Refactoring Legacy JavaScript Code to Use Classes: The Good, The Bad and The Ugly

Leonardo Humberto Silva[1] (0000-0003-2807-6798), Marco Tulio Valente[2] (0000-0002-8180-7548), and Alexandre Bergel[3] (0000-0001-8087-1903)

[1] Federal Institute of Northern Minas Gerais, Salinas, Brazil
`leonardo.silva@ifnmg.edu.br`
[2] Federal University of Minas Gerais, Belo Horizonte, Brazil
`mtov@dcc.ufmg.br`
[3] Pleiad Lab - DCC - University of Chile, Santiago, Chile
`abergel@dcc.uchile.cl`

Abstract. JavaScript systems are becoming increasingly complex and large. To tackle the challenges involved in implementing these systems, the language is evolving to include several constructions for programming-in-the-large. For example, although the language is prototype-based, the latest JavaScript standard, named ECMAScript 6 (ES6), provides native support for implementing classes. Even though most modern web browsers support ES6, only a very few applications use the class syntax. In this paper, we analyze the process of migrating structures that emulate classes in legacy JavaScript code to adopt the new syntax for classes introduced by ES6. We apply a set of migration rules on eight legacy JavaScript systems. In our study, we document: (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). Six out of eight systems (75%) contain instances of bad and/or ugly cases. We also collect the perceptions of JavaScript developers about migrating their code to use the new syntax for classes.

Keywords: JavaScript · Refactoring · ECMAScript 6

1 Introduction

JavaScript is the most dominant web programming language. It was initially designed in the mid-1990s to extend web pages with small executable code. Since then, its popularity and relevance only grew [1–3]. Among the top 2,500 most popular systems on GitHub, according to their number of stars, 34.2% are implemented in JavaScript [4]. To mention another example, in the last year, JavaScript repositories had twice as many pull requests (PRs) than the second language, representing an increase of 97% over the previous year.[4] The language can be used to implement both client and server-side applications. Moreover,

[4] `https://octoverse.github.com/`

JavaScript code can also be encapsulated as libraries and referred to by web pages. These characteristics make JavaScript suitable for implementing complex, single-page web systems, including mail clients, frameworks, mobile applications, and IDEs, which can reach hundreds of thousands of lines of code.

JavaScript is an imperative and object-oriented language centered on prototypes [5, 6]. Recently, the release of the new standard version of the language, named ECMAScript 6 (or just ES6, as used throughout this paper), represented a significant update to the language. Among the new features, particularly important is the syntactical support for classes [7]. With ES6, it is possible to implement classes using a syntax very similar to the one of mainstream class-based object-oriented languages, such as Java and C++. However, although most modern browsers already support ES6, there is a large codebase of legacy JavaScript source code, i.e., code implemented in versions prior to the ES6 standard. Even in this code, it is common to find structures that in practice are very similar to classes, being used to encapsulate data and code. Although not using appropriate syntax, developers frequently emulate class-like structures in legacy JavaScript applications to easily reuse code and abstract functionalities into specialized objects. In a previous study, we show that structures emulating classes are present in 74% of the studied systems [8]. We also implemented a tool, JSClassFinder [9], to detect classes in legacy JavaScript code. Moreover, a recent empirical study shows that JavaScript developers are not fully aware of changes introduced in ES6, and very few are currently using object-oriented features, such as the new class syntax [10].

In this paper, we investigate the feasibility of rejuvenating legacy JavaScript code and, therefore, to increase the chances of code reuse in the language. Specifically, we describe an experiment on migrating eight real-world JavaScript systems to use the native syntax for classes provided by ES6. We first use JSClassFinder to identify class like structures in the selected systems. Then we convert these classes to use the new syntax.

This paper makes the following contributions:

- We present a basic set of rules to migrate class-like structures from ES5 (prior version of JavaScript) to the new syntax for classes provided by ES6 (Section 3.1).
- We quantify the amount of code (churned and deleted) that can be automatically migrated by the proposed rules (the good parts, Section 4.1).
- We describe the limitations of the proposed rules, i.e., a set of cases where manual adjusts are required to migrate the code (the bad parts, Section 4.2).
- We 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 (the ugly parts, Section 4.3).
- We document a set of reasons that can lead developers to postpone/reject the adoption of ES6 classes (Section 5). These reasons are based on the feedback received after submitting pull requests suggesting the migration to the new syntax.

2 Background

2.1 Class Emulation in Legacy JavaScript Code

Using functions is the most common strategy to emulate classes in legacy JavaScript systems. Particularly, any function can be used as a template for the creation of objects. When a function is used as a class constructor, the `this` variable is bound to the new object under construction. Variables linked to `this` define properties that emulate attributes and methods. If a property is an inner function, it represents a *method*; otherwise, it is an *attribute*. The operator `new` is used to instantiate class objects.

To illustrate the emulation of classes in legacy JavaScript code, we use a simple `Queue` class. Listing 1.1 presents the function that defines this class (lines 1-8), which includes one attribute (`_elements`) and three methods (`isEmpty`, `push`, and `pop`). The implementation of a specialized queue is found in lines 9-17. `Stack` is a subclass of `Queue` (line 15). Method `push` (line 17) is overwritten to insert elements at the first position of the queue.

```
 1 // Class Queue
 2 function Queue() { // Constructor function
 3   this._elements = new LinkedList();
 4   ...
 5 }
 6 Queue.prototype.isEmpty = function() {...}
 7 Queue.prototype.push = function(e) {...}
 8 Queue.prototype.pop = function() {...}
 9 // Class Stack
10 function Stack() {
11   // Calling parent's class constructor
12   Queue.call(this);
13 }
14 // Inheritance link
15 Stack.prototype = new Queue();
16 // Overwritten method
17 Stack.prototype.push = function(e) {...}
```

Listing 1.1: *Class* emulation in legacy JavaScript code

The implementation in Listing 1.1 represents one possibility of class emulation in JavaScript. Some variations are possible, like implementing methods inside/outside class constructors and using anonymous/non-anonymous functions [8, 11].

2.2 ECMAScript 6 Classes

ES6 includes syntactical support for classes. Listing 1.2 presents an implementation for classes `Queue` and `Stack` (Listing 1.1) in this latest JavaScript standard. As can be observed, the implementation follows the syntax provided by mainstream class-based languages. We see, for example, the usage of the keywords `class` (lines 1 and 11), `constructor` (lines 2 and 12), `extends` (line 11), and `super` (line 13). Although ES6 classes provide a much simpler and clearer syntax to define classes and deal with inheritance, it is a syntactical sugar over JavaScript's existing prototype-based inheritance. In other words, the new syntax does not impact the semantics of the language, which remains prototype-based.[5]

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

```
class Queue {
  constructor() {
    this._elements = new LinkedList();
    ...
  }
  // Methods
  isEmpty() {...}
  push(e) {...}
  pop() {...}
}
class Stack extends Queue {
  constructor() {
    super();
  }
  // Overwritten method
  push(e) {...}
}
```

Listing 1.2: Class declaration using ES6 syntax

3 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 describe the rules followed to conduct this migration (Section 3.1). Then, we present the set of selected systems in our dataset (Section 3.2). The results are discussed in Section 4.

3.1 Migration Rules

Figure 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 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 [8, 11]. 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 implemented outside the class constructor but it is not bound to the prototype.[6] 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.

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

Rule #1: Classes

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; }
  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;
}
```

Fig. 1: Migration rules (p_i is a formal parameter list and B_i is a block of statements)

There are no rules for migrating fields, because they are declared with the same syntax both in ES5 and ES6 (see Listing 1.1, line 3; and Listing 1.2, line 3). 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.

3.2 Dataset

We select systems that emulate classes in legacy JavaScript code in order to migrate them to the new syntax. In a previous work [8], we conducted an empirical study on the use of classes with 50 popular JavaScript systems, before the release of ES6. In this paper, we select eight systems from the dataset used in this previous work. The selected systems have at minimum one and at maximum 100 classes, and 40 KLOC.

Table 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) [8]. JSClassFinder [9] was used to identify the classes emulated in legacy code and to compute the measures presented in Table 1. The selection includes well-known and widely used JavaScript systems, from different domains, 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 1: JavaScript systems 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

4 Migration Results

We followed the rules presented in Section 3 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 3.1. As future work, we plan to develop a refactoring tool to handle these cases.
– *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.

4.1 The Good Parts

As mentioned, the "good parts" are the ones handled by the rules presented in Section 3.1. To measure the amount of source code converted we use the following churn metrics [12]: (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 2 presents the measures for the proposed code churn metrics. PIXI.JS 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.

Table 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

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.

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

Accessing `this` *before* `super`. To illustrate this case, Listing 1.3 shows the emulation of class `PriorityQueue` which inherits from `MinHeap`, in ALGORITHMS.JS. In this example, lines 7-8 call the super class constructor using a function as argument. This function makes direct references to `this` (line 8). 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 function PriorityQueue() {
 7   MinHeap.call(this, function(a, b) {
 8     return this.priority(a) < this.priority(b) ? -1 : 1;
 9   });
10   ...
11 }
12 PriorityQueue.prototype = new MinHeap();
```

Listing 1.3: Passing `this` as argument to super class constructor

Listing 1.4 presents the solution adopted to migrate the code in Listing 1.3. 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 10) and then we call the created *setter* method (lines 11-14). 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 class PriorityQueue extends MinHeap {
 9   constructor() {
10     super();
11     this.setComparator(
12       (function(a, b) {
13         return this.priority(a) < this.priority(b) ? -1 : 1;
14       }).bind(this));
15     ...
16   }
17 }
```

Listing 1.4: 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 [17]. As an example, Listing 1.5 shows part of a `Server` class implementation in SOCKET.IO. 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. However, this class invocation without having an instance is not allowed in ES6.

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

Listing 1.5: Constructor of class `Server` in system SOCKET.IO

Listing 1.6 shows the solution we 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 7). This solution does not have any impact in client systems.

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

Listing 1.6: Workaround to allow calling `Server` with or without `new`

We found one case of calling a class constructor without *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.[7] As a result, in ES6 we first need to declare a class before making reference to it. As an example, Listing 1.7 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.

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

Listing 1.7: Function `Namespace` is referenced before its definition

Listing 1.8 shows another example of a hoisting problem, this time in PIXI.JS. In this case, a global variable receives an instance of the class `DisplayObject` (line 2) before the class definition (lines 3-6). However, in this case the variable `_tempDisplayObjectParent` is also used by the class `DisplayObject` (line 5). Furthermore, PIXI.JS uses a lint-like static checker, called ESLint[8], 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 1.7.

```
1 // Legacy code
2 var _tempDisplayObjectParent = new DisplayObject();
3 DisplayObject.prototype.getBounds = function(..) {
4   ...
5   this.parent = _tempDisplayObjectParent;
6 }
```

Listing 1.8: Hoisting problem in PIXI.JS

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

[8] http://eslint.org/

Listing 1.9 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` (line 4). Then we assign the original value, which makes reference to `DisplayObject`, after the class declaration.

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

Listing 1.9: 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 compatibility. Listing 1.10 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 1.10: 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 1.11. In this example, `addSlide` (line 6) just delegates any calls to `slickAdd` (line 4).

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

Listing 1.11: 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).

4.3 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 features, `__defineGetter__` and `__defineSetter__`, that allow binding an object's property to functions that work as *getters* and *setters*, respectively.[9] Listing 1.12 shows an example in SOCKET.IO. In this code, the first argument passed to `__defineGetter__` (line 2) is the name of the property and the second one (line 3) is the function that will work as *getter* to this property.

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

Listing 1.12: *Getter* definition in SOCKET.IO using `__defineGetter__`

ES6 provides specific syntax to implement *getters* and *setters* within the body of the class structure. Listing 1.13 presents the ES6 version of the example shown in Listing 1.12. Declarations of *setters* follow the same pattern.

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

Listing 1.13: *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 1.14 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.

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

Listing 1.14: *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 1.15 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 1.15: Static properties defined over the prototype in PARALLAX

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

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Optional features. Among the meta-programming functionalities supported by ES5, we found classes providing optional features by implementing them in separated modules [13]. Listing 1.16 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` (line 4) is a feature that is only incorporated to the system's core when the module implemented in Listing 1.16 is used.

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

Listing 1.16: 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 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 every system, we create a pull request (PR) with the migrated code, suggesting the adoption of ES6 classes. Table 3 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 3: Created Pull Requests

System	ID	#Comments	Opening Date	Status
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.[10] However, he does not want the project to depend on a transpiler, such as `Babel`[11], 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`, 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*)*

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." (Developer of PIXI.JS*)*

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 PIXI.JS*)*

Finally, 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 PIXI.JS*)*

[10] 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.

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

6 Threats to 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. However, we tackled this threat with two procedures. First, all systems in our dataset include a large number of tests. We assure that all tests also pass in the ES6 code. Second, we submitted our changes to the system’s developers. They have not pointed any changes in the behavior of their code.

Construct Validity. The classes emulated in the legacy code were detected by JSClassFinder [8,9]. 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 pull requests did not complain about such problems.

7 Related Work

In a previous work, we present a set of heuristics followed by an empirical study to analyze the prevalence of class-based structures in legacy JavaScript code [8]. The study was conducted on 50 popular JavaScript systems, all implemented according to ES5. The results indicated that class-based constructs are present in 74% of the studied systems. We also implemented a tool, JSClassFinder [9], to detect classes in legacy JavaScript code. We use this tool to statically identify class dependencies in legacy JavaScript systems [?] and also to identify the classes migrated to ES6 in this paper.

Hafiz et al. [10] present an empirical study to understand how different language features in JavaScript are used by developers. The authors conclude that: (a) developers are not fully aware about newly introduced JavaScript features; (b) developers continue to use deprecated features that are no longer recommended; (c) very few developers are currently using object-oriented features, such as the new class syntax. We believe this last finding corroborates the importance of our work to assist developers to start using ES6 classes.

Rostami et al. [14] propose a tool to detect constructor functions in legacy JavaScript systems. They first identify all object instantiations, even when there is no explicit object instantiation statement (*e.g.,* the keyword `new`), and then link each instance to its constructor function. Finally, the identified constructors represent the emulated classes and the functions that belong to these constructors (inner functions) represent the methods.

Gama et al. [11] identify five styles for implementing methods in JavaScript: inside/outside constructor functions using anonymous/non-anonymous functions and using prototypes. Their main goal is to implement an automated approach to normalize JavaScript code to a single consistent style. The migration algorithm used in this paper covers the five styles proposed by the authors. Additionally, we also migrate static methods, *getter* and *setters*, and inheritance relationships.

Feldthaus et al. [15] describe a methodology for implementing automated refactorings on a nearly complete subset of the JavaScript language. The authors specify and implement three refactorings: *rename property*, *extract module*, and *encapsulate property*. In summary, the proposed refactorings aim to transform ES5 code in code that is more maintainable. However, they do not transform the code to the new JavaScript standard.

Previous works have also investigated the migration of legacy code, implemented in procedural languages, to object-oriented code, including the transformation of C functions to C++ function templates [**?**] and the adoption of class methods in PHP [**?**].

8 Final Remarks

In this paper, we report 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 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 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) 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.

As future work, we intend to enrich our research in two directions. First, we plan to extend our study migrating a larger set of JavaScript systems. In this way, we can identify other instances of bad and ugly cases. Second, we plan to implement a refactoring tool for a JavaScript IDE. This tool should be able to semi-automatically handle the good cases, and also alert developers about possible bad and ugly cases.

Acknowledgments. This research is supported by CNPq, CAPES and Fapemig.

References

1. H. Kienle, "It's about time to take JavaScript (more) seriously," *IEEE Software*, vol. 27, no. 3, pp. 60–62, May 2010.

2. F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *22nd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2011, pp. 100–109.
3. A. Nederlof, A. Mesbah, and A. van Deursen, "Software engineering for the web: the state of the practice," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 4–13.
4. H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors that Impact the Popularity of GitHub Repositories," in *32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 1–10.
5. A. H. Borning, "Classes versus prototypes in object-oriented languages," in *ACM Fall Joint Computer Conference*, 1986, pp. 36–40.
6. A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *24th European Conference on Object-Oriented Programming (ECOOP)*, 2010, pp. 126–150.
7. "European Association for Standardizing Information and Communication Systems (ECMA). ECMAScript Language Specification, 6th edition," 2015.
8. L. H. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil, "Does JavaScript software embrace classes?" in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 73–82.
9. L. H. Silva, D. Hovadick, M. T. Valente, A. Bergel, N. Anquetil, and A. Etien, "JSClassFinder: A Tool to Detect Class-like Structures in JavaScript," in *6th Brazilian Conference on Software (CBSoft), Tools Demonstration Track*, 2015, pp. 113–120.
10. M. Hafiz, S. Hasan, Z. King, and A. 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 (JSS)*, vol. 121, pp. 191–208, 2016.
11. W. Gama, M. Alalfi, J. Cordy, and T. Dean, "Normalizing object-oriented class styles in JavaScript," in *14th IEEE International Symposium on Web Systems Evolution (WSE)*, 2012, pp. 79–83.
12. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th International Conference on Software Engineering (ICSE)*, 2005, pp. 284–292.
13. M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
14. S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.
15. L. H. Silva, M. T. Valente, and A. Bergel, "Statically Identifying Class Dependencies in Legacy JavaScript Systems: First Results" in *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), Early Research Achievements (ERA) track*, 2017, pp. 1–5.
16. S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis, "Detecting function constructors in JavaScript," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 1–5.
17. A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip, "Refactoring towards the good parts of JavaScript," in *26th Conference on Object-Oriented Programming (OOPSLA)*, 2011, pp. 189–190.
18. M. Siff and T. Reps, "Program Generalization for Software Reuse: From C to C++," in *4th Symposium on Foundations of Software Engineering (FSE)*, 1996, pp. 135–146.
19. P. Kyriakakis and A. Chatzigeorgiou, "Maintenance Patterns of Large-Scale PHP Web Applications," in *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 381–390.