   }
 7   this.setColor= setColor;  // function -> method
 8 }
 9 function setColor(c) { // function
10   this.color = c;  // property -> attribute
11 }
12 // Circle instance -> object
13 var myCircle = new Circle (10, 0x0000FF);
```

Listing 2.1: Class declaration and object instantiation

In prototype-based languages, objects inherit their properties and methods from their prototypes. New objects are produced by copying and modifying prototypes, rather than by instantiating classes. Therefore, a `prototype` can be seen as a standard model instance [Borning, 1986]. In JavaScript, each object has an implicit `prototype` property that refers to another object (all objects in JavaScript initially descend from a base class called `Object`). To evaluate an expression like `obj.p`, in JavaScript, the runtime starts searching for property `p` in `obj`, then in `obj.prototype`, then in `obj.prototype.prototype`, and so on until it finds the desired property or fails in its search. When an object is created using `new C`, for example, its `prototype` is set to the `prototype` of the function `C`, which by default is defined as pointing to `Object`. Therefore, a chain of prototype links usually ends at `Object`.

By manipulating the `prototype` property of a function, we can define a method whose implementation is shared by all object instances of that function. It is also possible to define attributes shared by all objects of a given class, akin to static attributes in class-based languages. In Listing 2.2, `Circle` includes a `pi` *static* attribute and a `getCircumference` method. It is worth noting that `getCircumference` is not a method attached to the class (as a `static` method in Java). It has for example access to the variable `this`, whose value is not determined using lexical scoping rules, but instead using the caller object.

```
// prototype property -> static attribute
Circle.prototype.pi = 3.14;
// function -> method
Circle.prototype.getCircumference= function () {
  return (2 * this.pi * this.radius);
}
```

Listing 2.2: Using `prototype` to define *methods* and *static attributes*

Prototypes can also be used to build inheritance hierarchies [Ungar and Smith, 1987, Borning, 1986]. In JavaScript, we can consider that a class `C2` is a *subclass* of `C1` if `C2`'s prototype refers to `C1`'s prototype or to an instance of `C1`. For example, Listing 2.3 shows a class `Circle2D` that extends `Circle` with its position in a Cartesian plane.

```
function Circle2D (x, y) {  // class Circle2D
  this.x = x;
  this.y = y;
}
// Circle2D is a subclass of Circle
Circle2D.prototype = new Circle(10, 0x0000FF);
// Circle2D extends Circle with new methods
Circle2D.prototype.getX = function () {
  return (this.x);
}
Circle2D.prototype.getY = function () {
  return (this.y);
}
```

Listing 2.3: Implementing *subclasses*

Alternatively, the *subclass* may refer directly to the prototype of the *superclass*, which is possible using the `Object.create()` method. This method creates a new object with the specified prototype object, as illustrated by the following code:

```
Circle2D.prototype = Object.create(Circle.prototype)
```

Table 2.1 summarizes the mechanisms presented in this section to map class-based object-oriented abstractions to JavaScript abstractions.

Table 2.1: Class-based languages vs JavaScript

Class-based languages	JavaScript
Class	Function
Attribute	Property
Method	Inner function
Static attribute	Prototype property
Inheritance	Prototype chaining

2.1.2 ECMAScript 6 Classes

ECMAScript 6 (ES6), released in 2015, includes, among other features, a syntactical support to classes.[5] For example, ES6 supports the following class definition:

```
class Circle {
  constructor (radius) {
    this.radius = radius;
  }
  getArea() {
    return (3.14 * this.radius * this.radius);
  }
}
```

However, this support to classes does not impact the semantics of the language, which remains prototype-based. For example, the previous class is equivalent to the following code:

```
function Circle (radius) {
  this.radius = radius;
}
Circle.prototype.getArea = function () {
  return (3.14 * this.radius * this.radius);
}
```

As we can see in the prior example, the new syntax for classes does not bring any new features to the language, it represents a "sugar" over the existing standard. In other words, we can accomplish the same results emulating classes using ECMAScript

[5]https://developer.mozilla.org/en/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla

version 5. However, it is possible that exposing classes according to the ES6 standard may encourage developers that have background in other class-based languages, such as Java and Smalltalk, to implement JavaScript systems.

2.1.3 Script Languages on Top of JavaScript

CoffeeScript is a language that compiles one-to-one into JavaScript code and aims to expose the "good parts of JavaScript" by only simplifying the language's syntax [Bates, 2012, MacCaw, 2012]. The creators of CoffeeScript state that it is better to write code in CoffeScript than writing pure JavaScript simply because there is less code to write. Listing 2.4 shows examples of snippets of code in CoffeScript compiled to JavaScript output. The last snippet presented is related to the emulation of classes.

```
# Assignment:                  | var number, opposite, square, Animal;
number    = 12                 | number = 12;
opposite = true                | opposite = true;
                               |
# Conditions:                  | if (opposite) {
number = -12 if opposite       |    number = -12;
                               | }
# Functions:                   |
square = (x) -> x * x          | square = function(x) {
                               |    return x * x;
# Classes:                     | };
class Animal                   |
constructor: (name) ->         | Animal = (function() {
@name = name                   |    function Animal(name) {
                               |       this.name = name;
                               |    }
                               |    return Animal;
                               | })();
```

Listing 2.4: CoffeeScript on the left compiled to JavaScript on the right

TypeScript[6] is another language that compiles to plain JavaScript. While every JavaScript program is a TypeScript program, TypeScript offers a module system, classes, interfaces and a gradual type system. The support for classes is aligned with proposals currently standardized for ECMAScript 6. Listing 2.5 shows an example of a class `Point` implemented in TypeScript (lines 1-11) and its instantiation (line 12). In this example, the attributes `x` and `y` are typed.

[6] www.typescriptlang.org/

```
class Point {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
  getDist() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}
var p = new Point(3,4);
```

Listing 2.5: Class emulation using TypeScript syntax

2.2 Dynamic Evaluation

In order to dynamically evaluate JavaScript code, the language provides the `eval` function to turn text into executable code at runtime [Flanagan, 2011, Crockford, 2008]. It parses a string argument as source code and immediately executes it. When invoked, `eval` executes with the privileges of the caller and returns the result of the last evaluated expression, or propagates any thrown exception. The `eval` function may be invoked directly, with the evaluated code having access to the variables lexically in scope, or indirectly, through an alias, with the evaluated code executing in the global scope. The use of `eval` has been discouraged[7], mainly because: (i) one may explore it to execute malicious code; (ii) it is generally slower than the alternatives because it cannot be previously optimized. Its use also makes more difficult any approach that uses static analysis because the code it executes is only known at runtime.

Richards et al. [2010] investigated a broad range of JavaScript dynamic features, not restricted to the use of `eval`. The goal was to characterize JavaScript program behavior by analyzing execution traces recorded from a corpus of real-world programs. To obtain those traces they instrumented and interacted with 103 web sites. The execution traces were then analyzed to produce behavioral data about the programs. In addition to web sites, they also analyzed three benchmark suites. They relied on traditional program metrics as well as metrics that are more indicative of the degree of dynamism exhibited by JavaScript programs. The authors concluded that libraries often change the prototype links dynamically, but such changes are restricted to built-

[7] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

in types, like `Object` and `Array`, and changes in user-created types are more rare. The authors also reported that most JavaScript programs do not delete attributes from objects dynamically.

The authors also conducted a large-scale study on the use of `eval` in JavaScript based on a corpus of more than 10,000 popular web sites [Richards et al., 2011]. The goal was to quantify the use of `eval` in web applications. For all web page executions, they obtained behavioral data with the aid of an instrumented JavaScript interpreter. In addition to page-load program executions, they used a random testing approach to automatically generate user input events to explore the state space of web applications. They also interacted manually with approximately 100 web sites to generate meaningful interactions. The authors concluded that `eval` is popular and not necessarily harmful, although its use can be replaced with equivalent and safer code or language extensions in most usage scenarios. Moreover, it is usually considered a good practice to use `eval` when loading scripts or data asynchronously.

2.3 Class Identification and Refactoring Support

Gama et al. [2012] proposed an automated approach to refactor JavaScript code to a single object-oriented style. After surveying JavaScript books, web tutorials and a set of 40 JavaScript applications (sizes ranging from 800 to 85,000 lines of code), the authors identified five styles for implementing methods in JavaScript: inside/outside constructor functions; using anonymous/non-anonymous functions; and using prototypes. The authors also reported examples of applications that use built-in functions and extensions to provide customized styles for class emulation. They claim that mixing styles in the same code may hinder program comprehension and make maintenance more difficult. Based on that, they proposed to modify the source code of a given application transforming all occurrences of the five styles they identified in one single style, the one using prototypes.[8] Their transformation process can be divided in two main steps:

1. Search for classes and tag them with the current implementation style. To distinguish classes from simple functions, the authors check if there is an object that is instantiated from that class using the keyword `new`.

2. Look for instances of each implementation style markup and transform the methods to prototype style.

[8]An example of the style for class implementation using prototypes can be seen in Listing 2.3

Although the authors identify different styles for method implementation, their approach does not address the problem of identifying classes in legacy JavaScript code. Instead, their purpose is to standardize a single programming style (*i.e.*, a normal form) for defining methods.

Feldthaus et al. [2011a,b] describe a methodology for implementing automated refactorings on a nearly complete subset of the JavaScript language (ECMAScript 5). The authors specified and implemented three refactorings: *rename property*, *extract module*, and *encapsulate property*. *Rename property* is similar to the refactoring *rename field* for typed languages. The main difference is that while fields in Java, for example, are statically declared within class definitions, properties in JavaScript are associated with dynamically created objects and are themselves dynamically created after first write. Besides, JavaScript has the ability to dynamically delete properties, change the prototype hierarchy, or reference a property by specifying its name as a dynamically computed string. The goal of the refactoring *extract module* is to use anonymous functions to make global functions become local. These anonymous functions will then return object literals with properties through which the previous global functions can be invoked. This allows the use of global variables representing modules that encapsulate a group of related functions. Finally, the *encapsulate property* refactoring can be used to encapsulate state by making a field private and redirecting access to that field via introduced getter and setter methods. It targets constructor functions that emulate classes in JavaScript. To determine if a function is used as a constructor, they look for functions that initialize an object when invoked, like those that are invoked with the commands `new` or `Object.create`().

2.4 Code Smells and JavaScript Patterns

A code smell is an indication that usually corresponds to a deeper problem in a system. The term was first coined by Kent Beck and Martin Fowler in their book about refactoring [Fowler and Beck, 1999].

Fard and Mesbah [2013] proposed a set of 13 JavaScript code smells, including seven generic smells (*e.g.*, long functions and dead code) and six smells specific to JavaScript (*e.g.*, creating closures in loops and accessing `this` in closures). They described a tool, called JSNose, for detecting code smells based on a combination of static and dynamic analysis. They also investigated 11 web applications to find out which smells are more prevalent. At a high level, their approach intercepts the JavaScript code of a given web application by setting up a proxy between the server and the browser, and parses the source code (`.js` and HTML files) into an Abstract Syntax Tree (AST). They analyze the AST by visiting all program entities, objects, properties, functions, and code blocks, and store their structure and relations. They extract patterns from the AST such as names of objects and functions, and infer JavaScript objects, their types, and properties by querying the browser at runtime. Finally, based on the static and dynamic data collected, they calculate the necessary metrics for code smells detection. Among the smells detected by their tool, lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables, are the most prevalent ones. Further, they report a strong positive correlation between the types of smells and lines of code, number of functions, number of JavaScript files, and cyclomatic complexity. However, among the proposed patterns for code smells, only Refused Bequest is directly related to class-emulation in JavaScript. In fact, this smell was originally proposed to class-based languages [Fowler and Beck, 1999, Lanza and Marinescu, 2006], to refer to subclasses that use only some of the methods and properties inherited from their parents.

Nguyen et al. [2014] presented an approach to mine JavaScript usage patterns in web applications where JavaScript usages involve unnamed data objects whose types are not statically revealed. Usage patterns are particularly useful for documentation, code smell and anomaly detection, defect and vulnerability detection, source code search and code completion features. The authors introduced JSModel, a graph representation for JavaScript code, and JSMiner, a tool that mines inter-procedural and data-oriented JavaScript usage patterns. Their experiments also showed JSModel's usefulness in detecting buggy patterns and documenting JS APIs.

Nicolay et al. [2015] presented an abstract machine for a core JavaScript-like language that tracks write side-effects in JavaScript functions to detect their purity. Anything a function does, besides producing a value, is called a side-effect. A function is considered pure only if it does not generate observable side-effects. The authors point out that purity aids program understanding, specification, testing, debugging, and maintenance. The proposed abstract machine generates appropriate write effects caused by writing to variables and object properties. Starting from an initial evaluation state, all possible successor states are explored resulting in a flow graph. They implemented a purity analysis for a subset of JavaScript, and experimented with it on common JavaScript benchmarks.

2.5 Dynamic and Static Analysis

Dynamic analysis contrasts static analysis approaches, where a program's text (source code) is examined to derive properties that hold for all executions. One advantage of static analysis is that it is not necessary to execute a program to analyze it. Without program execution, it is not necessary to set a specific environment, initialize software parameters, install external dependencies, nor exercise different test cases in order to have a coverage of as many features as possible. On the other hand, dynamic analysis has the potential to discover dependencies between program entities widely separated in the execution path, which can be tricky to find statically. Considering JavaScript, for example, some characteristics make static analysis challenging: (i) the dynamic nature of the language; (ii) the absence of types; (iii) the existence of functions such as `eval` and `delete`, that can change program's structure and behavior dynamically.

Despite the differences, dynamic and static techniques can complement each other in order to improve completeness and precision of software analysis. If the results of dynamic and static analyses disagree, usually there are two possibilities: (i) the dynamic analysis is wrong because it did not cover a sufficient number of execution paths; (ii) the static analysis is wrong because it analyzed unfeasible paths (paths that can never be reached during execution). Dynamic analysis, by definition, considers fewer execution paths than static analysis [Ball, 1999]. There are different approaches that combine dynamic and static analyses to master their specific problems. As examples, we can mention Odgaard [2014] for type inference, Fard and Mesbah [2013] for code smells detection, and Alimadadi et al. [2015] to capture event-based interactions. The next subsections present in more details static and dynamic approaches to analyze JavaScript programs.

2.5.1 Static Analysis for JavaScript

Feldthaus et al. [2013] presented a flow analysis specifically designed to compute approximate call graphs for JavaScript programs. The authors proposed two variants of a field-based flow analysis for JavaScript that only tracks function objects and ignores dynamic property reads and writes. The first variant is a standard optimistic analysis that starts out with an empty call graph, which is gradually extended as new flows are discovered. The second is a pessimistic analysis that does not reason about interprocedural flow and ignore call targets that may depend on such flow, except in cases where the callee can be determined purely locally. The authors argument that these design decisions improve scalability and do not hamper precision. They implemented both techniques and performed an empirical evaluation on ten large web applications to show the feasibility of using their analyses in an IDE for JavaScript.

Gallaba et al. [2015] performed an empirical study to characterize JavaScript callback usage across a corpus of 138 JavaScript programs. They found that, on average, every 10th function definition takes a callback argument, and that over 43% of all callback-accepting function callsites are anonymous. Furthermore, the majority of callbacks are nested, more than half of all callbacks are asynchronous, and asynchronous callbacks, on average, appear more frequently in client-side code than server-side.

Ocariza et al. [2015] proposed an approach to detect inconsistencies in web applications that use MVC frameworks. Such frameworks are meant to simplify JavaScript development creating abstractions for DOM method calls. This is accomplished by giving programmers the ability to define model objects, which are then directly embedded in the HTML code. The frameworks thus eliminate the need for web programmers to explicitly set up DOM interactions in JavaScript. However, MVC frameworks are susceptible to inconsistencies between the identifiers and types of variables and functions used throughout the application. To tackle this problem, the authors suggested the use of static analysis to look for identifiers in the three main components (model, view, and controller) and then compare the collected identifiers to the inferred type information to determine any inconsistencies.

2.5.2 Dynamic Analysis for JavaScript

Zaidman et al. [2013] presented a tool, called FireDetective, to record execution traces of the JavaScript code that is executed in the browser (client) and also in the server. The level of detail used is the call level: the tool records the names of all functions and methods that were called, and in what order they were called, allowing the reconstruction of a call tree representation of each trace. The tool also records information about abstractions that are specific to the Ajax/web-domain, such as Ajax requests, DOM events, and time-outs. JavaScript function calls and Java calls (server-side) are recorded using Firefox' debugger interface and the Java VM tool interface, respectively. This has the advantage that no code needs to be instrumented, and that the approach also works for JavaScript code dynamically generated. The authors also carried out a field study with two Ajax developers that gave additional insight about how they use FireDetective for understanding complex web applications.

Alimadadi et al. [2014] presented a tool, called Clematis, for supporting software comprehension in web applications. This tool captures a detailed trace of a web application's behaviour during a particular user session through a combination of automated JavaScript code instrumentation and transformation. Afterwards, their approach transforms the trace into an abstract behavioural model, including relations within and between involved components. This model is then transformed into an interactive visualization, representing episodes of triggered causal and temporal events, related JavaScript code executions, and their impact on the dynamic DOM state. The model is then presented to the developers as an interactive visualization that depicts the creation and flow of triggered events, the corresponding executed JavaScript functions, and the mutated DOM nodes, within each episode.

Alimadadi et al. [2015] conducted an exploratory study to investigate the role of JavaScript's DOM-related event-based and dynamic features in code change propagation. For this study, they selected ten web applications that make extensive use of JavaScript on the client-side. For each DOM access that occurred during the execution, they collected the accessed entity, the JavaScript function that accesses the DOM, and the access type. Each application was manually exercised in different scenarios and the results showed that, on average, 42% of the accessed DOM elements are part of an impact path between two functions. An impact path of an entity (`P(e)`) is a directed acyclic path starting from entity `e`, where the nodes on the path are entities in the system and the edges are the directed impact relations that connect those entities. Moreover, about 14% of the executed event handlers are invoked through event propagation mechanisms. Giving the importance of DOM elements in transferring the impact of changes, the authors proposed a technique, called Tochal, that builds a hybrid system dependency graph by inferring and combing static and dynamic call graphs. Their technique ranks the detected impact set based on the relative importance of the entities in this hybrid graph.

Andreasen et al. [2017] provided a survey of dynamic analysis and test generation techniques for JavaScript. They summarized and compared existing approaches in a structured way, in order to provide an overview of this field. They concluded that the current state of the art successfully addresses the most common software goals, *i.e.*, correctness, reliability, security, privacy, and performance. However, they pointed some important research challenges that still need to be addressed. These include better support for code refactoring and program repair, analyses targeted at emerging execution platforms and usage scenarios for JavaScript, and the combination of test generation and dynamic analysis. They also suggested the development of analysis tools that are easily usable by regular developers.

2.6 Type Inference for JavaScript

A type can be considered a collection of program entities that share some common properties. An important advantage of type information is that types may be checked by the programming language interpreter or compiler. In general, we refer to the process of reasoning about unknown types as type inference. A type inference mechanism analyzes a program to infer the types of some or all of its expressions. Commonly, a type checker verifies if all the types are properly defined and used according to the semantics of the programming language. Statically typed languages, such as Java and C++, do type checking at compile-time, demanding the developers to explicitly write the types in the source code. Dynamically typed programming languages, such as JavaScript and Lisp, can only check types at runtime, leaving the source code cleaner and more flexible, without explicit type information. This "cleanliness" may please the developers but it also makes programming tasks more error-prone. Some bugs that could have been prevented at compile-time are just revealed during program's execution [Palsberg and Schwartzbach, 1991].

Flow[9] is a static type checker for JavaScript designed by Facebook. Flow employs a control-flow analysis that compilers typically perform to extract semantic information from code, and then uses this information for type inference. When Flow detects any incompatibilities during type checking it reports the errors found to the developer. Listing 2.6 is one example of code in which Flow can detect incompatible types involving a mathematical expression (line 3) and the type passed as argument of the function call (line 5). The comment in line 1, with the tag `@flow`, is manually included to inform the type checker that this file must be verified. It allows developers to check their JavaScript files gradually if wanted, instead of the whole system at once. Listing 2.7 shows the result of applying Flow to the code in Listing 2.6. As we can see, Flow indicates a type error in line 3 (a string is used as an operand in a multiplication).

```
1 /* @flow */
2 function foo(x) {  // this function expects a number as argument
3   return x * 10;
4 }
5 foo('Hello, Flow!');
```

Listing 2.6: Example of incompatible types detected by Flow

[9]http://flowtype.org/

```
foo.js:5
5: foo('Hello, Flow!');
^^^^^^^^^^^^^^^^^^^ function call
3:  return x * 10;
^ string. This type is incompatible with
3:  return x * 10;
^^^^^^ number
```

Listing 2.7: Warning messages from Flow for the code in Listing 2.6

Developers can use type annotations to better document their systems and also to help Flow during type checking, although this is not usually necessary. Flow only requires annotations at the boundaries of modules, *i.e.*, when a function declaration and its respective call are in separated modules (files). Type annotations are generally prefixed by ":", and they can be placed on function parameters, function return types and variable declarations. Listing 2.8 is an example of function with annotations in the source code. They explicitly indicate that function `size` expects a string and returns a number. In order to execute the code, Flow provides a feature to convert it back to pure JavaScript.

```
/* @flow */
function size(input: string): number {
  return input.length;
}
```

Listing 2.8: Example of function with type annotations

Anderson et al. [2005] developed a formalism for a subset of the JavaScript language, called JS_0, that includes dynamic addition/modification of fields and methods, and the creation of objects using the operator `new`. They demonstrate how type inference for JS_0 can be expressed as a finite system of constraints between type variables. Type variables are used to represent the type of an expression. Constraints represent the expected relationships between types in the program. If the combined constraints have a solution, it can be used to translate a JS_0 program into an equivalent typed version of the language. This involves annotating the JS_0 program with type declarations.

Odgaard [2014] proposed a JavaScript type inference and annotation system mechanism. His approach uses dynamic analysis based on test cases to automatically generate type annotations for JavaScript programs. The process uses a combination of static and dynamic code instrumentation, and requires two inputs: (i) an AST with the assignment of unique identifiers to each global variable declaration, function declaration, function expression, and object instantiation; (ii) the original source code instrumented by inserting function calls that write tracing information to a log file during execution. A scope map object is generated to map the identifiers assigned to information about the AST nodes with data extracted from execution traces. Types are inferred and the annotations are generated according to these information. The used type annotations follow the specifications of JSDoc[10], which is a markup language used to document JavaScript source code (similar to the JavaDoc format used for documenting Java code).

Milojković and Nierstrasz [2016] proposed two heuristics to statically infer types of variables in dynamically typed languages. While the aim of the existing type inference algorithms is to narrow down the list of classes that represent possible types bound to a variable at runtime, the authors proposed to order the list of possible types of a variable according to how frequently those classes occur in the source code. They hypothesize that class usage frequency serves as a reliable proxy for the likelihood that a variable belongs to that class at runtime. In order to determine the classes more frequently used throughout the source code, the authors implemented and evaluated two possible heuristics: (i) name occurrence heuristic; and (ii) class instantiation heuristic. The first heuristic is based on the calculation of the occurrences of class names throughout the source code. The second focus on the places in source code where a class name is used to instantiate a new object. Their heuristics produced results comparable with the existing type inference algorithms.

2.7 Final Remarks

In this chapter, we provided background information to better understand the state of the art related to this PhD thesis. Firstly, we presented an overview of the JavaScript language (Section 2.1), the concepts related to the different styles for class emulation in JavaScript (Subsection 2.1.1), the new syntax for class emulation that comes with ECMAScript 6 (Subsection 2.1.2), and a brief introduction on scripting languages that work on top of JavaScript (Subsection 2.1.3).

In Section 2.2, we showed that the behavior of a program can be modified dynamically, especially using the `eval` operator.

In Section 2.3 we presented existing techniques for code transformation (refactoring) in JavaScript. Some of these techniques rely on the identification of class-like structures to implement the proposed refactorings.

[10]`http://usejsdoc.org/`

In Section 2.4, we investigated approaches that aim to detect code smells, source code patterns and anti-patterns in JavaScript systems. We recognized one smell (Refused Bequest) related to the emulation of classes in JavaScript.

In Section 2.5, we showed related work that rely on program analysis for JavaScript. We also presented some pros and cons of dynamic analysis in comparison with static analysis.

Finally, in Section 2.6, we discussed type inference techniques for JavaScript systems. Since we analyze the use of class-like structures in legacy JavaScript systems, we feel these are important concepts to discuss in this chapter.

Chapter 3

Proposed Approach

In this chapter, we shed light on the use of class-like structures in legacy JavaScript systems. We formalize our strategy to detect classes in Section 3.1. We describe the design of the studies performed to evaluate our strategy in Section 3.2, and then we present the results in Section 3.3. Threats to validity are exposed in Section 3.4. We conclude this chapter by summarizing our findings in Section 3.5.

3.1 Detecting Classes in Legacy JavaScript

In this section, we describe our strategy to statically detect classes in legacy JavaScript source code (Subsection 3.1.1). Subsection 3.1.2 describes the tool we implemented for this purpose. We also report limitations of this strategy, mainly due to the dynamic nature of JavaScript (Subsection 3.1.3).

3.1.1 Strategy to Detect Classes

We define our heuristics for a subset of JavaScript which includes the key elements of the language related to class emulation. The use of a language subset for studying a particular aspect of JavaScript is also adopted by other works. For example, Anderson et al. [2005] rely on a language subset when proposing a type inference algorithm for JavaScript. The subset of JavaScript we describe in the following grammar (Figure 3.1) assumes that a program is composed of functions and prototype declarations. The expressions of interest are the ones that create objects and add properties to functions via `this` or `prototype`. For the sake of clarity, we removed from our subset the JavaScript features that are not related to class emulation, such as variable declarations and assignments, function invocations, return statements, etc.

```
Program      ::= (FuncDecl | ProtoAssign)*
FunDecl      ::= function Identifier() { (Exp | ProtoAssign)*}
Exp          ::= new Identifier(); |
                 Object.create(Identifier.prototype); |
                 this.Identifier = Exp; |
                 this.Identifier = function { Exp }
ProtoAssign  ::= Identifier.prototype.Identifier = Exp; |
                 Identifier.prototype.Identifier = function { Exp } |
                 Identifier.prototype = new Identifier(); |
                 Identifier.prototype = Object.create(Identifier.prototype);
```

Figure 3.1: Syntax to detect class emulation in a subset of JavaScript language

Definition #1: A class is a tuple $(C, \mathcal{A}, \mathcal{M})$, where C is the class name, $\mathcal{A} = \{a_1, a_2, \ldots, a_p\}$ are the attributes defined by the class, and $\mathcal{M} = \{m_1, m_2, \ldots, m_q\}$ are the methods. Moreover, a class $(C, \mathcal{A}, \mathcal{M})$, defined in a JavaScript program P, must respect the following conditions:

- P must have a function with name C.
- For each attribute $a \in \mathcal{A}$, the class constructor or one of its methods must include an assignment `this.a` = `Exp` or P must include an assignment `C.prototype.a` = `Exp`.
- For each method $m \in \mathcal{M}$, function C must include an assignment `this.m` = `function` {`Exp`} or P must include an assignment `C.prototype.m` = `function` {`Exp`}.

However, when functions matching *Definition #1* are implemented in the same lexical scope, as functions `Circle` and `setColor` in Listing 2.1, we must distinguish those that are class constructors from those that are methods. To achieve that, we do not consider as a class constructor a function that: (i) has no inner functions bound to `this`, (ii) does not participate in inheritance relationships defined using prototypes, and (iii) is never instantiated with neither `new` nor `Object.create`. In Listing 2.1, function `setColor` does not have inner functions bound to `this` nor inheritance relationships and it is never instantiated. Therefore, it is not considered a constructor function, but a method.

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

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

In the examples of Section 2.1.1, `Circle2D` is a *subclass* of `Circle`.

3.1.2 Tool Support

We implemented a tool, called JSClassFinder [Silva et al., 2015a], for identifying classes in legacy JavaScript programs. As illustrated in Figure 3.2, this tool works in two steps. In the first step, Esprima[1]—a widely used JavaScript Parser—is used to generate a full abstract syntax tree (AST), in JSON[2] format. In the second step, the "Class Detector" module is responsible for identifying classes in the JavaScript AST and producing an object-oriented model of the source code.

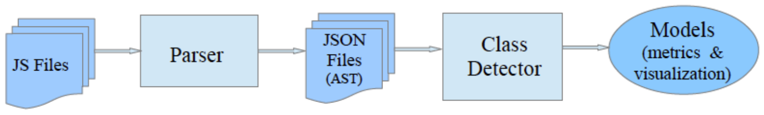


Figure 3.2: JSClassFinder's architecture

The models generated by JSClassFinder are integrated with Moose[3], which is a platform for software and data analysis [Nierstrasz et al., 2005]. This platform provides visualizations to interact with the tool and to "navigate" the application's model. All information about classes, methods, attributes, and inheritance relationships is available. Users can interact with a Moose model to access all visualization features and metric values. This model also allows the use of drill-down and drill-up operations when an entity is selected. The visualization options include UML class diagrams [Jacobson et al., 1999], distribution maps [Ducasse et al., 2006], and tree views.

It is possible for a user to customize the diagrams and to choose which elements to expose. For example, Figure 3.3 shows a distribution map for the system JADE[4], which is a template engine for `Node.js`. In this visualization, classes are represented by external rectangles, the small internal squares are methods, and the links between

[1]`http://esprima.org`
[2]`http://www.json.org/`
[3]`http://www.moosetechnology.org/`
[4]`http://jade-lang.com/`

classes represent inheritance relationships. It is also possible to show a similar diagram where the external squares are JavaScript files and the internal squares are classes.

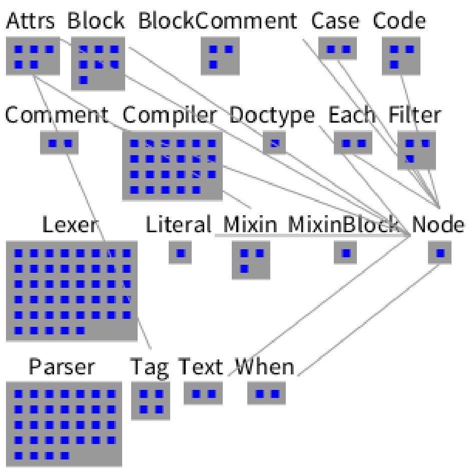


Figure 3.3: Example of distribution map for system JADE, generated by JSClassFinder

JSClassFinder also collects the following metrics: Number of Attributes (NOA), Number of Methods (NOM), Depth of Inheritance Tree (DIT), and Number of Children (NOC) [Chidamber and Kemerer, 1994].

JSClassFinder is implemented in Pharo[5], which is a complete Smalltalk environment for developing and executing object-oriented code. Pharo also offers strong live programming features such as immediate object manipulation, live update, and hot recompilation. Moreover, we chose to implement our tool using Pharo because of the possibility to integrate it with the Moose platform.

3.1.3 Limitations

We acknowledge that there is not a single strategy to emulate classes in JavaScript. For example, it is possible to create "singleton" objects directly, without using any class-like constructions, as in Listing 3.1. Even though, we do not consider such objects as classes. Instead, we chose to follow the definition presented in Booch et al. [2004], in which the authors state that classes and objects are tightly interwoven, but there are important differences between them ("a class is a set of objects that share a common structure, common behavior, and common semantics", "a single object is simply an instance of a class", page 93).

[5]http://pharo.org/

```
var myCircle = {
  radius: 10,
  pi: 3.14,
  getArea: function () { ... }
}
```

Listing 3.1: Example of “singleton” object

In addition, there are various JavaScript frameworks, like Prototype[6] and ClazzJS[7], that support their own style for implementing class-like abstractions. For this reason, we do not struggle to cover the whole spectrum of alternatives to implement classes. Instead, we consider only the strategy closest to the syntax and semantics of class-based languages and that ES6 code can be directly translated to (as discussed in Subsection 2.1.2).

Moreover, there are object-oriented abstractions that are more difficult to emulate in JavaScript, like abstract classes and interfaces. Encapsulation is another concept that does not have a straightforward mapping to JavaScript. A common workaround to simulate private members in JavaScript is by using local variables and closures. As shown in Listing 3.2, an inner function `f2` in JavaScript has access to the variables of its outer function `f1`, even after `f1` returns. Therefore, local variables declared in `f1` can be considered as private, because they can only be accessed by the “private function” `f2`. However, we decided not to classify `f2` as a private method, mainly because it cannot be accessed from the object `this`, nor can it be directly called from the public methods associated to the prototype of `f1`.

```
function f1 () {     // outer function
  var x;                       // local variable
  function f2 () {             // inner function
    // can access "x"
    // cannot be called outside "f1"
  }
}
```

Listing 3.2: Using closures to implement “private” inner functions

In JavaScript, it is possible to remove properties from objects dynamically, *e.g.*, by calling `delete myCircle.radius`. Therefore, at runtime, an object can have less attributes than the ones initially defined. It is also possible to modify the prototype chains dynamically, which would mean modifying the “inheritance” links. Finally, the behavior of a program can also be dynamically modified using the `eval` operator [Richards

[6]http://prototypejs.org

[7]https://github.com/alexpods/ClazzJS

et al., 2011, Meawad et al., 2012]. However, we do not consider the impact of `eval`'s in the strategy described in Subsection 3.1.1. For example, we do not account for classes entirely or partially created by means of `eval`.

Still due to the dynamic nature of JavaScript, if a class has a property that receives the return of a function call, this property is classified as an attribute, even if this call returns another function. Listing 3.3 shows one example of this case, in which the property `this.x` (line 6) is classified as an attribute, instead of a method, because the language is loosely typed and we do not evaluate the results of function calls.

```
1 function getF () {
2   // getF() returns another function
3   return function () {...};
4 }
5 function f1 () {     // class constructor
6   this.x = getF();   // property x
7   ...
8 }
```

Listing 3.3: Property that receives a function as the return of a function call

Finally, we do not evaluate code structures involved in indirect calls. For example, Listing 3.4 shows the use of a JavaScript function, called `setParentClass` (lines 6-8), to set a parent for class `Circle`, establishing an inheritance relationship. Line 10 shows a call to this function passing `Figure` as argument, to be the superclass of `Circle`. In this case, we cannot detect the inheritance between the two classes because the code in line 7, that creates the link, uses the parameter `superclass` instead of a direct reference to the constructor function `Figure`. If we replaced the code in line 10 by `Circle.prototype` = `new Figure`(); then we would be able to detect this inheritance relationship.

```
1 // class constructors
2 function Figure () { ... }
3 function Circle () { ... }
4 // method to set a parent class
5 Circle.setParentClass =
6   function (superclass) {
7     Circle.prototype = new superclass();
8   }
9 // establishing the link between Figure and Circle
10 Circle.setParentClass(Figure);
```

Listing 3.4: Inheritance relationship not detect due to indirect function call

In Section 3.3.5, we report a validation of our heuristics with the developers of 60 legacy JavaScript systems. As a result of this validation, we discuss the impact of some of the aforementioned limitations.

3.2 Evaluation Design

In this section, we describe the methodology we use to evaluate and to validate the strategy proposed to detect classes in legacy JavaScript code. We first present the questions that motivate our research (Subsection 3.2.1). Next, we describe the process we follow to select JavaScript repositories on GitHub and to carry out the necessary clean up of the downloaded code (Subsection 3.2.2). The metrics we use in our evaluation are described in Subsection 3.2.3. Finally, we report the design of a field study with JavaScript developers in Subsection 3.2.4.

3.2.1 Research Questions

Our goal is to evaluate the strategy we propose to detect class-like abstractions in legacy JavaScript software. To achieve this goal, we pose the following research questions:

- RQ #1: Do developers emulate classes in legacy JavaScript applications?
- RQ #2: Do developers emulate subclasses in legacy JavaScript applications?
- RQ #3: Is there a relation between the size of a JavaScript application and the number of class-like structures?
- RQ #4: What is the shape of the classes emulated in legacy JavaScript code?
- RQ #5: How accurate is our strategy to detect classes?
- RQ #6: Do developers intend to use the new support for classes that comes with ECMAScript 6?

With RQ #1, we check if the emulation of classes is a common practice in legacy JavaScript applications. RQ #2 checks the usage of prototype-based inheritance. With RQ #3, we verify if the number of JavaScript classes in a system, as detected by JSClassFinder, is related to its size, measured in lines of code. With RQ #4, we analyze the shape of JavaScript classes regarding the relation between the number of attributes and the number of methods. With RQ #5, we evaluate the accuracy of the proposed approach to identify class-like structures. With RQ #6, we verify if developers intend to use the concrete syntax to define classes provided by ES6.

Although RQs #3 and #4 are not directly related to the identification of classes, we decided to investigate the shape of the analyzed classes to better understand their use in JavaScript legacy systems.

3.2.2 Dataset

Our dataset includes the last version of the top 1,000 JavaScript projects on GitHub, according to the number of stars. This selection was performed in July, 2015. After cloning the repositories, we used an external library called Linguist[8] to clean up the source code files. Linguist is used by GitHub to ignore binary, third-party, and automatically generated files when computing statistics on the programming languages used by a repository. After running Linguist, we also performed a custom-made script to remove tests, examples, documentation, and configuration files. More specifically, this script removes the following files: `gulpfile.js`, `gruntfile.js`, `package.js`, `*thirdparty.js`, `*_test.js`, `*_tests.js`, `test.js`, `tests.js`, `license.js`; and the following folders: `test`, `tests`, `examples`, `example`, `build`, `dist`, `spec`, `demos`, `demo`, `minify`, `release`, `releases`, `docs`, `bin`, `test-*`, and `testing`.

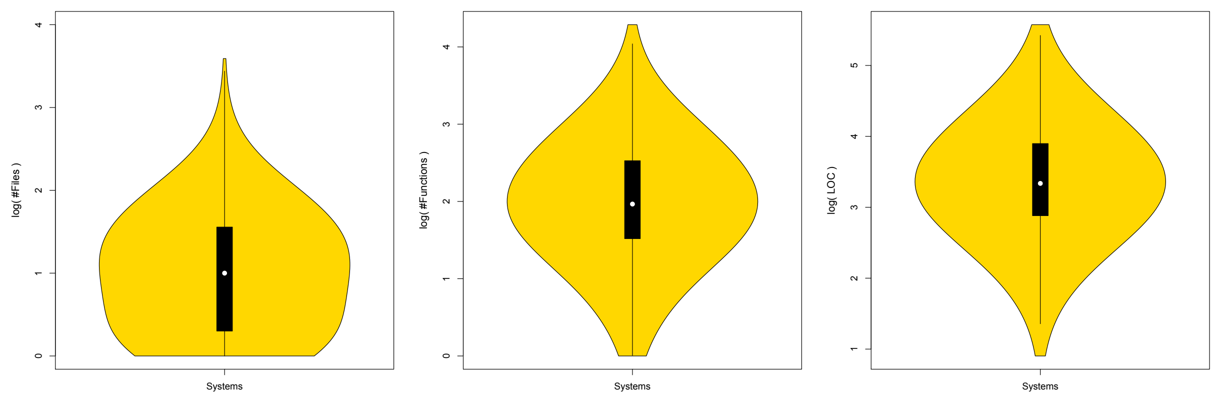
After this clean up process, 82 systems were not exploitable because they did not contain any significant contributions, *i.e.*, they remained with no source code files. Therefore, the final dataset was composed of 918 systems. Figure 3.4 shows violin plots[9] with the distribution of number of files, number of functions, and lines of code (LOC) in logarithm scale (base 10). The width of the "violin plot" correlates with the number of systems for a given value. The largest system (`gaia`) has 375,615 LOC and 1,650 files with `.js` extension. The smallest system (`jswiki`) has 8 LOC and a single file. The average size is 8,778 LOC (standard deviation 21,801 LOC) and 41 files (standard deviation 163 files). The median is 2,170 LOC and 10 files.

3.2.3 Metrics

In the following we describe the metrics we use to answer the first four research questions proposed in Subsection 3.2.1.

[8]`https://github.com/github/linguist`

[9]A violin plot is used to visualize the distribution of the data and its probability density. The thick black bar in the center represents the interquartile range, the thin black line extended from it represents the confidence intervals, and the white dot is the median.



(a) # Files (b) # Functions (c) LOC

Figure 3.4: Dataset size distributions (log scales)

3.2.3.1 Class Density (CD)

To measure the amount of source code related to the emulation of classes we propose a metric called *Class Density (CD)*, which is defined as:

$$CD = \frac{\#\ function\ methods + \#\ classes}{\#\ functions}$$

This metric is the ratio of functions in a program that are related to the implementation of classes, *i.e.*, that are methods or that are classes themselves. It ranges between 0 (system with no functions related to classes) to 1 (a fully class-oriented system, where all functions are used to support classes). The denominator includes all functions in a JavaScript program. We use the number of functions to implement methods (*function methods*) instead of the number of methods because, in JavaScript, it is possible to share the same function to implement multiple methods. Listing 3.5 shows an example found in the system SLICK, where a function body is shared by two methods. In this example, the `Slick` class provides two methods (`getCurrent` and `slickCurrentSlide`) that perform the same action when called. Therefore, the number of methods is equal to two, but the number of *function methods* is one.

```
Slick.prototype.getCurrent =
  Slick.prototype.slickCurrentSlide = function() {
    var _ = this;
    return _.currentSlide;
  };
```

Listing 3.5: Methods sharing the same body in system SLICK

We used CD to classify the systems in four main groups:

- *Class-free*: systems that do not use classes at all ($CD = 0$).
- *Class-aware*: systems that use classes, but marginally ($0 < CD \leq 0.25$).
- *Class-friendly*: systems with an important usage of classes ($0.25 < CD \leq 0.75$)
- *Class-oriented*: systems where most structures are classes ($CD > 0.75$).

3.2.3.2 Subclass Density (SCD)

To evaluate the usage of inheritance, we propose a metric called *Subclass Density (SCD)*, defined as:

$$SCD = \frac{\mid \{\, C \in Classes \mid DIT(C) \geq 2 \,\} \mid}{\mid Classes \mid - 1}$$

where *Classes* is the set of all classes in a given system and *DIT* is the *Depth of Inheritance Tree*. *Classes* with $DIT = 1$ only inherit from the common base class (`Object`). *SCD* ranges from 0 (system that does not make use of inheritance) to 1 (system where all classes inherit from another class, except the class that is the root of the class hierarchy). *SCD* is only defined for systems that have at least two classes.

3.2.3.3 Data-Oriented Class Ratio (DOCR)

In a preliminary analysis, we noticed many classes having more attributes than methods. This contrasts to the common shape of classes in class-based languages, when classes usually have more methods than attributes [Terra et al., 2013]. To better understand the members of JavaScript classes, we propose a metric called *Data-Oriented Class Ratio (DOCR)*, defined as follows:

$$DOCR = \frac{\mid \{\, C \in Classes \mid NOA(C) > NOM(C) \,\} \mid}{\mid Classes \mid}$$

where *Classes* is the set of all classes in a system. *DOCR* ranges from 0 (system where all classes have more methods than attributes or both measures are equal) to 1 (system where all classes are data-oriented classes, *i.e.*, their number of attributes is greater than the number of methods). *DOCR* is only defined for systems that have at least one class.

3.2.4 Field Study Design

To validate our strategy for detecting classes, we perform a field study with the developers of 60 JavaScript applications, including 50 systems from our previous conference paper [Silva et al., 2015b], and 10 new systems. These systems have at least 1,000 stars on GitHub, 150 commits, and are not forks of other projects. After checking out each system, we cleaned up the source code to remove unnecessary files, as we did for the dataset described in Subsection 3.2.2.

The systems considered in the field study are presented in Table 3.1, including their version, a brief description, size (in lines of code), number of files, and number of functions. The selection includes well-known and widely used JavaScript systems, from different domains, covering frameworks (*e.g.*, ANGULAR.JS and JASMINE), editors (*e.g.*, BRACKETS), browser plug-ins (*e.g.*, PDF.JS), games (*e.g.*, 2048 and CLUMSY-BIRD), etc. The largest system (ACE) has 140,023 LOC and 594 files with `.js` extension. The smallest system (MASONRY) has 208 LOC and a single file. The average size is 12,870 LOC (standard deviation 25,961 LOC) and 56 files (standard deviation 101 files). The median size is 3,363 LOC and 13 files.

This field study was conducted between March and June, 2015, during a sandwich Ph.D internship at INRIA Research Centre Lille Nord-Europe, under the supervision of professor Nicolas Anquetil. For each system, we performed the following steps:

1. We downloaded the latest version on GitHub and cleaned up the source code;

2. We executed the parser (Esprima) to generate the AST;

3. We executed JSClassFinder to identify class-like structures and to build a class diagram;

4. We used the information available on GitHub to identify the main developers of each system in the dataset. For systems supported by a team of developers, the developer selected was the one with the highest number of commits in the previous three months. We then sent an email to the application's main developer with the class diagram attached and asked him to validate the detected classes. Figure 3.5 shows the class diagram sent to the developer of ALGORITHMS.JS. This diagram includes 14 classes representing common data structures, such as `Stack`, `LinkedList`, `Graph`, `HashTable`, etc.

5. We analyzed and categorized the developer's responses.

Table 3.1: JavaScript systems (ordered by the CD column, see description in accompanying text). SCD can only be computed for systems with 2 or more classes. DOCR can only be computed for systems with at least one class.

System	Version	LOC	#Files	#Func	#Class	#Meth	#Attr	CD	SCD	DOCR
masonry	3.2.3	208	1	10	0	0	0	0.00	-	-
randomColor	0.2.0	373	1	16	0	0	0	0.00	-	-
respond	1.4.2	460	3	15	0	0	0	0.00	-	-
resume	-	460	1	19	0	0	0	0.00	-	-
clumsy-bird	-	672	7	36	0	0	0	0.00	-	-
impress.js	0.5.3	769	1	24	0	0	0	0.00	-	-
jquery-pjax	1.9.3	913	1	33	0	0	0	0.00	-	-
async	1.1.0	1,114	1	100	0	0	0	0.00	-	-
modernizr	2.8.3	1,382	1	69	0	0	0	0.00	-	-
deck.js	1.1.0	1,473	6	51	0	0	0	0.00	-	-
zepto.js	1.1.6	2,497	17	233	0	0	0	0.00	-	-
photoSwipe	4.0.7	4,401	9	185	0	0	0	0.00	-	-
semantic-UI	1.12.3	18,369	23	1,191	0	0	0	0.00	-	-
jQueryFileUp	9.9.3	4,011	14	179	1	1	3	0.01	-	1.00
leaflet	0.7.3	8,711	75	677	4	0	7	0.01	0.00	1.00
backbone	1.1.2	1,681	2	115	1	1	0	0.02	-	0.00
chart.js	1.0.2	3,463	6	189	2	2	5	0.02	0.00	0.50
turn.js	4.0.0	6,916	5	267	3	3	6	0.02	0.00	1.00
react	0.13.2	16,654	143	608	7	8	17	0.02	0.00	0.57
meteor	1.1.0.2	41,195	72	1,378	15	12	14	0.02	0.21	0.20
underscore	1.8.2	1,531	1	123	1	5	1	0.03	-	0.00
jasmine	2.2.1	7,749	62	892	3	8	11	0.03	0.00	0.67
paper.js	0.9.22	26,039	65	1,071	30	10	115	0.04	0.00	0.90
typeahead.js	0.10.5	2,576	19	233	11	1	72	0.05	0.00	1.00
d3	3.5.5	13,079	268	1,259	19	45	41	0.05	0.22	0.58
wysihtml5	0.3.0	5,913	69	343	2	17	8	0.06	0.00	0.00
sails	0.11.0	12,724	101	425	8	23	40	0.07	0.00	0.25
ionic	1.0.0.4	19,322	103	492	8	26	21	0.07	0.29	0.50
jquery	2.1.4	7,736	79	330	6	25	31	0.09	0.00	0.50
ghost	0.6.2	15,290	142	659	15	47	44	0.09	0.00	0.27
timelineJS	2.35.6	18,371	93	896	12	69	11	0.09	0.00	0.08
express	4.12.3	3,590	11	131	3	12	14	0.11	0.00	0.67
reveal.js	3.0.0	5,811	16	242	5	22	18	0.11	0.00	0.40
video.js	4.12.5	9,823	46	586	6	63	17	0.11	0.00	0.50
three.js	0.0.71	39,449	202	1,266	99	48	544	0.12	0.00	0.92
numbers.js	-	2,965	10	132	2	16	4	0.14	0.00	0.00
polymer	0.5.5	11,849	1	763	22	103	68	0.16	0.00	0.41
grunt	0.4.5	1,932	11	103	1	16	8	0.17	-	0.00
skrollr	0.6.29	1,772	1	58	1	12	0	0.22	-	0.00
ace	1.1.9	140,023	594	4,337	291	673	785	0.22	0.01	0.46
mousetrap	1.5.3	1,281	5	46	1	10	0	0.24	-	0.00
hammer.js	2.0.4	2,348	19	124	6	33	25	0.31	0.00	0.33
brackets	1.3.0	130,770	392	4,298	173	1,239	750	0.33	0.09	0.31
angular.js	1.4.0.1	49,220	191	981	61	276	171	0.34	0.03	0.21
intro.js	1.0.0	1,255	1	42	1	14	2	0.36	-	0.00
algorithms	0.8.1	3,263	58	165	14	59	32	0.44	0.23	0.21
pdf.js	1.1.1	57,359	88	2,277	181	895	795	0.47	0.11	0.44
bower	1.4.1	8,464	60	304	15	143	97	0.51	0.00	0.40
mustache.js	2.0.0	594	1	33	3	15	7	0.55	0.00	0.33
less.js	2.3.1	12,045	99	707	64	327	278	0.55	0.21	0.34
gulp	3.8.11	99	3	5	1	2	6	0.60	-	1.00
fastclick	1.0.6	841	1	23	1	16	10	0.74	-	0.00
pixiJS	3.0.2	21,024	113	703	87	453	546	0.76	0.33	0.46
isomer	0.2.4	770	7	47	7	31	27	0.81	0.00	0.57
2048	-	873	10	76	7	62	29	0.91	0.00	0.14
slick	1.5.2	2,300	1	81	1	86	0	0.93	-	0.00
floraJS	3.1.1	2,942	20	86	18	62	315	0.93	0.00	0.94
parallax	2.1.3	1,007	3	57	2	56	75	0.95	0.00	1.00
jade	1.9.2	11,427	27	169	19	142	73	0.95	0.83	0.26
socket.io	1.3.5	1,297	4	57	4	58	46	1.00	0.00	0.00

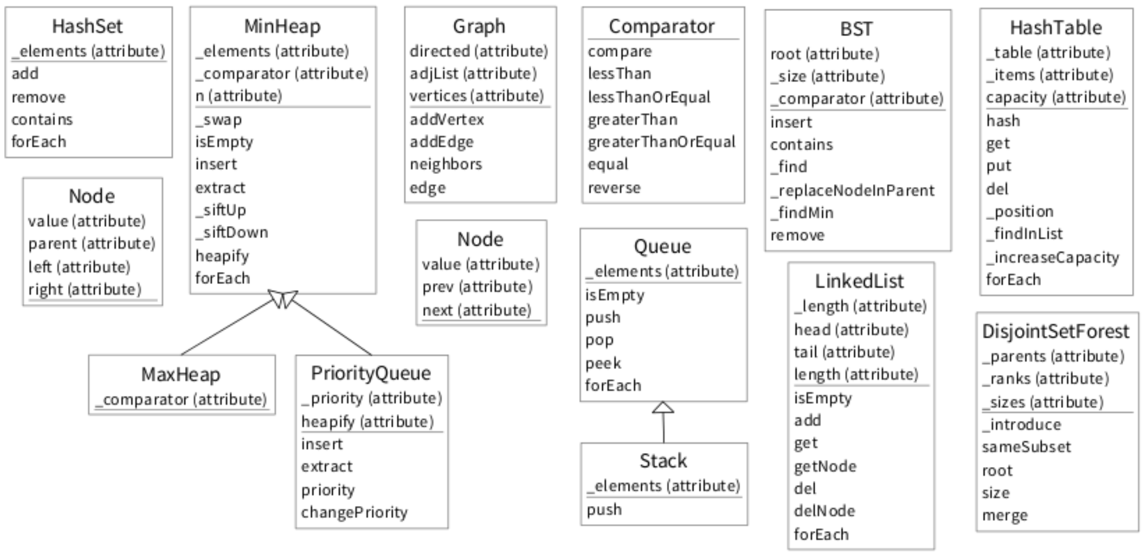


Figure 3.5: Class diagram for ALGORITHMS.JS, generated by JSClassFinder

In the mails to the developers, we asked two questions:

- Do you agree that the classes in the attached class diagram are correct?
- Do you intend to use the new support for classes that comes with ES6? Why?

The developers had to answer the questions and point out their reasons. The first question aims to evaluate the accuracy of our approach to detect class-like structures (RQ #5). The second question aims to measure the interest in a concrete syntax to implement classes in JavaScript (RQ #6). In the cases where, after one month, an answer was not received, a gentle reminder was sent. For the systems where we did not find any classes, we also sent emails requesting the developers to confirm that they really do not emulate classes in their systems.

We sent 60 emails and received 33 answers, which represents a response ratio of 55%. Out of the 33 answers, 29 were obtained after a first round, and the other four after sending a gentle reminder.

We had three answers that could not be properly classified in our study. The first came from a developer who said he agreed with our findings but he was not totally sure. In the second case, the developer sent a web link which contains the API documentation of his application, and he recommended us to validate the classes ourselves. In the last case, the developer just stated that we should never use classes. Therefore, after discarding these cases, we have 30 valid answers.

Figure 3.6 shows the distribution of the valid answers per group of systems according to the class density (CD values). The distribution indicates that our field study includes systems in all four main groups: *class-free* (4 answers), *class-aware* (15 answers), *class-friendly* (7 answers) and *class-oriented* (4 answers).

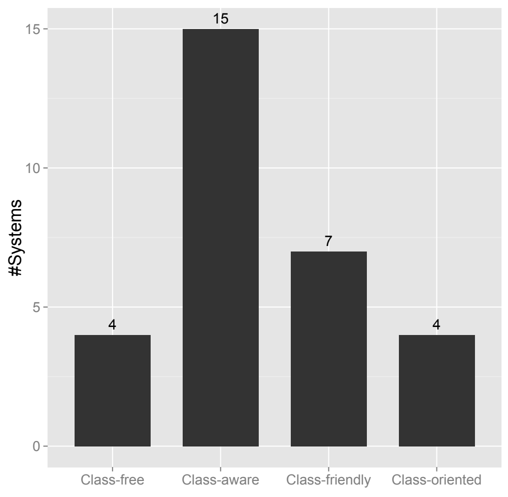


Figure 3.6: Distribution of valid answers per group

Finally, we use developers' answers to measure precision, recall, and F-score for the classes, methods, and attributes identified by our tool. These measures are calculated as follows:

$$\textit{Precision (P)} = \frac{TP}{TP + FP}$$

$$\textit{Recall (R)} = \frac{TP}{TP + FN}$$

$$\textit{F-score } (F_1) = 2 \times \frac{P \times R}{P + R}$$

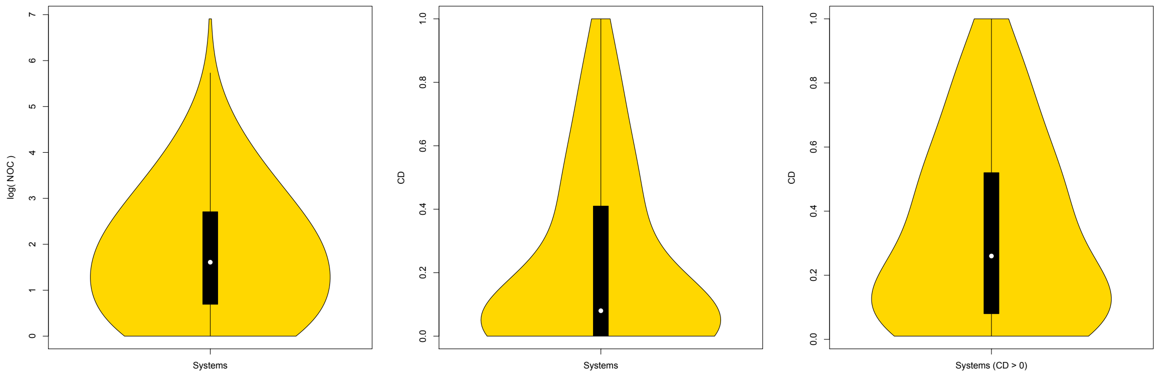
where TP represents the true positives, FP the false positives, and FN the false negatives. For classes, TP is the number of class-like structures correctly identified by our tool, FP is the number of class-like structures erroneously identified, and FN is the number of existing class-like structures that are not identified. F-score is the harmonic mean of precision and recall. For methods and attributes, the measures are defined in a similar way, but searching for method-like and attribute-like structures, respectively.

3.3 Results

In this section, we present the answers to the six proposed research questions.

3.3.1 Do developers emulate classes in legacy JavaScript applications?

We found classes in 623 out of 918 systems (68%). The system with the largest number of classes is GAIA (1,001 classes), followed by NODEINSPECTOR (330 classes), and BABYLON.JS (294 classes). MathJax is the largest system (122,683 LOC) in which we could not identify any class. Figure 3.7(a) shows the distribution of the number of classes for the systems that have at least one class. The first quartile is two (lower bound of the black box within the "violin") with 135 systems having only one class. The median is 5 and the third quartile is 15 (upper bound of the black box). Listing 3.6 shows an example of a class `Color`, detected in the system THREE.JS. We omit part of the code for the sake of readability.



(a) # Classes (log scale) (b) CD (All Systems) (c) CD

Figure 3.7: Metric distributions. Results in (a) and (c) are reported only for systems with at least one class.

```
THREE.Color = function ( color ) {  // Constructor
  ...
  return this.set( color )
};
THREE.Color.prototype = {
  r: 1, g: 1, b: 1,  // Attributes
  // Methods
  set: function ( value ) { ... },
  setRGB: function ( r, g, b ) { ... },
  ...
}
```

Listing 3.6: Example of class in THREE.JS

Figure 3.7(b) shows the distribution of the CD values. We found that 295 systems have CD equal to zero. In other words, 32% of the systems do not use classes at all or are using an abstraction other than the one we are looking for. The median is 0.08 and the third quartile is 0.41. We also found seven fully class-oriented systems (CD=1). Table 3.2 shows the ten systems with the highest values of CD.

Table 3.2: Top-10 systems with highest CD values

System	CD	#Class	LOC
SKEUOCARD	1.00	8	1,685
RAINYDAY.JS	1.00	5	1,005
SIDE-COMMENTS	1.00	3	523
ZOOM.JS	1.00	2	229
STEADY.JS	1.00	1	215
TMI	1.00	1	203
LAYZR.JS	1.00	1	164
SOCKET.IO	0.97	4	1,350
CLNDR	0.97	1	1,197
SLAP	0.97	11	938

Figure 3.7(c) shows the CD distribution when we only consider the systems with CD greater than zero. The first quartile is 0.08, the median is 0.26, and the third quartile is 0.52. In other words, the emulation of classes represents on the median 26% of the functions, for the systems that include at least one class.

Figure 3.8 shows the number of systems in each of the four proposed groups (*class-free*, *class-aware*, *class-friendly*, and *class-oriented systems*) according to the CD values. The largest group is the *class-aware* (34%), in which systems use classes but they correspond to less than 25% of the implemented functions. *Class-oriented* is the smallest group, including systems that use more than 75% of their functions to emulate classes.

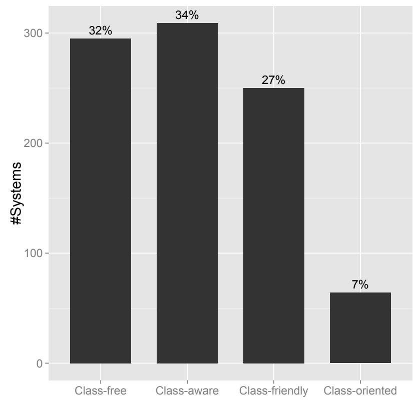


Figure 3.8: Class Density (CD) groups

Summary: We found classes in 623 out of 918 systems (68%). Therefore, 32% of the systems are *class-free*. Moreover, *class-aware* and *class-friendly* systems correspond to 34% and 27% of the systems; 7% are *class-oriented* systems.

3.3.2 Do developers emulate subclasses in legacy JavaScript applications?

As shown in Figure 3.9, the use of prototype-based inheritance is rare in JavaScript systems. First, we counted 499 systems (54%) having two or more classes, *i.e.*, systems where it is possible to detect the use of inheritance. However, in 429 of such systems (86%), we did not find any *subclasses* ($SCD = 0$). The system with the highest use of inheritance is PROGRESSBAR.JS ($SCD = 0.8$). Figure 3.10 shows the class diagram for this system. As can be seen, the `Shape` class has four *subclasses*: `Circle`, `Line`, `SemiCircle`, and `Square`.

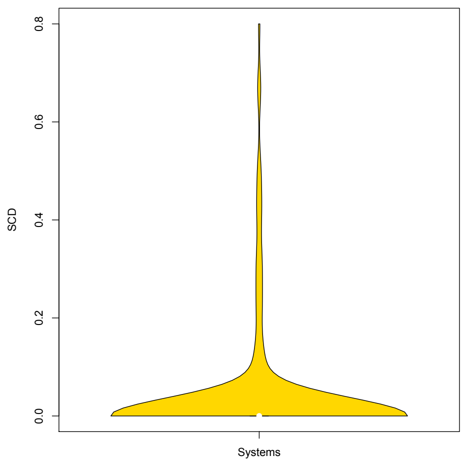


Figure 3.9: Subclass Density (SCD) distribution

Summary: We found subclasses in only 70 out of 918 systems (8%).

3.3.3 Is there a relation between the size of a JavaScript application and the number of class-like structures?

Figure 3.11 shows scatterplots with size metrics on the x-axis in a logarithmic scale and CD on the y-axis. We also computed the Spearman's rank correlation coefficient between CD and the following size metrics: KLOC, number of files, and number of

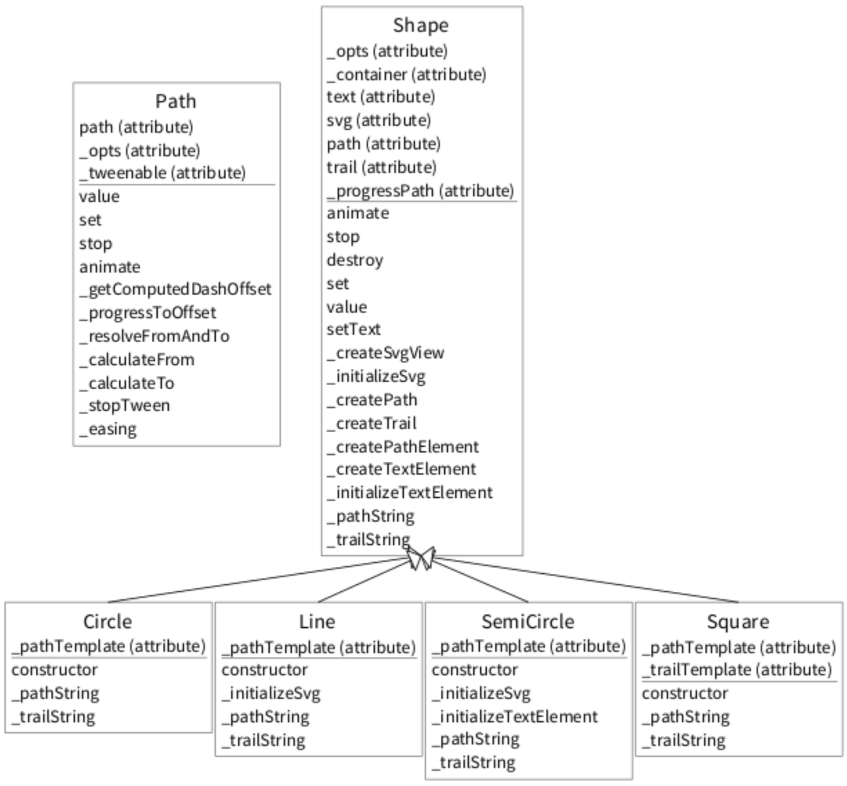
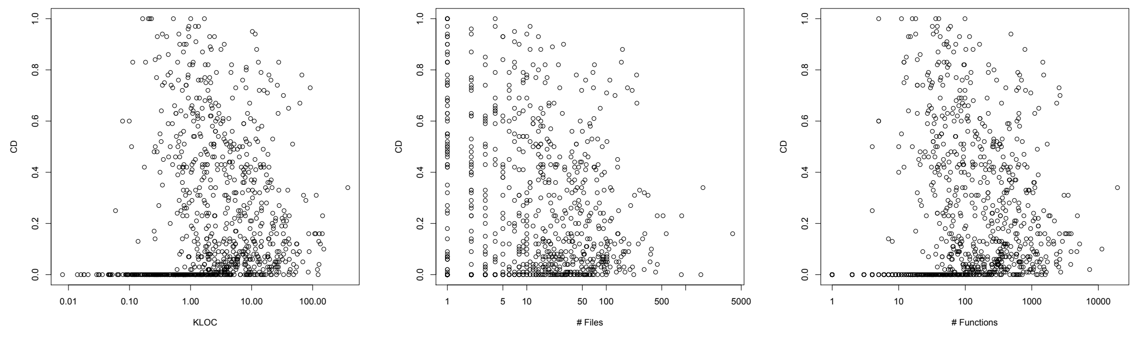


Figure 3.10: Inheritance in system PROGRESSBAR.JS

functions. The results are presented in Table 3.3. We found a weak correlation for KLOC (ρ=0.250), number of files (ρ=0.178), and for number of functions (ρ=0.289). For example, there are systems with similar sizes having both low and high class densities. ALOHA-EDITOR is an example of a system with a considerable size (69 KLOC) and low class density (CD = 0.05). By contrast, END-TO-END is also a large system (67 KLOC) but with a high class density (CD = 0.78).



(a) KLOC vs CD (b) # Files vs CD (c) # Func vs CD

Figure 3.11: Size metrics vs Class Density (CD)

Table 3.3: Correlation between CD and size metrics

	KLOC	# Files	# Func
Spearman	0.250	0.178	0.289
p-value	1.407e-14	6.216e-08	< 2.2e-16

We also used the Kruskal-Wallis test to check if the LOC distributions in all four groups (*class-free*, *class-aware*, *class-friendly*, and *class-oriented systems*) are equal. The test resulted in a p-value < 2.2e-16, leading us to reject the null hypothesis (the groups have systems with equal size), at a 5% significance level. In fact, the median measures of each tested group are quite different (690; 5,667; 2,578; and 1,150; respectively).

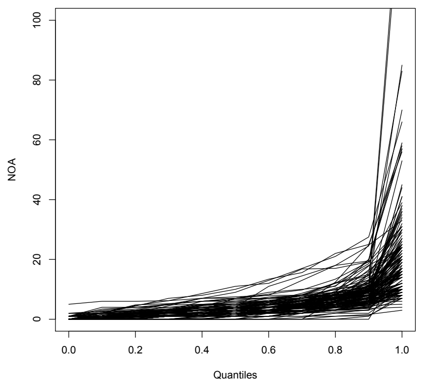
Summary: We found no correlation between the size of a JavaScript application and the number of class-like structures.

3.3.4 What is the shape of the classes emulated in legacy JavaScript code?

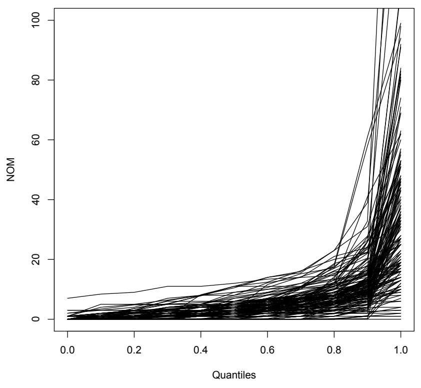
To verify the shape of JavaScript classes, regarding the number of methods and attributes, we focus on systems that have the number of classes greater than or equal to 15 (which represents the 3rd quartile of this distribution).[10] Figure 3.12 shows the quantile functions for the Number of Attributes (NOA) and Number of Methods (NOM) in such systems. The x-axis represents the quantiles and the y-axis represents the metric values for the classes in a given quantile. For example, suppose the value of a quantile p (x-axis) is k (y-axis), for NOA. This means that p% of the classes in this system have at most k attributes. As can be observed, the curves representing the systems have a right-skewed (or heavy-tailed) behavior. In fact, this behavior is common in source code metrics [Baxter et al., 2006, Louridas et al., 2008, Wheeldon and Counsell, 2003].

Regarding NOA, the quantile functions reveal that the vast majority of the classes have at most 28 attributes (90th percentile). Regarding NOM, the vast majority of the classes have less than 61 methods (90th percentile). To compare NOA and NOM measures, Figure 3.13 shows the *DOCR* distribution using a violin plot. The median DOCR value is 0.39, which is a high measure when compared to other languages. For example, metric thresholds for Java suggest that classes should have at most 8 attributes and 16

[10]Using this criteria, we reduced the number of selected systems but we continued with a considerable number of classes to analyze. We made this decision to improve the readability of the quantile functions (Figure 3.12).



(a) NOA



(b) NOM

Figure 3.12: Quantile functions

methods, [Oliveira et al., 2014]. By contrast, half of the JavaScript systems we studied have more than 39% of their classes with more attributes than methods. We hypothesize that it is due to JavaScript developers placing less importance on encapsulation. For example, getters and setters are rare in JavaScript.

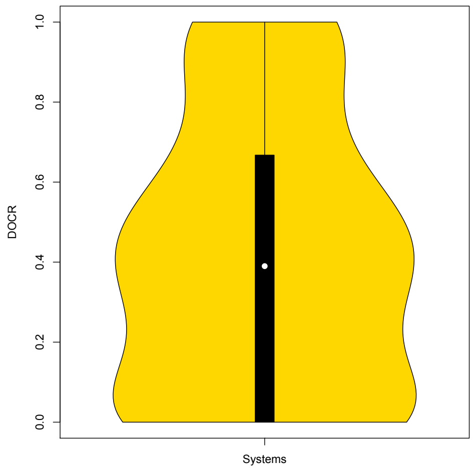


Figure 3.13: Data-Oriented Class Ratio (DOCR) distribution

Summary: The identified classes in JavaScript have usually less than 28 attributes and 61 methods (90th percentile measures). It is also common to have data-oriented classes, *i.e.*, classes with more attributes than methods. In half of the systems, we have at least 39% of such classes.

3.3.5 How accurate is our strategy to detect classes?

As described in Subsection 3.2.4, we measure accuracy using precision, recall, and F-score. Table 3.4 summarizes the results according to the developers' answers. The developers of 21 out of 30 systems (70%) fully agreed that the class diagrams correctly model the classes of their systems. Therefore, precision, recall, and F-score for these systems are equal to 100%. The following two comments are examples of answers we received for such systems:

"Yes, everything looks like it actually is in the code base." (Developer of system LESS.JS*)*

"I do in fact agree with your findings on classes/methods/attributes. In building numbers.js I did have OOP in mind." (Developer of system NUMBERS.JS*)*

Table 3.4: Precision, Recall, and F-Score results

	Precision (%)			**Recall (%)**			**F-Score (%)**		
Systems	Classes	Meth.	Attr.	Classes	Meth.	Attr.	Classes	Meth.	Attr.
ACE	93	100	100	100	100	100	96	100	100
ALGORITHMS.JS	100	100	100	100	100	100	100	100	100
ANGULAR.JS	92	100	87	100	93	100	96	96	93
BOWER	100	100	100	100	100	100	100	100	100
CLUMSY-BIRD	100	100	100	0	0	0	0	0	0
D3	100	100	100	83	48	79	91	65	88
EXPRESS	100	100	100	60	36	56	75	53	72
INTRO.JS	100	100	100	100	100	100	100	100	100
JADE	100	100	100	100	100	100	100	100	100
JASMINE	100	100	100	7	5	24	13	10	39
JQUERY	100	100	100	100	100	100	100	100	100
JQUERYFILEUP	100	100	100	100	100	100	100	100	100
LEAFLET	100	100	100	9	0	4	17	0	8
LESS.JS	100	100	100	100	100	100	100	100	100
MASONRY	100	100	100	100	100	100	100	100	100
MODERNIZR	100	100	100	100	100	100	100	100	100
MOUSETRAP	100	100	100	100	100	100	100	100	100
MUSTACHE.JS	100	100	100	100	100	100	100	100	100
NUMBERS.JS	100	100	100	100	100	100	100	100	100
PAPER.JS	100	100	100	100	3	59	100	6	74
PDF.JS	100	100	100	100	100	100	100	100	100
PIXIJS	100	100	100	100	100	100	100	100	100
RANDOMCOLOR	100	100	100	100	100	100	100	100	100
SAILS	100	100	100	100	100	100	100	100	100
SKROLLR	100	100	100	100	100	100	100	100	100
SLICK	100	100	100	100	100	100	100	100	100
SOCKET.IO	100	100	100	100	100	100	100	100	100
THREE.JS	100	100	100	100	100	100	100	100	100
UNDERSCORE	100	100	100	100	100	100	100	100	100
VIDEO.JS	100	100	100	11	15	16	20	26	28

3.3.5.1 Precision

We achieve a precision of 100% in 28 out of 30 systems for classes; in all 30 systems for methods; and in 29 systems for attributes. In the following paragraphs we discuss the false positives we detected for classes and attributes.

False positives for classes. The developers of systems ACE and ANGULAR.JS pointed out that our strategy incorrectly identified some entities as classes. In both cases, the false positives are due to a limitation regarding JavaScript scoping rules. Listing 3.7 shows an example for the system ANGULAR.JS. In this example, we have a `MessageFormatParser` class, with a method `startStringAtMatch` (lines 4-6). Since there is also a function `match` in the global scope (line 1) our tool initially classifies `stringQuote` as a method (line 5). However, due to the scoping rules of JavaScript, this property is initialized with the formal parameter of `startStringAtMatch`, which is also named `match`. Moreover, `match` always receives a non-function value and therefore it should have been classified as an attribute. A similar issue happens in ACE.

```
1 function match() {...};
2
3 MessageFormatParser.prototype.startStringAtMatch =
4   function startStringAtMatch(match) {
5     this.stringQuote = match;
6     ...
7   };
```

Listing 3.7: Example of method incorrectly identified as a class in ANGULAR.JS

False positives for attributes. We have two situations in which methods are indeed identified as attributes in the system ANGULAR.JS. Listing 3.8 shows part of the implementation for the class `JQLite`. Our strategy correctly classifies the property `ready` (line 2) as a method, but it is not able to do the same with the property `splice` (line 3). The function [].`splice` is not recognized as a function because its implementation is not part of the source code of ANGULAR.JS (it is a JavaScript native function from `Array` object). Currently, our implementation does not recognize as methods functions that are initialized with JavaScript built-in functions.

Listing 3.9 shows another example of a property that is not identified as a method in ANGULAR.JS, as we can see in the following comment:

"$get is marked as attribute a lot, it should always be a method." (Developer of system ANGULAR.JS*)*

```
1 JQLite.prototype = {
2   ready: function(fn) {...},
3   splice: [].splice,
4   ...
5 };
```

Listing 3.8: Example of missing method (line 3 - system ANGULAR.JS)

In this case, the property `$get` receives an array that contains a function in its second element. Although the developer considers that this property is a method, our approach identifies it as an array and therefore classifies it as an attribute.

```
this.$get = ['$window', function(\$window) {...}];
```

Listing 3.9: Example of an array that contains a function (system ANGULAR.JS)

3.3.5.2 Recall

We achieve a recall of 100% in 24 out of 30 systems for classes; in 22 systems for methods; and in 23 systems for attributes. In the following paragraphs we discuss the false negatives we detected for classes, methods, and attributes.

False negatives for classes. Six developers pointed out at least one missing class in their systems. In the case of the system CLUMSY-BIRD, the base class constructors are not available in the GitHub repository. The application imports an external file, which contains these base classes[11]. The import statement is placed directly in the main HTML file. For this reason, we were not able to detect classes in this system.

As a second case, EXPRESS' developer stated that our tool missed two classes, as shown in the following answer excerpt:

"So I have taken a look at the UML diagram you attached to the email and they do look mostly right. The main thing missing is there is also an Application class and a Router class, to round out a total of five main classes. The three you have there do look right, though." (Developer of system EXPRESS*)*

According to our strategy, `Application` and `Router` are not classes. `Application` is implemented as a singleton object, and we do not identify such structures as classes, as commented in Subsection 3.1.3. `Router` is not a class because its methods and attributes are not directly bound to `this` nor `prototype`. Instead, the constructor

[11] `http://cdn.jsdelivr.net/melonjs/2.0.2/melonJS.js`

function uses `__proto__` (an accessor property), as we can see in Listing 3.10 (line 5). In fact, `__proto__` is a special name used by Mozilla's JavaScript implementation to expose the internal `prototype` of the object through which it is accessed. However, the use of `__proto__` has been discouraged[12], mostly because it is not supported by other browsers.

```
1  var proto = module.exports = function() {
2    function router() {
3      ...
4    }
5    router.__proto__ = proto;
6    router.params = {};
7    router.stack = [];
8    ...
9  };
10 proto.param = function param(name, fn) {...};
11 proto.handle = function() {...};
12 ...
```

Listing 3.10: Example of function `router` which is not detected as a class in system EXPRESS

In the four remaining systems (D3, JASMINE, VIDEO.JS, and LEAFLET), the causes for missing classes are related to the use of external frameworks and libraries that provide their own style for implementing class-like abstractions. The following comments are examples of answers in this category:

"The classes you found are only a small part of Leaflet classes. This is because Leaflet uses its own class utility: `https://github.com/Leaflet/Leaflet/blob/master/src/core/Class.js`" (Developer of system LEAFLET*)*

"From a pure Object Orientation point of view, I would probably call almost every file inside 'src/core' in the jasmine repo its own class (minus a few like 'util.js' and 'base.js' at least), which is more like 45 classes." (Developer of system JASMINE*)*

False negatives for methods and attributes. In all six systems with missing classes we also have, as consequence, missing methods and attributes. Besides these cases, developers of other two systems pointed out missing methods. In the first case, for system ANGULAR.JS, our approach identified some methods as attributes, as discussed in the previous subsection (precision). In the second case, PAPER.JS's developers use a customized implementation that allows our approach to identify the classes,

[12]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto`

but not the methods. Listing 3.11 illustrates this issue for the class `Line`. In this case, the association between the constructor function `Line` (line 3) and the methods `getPoint`(), `getVector`(), etc (lines 9-11) is built by using a project-specific function called `Base.extend` (line 1). The usage of this function hides the methods and some attributes from our tool.

```
1  var Line = Base.extend({
2    _class: 'Line',
3    initialize: function Line(arg0, arg1, ...) {
4      // Attributes
5      this._px = arg0;
6      this._py = arg1;
7      ...
8    },
9    getPoint: function() {...},
10   getVector: function() {...},
11   ...
12 }
```

Listing 3.11: Class implementation for system paper.js which uses a project-specific function (`Base.extend`) to implement classes

Assessing the impact of the heuristics limitations. In Section 3.1.3, we listed five limitations of the proposed heuristics: (1) use of "singleton" objects; (2) use of JavaScript frameworks to implement class-like abstractions; (3) use of dynamic features to alter class-like structures; (4) use of properties that receive the return of function calls; (5) use of indirect calls. We mapped the false positives and negatives detected in the RQ #5 to these limitations. We found that six out of eight identified causes of false positives/negatives are not related to any limitation. In other words, they refer to other limitations which were not previously listed. However, two false negatives for class identification are related to previously known limitations (system Express contains one instance of "singleton" object and systems D3, jasmine, video.js, and Leaflet make use of JavaScript frameworks to implement class-like abstractions).

3.3.5.3 F-Score

Table 3.4 also reports the F-score results. The measures are equal to 100% in 22 out of 30 systems for classes, methods, and also for attributes. In the remaining systems, the measures range from 0% (clumsy-bird) to 96% (ace and angular.js) for classes, from 0% (clumsy-bird and leaflet) to 96% (angular.js) for methods, and from 0% (clumsy-bird) to 93% (angular.js) for attributes.

The system CLUMSY-BIRD has F-score equal to zero because it uses base class constructors that are not available in its source code repository, as discussed in Subsection 3.3.5.2.

3.3.5.4 Overall results

Figure 3.14 presents the results for precision, recall, and F-score considering the whole population of classes, methods, and attributes, independently from system. The overall measurements range from 97% (classes) to 100% (methods) for precision, from 70% (methods) to 89% (attributes) for recall, and from 82% (methods) to 94% (attributes) for F-score.

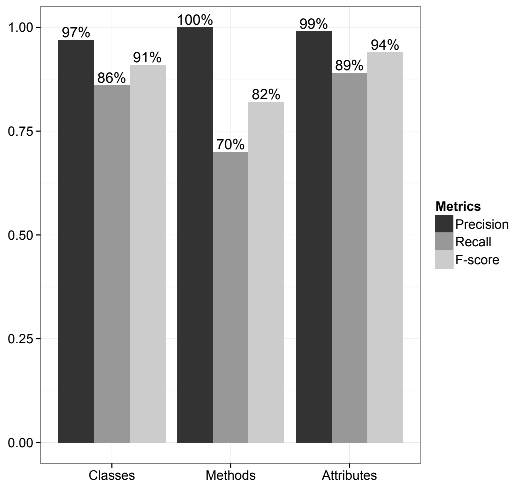


Figure 3.14: Overall results for precision, recall, and F-score

3.3.6 Do developers intend to use the new support for classes that comes with ECMAScript 6 ?

Table 3.5 summarizes the answers for this question. Nineteen developers (58%) answered that they intend to use the new syntax. Two of them declared to have plans to migrate their systems to the new syntax, while the others stated that they intend to use it only when implementing new features and projects, as stated in the following answer:

"I'm quite confident that ES6 will make for a more robust codebase. And I think the most interesting point is that it can be applied progressively. We don't have to make a massive rewrite. Any new code we add can be ES6, and then we can slowly rewrite old code to be ES6 as well." (Developer of system SOCKET.IO*)*

Table 3.5: Intention to use ES6 classes

Type of answer	#	%
Yes	19	58
No	12	36
Did not know	2	6

Twelve developers (36%) declared they do not intend to use ES6 syntax for classes, because they have to keep compatibility with legacy code, as stated in the following answer:

"For us right now it makes more sense to use CJS patterns and integrate with existing module ecosystems. When the ES6 penetration and support is higher, maybe we will switch." (Developer of system PIXIJS*)*

Summary: 58% of JavaScript developers intend to use the ES6 syntax for classes, but mostly for new features and projects.

3.4 Threats to Validity

This section presents threats to validity according to the guidelines proposed by Wohlin et al. [2012]. These threats are organized in three categories, addressing internal, external, and construct validity.

Internal Validity. In the field study, to address RQ #5, we recognize two internal threats. First, we consider that the developers correctly evaluated all elements we provided in the class diagrams of their systems. We acknowledge this activity is error-prone, and that they may have missed some class elements, like methods and attributes. However, we asked the main developers of each system, who are probably the most qualified people to conduct such evaluation. The second internal threat is related to the identification of false negatives for the analysis of recall. Since some developers did not provide the names of all classes that represent false negatives in their systems, we performed a manual verification in the related source code files in order to identify the remaining structures.

External Validity. To address the first four research questions, we used a dataset of 918 JavaScript systems. For research questions RQ #5 and RQ #6, which involved contacting developers, we used a dataset of 60 JavaScript systems. As a threat, our datasets, both obtained from GitHub repository, might not represent the whole population of JavaScript systems. Our selection excludes web applications that use JavaScript together with other languages. For example, we do not consider JavaScript code inserted between `<script>` and `</script>` tags, in HTML files. But, at least, we selected a representative number of popular and well-known systems, of different sizes and covering various domains.

Construct Validity. We use the library Linguist and a custom-made script, as described in Subsection 3.2.2, to remove unnecessary files from our dataset. We assume that this clean up process does not remove any source code files that could be used to implement classes. It is also possible that some of the selected repositories do not represent real systems, but other types of applications, *e.g.*, tutorials and toy examples.

3.5 Final Remarks

This chapter provides a large-scale study on the usage of class-like structures in JavaScript, a language that is used nowadays to implement complex single-page applications for the Web. We propose a strategy to statically detect class emulation in legacy JavaScript code and the JSCLASSFINDER tool, that supports this strategy. We use JSCLASSFINDER on a corpus of 918 popular JavaScript applications, with different sizes and from multiple domains, in order to describe the usage of class-like structures in legacy JavaScript systems. This study shows that: (i) JavaScript developers often emulate classes in legacy code to tackle the complexity of their systems; (ii) there is no significant relation between size and class usage; (iii) prototype-based inheritance is not popular in JavaScript; (iv) it is common to have JavaScript classes with more attributes than methods; (v) JavaScript developers that emulate classes tend to use the ES6 new syntax, mostly for new features and projects. We also perform a field study with JavaScript developers to evaluate the accuracy of our strategy and tool. Precision, recall and F-score measures indicate that our tool is able to identify the classes, methods, and attributes in JavaScript systems. The overall results range from 97% to 100% for precision, from 70% to 89% for recall, and from 82% to 94% for F-score.

Chapter 4

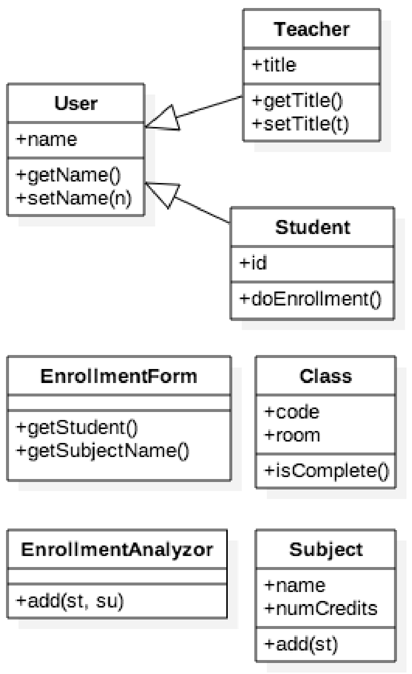
Identifying Class Dependencies

In this chapter, we evaluate the use of Flow type inferencer to extract dependencies between classes in legacy JavaScript code. Section 4.1 provides information about class dependencies in JavaScript. Section 4.2 describes the methodology used in a study to detect class-to-class dependencies in legacy JavaScript code. Section 4.3 describes the research questions that guided this study, along with the dataset and the metrics we used. We discuss answers to the proposed research questions in Section 4.4. We explain and discuss Flow limitations in Section 4.5 and discuss lessons learned in Section 4.6. Threats to validity are exposed in Section 4.7. We conclude this chapter by summarizing our findings in Section 4.8.

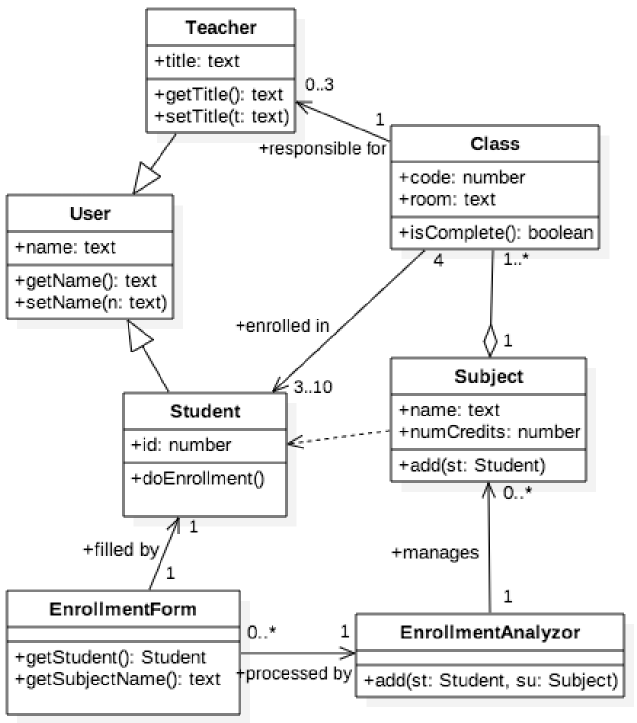
4.1 Motivation

Highly-coupled systems, in which modules have unnecessary dependencies, are hard to maintain and evolve because these modules cannot be easily understood separately [Parnas, 1978, Sangal et al., 2005b]. The same rationale applies to class-to-class dependencies in object-oriented code. In some cases, an excessive number of class dependencies may be considered an indicator of poor software design. In JavaScript, the identification of such dependencies is specially challenging due to the lack of types in the source code. These dependencies form the basis to provide, for example, class diagrams for JavaScript applications.

In Chapter 3, we proposed a strategy to statically identify classes in legacy JavaScript code. Figure 4.1 resumes how we intend to enrich this strategy with class-to-class dependencies. The figure illustrates two class diagrams for a simplified version of an enrollment system. The first diagram (a) contains information about classes, methods, attributes, and inheritance relationships. The second diagram (b) complements the first one with type information and dependencies between classes. The current implementation of JSCLASSFINDER provide the diagram (a). In this chapter, we investigate the use of type inference to produce a diagram similar to diagram (b).



(a) Class diagram as currently identified by JSCLASSFINDER



(b) Class diagram with dependencies and types

Figure 4.1: Class diagrams for a simplified enrollment system

Based on the UML specification [Fowler, 2003], we consider two types of class-to-class dependencies in JavaScript. *Associations* are particular cases of dependencies in which a class contains one or more attributes that are bound to instances of other classes. Figure 4.2 shows two examples of associations in JavaScript. In code (1), the constructor function for class `Z` is implemented in lines 8-11. The attribute `x`, which is part of class `Z`, receives an instance of class `X` (line 10), creating an association between the two classes. In code (2), the constructor function `Z` has a parameter `x` (line 8), which is assigned to the attribute `this.x` (line 9). In line 12, an instance of `Z` is created, with the parameter `x` bound to a newly created object of type `X`. If we consider the source code of the constructor functions, in code (1) we clearly notice the association between the two classes. However, in code (2), we need a client code (line 12), with the instantiation of classes `Z` and `X`, to establish the association.

```
// class X
function X()
{
  ...
}
...
// class Z
function Z()
{
  this.x = new X();
}
```

(1)

```
// class X
function X()
{
  ...
}
...
// class Z
function Z(x) {
  this.x = x;
}
...
var z = new Z(new X());
```

(2)

Figure 4.2: Examples of associations (from class `Z` to class `X`)

When a relationship between classes does not involve assignments of objects to class attributes, we have a *uses* relationship, because one class just uses the other. Figure 4.3 shows two examples of dependencies in which a class `Z` *uses* another class `X`. In code (1), we can see a method `foo` (lines 1-6) of a class `Z`. This method creates an instance of `X` and stores it in a local variable `_x` for later use (line 4). In code (2), method `foo` receives an object as argument and *uses* it to invoke its method `bar` (line 4). The identification of the class dependency is more challenging in code (2) because there is no direct reference to the "used" class in the source code of the method.

```
1 Z.prototype.foo =
2   function()
3   {
4     var _x = new X();
5     ...
6   }
```

(1)

```
1 Z.prototype.foo =
2   function(x)
3   {
4     var _bar = x.bar();
5     ...
6   }
```

(2)

Figure 4.3: Examples of dependencies of type "uses"

In this chapter, the term dependency is used generically to denote associations and dependencies of the type *uses*.

4.2 Using Flow to Infer Dependencies

In this section, we describe the usage of a static type checker (Facebook Flow) to identify dependencies between structures that emulate classes in legacy JavaScript code. Figure 4.4 presents an overview of this methodology. Given a JavaScript legacy application, we perform the following steps.

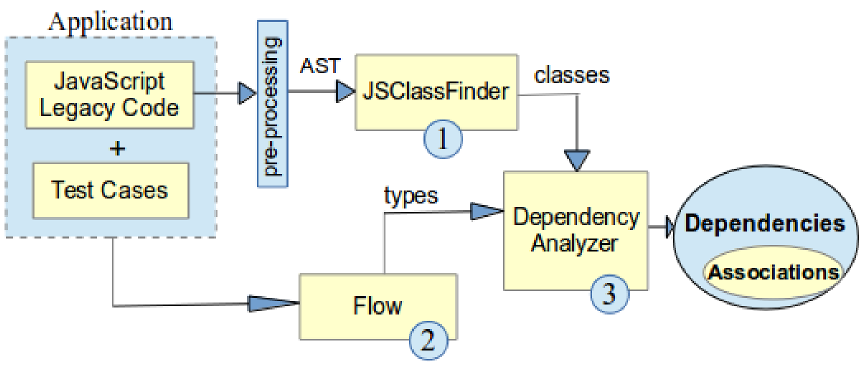


Figure 4.4: Overview of the evaluated approach

Step 1: Identify classes. In this first step, we identify the classes emulated in a legacy JavaScript codebase using JSCLASSFINDER, which works on the application's abstract syntax tree that is generated after pre-processing the source code. The tool then applies a set of heuristics to identify classes, methods, attributes, and inheritance relationships, as described in Chapter 3.

Step 2: Infer types. The decision to use Flow for type inference was taken basically because it is a very active open source project, with more than 390 contributors, 4,800 commits, and almost 13K stars (data collected in August 2017).[1] We execute Flow passing the application's source code and tests as input. In this case, the tests are important to determine some of the types involved in class instantiations and method calls, like in the association showed in code (2) of Figure 4.2. In that example, it is not possible to establish the association between classes `Z` and `X` without the object instantiation in line 12. Usually, these kind of object instantiation is only present outside the respective class implementation, *e.g.*, in a test code. By contrast, the class identification (step 1) is based only on the source files, without the tests, to avoid false positives for classes that are not part of application's source code.

The output produced by Flow is a text file that contains the coordinates (line, column) for every element of the source code (variable, function, object, etc), and their respective types. Figure 4.5 shows the emulation of a simple class `Point` (code on the left) and the correspondent Flow's output (text on the right). We can see that there is a function in line 1 of the output file, denoted by the arrow ("=>"), whose name is located in line 4 of the file `Point.js`, between columns 10 and 14. This function has two arguments (`x` and `y`) of type `number` and returns `void`. Every time the keyword `this` is used, Flow infers the structure of the object that it represents as its type. For example, the structure of the object `this` that appears in lines 5 and 6 of the legacy

[1] `https://github.com/facebook/flow`

code is {x : number, y : number} (see lines 4 and 7 of Flow's output file). In this case, the inferred object represents the attributes of `Point`. In other words, when a reference to an emulated class is found, Flow infers the structure of the object that represents the class as its type.

In JavaScript, the value of `this` depends on how a function is called, and it may be different each time the function is called.[2] For this reason, we do not have the return type of function `getX` in line 11 of the output file. In this case, since `getX` is not invoked in the source code, Flow does not infer neither its return type nor the type of `this.x`, used inside `getX` (lines 11-15 of the output file). To overcome this problem, it is important to pass the tests together with the source code as input for Flow, using as many function invocations as possible. If we add a call to `getX` at the end of the source code on the left, for example `p.getX()`, then Flow gives the following output: `getX : () => number`, identifying that the function returns a number.

Point.js

```
 1 /*
 2  Constructor function
 3 */
 4 function Point (x, y) {
 5   this.x = x;
 6   this.y = y;
 7 }
 8
 9 /*
10  Method getX()
11 */
12 Point.prototype.getX =
13   function() {
14     return this.x;
15   }
16
17 // Creating an instance
18 var p = new Point(2,3);
```

⇒

Flow's output (text file with types)

```
 1 Point.js:4:10-14: (x:number, y:number)=>void
 2 Point.js:4:17: number
 3 Point.js:4:20: number
 4 Point.js:5:2-5: {x:number, y:number}
 5 Point.js:5:2-11: number
 6 Point.js:5:11: number
 7 Point.js:6:2-5: {x:number, y:number}
 8 Point.js:6:2-11: number
 9 Point.js:6:11: number
10 Point.js:12:1-5: (x:number, y:number)=>void
11 Point.js:12:1-15: {getX: () => }
12 Point.js:12:1,15:2: () =>
13 Point.js:13:2,15:2: () =>
14 Point.js:14:13-16:
15 Point.js:14:13-18:
16 Point.js:18:5: {x:number, y:number}
17 Point.js:18:9-22: {x:number, y:number}
18 Point.js:18:13-17: (x:number, y:number)=>void
19 Point.js:18:19: number
20 Point.js:18:21: number
```

Figure 4.5: Example of Flow's output file

Step 3: Locate the dependencies. The classes detected by JSCLASSFINDER (step 1) and the types inferred by Flow (step 2) are used by a component we implemented for this study, called Dependency Analyzer (see Figure 4.4). This component uses structural equivalence to identify associations and *uses* dependencies, since Flow infers the structures of objects, including class instances, as their types. In structural equivalence, two values have equivalent types if the types have isomorphic structures [Connor et al., 1990].

[2]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this

4.3 Study Design

In this section, we detail the design of a study proposed to evaluate the usage of Flow to detect dependencies between classes in legacy JavaScript systems. First, we present the dataset and the oracle that we use (Section 4.3.1). Then, we present the research questions that guide this study (Section 4.3.2).

4.3.1 Dataset and Oracle

For our study, we need systems that emulate classes in legacy JavaScript in order to find the dependencies between these classes. We also need an oracle of class dependencies to measure the accuracy of the types inferred by Flow. For these reasons, we use two TypeScript open-source projects to build this oracle and minimize possible bias. TypeScript is an extension of JavaScript that offers classes, interfaces, and a gradual type system [Nance, 2014]. Therefore, by parsing and extracting explicit types in TypeScript programs, class-to-class dependencies can be found and added to an oracle. Before performing our analyses, TypeScript code is *transpiled*[3] to vanilla JavaScript. The automatically generated code contains all constructor functions and methods, along with their dependencies, implemented according to JavaScript ES5 syntax. The strategy of using TypeScript projects to build an oracle is also adopted by Rostami et al. [2016] in their work to detect constructor functions in JavaScript.

Table 4.1 presents the main characteristics of the two TypeScript projects in our dataset, including version number, size (LOC), number of classes, number of dependencies, and number of dependencies that are class associations. INVERSIFYJS[4] is a lightweight inversion of control container for TypeScript and JavaScript applications. SATELLIZER[5] is an end-to-end token-based authentication module for `AngularJS` with built-in support to different service providers.

Table 4.1: Characteristics of the analyzed systems

System	Version	LOC	# Classes	# Class Dependencies	# Class Associations
INVERSIFYJS	2.0.1	1,527	20	184	26
SATELLIZER	0.15.5	990	11	46	20

[3]A transpiler is a source-to-source compiler. Transpilers are used, for example, to convert from TypeScript to JavaScript, in order to guarantee compatibility with existing browsers and runtime tools.

[4]`https://github.com/inversify/InversifyJS`

[5]`https://github.com/sahat/satellizer`

As an example of a class in TypeScript, Listing 4.1 shows part of the implementation of class `QueryString` in INVERSIFYJS. We can see explicit type annotations in attributes (line 2), which are used to infer associations, and in parameters (lines 4 and 7), which are used to infer *uses* relations, when building the oracle of class dependencies used in this chapter.

```
1  class QueryableString {
2    private str: string;
3
4    constructor(str:string) {
5      this.str = str;
6    }
7    public startsWith (searchString: string): boolean {
8      ...
9    }
10 }
```

Listing 4.1: Example of class in TypeScript

Listing 4.2 shows a second example that contains class associations in INVERSIFYJS. The import statements (lines 1-3) link the variables (`Planner`, `Resolver`, and `Lookup`) to the respective classes, implemented in other files. Lines 5-19 contain the implementation of class `Kernel`, whose attributes are defined in lines 6-9 and initialized in the constructor function (lines 12-17). The attributes `_planner`, `_resolver`, and `_bindingDictionary` receive new instances of the classes `Planner`, `Resolver`, and `Lookup`, respectively (lines 14-16). Therefore, we count three class associations in this example.

```
1  import Planner from "../planning/planner";
2  import Resolver from "../resolution/resolver";
3  import Lookup from "./lookup";
4
5  class Kernel {
6    public guid: string;
7    private _planner: Planner;
8    private _resolver: Resolver;
9    private _bindingDictionary: Lookup<Binding<any>>;
10
11   // Initialize private properties
12   public constructor() {
13     this.guid = guid();
14     this._planner = new Planner();
15     this._resolver = new Resolver();
16     this._bindingDictionary = new Lookup<Binding<any>>();
17   }
18   ...
19 }
```

Listing 4.2: Example of class associations in TypeScript

It is important to mention that the TypeScript code is only used to produce the oracle of class dependencies (which was performed manually, by this thesis' author, by inspecting the source code of the two systems described in Table 4.1). In the study, Flow is always executed on the legacy JavaScript code, produced by the TypeScript's transpiler.

4.3.2 Research Questions

RQ #1 - What is the accuracy of Flow in detecting class-to-class dependencies?

Answering this first research question is important to assess how accurate and complete are the class-to-class dependencies identified by Flow. The oracle with all class dependencies allows the measurement of precision and recall.

RQ #2 - Can we improve the accuracy of Flow by expanding `require` statements?

Both analyzed systems address module dependencies that comply with `CommonJS`[6] standard, which is implemented by `Node.js`. In this module system, JavaScript files use the built-in function `require()` to load modules, which can reference other JavaScript files or entire folders. In this second research question, we intend to improve Flow's accuracy by eliminating the need of variables that make reference to class constructor functions implemented in other modules, using instead the names of these functions directly. We use the example in Figure 4.6 to illustrate this approach. In this figure, we initially represent two class constructor functions, `X` and `Z`, implemented in files `fileX.js` and `fileZ.js`, respectively. There is a dependency between them, as we can see in the implementation of `fileZ.js` (lines 2 and 5). The `require()` invocation in line 2 assigns the implementation of `X` to variable `x_1`. This variable is then used to establish the association between `X` and `Z` (line 5). In order to eliminate the need of the variable `x_1` and, by consequence, the need of the `require` statement, in this research question we propose to "expand" the source code of `fileX.js`, which is passed as argument to `require`, in `fileZ.js`. As a result, the implementations of `X` and `Z` are bundled together, as we can see in the final implementation of `fileZ.js`. Therefore, after bundling together the files that depend on each other, dependencies can be established by making direct reference to other class constructor functions, like in line 8 of the bundled file.

[6]`http://www.commonjs.org/`

```
// fileX.js
function X()
{
  ...
}
```

fileX.js ↘

```
// fileZ.js
var x_1 = require("fileX.js");

function Z() {
  this.x = new x_1();
}
```

↙ fileZ.js

```
// fileX.js
function X() {
  ...
}

// fileZ.js
function Z() {
  this.x = new X();
}
```

fileZ.js (after expansion)

Figure 4.6: Example of constructor functions `Z` and `X` bundled together in a same file

To answer RQ #2, we expanded the source code of all `require` statements that import the definitions of other class constructor functions, in both analyzed systems. Then, we used the resulting files as input for Flow. The transformation proposed in this research question is similar to the one performed by the C preprocessor when handling #`include` directions. The main difference is the need to replace every use of the variable with the `require` result by the imported class constructor.

4.4 Results

4.4.1 What is the accuracy of Flow in detecting class-to-class dependencies?

In this first research question, after executing Flow, we could not identify any class-to-class dependencies; therefore, we have (P)recision = (R)ecall = 0. After a manual inspection and analysis, we found that this result is due to the use of variables that denote constructor functions implemented in other files, like in the example shown in Listing 4.3. In this case, we can see two import statements (lines 1-2) and the constructor function of class `BindingInWhenOnSyntax` (lines 4-9) in INVERSIFYJS. Since Flow cannot determine the return type of function `require` (lines 1-2), the types of variables `binding_when_syntax_1` (lines 1 and 6) and `binding_on_syntax_1` (lines 2 and

7) are unknown. Both analyzed systems rely on variables to import class definitions from other files. This observation motivated the investigation proposed in RQ #2.

```
var binding_when_syntax_1 = require("./binding_when_syntax");
var binding_on_syntax_1 = require("./binding_on_syntax");

function BindingInWhenOnSyntax (binding) {
  this._binding = binding;
  this._bindingWhenSyntax= new binding_when_syntax_1(this._binding);
  this._bindingOnSyntax = new binding_on_syntax_1(this._binding);
  ...
}
```

Listing 4.3: JavaScript code using variables denoting constructor functions in INVERSIFYJS

Summary: In the configuration assumed by RQ #1, Flow cannot identify the expected class-to-class dependencies in the analyzed systems, having (P)recision = (R)ecall = 0.

4.4.2 Can we improve the accuracy of Flow by expanding `require` statements?

In this second research question, we expand the source code of classes implemented in other modules, bundling together the files that depend on each other, and replace variable references by direct references to the respective class constructor functions. Table 4.2 summarizes the results according to these transformations. In this case, we start to have positive results. First, we have high precision values (100%) in both analyzed systems. In the case of recall, we observe INVERSIFYJS with 86% for all class dependencies and 96% if we only consider class associations. In the case of SATELLIZER, recall is 80% for all dependencies and 85% for class associations. In essence, these results suggest that the type inference mechanism is very conservative. Flow only makes the decision about a type after finding strong evidences of its use.

Table 4.2: Precision and Recall results

System	All Dependencies					Associations				
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
INVERSIFYJS	159	0	25	100%	86%	25	0	1	100%	96%
SATELLIZER	37	0	9	100%	80%	17	0	3	100%	85%

Summary: By eliminating and expanding the code of `require` statements precision improves to 100% in both systems. The values of recall range from 80% to 86% for all dependencies and from 85% to 96% for associations.

4.5 Explaining the Recall Results in RQ #2

We manually analyzed the source code to understand the cases in which the class-to-class dependencies could not be identified by Flow when answering the second research question. We categorized and quantified the false negatives, which are described next.

Use of Dependency Injection (DI). `AngularJS`[7] is a JavaScript-based front-end web application framework. One of its features is a built-in dependency injection (DI) subsystem provided as a service [Lerner, 2013, Jain et al., 2015, Ramos et al., 2017]. With this mechanism, instead of creating a dependency directly, developers can request this service using the directive `$inject`, and then `AngularJS` answers providing the instance they need. Listing 4.4 shows part of the implementation of class `Interceptor` in SATELLIZER. The constructor function receives three parameters that represent instances of other classes (line 2). There is also a factory method that calls this constructor function (lines 6-9). However, Flow cannot establish the class-to-class dependencies in this case because the factory method is not called directly in the source code. Instead, developers use the DI mechanism setting the directive `$inject` like is shown in Listing 4.5.

```
1 // Constructor function (class Interceptor)
2 function Interceptor(SatellizerConfig, SatellizerShared, SatellizerStorage) {
3   ...
4 }
5 // Factory method
6 Interceptor.Factory =
7   function (SatellizerConfig, SatellizerShared, SatellizerStorage) {
8       return new Interceptor(SatellizerConfig, SatellizerShared,
          SatellizerStorage);
9   };
```

Listing 4.4: Factory method for class `Interceptor` in SATELLIZER

```
// Setting AngularJS directive for dependency injection
Interceptor.$inject = ['SatellizerConfig', 'SatellizerShared', '
   SatellizerStorage'];
Interceptor.Factory.$inject = ['SatellizerConfig', 'SatellizerShared', '
   SatellizerStorage'];
```

Listing 4.5: Setting dependency injection directives in SATELLIZER

[7]`https://angular.io/`

We found nine dependencies that could not be identified by Flow due to the use of dependency injection in SATELLIZER, including three associations and six dependencies of type "*uses*".

Absence of method invocation. Some methods are never called in any part of the source code nor tests due to incomplete test coverage or because they represent dead code. Like pointed out in Section 4.2, Flow may miss types for methods that are not called. Listing 4.6 shows the implementation of method `removeByModuleId` from class `Lookup`, in INVERSIFYJS. Checking the oracle we can see that the parameter `moduleId` (line 1) is of type `string`, and that `this` (line 2) is an instance of `Lookup`. However, in the legacy code, we cannot determine these types because the method `removeByModuleId` is never called in the existing codebase. Moreover, in JavaScript, the value of `this` is determined by how a function is called, and it may be different each time the function is called. Therefore, Flow does not infer the types of `moduleId` (line 1) and `this` (line 2). By consequence, the types of `_dictionary` (line 2), `keyValuePair` (line 2), and `binding` (line 4) are also unknown.

```
1 Lookup.prototype.removeByModuleId = function (moduleId) {
2     this._dictionary.forEach(function (keyValuePair) {
3       keyValuePair.value =
4         keyValuePair.value.filter(function (binding) {
5           return binding.moduleId !== moduleId;
6         });
7     });
8     ...
9 };
```

Listing 4.6: Example of method that is not used in INVERSIFYJS

We found eight dependencies of type "*uses*" in INVERSIFYJS that could not be identified by Flow because their enclosing methods are not invoked.

Use of the object `arguments`*.* JavaScript functions have a built-in object called `arguments`. This object contains an array of the arguments used when the function is invoked. This way, one can call a function passing arguments even when the function's signature does not expect any arguments. As an example, Listing 4.7 shows the implementation of method `load` from class `Kernel`, in INVERSIFYJS. We can see by the function's signature (line 1) that the method expects no arguments. However, the loop in lines 5-7 transfers all elements from `arguments` to the array named `modules`. Therefore, method `load` can be called with arguments, like in the test code shown in lines 11-14. In line 14, we can see the invocation of method `load` passing two parameters of type `KernelModule`. As expected, Flow does not infer the content of the object `arguments` (lines 5-6) and, therefore, cannot identify the dependency between classes `Kernel` and `KernelModule`, which is present in this example. By contrast, in the TypeScript version of method `load`, this dependency is explicit, as we can see in Listing 4.8. The argument `modules` (line 3) is an array of type `KernelModule`. In this line code, specifically, the ellipses ("...") placed before the name of the argument is part of TypeScript syntax, indicating that the number of elements in `modules` may vary.

```
1  Kernel.prototype.load = function () {
2      var _this = this;
3      var modules = [];
4
5      for (var _i = 0; _i < arguments.length; _i++) {
6        modules[_i - 0] = arguments[_i];
7      }
8      ...
9  };
10 // Test code
11 var warriors = new KernelModule(..);
12 var weapons = new KernelModule(..);
13 var kernel = new Kernel();
14 kernel.load(warriors, weapons);
```

Listing 4.7: Example of method that uses the built-in object `arguments`

```
1  class Kernel {
2    ...
3    public load(...modules: KernelModule[]): void {
4        ...
5        modules.forEach((module) => {
6          let bindFunction = getBindFunction(module.guid);
7          module.registry(bindFunction);
8      });
9    }
10   ...
11 }
```

Listing 4.8: TypeScript version of method `load` from class `Kernel`

We found 12 dependencies of type "*uses*" in INVERSIFYJS that are not identified by Flow because the methods make use of the `arguments` object.

Attributes of type `Array`. `Array` is a JavaScript global object that is used in the construction of arrays, i.e., list-like objects. Neither the length of a JavaScript array nor the types of its elements are statically pre-defined.[8] As an example, Listing 4.9 shows the constructor function of class `Target` in INVERSIFYJS. This class has an attribute of type `Array` (line 3), called `metadata`, which is initialized with a new instance of `Array`. The conditional statements (lines 4-13) control when elements are added to the array using the method `push` (line 12). In this case, when Flow applies the algorithm for type inference, it indicates that the type of the attribute `metadata` is simply `Array`, without specifying what kind of elements are added. Therefore, we cannot establish the association between classes `Target` and `Metadata`. In the TypeScript code, this dependency becomes explicit when the attribute of type `Array` is declared, as we can see in line 3 of Listing 4.10.

```
 1 function Target (type, name, serviceIdentifier, namedOrTagged) {
 2   ...
 3   this.metadata = new Array();
 4   var metadataItem = null;
 5   if (typeof namedOrTagged === "string") {
 6     metadataItem = new Metadata(NAMED_TAG, namedOrTagged);
 7   }
 8   else if (namedOrTagged instanceof Metadata) {
 9     metadataItem = namedOrTagged;
10   }
11   if (metadataItem !== null) {
12     this.metadata.push(metadataItem);
13   }
14 }
```

Listing 4.9: Example of class constructor that has an attribute of type `Array`

```
1 class Target  {
2   ...
3   public metadata: Array<Metadata>;
4   ...
5 }
```

Listing 4.10: TypeScript implementation of a class that has an attribute of type `Array` < `Metadata` >

We found five cases in INVERSIFYJS, including one association and four dependencies of type "*uses*", that could not be identified by Flow because the respective attribute is an instance of `Array`.

Concluding our analysis of false negatives, Table 4.3 summarizes all the identified causes along with the number of instances found in both analyzed systems.

[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Table 4.3: Classification of false negatives

Causes of False Negatives	All Dependencies		Associations	
	INVERSIFYJS	SATELLIZER	INVERSIFYJS	SATELLIZER
Use of Dependency Injection	0	9	0	3
Absence of method invocation	8	0	0	0
Use of the object `arguments`	12	0	0	0
Attributes of type `Array`	5	0	1	0

4.6 Discussion and Lessons Learned

We could not find any class-to-class dependencies when Flow is used in its default configuration, as in RQ #1. The reason is that Flow cannot infer types for references across modules. In both analyzed systems, these modules are imported by calling the function `require`, which is part of `CommonJS` module specification. In these cases, the imported modules are assigned to variables for further use, as showed when investigating the first research question. As a result, the types of these variables are unknown. To overcome this problem, we decided to replace the variables that make reference to constructor functions implemented in other modules by the names of the respective constructor functions. To achieve this, we expanded the source code imported by the `require` statements, gathering the classes that depend on each other in the same module (file). This last approach achieved the best results for precision and recall. Finally, we manually analyzed and categorized all false negatives identified in our study, considering RQ #2.

Due to the dynamic nature of JavaScript, we hypothesize that it would be necessary to combine static and dynamic analysis to tackle the identified causes of false negatives, improving recall and providing more reliable information. However, it is known that dynamic analysis brings a new set of challenges to software engineers, such as code instrumentation and the need to set up and execute every program under analysis. Therefore, we conclude that the use of Flow to statically infer types and spot class-to-class dependencies in JavaScript legacy code can achieve an accuracy sufficient to support its usage by reverse engineering tools. However, we need to expand the source code in the `require` statements and to eliminate variable references to classes implemented in other files.

4.7 Threats to Validity

This section presents threats to validity according to the guidelines proposed by Wohlin et al. [2012]. These threats are organized in external, internal, and construct validity.

External Validity. We studied two open-source JavaScript/TypeScript systems. For this reason, our results might not represent all possible cases of class dependencies and associations. If other systems are considered, the effect of removing references to classes implemented in other modules can vary. To allow the replication of our study, the oracle along with the detected dependencies for both systems is available on-line.[9]

Other external validity is related to the fact that we only address module dependencies that comply with `CommonJS`. We acknowledge that there are different strategies to incorporate modules into JavaScript programs, *e.g.*, global functions and AMD[10], which also support modules.

Internal Validity. We analyze JavaScript legacy code automatically generated by the TypeScript's transpiler. Perhaps the use of a typed language can favor simpler and less dynamic applications. It is also possible that applications originally implemented in JavaScript use different implementation strategies that we do not consider in our analysis.

Construct Validity. To build the oracle, we manually inspect the TypeScript systems looking for types representing classes. It is possible that we miss dependencies during this inspection. However, the analyzed systems are not large, and, since the types are explicit in the source code, we claim that the oracle contains all class-to-class dependencies.

4.8 Final Remarks

In this chapter, we evaluate the usage of Flow, a static type-checker for JavaScript, to infer class-to-class associations and dependencies of type *uses* in legacy JavaScript code. We report a study on two open-source projects whose code is transpiled from TypeScript to JavaScript. We could not find any class-to-class dependencies when Flow is used in its default configuration, as in the first research question. After expanding the source code in the `require` statements to eliminate variable references to class constructor functions implemented in other modules, our results show that precision reaches 100% in the two evaluated systems. Flow's recall ranges from 80% to 86% for

[9]`https://github.com/leonardo-silva/JSClassDependencies`
[10]`https://github.com/amdjs/amdjs-api/wiki/AMD`

all dependencies and from 85% to 96% for associations. Finally, we manually analyzed and categorized all false negatives identified in our study. These false negatives are due to: (i) use of dependency injection (9 instances); (ii) absence of method invocation (8 instances); (iii) use of the object `arguments` (12 instances); and (iv) use of attributes of type `Array` (5 instances). The overall results suggest that this approach can be used to support the implementation of reverse engineering tools for JavaScript. However, we envision that future solutions should also make use of dynamic analysis to cover all possible dependencies in JavaScript code.

Chapter 5

Refactoring JavaScript Classes

In this chapter, we investigate the feasibility of rejuvenating legacy JavaScript classes and, therefore, to increase the chances of code reuse in the language. First, we describe a study on migrating a dataset of real-world JavaScript systems to adopt the new syntax for classes provided by ES6 (Section 5.1). We present the set of rules followed to conduct this migration (Section 5.1.1) and the dataset we use (Section 5.1.2). Then, we quantify the amount of code (churned and deleted) that can be automatically migrated by the proposed rules (Section 5.2.1) and we describe a set of cases where manual adjusts are required to migrate the code (Section 5.2.2). We also describe the limitations of the new syntax for classes provided by ES6, i.e., the cases where it is not possible to migrate the code and, therefore, we should expose the prototype-based object system to ES6 maintainers (Section 5.2.3). Threats to validity are presented in Section 5.3. Based on the results of our study, we propose a tool to automatically refactor the cases covered by the proposed set of rules and to warn developers about the cases that cannot be automatically migrated (Section 5.4). In Section 5.5, we document a set of reasons that can lead developers to postpone/reject the adoption of ES6 classes. These reasons are based on the feedback received after submitting pull requests suggesting the migration to the new syntax. Finally, we conclude this chapter with general remarks on the refactoring of legacy JavaScript code to use ES6 classes (Section 5.6).

5.1 Study Design

In this section, we describe our study to migrate a set of legacy JavaScript systems (implemented in ES5) to use the new syntax for classes proposed by ES6. First, we present the rules followed to conduct this migration (Section 5.1.1). Then, we present the set of selected systems in our dataset (Section 5.1.2).

Rule #1: Class

ES5

```
function C(p0) {
  B0;
  this.m1 = function(p1) {
    B1;
  }
  B2;
}
C.prototype.m2 = function(p2) {
  B3;
}
C.m3 = function(p3) {
  B4;
}
```

$\Longrightarrow$

ES6

```
class C {
  constructor(p0) {
      B0; B2;
  }
  m1(p1) {
    B1;
  }
  m2(p2) {
    B3;
  }
  static m3(p3) {
    B4;
  }
}
```

Rule #2: Subclasses

ES5

```
class C {
  B0;
}
C.prototype = new D();
```

$\Longrightarrow$

ES6

```
class C extends D {
  B0;
}
```

Rule #3: `super()` *calls*

ES5

```
class C extends D {
  B0;
  constructor(p0) {
    B1; D.call(this, p1); B2;
  }
  B3;
  m1(p2) {
    B4;
    D.m2.call(this, p3);
    B5;
  }
  B6;
}
```

$\Longrightarrow$

ES6

```
class C extends D {
  B0;
  constructor(p0) {
    B1; super(p1); B2;
  }
  B3;
  m1(p2) {
    B4;
    super.m2.(p3);
    B5;
  }
  B6;
}
```

Figure 5.1: Migration rules (p_i is a formal parameter list and B_i is a block of statements)

5.1.1 Migration Rules

Figure 5.1 presents three basic rules to migrate classes emulated in legacy JavaScript code to use the ES6 syntax. Each rule defines a transformation that, when applied to legacy JavaScript code (program on the left) produces a new code in ES6 (program on the right). Starting with Rule #1, each rule should be applied multiple times, until a fixed point is reached. After that, the migration proceeds by applying the next rule. The process finishes after reaching the fixed point of the last rule.

For each rule, the left side is the result of "desugaring" this program to the legacy syntax. The right side of the rule is a template for an ES6 program using the new syntax. Since there is no standard way to define classes in ES5, we consider three different patterns of method implementation, including methods inside/outside class constructors and using prototypes [Silva et al., 2015b, Gama et al., 2012]. Rule #1 defines the migration of a class `C` with three methods (`m1`, `m2`, and `m3`) to the new class syntax (which relies on the keywords `class` and `constructor`). Method `m1` is implemented inside the body of the class constructor, `m2` is bound to the prototype of `C`, and `m3` is also implemented outside the class constructor but it is not bound to the prototype (static method).[1] Rule #2, which is applied after migrating all constructor functions and methods, generates subclasses in the new syntax (by introducing the `extends` keyword). Finally, Rule #3 replaces calls to super class constructors and to super class methods by making use of the `super` keyword.

There are no rules for migrating fields, because they are declared with the same syntax both in ES5 and ES6 (see Chapter 2, Section 2.1.2). Moreover, fields are most often declared in constructor functions or less frequently in methods. Therefore, when we migrate these elements to ES6, the field declarations performed in their code are also migrated.

5.1.2 Dataset

We select systems that emulate classes in legacy JavaScript code in order to migrate them to the new syntax. In Chapter 3, we presented an empirical study on the use of classes in legacy JavaScript systems. In this chapter, we select eight systems from the dataset used in this previous study. The selected systems have at minimum one and at maximum 100 classes, and 40 KLOC. For each system, this study includes the checkout from GitHub, the setup of development environment, and the execution of the existing tests before and after the migration.

Table 5.1 presents the selected systems, including a brief description, checkout date, size (LOC), number of files, number of classes (NOC), number of methods (NOM), and class density (CD). CD is the ratio of functions in a program that are related to the emulation of classes (i.e., functions which act as methods or class constructors) [Silva et al., 2015b]. JSClassFinder [Silva et al., 2015a] was used to identify the classes emulated in legacy code and to compute the measures presented in Table 3.1. The selection includes well-known and widely used JavaScript systems, from different domains,

[1]For the sake of legibility, Rule #1 assumes a class with only one method in each idiom. The generalization for multiple methods is straightforward.

covering frameworks (SOCKET.IO and GRUNT), graphic libraries (ISOMER), visualization engines (SLICK), data structures and algorithms (ALGORITHMS.JS), and a motion detector (PARALLAX). The largest system (PIXI.JS) has 23,952 LOC, 83 classes, and 134 files with `.js` extension. The smallest system (FASTCLICK) has 846 LOC, one class, and a single file. The average size is 4,681 LOC (standard deviation 7,881 LOC), 15 classes (standard deviation 28 classes) and 29 files (standard deviation 48 files).

Table 5.1: JavaScript systems whose classes where migrated to ES6, ordered by the number of classes.

System	Description	Checkout Date	LOC	Files	Classes	Methods	Class Density
FASTCLICK	Library to remove click delays	01-Sep-16	846	1	1	16	0.74
GRUNT	JavaScript task runner	30-Aug-16	1,895	11	1	16	0.16
SLICK	Carousel visualization engine	24-Aug-16	2,905	1	1	94	0.90
PARALLAX	Motion detector for devices	31-Aug-16	1,018	3	2	56	0.95
SOCKET.IO	Realtime app framework	25-Aug-16	1,408	4	4	59	0.95
ISOMER	Isometric graphics library	02-Sep-16	990	9	7	35	0.79
ALGORITHMS.JS	Data structures & algorithms	21-Aug-16	4,437	70	20	101	0.54
PIXI.JS	Rendering engine	05-Sep-16	23,952	134	83	518	0.71

5.2 Migration Results

We follow the rules presented in Section 5.1 to migrate the systems in our dataset to ES6. We classify the migrated code in three groups:

- *The Good Parts.* Cases that are straightforward to migrate, without the need of further adjusts, by just following the migration rules defined in Section 5.1.1. After this study, we used the lessons learned to develop a refactoring tool that can handle the good cases automatically, as we discuss in Section 5.4.
- *The Bad Parts.* Cases that require manual and ad-hoc migration. Essentially, these cases are associated with semantic conflicts between the structures used to emulate classes in ES5 and the new constructs for implementing classes in ES6. For example, function declarations in ES5 are hoisted (i.e., they can be used before the point at which they are declared in the source code), whereas ES6 class declarations are not.
- *The Ugly Parts.* Cases that cannot be migrated due to limitations and restrictions of ES6 (*e.g.*, lack of support to static fields). For this reason, in such cases we need to keep the legacy code unchanged, exposing the prototype mechanism of ES5 in the migrated code, which in our view results in "ugly code". As a result, developers are not shielded from manipulating prototypes.

In the following sections, we detail the migration results according to the proposed classification.

5.2.1 The Good Parts

As mentioned, the "good parts" are the ones handled by the rules presented in Section 5.1.1. To measure the amount of source code converted by the proposed algorithm we use the following churn metrics [Nagappan and Ball, 2005]: (a) `Churned LOC` is the sum of the added and changed lines of code between the original and the migrated versions, (b) `Deleted LOC` is the number of lines of code deleted between the original and the migrated version, (c) `Files churned` is the number of source code files that churned. We also use a set of relative churn measures as follows: `Churned LOC` / `Total LOC`, `Deleted LOC` / `Total LOC`, `Files churned` / `File count`, and `Churned LOC` / `Deleted LOC`. This last measure quantifies new development. Churned and deleted LOC are computed by GitHub. `Total LOC` is computed on the migrated code.

Table 5.2 presents the measures for the proposed code churn metrics. PIXI.JS is the system with the greatest number of classes and methods, 83 and 518 respectively. It also has the greatest absolute churned and deleted LOC, 8,879 and 8,805 lines of code, respectively. The smallest systems in terms of number of classes and methods are FASTCLICK and GRUNT. For this reason, they have the lowest values for absolute churned measures. Regarding the relative churn metrics, PARALLAX and SOCKET.IO are the systems with the greatest values for class density, 0.95 each, and they have the highest relative churned measures. PARALLAX has relative churned equals 0.76 and relative deleted equals 0.75. SOCKET.IO has relative churned equals 0.77 and relative deleted equals 0.75. Finally, the values of `Churned` / `Deleted` are approximately equal one in all systems, indicating that the impact in the size of the systems was minimum. In fact, the proposed migration algorithm is responsible for many "move" operations to place the functions (methods and class constructors) in the body of the newly created class structures.

In summary, the relative measures to migrate to ES6 range from 0.16 to 0.77 for churned code, from 0.15 to 0.75 for deleted code, and from 0.21 to 1.11 for churned files. Essentially, these measures correlate with the class density of the legacy systems.

5.2.2 The Bad Parts

As detailed in the beginning of this section, the "bad parts" are cases not handled by the proposed migration rules. To make the migration possible, these cases require

Table 5.2: Churned metric measures

System	Absolute Churn Measures			Relative Churn Measures			Churned /
	Churned	Deleted	Files	Churned	Deleted	Files	Deleted
FASTCLICK	635	630	1	0.75	0.74	1.00	1.01
GRUNT	296	291	1	0.16	0.15	0.09	1.02
SLICK	2,013	1,987	1	0.69	0.68	1.00	1.01
PARALLAX	772	764	2	0.76	0.75	0.67	1.01
SOCKET.IO	1,090	1,053	4	0.77	0.75	1.00	1.04
ISOMER	701	678	10	0.71	0.68	1.11	1.03
ALGORITHMS.JS	1,379	1,327	15	0.31	0.30	0.21	1.04
PIXI.JS	8,879	8,805	82	0.37	0.37	0.61	1.01

manual adjustments in the source code. We found four types of "bad cases" in our study, which are described next.

Accessing `this` *before* `super`. To illustrate this case, Listing 5.1 shows the emulation of class `PriorityQueue` which inherits from `MinHeap`, in ALGORITHMS.JS. In this example, lines 8-9 implement a call to the super class constructor using a function as argument. This function makes direct references to `this` (line 9). However, in ES6, these references yield an error because `super` calls must proceed any reference to `this`. The rationale is to ensure that variables defined in a superclass are initialized before initializing variables of the current class. Other languages, such as Java, have the same policy regarding class constructors.

```
 1 // Legacy code
 2 function MinHeap(compareFn) {
 3   this._comparator = compareFn;
 4   ...
 5 }
 6
 7 function PriorityQueue() {
 8   MinHeap.call(this, function(a, b) {
 9     return this.priority(a) < this.priority(b) ? -1 : 1;
10   });
11   ...
12 }
13
14 PriorityQueue.prototype = new MinHeap();
```

Listing 5.1: Passing `this` as argument to super class constructor

Listing 5.2 presents the solution adopted to migrate the code in Listing 5.1. First, we create a *setter* method to define the value of the `_comparator` property (lines 4-6).

Then, in the constructor of `PriorityQueue` we first call `super`() (line 11) and then we call the created *setter* method (lines 12-15). In this way, we guarantee that `super`() is used before `this`.

```
 1 // Migrated code
 2 class MinHeap {
 3   ...
 4   setComparator(compareFn) {
 5     this._comparator = compareFn;
 6   }
 7 }
 8
 9 class PriorityQueue extends MinHeap {
10   constructor() {
11     super();
12     this.setComparator(
13       (function(a, b) {
14         return this.priority(a) < this.priority(b) ? -1 : 1;
15       }).bind(this));
16     ...
17   }
18 }
```

Listing 5.2: By creating a setter method (lines 4-6) we guarantee that `super` is called before using `this` in the migrated code

We found three instances of classes accessing *this* before *super* in our study, two instances in ALGORITHMS.JS and one in PIXI.JS.

Calling class constructors without `new`. This pattern is also known as "factory method" in the literature [Fowler and Beck, 1999]. To illustrate this second "bad case" faced in the migration to ES6, Listing 5.3 shows part of a `Server` class implementation in SOCKET.IO. In this example, the conditional statement (line 3) verifies if `this` is an instance of `Server`, returning a `new Server` otherwise (line 4). This implementation allows calling `Server` with or without creating an instance first, as illustrated in Listing 5.4. However, ES6 does not allow class invocations as the one presented in line 2 of Listing 5.4.

```
1 // Legacy code
2 function Server(srv, opts){
3   if (!(this instanceof Server))
4     return new Server(srv, opts);
5   ...
6 }
```

Listing 5.3: Constructor of class `Server` in system SOCKET.IO

```
// Legacy code
var io = Server();
 // or
var io = new Server();
```

Listing 5.4: Two class instantiation idioms in SOCKET.IO

Listing 5.5 shows the solution adopted in this case. We first renamed class `Server` to `_Server` (line 2). Then we changed the function `Server` from the legacy code to return an instance of this new type (line 10). This solution does not have any impact in client systems, i.e., it is able to handle both class instantiation idioms presented in Listing 5.4.

```
 1 // Migrated code
 2 class _Server{
 3   constructor(srv, opts){
 4     ...
 5   }
 6 }
 7
 8 function Server(srv, opts){
 9   if (!(this instanceof _Server))
10     return new _Server(srv, opts);
11 }
```

Listing 5.5: Workaround to allow calling `Server` with or without `new`

In our study, we found one instance of class that allows calling its constructor method without using *new*, in SOCKET.IO.

Hoisting. In programming languages, hoisting denotes the possibility of referencing a variable anywhere in the code, even before its declaration. In ES5, legacy function declarations are hoisted, whereas ES6 class declarations are not.[2] As a result, in ES6 we first need to declare a class before making reference to it. As an example, Listing 5.6 shows the implementation of class `Namespace` in SOCKET.IO. `Namespace` is assigned to `module.exports` (line 2) before its constructor is declared (line 3). Therefore, in the migrated code we needed to change the order of these declarations.

Listing 5.7 shows another example of a hoisting problem, this time in PIXI.JS. In this example, a global variable receives an instance of the class `DisplayObject` (line 2) before the class definition. However, in this case the variable `_tempDisplayObjectParent` is also used by the class `DisplayObject` (line 7).

[2] `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes`

```
// Legacy code
module.exports = Namespace;
function Namespace {...}  // constructor function
```

Listing 5.6: Function `Namespace` is referenced before its definition

Furthermore, PIXI.JS uses a lint-like static checker, called ESLint[3], that prevents the use of variables before their definitions. For this reason, we cannot just reorder the statements to solve the problem, as in Listing 5.6.

```
// Legacy code
var _tempDisplayObjectParent = new DisplayObject();

DisplayObject.prototype.getBounds = function(..)
{
  ...
  this.parent = _tempDisplayObjectParent;
}
```

Listing 5.7: Hoisting problem in PIXI.JS

Listing 5.8 shows the adopted solution in this case. First, we assigned `null` to `_tempDisplayObjectParent` (line 2), but keeping its definition before the implementation of class `DisplayObject` (lines 4-6). Then we assign the original value, which makes reference to `DisplayObject`, after the class declaration (line 7).

```
1 // Migrated code
2 var _tempDisplayObjectParent = null;
3
4 class DisplayObject {
5   ...
6 }
7 _tempDisplayObjectParent = new DisplayObject();
```

Listing 5.8: Solution for hoisting problem in PIXI.JS

We found 88 instances of hoisting problems in our study, distributed over three instances in ALGORITHMS.JS, four instances in SOCKET.IO, one instance in GRUNT, and 80 instances in PIXI.JS.

Alias for method names. Legacy JavaScript code can declare two or more methods pointing to the same function. This usually happens when developers want to rename a method without breaking the code of clients. The old name is kept for the sake of

[3]`http://eslint.org/`

compatibility. Listing 5.9 shows an example of alias in Slick. In this case, Slick clients can use `addSlide` or `slickAdd` to perform the same task.

```
// Legacy code
Slick.prototype.addSlide =
  Slick.prototype.slickAdd =
    function(markup, index, addBefore) { ... };
```

Listing 5.9: Two prototype properties sharing the same function

Since we do not have a specific syntax to declare method alias in ES6, the solution we adopted was to create two methods and to make one delegate the call to the other one that implements the feature, as presented in Listing 5.10. In this example, `addSlide` (lines 7-9) just delegates any calls to `slickAdd` (line 4).

```
 1 // Migrated code
 2 class Slick {
 3   ...
 4   slickAdd(markup, index, addBefore) { ... }
 5
 6   // Method alias
 7   addSlide(markup, index, addBefore) {
 8     return slickAdd(markup, index, addBefore);
 9   }
10 }
```

Listing 5.10: Adopted solution for method alias in Slick

We found 39 instances of method alias in our study, distributed over 25 instances in Slick (confined in one class), 8 instances in Socket.io (spread over three classes), and 6 instances in Pixi.js (spread over six classes).

5.2.3 The Ugly Parts

The "ugly parts" are the ones that make use of features not supported by ES6. To make the migration possible, these cases remain untouched in the legacy code.

Getters and setters only known at runtime (meta-programming). In the ES5 implementation supported by Mozilla, there are two prototype features, `__defineGetter__` and `__defineSetter__`, that allow binding an object's property to functions that work as *getters* and *setters*, respectively.[4] Listing 5.11 shows an example in Socket.io.

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_getters_and_setters`

In this code, the first argument passed to `__defineGetter__` (line 2) is the name of the property and the second one (lines 3-5) is the function that will work as *getter* to this property.

```
1 // Legacy code
2 Socket.prototype.__defineGetter__('request',
3   function(){
4     return this.conn.request;
5   }
6 );
```

Listing 5.11: *Getter* definition in socket.io using `__defineGetter__`

By contrast, ES6 provides specific syntax to implement *getters* and *setters* within the body of the class structure. Listing 5.12 presents the ES6 version of the example shown in Listing 5.11. Declarations of *setters* follow the same pattern.

```
// Migrated code
class Socket {
  get request() {
    return this.conn.request;
  }
  ...
}
```

Listing 5.12: *Getter* method in ES6

However, during the migration of a *getter* or *setter*, if the property's name is not known at compile time (*e.g.*, if it is denoted by a variable), we cannot migrate it to ES6. Listing 5.13 shows an example from socket.io. In this case, a new *getter* is created for each string stored in an array called `flags`. Since the string values are only known at runtime, this implementation was left unchanged in our migration.

```
// Legacy code
flags.forEach(function(flag){
  Socket.prototype.__defineGetter__(flag,
    function(){
      ...
    });
});
```

Listing 5.13: *Getter* methods only known in execution time

We found five instances of *getters* and *setters* defined for properties only known at runtime, all in socket.io.

Static data properties. In ES5, usually developers use prototypes to implement static properties, i.e., properties shared by all objects from a class. Listing 5.14 shows two examples of static properties, `ww` and `orientationStatus`, that are bound to the prototype of the class `Parallax`. By contrast, ES6 classes do not have specific syntax for static properties. Because of that, we adopted an "ugly" solution leaving code defining static properties unchanged in our migration.

```
// Prototype properties (legacy code)
Parallax.prototype.ww = null;
Parallax.prototype.orientationStatus = 0;
```

Listing 5.14: Static properties defined over the prototype in Parallax

We found 42 instances of static properties in our study, 28 in PARALLAX and 14 in PIXI.JS.

Optional features. Among the meta-programming functionalities supported by ES5, we found classes providing optional features by implementing them in separated modules [Apel and Kästner, 2009]. Listing 5.15 shows a feature in PIXI.JS that is implemented in a module different than the one where the object's constructor function is defined. In this example, the class `Container` is defined in the module `core`, which is imported by using the function `require` (line 2). Therefore, `getChildByName` (lines 4-5) is an optional feature that is only incorporated to the system's core when the module implemented in Listing 5.15 is used.

```
1 // Legacy code
2 var core = require('../core');
3
4 core.Container.prototype.getChildByName =
5   function (name) { ... };
```

Listing 5.15: Method `getChildByName` is an optional feature in class `Container`

In our study, the mandatory features implemented in module `core` were properly migrated, but `core`'s optional features remained in the legacy code. Moving these features to `core` would make them mandatory in the system.

We found six instances of classes with optional features in our study, all in PIXI.JS.

5.3 Threats to Validity

This section presents threats to validity according to the guidelines proposed by Wohlin et al. [2012]. These threats are organized in external, internal, and construct validity.

External Validity. We studied eight open-source JavaScript systems. For this reason, our collection of "bad" and "ugly" cases might not represent all possible cases that require manual intervention or that cannot be migrated to the new syntax of ES6. If other systems are considered, this first catalogue of bad and ugly cases can increase.

Internal Validity. It is possible that we changed the semantics of the systems after the migration to ES6. However, we tackled this threat with two procedures. First, all systems in our dataset include a large number of tests. We assure that all the tests also pass in the ES6 code. Second, we submitted our changes to the system's developers (Section 5.5). They have not appointed changes in behavior in our code.

Construct Validity. The classes emulated in the legacy code were detected by an in-house tool, called JSClassFinder [Silva et al., 2015b,a]. Therefore, it is possible that JSClassFinder wrongly identifies some structures as classes (false positives) or that it misses some classes in the legacy code (false negatives). However, the developers who analyzed our migrated code did not complain about such problems.

5.4 Tool Support

Based on the lessons learned in our study to migrate JavaScript legacy code to use ES6 classes (Section 5.2), we propose a tool, named JSClassRefactor, to handle the migration of the good cases and to report occurrences of bad and ugly cases. First, we describe design and implementation details (Section 5.4.1) and then we present an evaluation of the proposed tool (Section 5.4.2).

5.4.1 Design and Implementation

Figure 5.2 presents an overview of the proposed migration tool. Given a legacy JavaScript application, we first use JSClassFinder to identify the class-like structures in the source code. This information is exported to a text file in CSV format, including, for each class: name, methods, parent class name, and LOC information ("start" and "end" line numbers) on every function that is either a class constructor or a method. The CSV file also contains information about bad and ugly cases, including the number of constructor calls without using `new`, aliases for methods, dynamic and static data properties, optional features, and if there is any occurrence of `this` used before `super` in the class. All this data serves as input for the module JSClassRefactor, in a second step. This module is responsible for producing a new version of the JavaScript application with the good cases migrated to the ES6 syntax for classes, and a list of the identified bad and ugly cases.

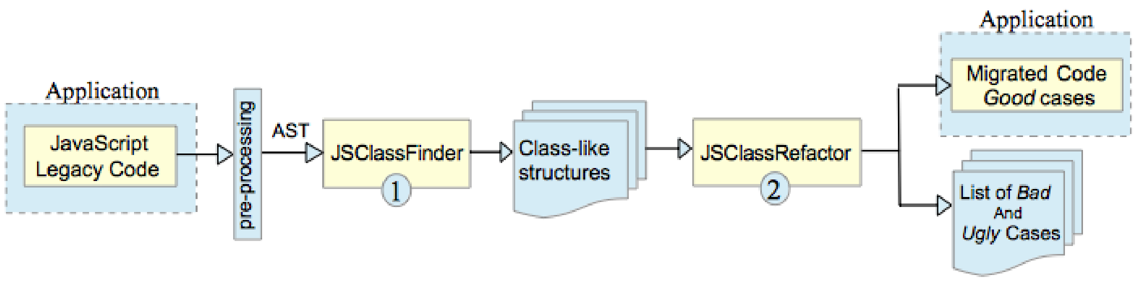


Figure 5.2: Migration Tool

As an example of the class-like structures used by JSClassRefactor as input, Table 5.3 presents part of the CSV file for class `TextStyle` in system pixi.js. The first seven columns (from `filePath` to `parent`) contain structural information necessary to migrate "the good parts" of an application. Start and end line numbers are used to locate a function in a file. We could also use start and end column numbers, if necessary, to be more precise. The column `type` indicates whether the function corresponds to a class constructor or method. The column `parent` is empty in this example because class `TextStyle` does not have a parent class in the application. The other columns (after `parent`) are specific to identify bad and ugly cases. In this example, these columns indicate that there is no call without `new` to the constructor function, that the methods do not have aliases, and that the class has one static data property.

Table 5.3: Part of the CSV file exported by JSClassFinder for class `TextStyle` in pixi.js.

filePath	class	function	start line	end line	type	parent	calls without new	aliases	static data prop.	...
../text/TextStyle	TextStyle	TextStyle	40	44	constr.		0		1	
../text/TextStyle	TextStyle	clone	83	91	method			0		
../text/TextStyle	TextStyle	reset	96	99	method			0		

5.4.2 Evaluation

We evaluated the JSClassRefactor tool using the same dataset of our study, presented in Section 5.1.2. The goal was to migrate the good cases automatically before

```
 1  /**
 2  * Binary Search Tree
 3  */
 4  function BST(compareFn) {
 5    this.root = null;
 6    this._size = 0;
 7    /**
 8    * @var Comparator
 9    */
10    this._comparator = new
         Comparator(compareFn);
11    ...
12  }
```

⟹

after

```
 1  class BST {
 2    constructor(compareFn) {
 3      this.root = null;
 4      this._size = 0;
 5      /**
 6      * @var Comparator
 7      */
 8      this._comparator = new
           Comparator(compareFn);
 9      ...
10    }
11  }
```

Figure 5.3: Migration of constructor function with comments in ALGORITHMS.JS

and to show the tool's accuracy on identifying the instances of bad and ugly cases. We summarize the results as follows, for the good parts and for each bad and ugly case.

The good parts. We implemented the migration rules defined in Section 5.1.1 to handle the good parts producing a new version of a given application, using the ES6 syntax for classes. We could verify a single limitation regarding comments written outside a function's body in the source code. These comments could not be migrated together with their respective functions. As an example, Figure 5.3 shows the migration of a constructor function in system ALGORITHMS.JS, with comments before (lines 1-3; left side) and inside (lines 7-9; left side) the function body. In this case, the comments inside the function are properly migrated to the new class structure (lines 5-7; right side), but the tool is not able to link the comments that come before or after the function declaration to the code to be migrated.

The bad parts. All instances pointed out in our study for "Accessing `this` before `super`" and "Alias for method names" were also identified by the tool. We highlight that this identification aims only to notify the user/developer, without performing any changes. The results for the other bad cases are reported next.

Calling class constructors without `new`*.* These cases can only be identified by JSClassRefactor if the call is made in the same module where the class is defined. Figure 5.4 shows an example of constructor call without `new` that could not be identified in SOCKET.IO. In this example, the constructor function for class `Server` (code on the left) is implemented in the `index.js` file, while there is a call to this constructor function in another file (`client.js`; code on the right). JSClassRefactor is not able to determine that the variable `Server`, in `client.js`, represents the class `Server`, in index.js

```
function Server() {
  ...
}
```

client.js

```
var Server = require('../index.js');
...
var io = Server();
```

Figure 5.4: Constructor declaration and constructor call in different modules

`index.js`. This is because the call to `require` (`client.js`) can return any object type. Therefore, it is not trivial to infer this kind of return type.

Hoisting. In its current version, JSCLASSREFACTOR is not able to determine if there is any reference to a class before its declaration (hoisting). This limitation exists because, when analyzing the AST of a JavaScript application, JSCLASSFINDER ignores values assigned to global variables. For example, the assignment in the second line of Listing 5.17 is ignored during analysis, and, as we pointed out in Section 5.2.2, this kind of assignment before the class definition breaks the hoisting condition.

```
1 // Legacy code
2 module.exports = SpriteMaskFilter;
3
4 function SpriteMaskFilter(sprite) {
5   ...
6 }
```

Listing 5.17: The value assigned to `module.exports` is ignored during the analysis of the AST

The ugly parts. All instances pointed out in our study for "*Getters* and *setters* only known at runtime (meta-programming)" and "*Static data properties*" were also identified by the tool. This identification, like happens with the bad cases, aims only to notify the user/developer, without performing any transformation. The results for the identification of "*Optional features*" are reported next.

Optional features. The proposed tool could identify one class with optional features, out of six detected in the manual study. Listing 5.18 shows an example of optional feature in PIXI.JS that could not be identified. In this case, a new method called `getGlobalPosition` is added to a class (`Container`), which is implemented in another module (`core`). The main problem with the identification of optional features is exactly the fact that they are implemented in other modules than the one where the class constructor is located. However, this case is slightly different from the one presented in Figure 5.4 for calling class constructors without `new`. Here we can expect that the variable represents a class because a new method is added to it, differently from a simple function call in another module. Therefore, JSClassRefactor can look for the specific class definition based on the left side of the assignment (line 4). Even though, the optional feature in listing 5.18 could not be identified because JSClassFinder could not link the class `Container` with the reference `core.Container` in line 4. Since JavaScript allows composite names for functions, `core.Container` can make reference to a function named `core.Container` or to a function named `Container` inside a module `core`. JSClassFinder lacks support for this kind of class reference in modular composite names.

```
1  // Import statement to use the module 'core'
2  var core = require('../core');
3
4  core.Container.prototype.getGlobalPosition =
5    function (point) { ... };
```

Listing 5.18: Optional feature that could not be detected

Listing 5.19 shows an optional feature for the class `DisplayObject`, in pixi.js, that could be identified by JSClassRefactor. The identification was possible due to the declaration of the variable `DisplayObject` (line 2) that eliminates the use of a composite name in the class reference (line 4). Even in this case, the identification can be considered "fragile" because the name of the variable may not match the name of the class (*e.g.*, the variable could be named "`Display`" or "`DisplayObj`")[5], and this would prevent the proposed tools from linking the variable to the correct class.

```
1 var core = require('../core'),
2 DisplayObject = core.DisplayObject;
3
4 DisplayObject.prototype._renderCachedWebGL =
5   function (renderer) { ... }
```

Listing 5.19: Optional feature detected by JSClassRefactor

5.5 Feedback from Developers

After migrating the code and handling the bad parts, we take to the JavaScript developers the discussion about accepting the new version of their systems in ES6. For

[5]In our experiments, we can see that developers usually declare this kind of variable using the same name of the class, but this is not mandatory in JavaScript.

every system, we create a pull request (PR) with the migrated code, suggesting the adoption of ES6 classes. Table 5.4 details these pull requests presenting their ID on GitHub, the number of comments they triggered, the opening date, and their status on the date when the data was collected (October 12th, 2016).

Table 5.4: Created Pull Requests.

System	ID	# Comments	Opening Date	Status on 12-Oct-16
FASTCLICK	#500	0	01-Sep-16	Open
GRUNT	#1549	2	31-Aug-16	Closed
SLICK	#2494	5	25-Aug-16	Open
PARALLAX	#159	1	01-Sep-16	Open
SOCKET.IO	#2661	4	29-Aug-16	Open
ISOMER	#87	3	05-Sep-16	Closed
ALGORITHMS.JS	#117	4	23-Aug-16	Open
PIXI.JS	#2936	14	09-Sep-16	Merged

Five PRs (62%) are still open. The PR for FASTCLICK has no comments. This repository seems to be sparsely maintained, since its last commit dates from April, 2016. The comments in the PRs for SLICK, SOCKET.IO, and PARALLAX suggest that they are still under evaluation by the developer's team. In the case of ALGORITHMS.JS, the developer is in favor of ES6 classes, although he believes that it is necessary to transpile the migrated code to ES5 for the sake of compatibility.[6] However, he does not want the project to depend on a transpiler, such as `Babel`[7], as stated in the following comment:

"I really like classes and I'm happy with your change. Even though most modern browsers support classes, it would be nice to transpile to ES5 to secure compatibility. And I'm not sure it would be good to add Babel as a dependency to this package. So for now I think we should keep this PR on hold for a little while..." (Developer of system ALGORITHMS.JS*)*

We have two closed PRs whose changes were not merged. The developer of GRUNT chose not to integrate the migrated code because the system has to keep compatibility with older versions of `node.js`[8] that do not support ES6 syntax, as stated in the following comment:

"We currently support node 0.10 that does not support this syntax. Once we are able to drop node 0.10 we might revisit this." (Developer of system GRUNT*)*

[6]A transpiler is a source-to-source compiler. Transpilers are used, for example, to convert back from ES6 to ES5, in order to guarantee compatibility with older browsers and runtime tools, like `node.js`.

[7]`https://babeljs.io/`

[8]`https://nodejs.org`

In the case of ISOMER, the developers decided to keep their code according to ES5, because they are not enthusiasts of the new class syntax in ES6:

"IMHO the class syntax is misleading, as JS "classes" are not actually classes. Using prototypal patterns seems like a simpler way to do inheritance." (Developer of system ISOMER*)*

The PR for system PIXI.JS was the largest one, with 82 churned files, and all the proposed changes were promptly accepted, as described in this comment:

"Awesome work! It is really great timing because we were planning on doing this very soon anyways."

The developers also mentioned the need to use a transpiler to keep compatibility with other applications that do not support ES6 yet, and they chose to use `Babel` for transpiling, as stated in the following comments:

"Include the babel-preset-es2015 module in the package.json devDependencies."... "Unfortunately, heavier dev dependencies are the cost right now for creating more maintainable code that's transpiled. Babel is pretty big and other tech like TypeScript, Coffeescript, Haxe, etc have tradeoffs too." (Developer of system PIXI.JS*)*

Finally, the PIXI.JS developers also discussed the adoption of other ES6 features, *e.g.*, using arrow functions expressions and declaring variables with `let` and `const`, as stated in the following comment:

"I think it makes more sense for us to make a new Dev branch and start working on this conversion there (starting by merging this PR). I'd like to make additional passes on this for const/let usage, fat arrows instead of binds, statics and other ES6 features." (Developer of system PIXI.JS*)*

5.6 Final Remarks

In this chapter, we report a study on migrating structures that emulate classes in legacy JavaScript code to adopt the new syntax for classes introduced by ES6. We present a set of migration rules based on the most frequent use of class emulations in ES5. We then convert eight legacy JavaScript systems to use ES6 classes. In our study, we detail cases that are straightforward to migrate (the good parts), cases that require manual and ad-hoc migration (the bad parts), and cases that cannot be migrated due to limitations and restrictions of ES6 (the ugly parts). This study indicates that the migration rules are sound but incomplete, since most of the studied systems (75%)

contain instances of bad and/or ugly cases. We also used the lessons learned during this study to propose a tool that automatically handles the good parts and exposes the bad and ugly parts. Finally, we collect the perceptions of JavaScript developers about migrating their code to use the new syntax for classes.

Our findings suggest that (a) proposals to automatically translate from ES5 to ES6 classes can be challenging and risky; (b) developers tend to move to ES6, but compatibility issues are making them postpone their decisions; (c) developers opinions diverge about the use of transpilers to keep compatibility with ES5; (d) there are demands for new class-related features, which can be introduced in future JavaScript versions, such as static fields, method deprecation, and partial classes (which can help in the implementation of optional features). In fact, we also found a proposal to include "class static fields" in ECMAScript.[9]

We collected evidences that developers plan to move to ES6 in the near future. Developers from system PIXI.JS indeed accepted our PR; developers from six other systems manifested interest in migrating their systems, but only postponed the decision due to compatibility concerns. Only one developer showed a critical view about classes, and manifested his preference for the original prototype mechanism. We conjecture that these concerns tend to disappear in the near future, specially because tools in the JavaScript ecosystem evolve very rapidly. Moreover, there are two other reasons that help to explain the low acceptance rate of the PRs. First, it is known that PRs that change large portions of the code base at a time have less changes of being understood and evaluated [Tsay et al., 2014]. Second, it is natural that changes submitted by someone outside the community take longer to be considered by project managers.

[9]`https://github.com/tc39/proposal-class-public-fields`

Chapter 6

Conclusion

In this chapter, we present our closing points and arguments. We summarize the outcome of this thesis (Section 6.1) and review our main contributions (Section 6.2). Finally, we outline possible ideas for future work (Section 6.3).

6.1 Summary

JavaScript is the most popular programming language for Web applications. The language is also gaining popularity in the server-side space since the release of `Node.js`. Due to the increasing usage of JavaScript, there is a great demand to write JavaScript code that is reliable and maintainable. Despite the effort in state-of-the-art research regarding software comprehension and analysis of legacy JavaScript systems, there is no solution that covers the identification of class-like structures that include attributes, methods, inheritance relationships, and coupling information between these entities. Moreover, most JavaScript applications can be considered as legacy code, *i.e.*, applications implemented according to versions prior to ECMAScript 6 (ES6), without syntactic support for classes. Considering this context, in this thesis, we define, implement, and evaluate: (i) a set of heuristics to identify class-like structures, and their dependencies, in legacy JavaScript code; (ii) a set of rules to migrate class-like structures from ES5 to ES6.

After the study reported in Chapter 3, we concluded that JavaScript developers often emulate classes in legacy code to tackle the complexity of their systems. We proposed a strategy to statically detect class emulation in legacy JavaScript code and the JSClassFinder tool, that supports this strategy. We used JSClassFinder on a corpus of 918 popular JavaScript applications, with different sizes and from multiple domains, in order to document the usage of class-like structures in legacy JavaScript systems. We also performed a field study with 60 JavaScript developers to evaluate the accuracy of our strategy and tool. We summarize our findings in this chapter as follows.

- There are essentially four types of JavaScript software regarding the usage of classes: *class-free* (systems that do not make any usage of classes), *class-aware* (systems that use classes marginally), *class-friendly* (systems that make relevant usage of classes), and *class-oriented* (systems that have the vast majority of their data structures implemented as classes). These categories represent, respectively, 32%, 34%, 27%, and 7% of the systems we studied. Precision, recall and F-score measures indicated that JSClassFinder is able to identify the classes, methods, and attributes in JavaScript systems. The overall results range from 97% to 100% for precision, from 70% to 89% for recall, and from 82% to 94% for F-score.

- We found that there is no significant relation between size and class usage. Therefore, we cannot conclude that the larger the system, the greater the usage of classes, at least in proportional terms. For this reason, we hypothesize that the background and experience of the systems' developers have more impact on the decision to design a system around classes, than its size.

- Prototype-based inheritance is not popular in JavaScript. We counted only 70 out of 918 systems (8%) using inheritance. We hypothesize that there are two main reasons for this: (i) even in class-based languages there are strong positions against inheritance, and a common recommendation is to "favor object composition over class inheritance" [Gamma et al., 1994, Stefanov, 2010]; (ii) prototype-based inheritance is more complex than the usual implementation of inheritance available in mainstream class-based object-oriented languages.

- Classes in JavaScript have usually less than 28 attributes and 61 methods (90th percentile measures). It is also common to have data-oriented classes, *i.e.*, classes with more attributes than methods. In half of the studied systems, we had at least 39% of such classes.

- 58% of JavaScript developers that answered our field study intended to use the ES6 new syntax for classes, but usually only for new features and projects. We acknowledge that ES6 specification has, besides the class support, other features the demand some changes in the source code, (*e.g.*, arrows, iterators, template strings, and enhanced object literals). It is understandable that developers take them all in consideration and plan carefully future refactorings to minimize effort.

In Chapter 4, we evaluated an approach that uses type inference to identify class-to-class dependencies and associations in legacy JavaScript code. We reported a study on two open-source projects whose code is transpiled from TypeScript to JavaScript. We could not find any class-to-class dependencies when Facebook's Flow (the tool considered in this chapter) is used in its default configuration. However, after pre-processing the source code in the `require` statements to eliminate variable references to class constructor functions implemented in other modules, our results showed that precision reaches 100%, and recall ranges from 80% to 86% for all dependencies and from 85% to 96% for associations, in the two evaluated systems. Finally, we manually analyzed and categorized all false negatives identified in our study. The results suggest that a tool like Facebook's Flow can be used to support the implementation of reverse engineering tools for JavaScript.

Finally, in Chapter 5, we reported a study on replacing structures that emulate classes in legacy JavaScript code by native structures introduced by ES6, which can contribute to foster software reuse. We presented a set of migration rules based on the most frequent use of class emulations in ES5, and the JSCLASSREFACTOR tool, that implements these rules. We converted eight legacy JavaScript systems to use ES6 classes. In our study, we detailed cases that are straightforward to migrate (the good parts), cases that require manual and ad-hoc migration (the bad parts), and cases that cannot be migrated due to limitations and restrictions of ES6 (the ugly parts). This study indicated that 75% of the studied systems contain instances of bad and/or ugly cases. We also collected the perceptions of JavaScript developers about migrating their code to use the new syntax for classes. Our findings suggest that: (a) proposals to automatically translate from ES5 to ES6 classes can be challenging and risky; (b) developers tend to move to ES6, but compatibility issues are making them postpone these decisions; (c) developer opinions diverge about the use of transpilers to keep compatibility with ES5; (d) there are demands for new class-related features in JavaScript, such as static fields, method deprecation, and partial classes.

6.2 Contributions

We summarize our contributions as follows:

- We proposed, implemented, and evaluated **a set of heuristics to identify classes in JavaScript legacy code.** We provided a thorough study on the usage of classes in a dataset of 918 JavaScript systems available on GitHub. This study showed that: (i) JavaScript developers often emulate classes in legacy code to tackle the complexity of their systems; (ii) there is no significant relation between size and class usage; (iii) prototype-based inheritance is not popular in JavaScript; (iv) it is common to have JavaScript classes with more attributes than methods; (v) JavaScript developers that emulate classes tend to use the ES6 new syntax, mostly for new features and projects.

- We developed **an open-source supporting tool**, called JSClassFinder [Silva et al., 2015a], that practitioners can use **to detect and inspect classes in legacy JavaScript code**.

- We performed **an evaluation on the usage of Facebook's Flow to infer class-to-class dependencies in legacy JavaScript code**. We performed a study with two open-source applications whose source code is transpiled from TypeScript to JavaScript. We showed that Flow can be used to support the implementation of reverse engineering tools for JavaScript.

- We presented **a set of rules to migrate class-like structures from ES5 to ES6**. We described the proposed rules and their limitations, *i.e.*, a set of cases where manual adjusts are required to migrate the code. We also identified some limitations of the new syntax for classes provided by ES6, *i.e.*, the cases where it is not possible to migrate the code and, therefore, we should expose the prototype-based object system to ES6 maintainers.

- We implemented **an open-source supporting tool**, called JSClassRefactor[1], that can be used **to migrate legacy code to use the new support for classes that comes with ES6**.

- We document **a set of reasons that can lead developers to postpone or reject the adoption of ES6 classes**. These reasons are based on the feedback received after submitting pull requests to JavaScript developers suggesting the migration of their systems.

6.3 Future Work

Future work includes the extension of our heuristics for class identification in order to: (a) measure other class properties, like coupling, cohesion, and complexity; (b) identify bad smells in JavaScript classes; (c) recommend best object-oriented programming

[1]`https://github.com/leonardo-silva/JSClassRefactor`

practices for JavaScript; (d) detect violations and deviations in the class-based architecture of JavaScript systems. It is also possible to extend JSCLASSFINDER to support the computation of metric thresholds for JavaScript class-like structures [Oliveira et al., 2014]. These thresholds could be used, for example, to spot class-level anomalies, such as *god class*, *feature envy*, *refused bequest*, and *lazy class* [Marinescu, 2004].

Regarding the identification of class-to-class dependencies, future work may include the investigation of systems whose native language is JavaScript. In Chapter 4, we reported a study on two open-source projects whose code is transpiled from TypeScript to legacy JavaScript. If other systems are considered, we can identify other instances of missing dependencies (false negatives). Moreover, the performance of Flow when analyzing systems that have been migrated to ES6 class syntax can also be evaluated. It is also possible to investigate other approaches to improve recall by combining Flow (or other static analysis tool) with dynamic analysis. We can instrument JavaScript code to record execution traces using, for example, tools like Aran [Christophe et al., 2015].

Regarding the refactoring of legacy applications to use the new syntax for classes provided by ES6, future work may include the investigation of other instances of bad and ugly cases migrating a larger set of JavaScript systems. It is also possible to extend our current refactoring tool implementation (JSCLASSREFACTOR) by improving its interface and documentation. Finally, another possibility is to work on recommendations to handle the bad cases, similar to the ones manually adopted in our study (Chapter 5).

Bibliography

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

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37, pages 321–345, 2015.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP)*, pages 428–452, 2005.

Esben Andreasen, Anders Møller, Liang Gong, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*, 1:1–36, 2017. Accepted for publication.

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Fault Localization for Dynamic Web Applications. *IEEE Transactions on Software Engineering*, 38(2):314–335, 2012.

Thoms Ball. The concept of dynamic analysis. In *7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 216–234. Springer-Verlag, 1999.

Mark Bates. *Programming in CoffeeScript.* Addison-Wesley Professional, 1st edition, 2012.

Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software. In *21st Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 397–412. ACM, 2006.

Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., 2004.

Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM Fall Joint Computer Conference*, pages 36–40. IEEE Computer Society Press, 1986.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. Poster: Dynamic Analysis Using JavaScript Proxies. In *37th International Conference on Software Engineering*, ICSE, pages 813–814. IEEE Press, 2015.

Richard C. H. Connor, Alfred L. Brown, Quintin I. Cutts, Alan Dearle, Ronald Morrison, and John Rosenberg. Type equivalence checking in persistent object systems. In *4th International Workshop on Persistent Objects (POS)*, pages 154–167, 1990.

Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

Serge Demeyer, StÃ©phane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009. ISBN 978-3-9523341-2-6.

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

ECMA-International. European Association for Standardizing Information and Communication Systems (ECMA). ECMA-262: ECMAScript Language Specification. edition 5.1. URL `https://www.ecma-international.org/ecma-262/5.1/`.

ECMA-International. European Association for Standardizing Information and Communication Systems (ECMA). ECMAScript Language Specification, 6th edition, 2015. URL `http://www.ecma-international.org/publications/standards/Ecma-402.htm`.

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

Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, and Frank Tip. Refactoring towards the good parts of JavaScript. In *26th Conference on Object-Oriented Programming (OOPSLA), Companion Proceedings*, pages 189–190, 2011a.

Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for JavaScript. In *26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 119–138, 2011b.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE Press, 2013.

David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 2011.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003.

Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.

Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 10 pages. IEEE Computer Society, 2015.

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

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *18th International Conference on World Wide Web (WWW)*, pages 561–570, 2009.

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

Brian Hackett and Shuyu Guo. Fast and precise hybrid type inference for JavaScript. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250, 2012. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254094.

Munawar Hafiz, Samir Hasan, Zachary King, and Allen Wirfs-Brock. Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving javascript features. *Journal of Systems and Software*, 121:191–208, 2016.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-57169-2.

Nilesh Jain, Priyanka Mangal, and Deepak Mehta. AngularJS: A modern MVC framework in JavaScript. *Journal of Global Research in Computer Science*, 5:17–23, 2015.

Holger M. Kienle. It's about time to take javascript (more) seriously. *IEEE Software*, 27(3):60–62, 2010.

Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer, 2006.

Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 113–122, 2009. ISBN 978-0-7695-3867-9. doi: 10.1109/WCRE.2009.11.

A. Lerner. *Ng-Book - the Complete Book on Angularjs.* Fullstack.io, 2013.

Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power Laws in Software. *ACM Transactions on Software Engineering and Methodology*, 18:1–26, 2008.

Alex MacCaw. *The Little Book on CoffeeScript.* O'Reilly Media, Inc., 2012.

Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *30th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 505–519. ACM, 2015.

Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: Semi-automated removal of eval from JavaScript programs. In *27th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620, 2012.

Nevena Milojković and Oscar Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 43–56, 2016. doi: 10.1145/2986012.2986017.

Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering (ICSE)*, pages 284–292. ACM, 2005. doi: 10.1145/1062455.1062514.

Christopher Nance. *TypeScript Essentials.* Packt Publishing, 2014. ISBN 1783985763, 9781783985760.

Alex Nederlof, Ali Mesbah, and Arie van Deursen. Software engineering for the web: the state of the practice. In *36th International Conference on Software Engineering (ICSE)*, pages 4–13, 2014.

Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *36th International Conference on Software Engineering (ICSE)*, pages 791–802, 2014.

Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting function purity in JavaScript. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015.

Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: An agile reengineering environment. In *10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 1–10. ACM, 2005.

Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *37th International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015.

Morten Passow Odgaard. JavaScript Type Inference Using Dynamic Analysis. Master's thesis, Aarhus University, 2014.

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

Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *6th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 146–161, New York, NY, USA, 1991. ACM. doi: 10.1145/117954.117965.

David L. Parnas. Designing software for ease of extension and contraction. In *3rd International Conference on Software Engineering (ICSE)*, pages 264–277. IEEE Press, 1978.

Caio Ribeiro Pereira. *Introduction to Node.js*, pages 1–3. Apress, 2016. ISBN 978-1-4842-2442-7. doi: 10.1007/978-1-4842-2442-7_1.

Thomas Powell. *Ajax: The Complete Reference*. McGraw-Hill, Inc., 1 edition, 2008.

Miguel Ramos, Marco Tulio Valente, and Ricardo Terra. AngularJS performance: A survey study. *IEEE Software*, 1(1):1–11, 2017.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *25th European Conference on Object-oriented Programming (ECOOP)*, 2011.

Shahriar Rostami, Laleh Eshkevari, Davood Mazinanian, and Nikolaos Tsantalis. Detecting function constructors in JavaScript. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME ERA)*, pages 1–5, 2016.

Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005a. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094824.

Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005b.

Leonardo Humberto Silva, Daniel Hovadick, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. JSClassFinder: A Tool to Detect Class-like Structures in JavaScript. In *6th Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Demonstration Track*, pages 113–120, 2015a.

Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. Does JavaScript software embrace classes? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 73–82, 2015b.

Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. Identifying classes in legacy JavaScript code. *Journal of Software: Evolution and Process*, 1(1):1–37, 2017.

S. Stefanov. *JavaScript Patterns*. O'Reilly Media, 2010.

Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *36th International Conference on Software Engineering (ICSE)*, pages 356–366. ACM, 2014.

David Ungar and Randall B. Smith. SELF: The power of simplicity. In *2nd Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 227–242. ACM, 1987.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

Richard Wheeldon and Steve Counsell. Power Law Distributions in Class Relationships. In *International Working Conference on Source Code Analysis and Manipulation*, pages 45–54, 2003.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012. ISBN 978-3-642-29043-5.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *34th Symposium on Principles of Programming Languages (POPL)*, pages 237–249, 2007.

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