Automated Refactoring of Client-Side JavaScript Code to ES6 Modules

Aikaterini Paltoglou, Vassilis E. Zafeiris, E. A. Giakoumakis, and N. A. Diamantidis
Department of Informatics, Athens University of Economics and Business,
76 Patission Str., Athens 104 34, Greece
{paltoglouk, bzafiris, mgia, nad}@aueb.gr

***Abstract*—JavaScript (JS) is a dynamic, weakly-typed and object-based programming language that expanded its reach, in recent years, from the desktop web browser to a wide range of runtime platforms in embedded, mobile and server hosts. Moreover, the scope of functionality implemented in JS scaled from DOM manipulation in dynamic HTML pages to full-scale applications for various domains, stressing the need for code reusability and maintainability. Towards this direction, the ECMAScript 6 (ES6) revision of the language standardized the syntax for class and module definitions, streamlining the encapsulation of data and functionality at various levels of granularity.**

This work focuses on refactoring client-side web applications for the elimination of code smells, relevant to global variables and functions that are declared in JS files linked to a web page. These declarations "pollute" the global namespace at runtime and often lead to name conflicts with undesired effects. We propose a method for the encapsulation of global declarations through automated refactoring to ES6 modules. Our approach transforms each linked JS script of a web application to an ES6 module with appropriate import and export declarations that are inferred through static analysis. A prototype implementation of the proposed method, based on WALA libraries, has been evaluated on a set of open source projects. The evaluation results support the applicability and runtime efficiency of the proposed method.

***Index Terms*—Refactoring, Client-side JavaScript, Global variables, ES6 Modules**

I. Introduction

Dynamic programming languages are most commonly used by developers in order to create stable and interactive web applications, because of the limited use of structures met in static programming languages. The main characteristics of these languages enable developing new and more abstract code in a short period of time. However, these characteristics may influence the quality of the program, undermining its understandability, maintainability and reusability. Program refactoring [1], performing code transformations while sustaining the external behavior of the program, aims to preserve the aforementioned software characteristics.

Research on refactoring applications created using dynamic programming languages and especially JavaScript is in its infancy, compared with research on refactoring applications designed using static programming languages [2]. In the former case, code smell detection and transformation can gain limited benefits from static type information, since a large amount of information needed is resolved at runtime. Thus, restructuring programs developed using dynamic languages is more fragile and prone to errors.

JavaScript treats functions as objects and makes use of a flexible object model [1]; object properties, either fields or methods, may be created or/and deleted at runtime. In addition, dynamic creation and execution of a script is allowed through the `eval` function; thus, source code may be created and executed at runtime. These language characteristics may downgrade the static analysis result quality and reliability, as detection of the code being executed and its behavior require information resolved at runtime, except from static type information.

Unlike other programming languages, JavaScript does not provide a formal mechanism regarding designing applications using functionality modules, that is autonomous parts of code created in order to be used by multiple program parts without further modification [3]. In practice, Singleton pattern and its variations, such as Module or Revealing Module pattern and One Global Approach or Zero Global Approach, are widely used by the developer community in order to address that problem.

On top of the aforementioned language characteristics, a significant amount of web applications is created using third-party libraries [4] and frameworks; third-party code facilitates common tasks, assisting developers in focusing on the basic functionality of their applications, and thus accelerates the application development process. Nevertheless, these libraries and frameworks make excessive use of a set of the language's characteristics that introduce complexity into the static analysis process.

This work is motivated by problems related to the management of program state through global variables [5]. Global variables introduce coupling between program units (common coupling) that complicates maintainance and program understanding [6]. The presence of global variables that are defined in different program units, creates confusion on the origin of variable values [7], especially in cases that these program units are declared in different files. In JavaScript, global variables are implicitly declared in cases that a variable is used without being declared (implied globals) [1]. Name collisions of implied globals with global variables declared in other files is a common source of bugs [7], [8]. Besides global

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

variables, function declarations, loaded from multiple sources in an ECMAScript 5 application, contribute to the "pollution" of the global namespace and increase the probability of name collisions with undesired effects on program behaviour.

We propose a method for dealing with the problem of global namespace "pollution" through refactoring to ECMAScript 6 (ES6) modules. The focus is on client-side web applications, implemented in ECMAScript 5 or below. Our approach transforms each linked JS script of the web application to an ES6 module with appropriate import and export declarations that are inferred through static analysis. We specify the refactoring identification algorithm that operates on the Abstract Syntax Tree representation of JavaScript (JS) source files, as well as on the application's Static Call Graph. Moreover, we identify refactoring preconditions and specify the source code transformation. The proposed method for automated refactoring to ES6 modules has been implemented in Java, on the basis of T.J. Watson Libraries for Analysis (WALA) [9], and evaluated on a set of open source projects. The evaluation results support the applicability and runtime efficiency of the proposed method.

The rest of this paper is organized as following: Section II presents related work on static analysis and refactoring of JavaScript code. In Section III, we introduce our method for automated refactoring to ES6 modules. Section IV presents an experimental evaluation of the proposed method. Limitations detected during the implementation of the proposed approach are discussed in Section V. Finally, conclusions are presented in Section VI.

II. Related Work

Research on static and dynamic analysis of JavaScript applications introduces many challenges, due to the dynamic nature of the language. In this section, we discuss proposed techniques and tools that address problems concerning several aspects of code maintainance, such as code smell detection, testing and refactoring. We refer to the proposed techniques and tools regarding static programming languages and we proceed with a reference to the approaches regarding dynamic languages and especially JavaScript.

A list of code smells in static object-oriented programming languages is proposed in [10]. Each code smell is assigned to a specific set of recommended refactorings. The first refactoring tools, such as SmallTalk Refactoring Browser [11], were designed and implemented in order to meet the need of performing simple transformations that result in a program more understandable and maintainable while preserving its external behavior [12].

A number of approaches and tools based on static analysis of JavaScript applications are formally proposed towards the direction of mitigating problems concerning the detection of potential errors, the detection of language feature usages and the creation of secure applications. A study concerning static analysis using abstract interpretation for JavaScript programs is available in [13]. Attention is paid to type inference information, which can be used to detect potential errors, to enable program understandability and to build tools that assist developers in creating and maintaining JavaScript programs.

JSDeodorant [14] is a tool that detects function constructors in legacy JavaScript systems. JSDeodorant detects object creation expressions; by performing a dataflow analysis, the tool matches each expression with the corresponding object function constructor. It is extended towards the identification of inheritance relations between classes that are implemented using functions [15]. In [16], attention is paid to the identification of the dependencies between classes using static type check and inference.

The semantics for a subset of JavaScript, supporting static scoping and nested local variable hiding, that assists developers in creating secure JavaScript applications providing restricted access to third-party untrusted code is presented in [17]. Based on the subset mentioned above, a tool which automatically verifies that an API cannot be compromised through exploiting the language semantics is proposed.

A significant number of tools is designed and implemented in order to enable and improve the source code maintenance process in production. Google Closure Compiler [18] is a tool aiming at statically analyzing and minimizing JavaScript source code, by removing comments and dead code. Mozilla Rhino [19] is an open-source tool aiming at analyzing JavaScript source code, by creating the Abstract Syntax Tree (AST), and at specifying the sets of declared variables and functions.

While a significant number of studies are proposed in the direction of static analysis of JavaScript applications, research on detecting and restructuring code smells is limited. A number of tools, each targeting at the identification of specific code smells, is either formally or informally proposed. An analysis of 13 common code smells occuring in JavaScript code is available in [20]. Among the proposed code smells, there are smells encountered in programs created using traditional programming languages, such as switch statements, and smells specific to dynamic languages, such as coupling JavaScript, HTML and CSS. The authors propose a technique for their identification, which combines static and dynamic analysis of the input program. Moreover, an empirical study on code smells occuring in popular JavaScript applications is available in [21]. A survival analysis is performed in order to compare the time till fault occurence in files with and without code smells. An investigation concerning the causes of the JavaScript code parts that are wrongly detected as smells is available in [22]. Among the causes, dynamic file loading, asychronous calls and dynamic code generation are identified.

Current work on detecting code smells in web applications includes IDEs and tools, mainly static code analyzers, that support this procedure automatically, such as JSHint [23] and JSLint [24]. A study concerning bad coding practices reported by JSHint and JSLint on JavaScript systems is available in [25]. The code practices proposed as harmful are divided in two categories; practices related to bad coding logic, i.e. usage of undefined variables, and to bad coding style, i.e. lack of proper indentation.

BUGAID corresponds to a semi-automatic technique targeting at detecting the most common JavaScript code smells restructured by developers by mining version control repositories [26]. BUGAID uses machine learning, in order to group bugs related to language misuses by the language constructs changes made during the bug fixing process. xWIDL [28] is an extension of the WebIDL language targeting at detecting misuses of JavaScript APIs, e.g. wrong number or type of function arguments.

DLINT [27] is a dynamic analysis technique for detecting JavaScript program parts throughout which violations of quality rules informally accepted among developers are met. DLINT aims at detecting rule violations that are not detected using static analysis approaches.

A restricted number of studies pointing out the need of creating tests in code understandability and bug detection is proposed. A study concerning the causes of JavaScript bugs and their impacts on developers focusing on creating and testing JavaScript applications and on creating static analysis tools is presented in [29]. Attention is paid to the client-side bugs, which are caused by the problematic interation of JavaScript applications with the Document Object Model (DOM).

An empirical study on characterizing the quality of JavaScript tests and demonstrating their weaknesses is available in [30]. Among the results, it is observed that a significant amount of the studied JavaScript projects has no tests. Moreover, a differentiation concerning quality is observed between tests in server-side and client-side code and between tests created using frameworks and tests created without using any framework.

Apart from code smell detection and testing, there is a growing interest in the application of static analysis techniques in the refactoring of JavaScript code. Ongoing research focuses mainly on: (a) base refactorings, (b) refactorings that eliminate code smells or (c) refactorings for migration of legacy JavaScript code to newer versions of the language. More specifically, [31], [32] propose a framework for the specification of base refactorings (RENAME, ENCAPSULATE PROPERTY and EXTRACT MODULE) that is based on pointer analysis. Extensions to these works focus on renaming object properties in a semi-automatic manner [2]. The elimination through refactoring of common uses of the `eval` language construct is studied in [33]. More recent works on refactoring JavaScript code focus on the identification of classes in legacy code and their refactoring to the class syntax introduced in ES6 [34], [35]. These works propose appropriate migration rules that are based on code structures that are frequently used for the emulation of classes in ECMAScript 5 or below.

In this study, we propose a method for automated refactoring of client-side JavaScript code, written in ECMAScript 5 (ES5) or below, to ECMAScript 6 (ES6) modules. Our work can be compared with [34], [35] in the context of migrating code to a newer version of the language. However, our focus is on mitigating the global namespace "pollution" problem through utilizing ES6 modules. To the best of our knowledge, this is the first work that studies automated refactoring to this language feature.

III. REFACTORING TO ES6 MODULES

A. JavaScript Scope System

Unlike traditional object-oriented programming languages, JavaScript does not provide a standard feature for classes. Classes are emulated with the help of the function feature. A function, which is a first class object, meets the need of resolving the scope of a variable. A variable declared in the scope of a function is considered local, namely accessible from the scopes that are enclosed by its declaration scope. A variable declared outside any function or used without being declared is considered global, i.e., accessible from any scope in the file it is introduced. The excessive use of global variables becomes a source of bugs, since the "pollution" of the global namespace [36] with variable bindings declared in different files often leads to name conflicts. Moreover, modification of the program state from multiple locations results in less understandable, maintainable and testable code.

Javascript provides function scope, also known as variable and function hoisting, unlike other programming languages that provide block scope [7]. Specifically, a variable whose declaration is introduced in a block enclosed by a function is accessible from all scopes introduced in that function. Consequently, function objects may be used in organizing multiple code clusters with different functionality towards providing one complicated functionality and, thus, meeting the need of code decoupling and reusability.

Multiple design patterns that are widely adopted by the developer community, target at mitigating the excessive use of global variables. "Singleton", "Module" and "Revealing Module" patterns and their extensions, "One Global Approach" and "Zero Global Approach", are among the most commonly used patterns [3], [7]. Moreover, various module systems have been proposed for the encapsulation of data and functionality through bundling them in a distinct global variable. CommonJS, Asynchronous Module Definition (AMD) and Universal Module Definition (UMD) are typical module systems used in ES5 and below, while ECMAScript 6 (ES6) introduces modules as a standard language feature. CommonJS and AMD aim at facilitating module definition in the Node.js and browser platforms, respectively, while UMD is created in the direction of combining CommonJS and AMD [3], generalizing, thus, the module creation process.

However, the module definition pattern proposed by UMD produced overhead due to the need of resolution of the execution environment, either the server or the client, of the module to be created. In ECMAScript 6 (ES6) the use of modules, adopting the principles utilized in CommonJS, AMD and UMD but defined in a more brief and concrete way, is proposed [3]. Each source file containing variable and function declarations corresponds to a module. In cases when a variable or function, defined in the top-level scope of the module, needs to be used in multiple modules, its scope may be extended beyond the scope of the module it is declared. Thus,

it can be included in the modules where its use is needed. Additionally, each module is a singleton, namely there is exactly one instance of the module on runtime regardless of its inclusions. The aforementioned features lead to the adoption of modules that meet the need of code simplification, while preserving asynchronous and configurable loading [37].

B. Identification of Refactoring Candidates

In this paper, we focus on the namespace "pollution" problem that is inherent in client-side web applications implemented in ES5 or below, without support of a module framework (e.g. AMD, UMD) or the use of appropriate design patterns. The proposed method processes the Javascript code embedded in HTML pages, as well as all imported JS scripts, and mitigates the problem through code migration to ES6 and refactoring to the ES6 module system. In the ES6 module system, each Javascript source file (JS file) corresponds to a module. The global variables and top-level function declarations of a JS file are assigned to the module scope and are not visible beyond that scope, unless explicitly exported through an `export` statement. The declarations that are exported in an ES6 module definition can be imported to the namespace of other modules through the use of `import` statements.

Our method for refactoring to ES6 modules involves the automated identification of imported and exported variable and function declarations for each JS source file of the project under analysis. The identification procedure results to a set of refactorings that can be applied to the project files. We represent each suggested refactoring as a triplet $s_i = (d_i, n_i, n_e)$, where d_i is the name of a top-level function or variable declaration, n_i is the name of the JS file that requires (imports) the declaration and n_e is the JS file that provides (exports) it to other modules. Suggested refactorings establish the data and function dependencies among the modules that will be introduced after migration of a project to ES6. Recall that, each JS file of the project is turned into an ES6 module after refactoring.

The refactoring identification algorithm analyzes the HTML and JS source files of a client-side web application and constructs a directed graph data structure that models the ES6 modules of the application and their dependencies after refactoring. We will, henceforth, refer to this data structure as *Module Dependence Graph (MDG)*. More formally, the Module Dependence Graph is an ordered pair $MDG = (M, R)$ where:

- $M = \{m_i\}$ is the set of graph nodes, where each node m_i represents an ES6 module of the refactored application. Each element of M is defined as a triple $m_i = (n_i, G_i, F_i)$, where n_i is the name of the JS file that defines the module and G_i, F_i are the sets of global variables and top-level functions declared in JS file n_i, respectively.
- $R = \{r_j\}$ is the set of graph edges. Each edge r_j models a dependency among two modules of the refactored application. We define each dependency as a tuple $r_j = (d_j, t_j, m_k, m_l)$, where d_j is the name of a global variable or function declaration, t_j is the type of the dependency, m_k is the module that imports the declaration and m_l the module that exports the declaration. Allowed values for the dependency type t_j are: *function* (F), *global use* (U) and *global definition* (D). A *function* dependency denotes that the imported declaration is a function. Morever, *global use* and *definition* dependencies indicate that module m_k imports variable d_j declared in m_l and uses its value or defines it to another value through assignment.

  Notice that the suggested refactorings for the JS files of the application can be directly extracted through processing the edge set R of the MDG.

Our refactoring identification procedure comprises five processing steps:

1) Module Dependence Graph initialization,
2) Detection of global variable declarations,
3) Resolution of the top-level function declarations, i.e., functions belonging to the global scope and are invoked, directly or indirectly, by the HTML pages of the web application,
4) Resolution of modules' data dependencies,
5) Resolution of modules' function dependencies.

The rest of this section provides a description of the static code analysis involved in each step of the refactoring identification procedure. Inline with the description of these steps, we specify refactoring preconditions that apply to the respective code analysis stage. The role of refactoring preconditions is to prevent the application of refactorings that would lead to erroneous refactored code, in the sense that it would either have runtime errors (e.g. use of undefined variables) or it would not preserve the external behavior of the application.

1) Module Dependence Graph Initialization: The initialization of the MDG involves creation of the node set M on the basis of the HTML and JS files of the web application. Specifically, for each JS file we introduce a node m_i that is initialized with the name of the JS file. A graph node is, also, introduced for each HTML file that contains Javascript code in the form of: (a) JS files referenced through script tags, (b) embedded code inside script tags, (c) embedded code in HTML event attributes. The graph node is assigned the HTML file name with a `.js` extension. In case that a JS file with the same name exists, the node creation is omitted. The proposed source code transformation involves moving all embedded JS code of the HTML file to the respective JS file that will be referred to as *JS Entry file*.

2) Detection of Global Variable Declarations: The detection of the global variables declared in each source file is based on processing their Abstract Syntax Tree (AST) representations. Each variable declaration in the AST of a JS file is represented as an AST node of the corresponding type. A function declaration AST node that is ancestor of a variable declaration node represents the variable's enclosing function. Global variables are declared outside any function and, thus, have no function declaration ancestors. We identify

global variable declarations by traversing the AST in search of variable declaration nodes without function declaration ancestors. Each identified global variable is appended to the global variable set G_i of the respective MDG node m_i.

Precondition 1: Each global variable must be declared in a single JS file. Different global variable declarations with the same name in different files correspond to the same variable binding in the global scope. Although the presence of such variables provides hints for potential bugs, their refactoring and transformation to module-local variables would change the external behaviour of the application, which is not desired.

3) Resolution of Top-level Function Declarations: The resolution of the application's top-level function declarations is based on the construction of a Static Call Graph starting from functions that are directly invoked from the HTML pages of the web application. The Static Call Graph enables the identification of reachable functions within each JS file, i.e., functions that are executed as part of the web application, while excluding functions that are never invoked. The subset of these functions that belong to the global scope are added to the set F_i of the respective MDG node m_i. These functions will either remain local to the module, after refactoring, or become exported functions in case that they are invoked from other JS files/modules.

Precondition 2: The names of top-level functions must be unique among the JS files of the web application. In case of name conflicts, the name binding in the global namespace refers to the most recently loaded function declaration. The other function declarations are overridden, leading, usually, to unexpected program behaviour. Although such name conflicts can be avoided with ES6 modules, automated refactoring is not possible since the programmer must manually indicate which function declaration, among conflicting ones, must be imported into modules using the conflicting name.

4) Resolution of Modules' Data Dependencies: The resolution of the data dependencies of a module m_i requires the identification of global variables that are: (a) defined or used in the m_i source code and (b) declared in the source code of another module. We identify the global variable uses and definitions by analyzing an Intermediate Representation (IR) of the code declared in the global scope and the reachable functions of a module. The IR is in Static Single Assignment - SSA form and is retrieved from the constructed Static Call Graph with the help of WALA libraries [9].

The IR representation of a function provides scope information, local or global, for all variables referenced in the function's source code. Our method iterates over the IR instructions of each function in search of uses or definitions of global variables, i.e. variables with global scope.

Let g_i be a global variable identified in the IR code of a function in module m_k. If g_i is declared in m_k, i.e., $g_i \in G_k$, then processing of the IR proceeds with the next global variable. In case that $g_i \notin G_k$, the variable name is looked up in the respective sets of other modules. Let $g_i \in G_l$, i.e., g_i is declared in module m_l; a new data dependency is created among modules m_k, m_l and added to the set R. Thus, $R = R \cup \{(g_i, U, m_k, m_l)\}$ for a variable use in m_k. An appropriate tuple is added to R for a g_i definition in m_k. In case that no module of the MDG includes a declaration for g_i, then g_i is an *implied* or accidental global variable [1]. The implied variable is added to the set of global variables of module m_k.

Precondition 3: The value of a global variable must not be defined through assignment outside its declaring JS source file. The reason is that in ES6, exported variables may be modified only within the context of the module that declares them. Thus, the imported variables or functions of a module comprise read-only bindings to the referred elements. A violation of this precondition introduces runtime errors to the refactored application.

Data dependencies of type *global definition* (D) enable the detection of precondition violations. Specifically, given a module m_k and a set $\{(g_i, D, m_{l_j}, m_k) : j = 1 \dots n\}$ of n incoming type D dependencies for variable $g_i \in G_k$, then modules m_{l_j} include violations of Precondition 3 for variable g_i. However, there is an exception to this rule: the module m_k has a single incoming type D dependency (g_i, D, m_l, m_k) for variable $g_i \in G_k$ and includes no value assignments for g_i. In other words, m_k just declares and uses the value of g_i. In this case the precondition can be negated by moving the declaration of g_i to module m_l. Module m_l, now, declares and defines g_i, while m_k imports and uses its value.

5) Resolution of Modules' Function Dependencies: The resolution of function dependencies requires a full traversal of the application's Static Call Graph. Each node of the call graph (CG node) corresponds to a function and includes a list of successor nodes that refer to functions that are invoked in the scope of that function. A CG node, also, integrates information on the scope of the function declaration.

Given a CG node n_a, that corresponds to a function $f \in F_i$ of module m_i, we iterate over the successors of n_a in search of top-level functions declared in another module. Let n_b be a successor of n_a associated with top-level function f_l declared in module m_l. In this case, a new *function* dependency edge is created among modules m_i, m_l and added to the set R, i.e., $R = R \cup \{(f, F, m_i, m_l)\}$. The procedure continues recursively from node n_b in search of function dependencies for module m_l.

Figure 1 presents the Module Dependence Graph that results from applying the refactoring identification procedure to the source files of the *hangman* project. The project belongs to the data set that we have used for the experimental evaluation of the proposed method, as explained in Section IV.

The project includes two HTML files, namely `hangman.html` and `index.html`, that contribute the `hangman.js` and `index.js` *JS Entry files* to the project after refactoring. The *JS Entry files* along with the JS source files of the application are represented as MDG nodes, as depicted in Figure 1. Each module dependency is represented as a directed edge, labelled with the name of a variable/function declaration. The edge starts from the module importing the declaration and points to the declaring module. For presentation purposes, we consolidate into a

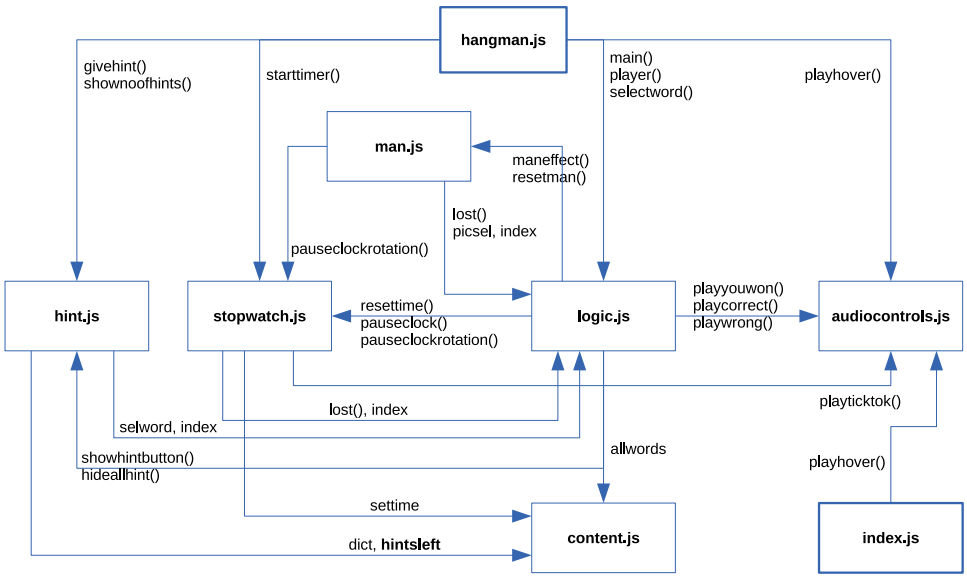

Fig. 1. The Module Dependence Graph for the *hangman* project.

single edge, two or more edges with the same direction among the same pair of nodes. For instance, edge (`man.js`, `logic.js`) represents a function dependency, `lost()`, and two data dependencies, `picsel`/`index`, among the respective modules. Module dependencies are *function* or *global use* dependencies, except for the (`hint.js`, `content.js`) dependency on `hintsleft` variable that is a *global definition* dependency. This type *D* dependency does not violate Precondition 3 and results to relocation of the `hintsleft` variable from module `content.js` to `hint.js`.

C. Source Code Transformation

The proposed source code transformation to ES6 modules is based on the Module Dependence Graph (MDG) constructed during the refactoring identification phase. The source code transformation involves two stages:

- Creation of the *JS Entry file* for each HTML page of the web application,
- Implementation of module dependencies through insertion of `import`, `export` declarations in JS files.

Notice that the source code transformation is aborted in case of refactoring precondition violations. Instead, appropriate warnings are presented to the developer, pinpointing the respective global variable/function name conflicts or global variables defined through assignment in multiple modules. The refactoring can be restarted after resolution of all precondition violations. Partial application of the refactoring is not possible, since migration to ES6 modules affects the entire project by encapsulating all non-exported declarations within their declaring module.

1) Creation of JS Entry Files: The refactoring procedure involves creation of a *JS Entry file* for each HTML file that embeds JavaScript code. *JS Entry file* creation for a given HTML file, named `[page].html`, comprises the following steps:

1) Create a JS file with name `[page].js`, if it does not exist, and move all JavaScript code embedded in HTML script tags to that file.
2) Remove all JavaScript code from HTML event attributes and generate the respective event registration code in the *JS Entry file*.
3) Replace all references to JS files in the HTML page with a single script tag that references the *JS Entry file*.
4) Implement module dependencies through appropriate `import` statements. A detailed description of this process will follow in Section III-C2.

Figure 2 demonstrates the creation of a *JS Entry file* for the `hangman.html` page of the *hangman* project. The HTML page embeds JavaScript code in script tags and event attributes. The figure highlights the refactoring steps with appropriate numbering. Specifically, the creation of the `hangman.js` file (1) is followed by registration of event listener functions (player, shownoofhints, selectword, startimer) to the `onload` event of the HTML `body` element (2). In the next step, all script references in the HTML file are replaced by a reference to `hangman.js` (3). Finally, appropriate `import` statements are introduced (4) on the basis of MDG module dependencies.

2) Implementation of Module Dependencies: Module dependencies are established through appropriate `import`, `export` declarations in the JS files of the web application. We determine the required module dependency declarations for each JS file through traversal of the Module Dependence Graph (MDG). The automated implementation of module dependencies involves two major steps:

1) Relocation of global variables referenced in *global definition* (type *D*) dependencies to the appropriate module. Recall from Section III-B4 that type *D* dependencies

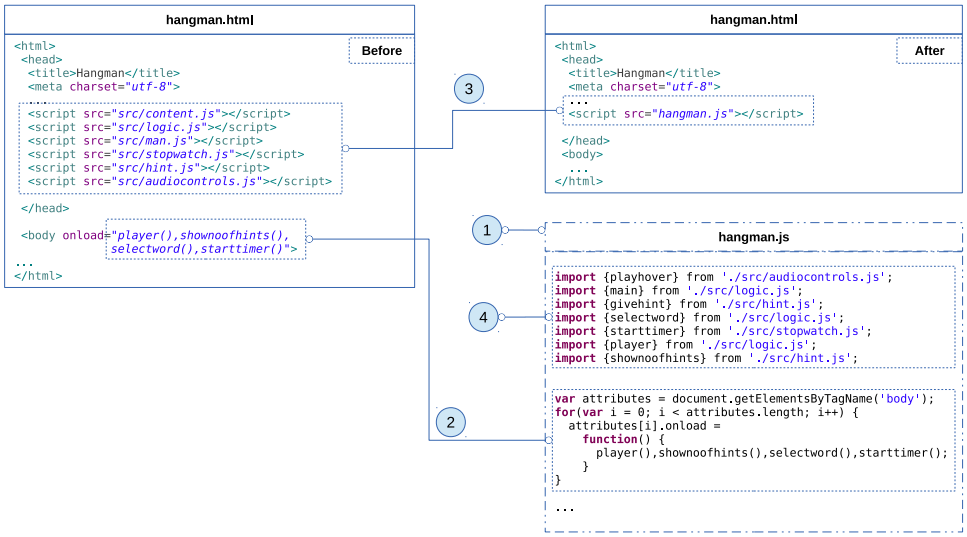

Fig. 2. Creation of the *JS Entry file* for the `hangman.html` page.

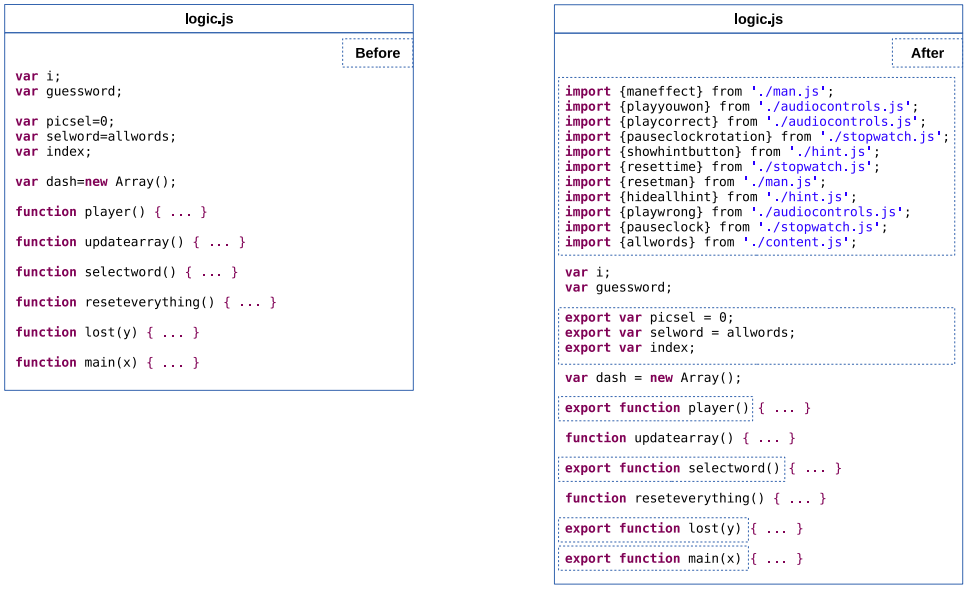

Fig. 3. Refactoring of the `logic.js` JS file.

that survive this stage of processing are associated with global variable relocations. Any other case of type D dependency would have violated Precondition 3, leading to rejection of the refactoring. The processing of each type D dependency (g_i, D, m_k, m_l) of the MDG involves:

- move of variable g_i declaration from original module m_l to m_k,
- redirection of all type U dependencies, that reference g_i, to module m_k that, now, declares g_i,
- elimination of the type D dependency from the MDG.

2) Introduction of `import`, `export` declarations to each JS file through iteration over the MDG nodes. The processing of each individual MDG node m_k involves the following refactorings to the respective JS file:

- add an `export` modifier to variable or function declarations that are referenced in incoming module dependence edges,
- add an `import` statement for each declaration referenced in outgoing module dependence edges.

Figure 3 presents the implementation of module dependencies for the `logic.js` file of the *hangman* project. The figure presents the file before and after the application of refactoring to ES6 modules. The exported declarations of the module correspond to incoming dependencies from *hangman*, `hint.js`, `man.js` and `stopwatch.js` modules, as depicted in Figure 1. Moreover, the module imports declarations, referenced in outgoing dependencies to modules

man.js, hint.js, stopwatch.js, content.js and audiocontrols.js.

IV. Evaluation

We have evaluated the proposed method for automated refactoring to ES6 modules by conducting an experimental study. The goal of this study is to analyze our method in order to assess: (i) its effectiveness in improving the application design without modifying its external behavior and (ii) its runtime efficiency. We conducted the experimental study from the perspective of a researcher seeking to evaluate the automation of a complex refactoring in the context of a set of open source applications.

A. Context Selection

The context of experimental evaluation comprises 6 open source web applications, published in GitHub and implemented in HTML and ECMAScript 5 (ES5) or below. The selection of these projects is based on the following requirements: (a) their source code comprises HTML and client-side JavaScript files written in ES5 or below, since our approach focuses on migration of legacy JavaScript code to ES6 language features, (b) they do not use any API for module definitions in ES5, such as AMD, (c) they are publicly available online, for study reproducibility reasons. Table I presents the web applications along with details on their implementation. Columns 3 (#HTML files) and 4 (#JS files) refer to the number of the HTML and JS files included in each application, respectively. Column 5 (#JS LOC), computed with CLOC [38], refers to the total lines of JavaScript code, while Column 6 includes third-party libraries or frameworks used by each application. Finally, Column 7 provides a link to the Github repository for each project. The web applications are sorted by ascending size (#JS LOC). The same ordering is applied to all tables presented in the evaluation section.

The static analysis required for the purposes of refactoring to ES6 modules is based on a Java implementation of the proposed refactoring algorithm. The implementation employs Mozilla Rhino [19], for Abstract Syntax Tree construction and traversal, and T.J.Watson Libraries for Analysis (WALA) [9], for Static Call Graph creation and extraction of each function's Intermediate Representation.

WALA comprises a set of Java libraries that provide static analysis capabilities for Java code. Currently, it provides support for performing static analysis on additional programming languages, such as JavaScript, by analyzing the source code given as input and translating it to a language close to the JVM bytecode. It supports various types of static program analysis, including pointer analysis and intraprocedural/interprocedural dataflow analysis, by relying on the call graph construction process.

On top of the typical Static Call Graph properties referenced in compiler textbooks [39], the WALA Call Graph contains a set of nodes corresponding to JavaScript and HTML DOM built-in functions. In addition, a Static Call Graph node represents either the construction or the execution of a JavaScript function. WALA, also, supports approximate creation of call graphs by performing a lightweight field-based flow analysis that associates each property name with an abstract location [40]. The analysis and, subsequently, the construction of the Field-based Call Graph addresses the problem of statically analyzing real-world applications designed using JavaScript third-party libraries, such as jQuery; these libraries make use of a set of the language's dynamic characteristics, for example reflection, that introduce complexity into the static analysis process.

Static analysis of JavaScript code in WALA is performed on an Intermediate Representation (IR) instead of the code itself [9]. Generation of the Intermediate Representation involves transforming each statement to Static Single Assignment (SSA) [39] register-based transfer language close to the JVM language, eliminating stack abstraction.

B. Research Questions

The experimental evaluation aims at answering the following research questions:

- RQ1: How effective is the proposed method in mitigating the global namespace "pollution" problem? This research question aims at evaluating the effectiveness of the proposed method, on the studied web applications, through estimation of: (i) the number of global variables and functions exposed to the global namespace before refactoring to ES6 modules and (ii) the number of global variables and functions encapsulated in ES6 modules after refactoring. The results will highlight the potential of the proposed method for improving application design through encapsulation of program state and behaviour.
- RQ2: What is the runtime efficiency of the refactoring identification process? This research question evaluates the practicality of the proposed method through estimation of the runtime performance of our prototype implementation on the web applications of Table I.

C. Evaluation Results

1) Research Question 1: How effective is the proposed method in mitigating the global namespace "pollution" problem? In order to measure effectiveness, we considered as metrics: (a) the total number of global variables and functions encountered in the HTML and JS files of each application and (b) the number of global-variables and top-level functions that are not exported by ES6 modules in the refactored code. Non exported functions and global variables are private to their declaring module and will be, henceforth, referred to as *Encapsulated Functions* and *Encapsulated Global Variables*, respectively.

The proposed method removes all global variables and top-level function declarations from the global scope and assigns them to the scope of the modules that declare or import them. Columns 4 and 6 of Table II provide an estimate of the name bindings that are removed from the global namespace. The effectiveness of the proposed method in encapsulating application behaviour is calculated as the ratio of *Encapsulated Functions* over the total number of functions for each project.

TABLE I
PROJECTS USED IN THE EVALUATION OF OUR APPROACH.

ID	Name	#HTML files	#JS files	#JS LOC	Third-party libraries/frameworks used	Resource
1	wksmine	1	3	318	jQuery	https://github.com/wks/wksmine
2	UltraTetris	1	5	326	jQuery	https://github.com/silviolucenajunior/UltraTetris
3	Hangman	2	7	346	-	https://github.com/aurobindodebnath/Hangman
4	SNAP-Calculator	1	0	458	-	https://github.com/JoshuaMGoodwin/SNAP-Calculator
5	TicTacToe	1	6	1055	jQuery, BootStrap	https://github.com/seanpr/TicTacToe
6	uki	1	3	2360	-	https://github.com/crcx/uki

TABLE II
GLOBAL VARIABLES AND FUNCTIONS ENCAPSULATED IN ES6 MODULES.

ID	Name	#Functions	#Encapsulated Functions	#Global Variables	#Encapsulated Global Variables
1	wksmine	28	28 (100%)	4	4 (100%)
2	UltraTetris	26	22 (84%)	14	7 (50%)
3	Hangman	24	5 (20%)	16	10 (62%)
4	SNAP-Calculator	10	10 (100%)	62	62 (100%)
5	TicTacToe	31	27 (87%)	13	13 (100%)
6	uki	15	0 (0%)	35	34 (97%)
	Average	**-**	**65.2%**	**-**	**84.8%**

We have computed the total number of functions, including anonymous functions, using ComplexityReport [41]. Effectiveness, in terms of encapsulating application state is calculated as the ratio of *Encapsulated Global Variables* over the total number of global variables. Table II results show an average effectiveness of 65.2% in encapsulating function declarations and 84.8% in encapsulating global variables.

The effectiveness of the proposed method in encapsulating state and behaviour depends on the usage of global variables and functions outside their declaring module. Full encapsulation (100%) of global variables in a given project (e.g. *wksmine*) denotes that all variables are used within their declaring module. A lower degree of encapsulation, e.g. 62% in *hangman* project, means that the remaining part of global variables (38%) are referenced outside their declaring module. As concerning functions, full encapsulation (100%) denotes that the project's JS files do not export any function. In this case, reachable functions are: (a) invoked by functions declared in the same module or (b) registered as DOM event listeners inside their declaring module. On the other hand, no encapsulation means that all top-level functions are marked as exported. For instance, all functions in the *uki* project are exported by their respective modules and imported in the *JS Entry file* where they are registered to HTML event attributes.

2) Research Question 2: What is the runtime efficiency of the refactoring identification process? For each web application, we measure the time required to analyze the application code along with the third-party library code. We, also, measure the execution time for Static Call Graph construction and present it comparatively to the total execution time. The results are presented in Table III. The experiment is performed on a computer with an Intel Core i7-3770 CPU @ 3.40GHz processor and 16GB RAM.

The execution of the implemented refactoring tool requires little user involvement, as she only needs to indicate the web application to be analyzed. Therefore, execution time depends, mainly, on the refactoring candidate identification process and, secondarily, on the JavaScript source code transformation process. The former is dependent on Rhino and the static analysis capabilities provided by T.J.Watson Libraries for Analysis (WALA). The latter is dependent on the construction of the Module Dependence Graph, as described in Section III. Runtime performance results in Table III show that execution time is dominated by the call graph creation process. Additionally, a delay in the execution of the implemented tool is observed in cases when the input application comprises a program designed using third-party libraries, such as jQuery. These libraries make use of a set of the dynamic characteristics provided by JavaScript, such as reflection, that introduce complexity into the static analysis process.

V. LIMITATIONS

In this section, we refer to a number of limitations that apply to the proposed method. A first limitation is that the JS files of the analyzed application must be free of syntactic errors. Since the detection of global variables is based on Rhino [19], syntax errors break the parsing process and, thus, interrupt the refactoring identification procedure.

Moreover, the proposed method does not analyze code that is dynamically generated and evaluated through the `eval()` function. The static analysis of such code involves several challenges, especially in cases that the code is generated at runtime [33].

The proposed refactoring migrates the ES5 source of a web application to ES6 syntax that is not yet fully supported by commercial and open source browsers. For this reason, the generated code is integrated with tools that guarantee its portability through transpiling to ES5 or earlier JavaScript versions and, thus, enabling the observation and the manual inspection of the external behaviour of the evaluated project before and after the application of the suggested refactoring. Specifically, the refactored application is integrated with the

TABLE III
RUNTIME PERFORMANCE OF REFACTORING IDENTIFICATION ALGORITHM.

ID	Name	Total Execution Time	Call Graph Creation Execution Time
1	wksmine	10.952 ms	8.739 ms (79.8%)
2	UltraTetris	10.916 ms	9.882 ms (90.5%)
3	Hangman	3.095 ms	2.533 ms (81.9%)
4	SNAP-Calculator	2.567 ms	2.087 ms (81.3%)
5	TicTacToe	23.098 ms	21.322 ms (92.3%)
6	uki	6.460 ms	5.949 ms (92.1%)

Babel [42] transpiler and *webpack* module bundler [43]. *Babel* transforms the ES6 code to ES5, while *webpack* bundles in a single file the application modules and their dependencies for direct loading by the browser. The implemented prototype generates all configuration files required for integrating Babel and webpack in the application build process.

The evaluation of the source code transformation, in terms of preserving the external code behaviour after refactoring, has been confined by the lack of test suites in the analyzed web applications. Thus, we have empirically evaluated its correctness through code inspection and manual testing of the applications' behaviour before and after the refactoring. As concerning the syntactic correctness of the refactored code, it has been automatically validated by *Babel* during transpilation of the refactored ES6 code to ES5.

VI. CONCLUSIONS AND FUTURE WORK

We have proposed a method for automated refactoring of client-side web applications with aim to eliminate code smells relevant to global variables and functions declarations. Such declarations are introduced to the global scope through execution of JavaScript code that is embedded or linked to a web page. These declarations "pollute" the global namespace at runtime and often lead to name conflicts with undesired effects. The proposed method encapsulates global declarations through automated refactoring to ES6 modules. Specifically, each linked JavaScript file of the web application is transformed to an ES6 module with appropriate import and export declarations that are inferred through static analysis. A prototype implementation of the proposed method, based on WALA libraries, has been evaluated on a set of open source projects. The evaluation results support the applicability and runtime efficiency of the proposed method.

Our future work will focus on the study of JavaScript code patterns that can be refactored to ES6 modules, such as Immediately-Invoked Function Expressions (IIFEs). Moreover, our study will extend to ES5 applications that already use a module API (e.g. AMD), in search of issues relevant to their migration to ES6 modules.

REFERENCES

[1] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[2] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for javascript," *SIGPLAN Not.*, vol. 48, no. 10, pp. 323–338, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544173.2509520

[3] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2012.

[4] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 435–458. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_20

[5] H. Sankaranarayanan and P. A. Kulkarni, "Source-to-Source Refactoring and Elimination of Global Variables in C Programs," *Journal of Software Engineering and Applications*, vol. 6, no. 5, 2013.

[6] W. Wulf and M. Shaw, "Global variable considered harmful," *SIGPLAN Not.*, vol. 8, no. 2, pp. 28–34, Feb. 1973. [Online]. Available: http://doi.acm.org/10.1145/953353.953355

[7] Nicholas C. Zakas, *Maintainable JavaScript*. O'Reilly Media, Inc., 2012.

[8] S. Stefanov, *JavaScript Patterns*. O'Reilly Media, Inc., 2010.

[9] IBM T.J. Watson Research, "WALA: T.J. Watson Libraries for Analysis," http://wala.sourceforge.net, 2017.

[10] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theor. Pract. Object Syst.*, vol. 3, no. 4, pp. 253–263, Oct. 1997.

[12] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[13] S. H. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *Proceedings of the 16th International Symposium on Static Analysis*, ser. SAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 238–255.

[14] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis, "Detecting Function Constructors in JavaScript," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 488–492.

[15] L. Eshkevari, D. Mazinanian, S. Rostami, and N. Tsantalis, "JSDeodorant: Class-awareness for JavaScript programs," in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

[16] L. H. Silva, M. T. Valente, and A. Bergel, "Statically identifying class dependencies in legacy javascript systems: First results," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[17] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated Analysis of Security-Critical JavaScript APIs," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 363–378. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.39

[18] Google, "Closure Compiler," https://developers.google.com/closure/compiler, 2017.

[19] Mozilla, "Rhino Javascript Engine," https://www.mozilla.org/rhino, 2017.

[20] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript Code Smells," in *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2013, pp. 116–125.

[21] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in JavaScript projects," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 294–305.

[22] S. R. Joonyoung Park, Inho Lim, "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild," in *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, May 2016.

[23] R. Waldron, C. Potter, M. Pennisi, and L. Page., "JSHint," http://jshint.com, 2017.

[24] Douglas Crockford, "JSLint," http://jslint.com, 2017.

[25] E. F. Adriano L. Santos, Marco Tulio Valente, "Using JavaScript Static Checkers on GitHub Systems: A First Evaluation," in *III Workshop on Sofware Visualization, Evolution and Maitenance*, 2015.

[26] Q. Hanam, F. S. D. M. Brito, and A. Mesbah, "Discovering bug patterns in JavaScript," in *24th ACM SIGSOFT International Symposium*, Nov 2016.

[27] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically Checking Bad Coding Practices in JavaScript," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771809

[28] Z. Zhang, "xWIDL: Modular and Deep JavaScript API Misuses Checking Based on eXtended WebIDL," in *Companion the 2016 ACM SIGPLAN International Conference*, Oct 2016.

[29] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A study of causes and consequences of client-side javascript bugs," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, Feb 2017.

[30] A. M. Amin Milani Fard, "JavaScript: The (Un)covered Parts," in *Verification and Validation (ICST), 2017 IEEE International Conference on Software Testing*, Mar 2017.

[31] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for javascript," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 119–138. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048078

[32] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Refactoring Towards the Good Parts of Javascript," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 189–190. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048200

[33] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 34–44. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336758

[34] L. H. Silva, M. T. Valente, A. Bergel, N. Anquetil, and A. Etien, "Identifying classes in legacy JavaScript code," *Journal of Software: Evolution and Process*, vol. 1, no. 1, pp. 1–37, 2017.

[35] L. H. Silva, M. T. Valente, and A. Bergel, "Refactoring legacy JavaScript code to use classes: The good, the bad and the ugly," in *16th International Conference on Software Reuse (ICSR)*, 2017, pp. 1–16.

[36] M. Haverbeke, *Eloquent JavaScript. A Modern Introduction to Programming*. No Starch Press, 2014.

[37] A. Rauschmayer, *Exploring ES6: Upgrade to the next version of JavaScript*, In 2017.

[38] Rick Waldron and Caitlin Potter and Mike Pennisi and Luke Page., "CLOC tool," https://github.com/AlDanial/cloc, 2017.

[39] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[40] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 752–761. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486887

[41] Complexity Report, "Software complexity analysis for JavaScript projects," https://github.com/escomplex/complexity-report, 2017.

[42] "Babel," https://babeljs.io/, 2017.

[43] "Webpack module bundler," https://webpack.js.org/, 2017.