Using JavaScript Static Checkers on GitHub Systems: A First Evaluation

Adriano L. Santos[1], **Marco Tulio Valente**[1], **Eduardo Figueiredo**[1]

[1]Department of Computer Sciente – Universidade Federal de Minas Gerais (UFMG)
Av. Pres. Antônio Carlos 6627 – 31.270-901 – Minas Gerais – MG – Brazil

{adrianolages,mtov,figueiredo}@dcc.ufmg.br

Abstract. *To improve code quality in JavaScript systems, static code analyzers are used to detect bad coding practices that can be potential bugs and can cause the system to not work properly. This paper investigates bad coding practices in JavaScript reported by two static analyzers (JSHint and JSLint) in order to verify if JavaScript systems are using this lint-like checkers to avoid bad coding practices and what are the most common warnings detected by them. Through an empirical study, we analyze 31 JavaScript systems and our results show that JSHint and JSLint are used by the development team of systems. We also found bad coding practices in all analyzed systems. In five systems, the number of warnings between old and new versions decreased by 14%.*

1. Introduction

Javascript is a widely used client-side language for develop web applications. It is used on various popular websites including Facebook, Gmail, and Twitter. More recently, JavaScript has found its way into server side platforms, such as Node.js[1]. These reasons contribute to a increasing popularity of the language, which became a fundamental piece in modern and interactive Web development [Cantelon et al. 2013].

To avoid pitfalls in JavaScript code, static code analyzers are used to detect problems that may not be verified when a developer is writing code. Static analyzers assist developers looking at code and they can find problems before running it. They inspect code and point some problems based on guidelines [Crockford 2008], [Kovalyov 2015]. In this work we present the results of a study comparing the warnings reported by two popular static code analyzers for JavaScript, called JSHint[2] and JSLint[3]. We observe two categories of warnings reported by these tools: warnings related to bad code logic (Group A) and warnings related to bad code style (Group B). JSHint and JSLint were choosen because they are widely accepted by developers and commonly used in industry.

Warnings of Group A identify pitfalls on code, e.g., mistyped keywords, mistyped variable names, attempts to use variables or functions not defined, use of `eval` function, bad use of operators, etc. These warnings are subtle and difficult to find and can lead to serious bugs. On the other hand, warnings of Group B enforce basic code style consistency rules that do not cause bugs in the code, but make it less readable and maintainable to other developers. This kind of warning is related to lack of proper indentation, lack of curly braces, no semicolon, no use of dot notation, etc.

[1]Node.js. https://nodejs.org/
[2]JSHint. http://jshint.com/.
[3]JSLint. http://www.jslint.com/.

This paper makes the following contributions: (i) We present a list of bad coding practices found by the two tools in an empirical study on 31 JavaScript systems located on GitHub. (ii) We look at whether the warnings indicated by the tools are increasing over a system development time and which warnings are more/less common in the studied systems. (iii) We verify if the systems studied on this present paper are making use of static code analyzers. (iv) We show that the use of static code analyzers are important for improving the code quality.

This paper is structured as follows: Section 2 documents some bad code practices in JavaScript. In Section 3 we detail our methodology to make the empirical study. We also present our results and our findings. Related work is presented in Section 5. Lastly, in Section 6 presents our final conclusions.

2. Bad Coding Practices in JavaScript

The goal of this work is to analyze violations of commonly accepted rules in JavaScript software, as reported by JSHint and JSLint. In this section, we document some of these rules.

Definition (Code quality rule) *A code quality rule is an informal description of a pattern of code or execution behavior that should be avoided or that should be used in a particular way. Following a code quality rule contributes to, for example, increased correctness, maintainability, code readability, or performance* [Gong et al. 2015].

2.1. Using Undeclared Variables

JavaScript allows a variable to be used without a previous declaration. JSHint and JSLint throws a warning when they encounter an identifier that has not been previously declared in a `var` statement or function declaration. In the example of Figure 2.1 we reference the variable `a` before we declare it. In this example both static code analyzers report the warning: `{a} was used before it was defined.`

```
function test() {
    a = 1; // 'a' was used before it was defined.
    var a;
    return a;
}
```

Fig 1: Code with a warning related to not declared variables.

The warning displayed in Figure 1 is raised to highlight potentially dangerous code. The code may run without error, depending on the identifier in question, but it is likely to cause confusion to other developers and the piece of code could in some cases cause a fatal error that will prevent the rest of script from executing.

2.2. Shadowing variables

Variable shadowing occurs when a variable declared within a certain scope (decision block or function) has the same name as a variable declared in an outer scope. This kind of problem generates `{a} is already defined` warning.

```
var currencySymbol = "$"; //This variable is shadowed by inner function variable.

function showMoney(amount) {
  var currencySymbol = "R$"; // '{currencySymbol}' is already defined.
  document.write(currencySymbol + amount); //R\$ sign will be shown.
}
showMoney("100");
```

Fig 2: Code with a warning related to shadowing variable.

In the example of Figure 2 the variable `currencySymbol` will retain the value of the final assignment. In this case, a `R$` sign will be shown, and not a dollar, because the `currencySymbol` containing the dollar is at a wider (global) scope than the `currencySymbol` containing the `R$` sign. This type of error can occur because the programmer can mistyped the identifier of one of the variables. Renaming one of them can solve this problem.

2.3. Mistakes in conditional expressions

Sometimes programmers forgotten to type an operator and instead of defining a conditional expression they define an assignment to a variable. JSHint and JSLint reports a warning called `expected a conditional expression and instead i saw an assignment` when this situation occur, as illustrated in Figure 3.

```
function test(someThing) {
    do {
        someThing.height = '100px';
    } while (someThing = someThing.parentThing); //expected a
    // conditional expression and instead i saw an assignment
        ...
}
```

Fig 3: Code with a warning related to confusing conditional expression.

3. Study

The JavaScript systems considered in this study are available at GitHub. We selected systems ranked with at least 1,000 stars at GitHub, whose sole language is JavaScript, and that have at least 150 commits. This search was performed on May, 2015 and resulted in 31 systems from different domains, covering frameworks, editors, games, etc [Silva et al. 2015]. After the checkout of each system, we manually inspected the source code to remove the following files: compacted files used in production to reduce network bandwidth consumption (which have the extension *.min.js), documentation files (located in directories called doc or docs), files belonging to third party libraries, examples and test files.

The selected systems are presented in Table 1, including their version, size (lines of code), LOC/#Total warnings detect by JSHint, LOC/# Total warnings detected by JSLint, number of warnings from Group A and Group B reported by JSHint and JSHint and the total number of warnings detected by the two static analyzers. We use Eclipse with JSHint plug-in (version 0.9.10) and JSLint plug-in (version 1.0.1) to analyze the systems in Table 1. In our study, we used the default settings of each plugin.

Table 1 presents the total number of warnings detected by JSHint and JSLint. Both static analyzers found pitfalls and code style warnings in all systems (100%). The number of warnings for each system in Table 1 is the total number of warnings found including repetitions of a same type of warning. JSHint and JSLint found 52 different types of warnings in the studied systems. The system with the largest number of warnings is `mocha` (2401 warnings), followed by `babel` (2328 warnings) and `wysihtml5` (2303 warnings). This high number of warnings correspond to warnings of group B (code style warnings). Figure 4 show the total warnings encountered by each static analyzers. The total number of warnings found by JSHint was 20.12% for Group A and 79.88% of Group B. JSLint reports 6.24% and 93.76% warnings of Group A and B, respectively. Furthermore, columns 4 and 5 of Table 1 show the density of warnings encountered by each static analyzer in the analyzed systems. For example, for `gulp`, a warning is found by JSHint for each 4.27 lines

Table 1. JavaScript systems (ordered on the LOC column).

System	Version	LOC	LOC/#Total JSHint	LOC/#Total JSLint	#Warnings Group A JSHint	#Warnings Group A JSLint	#Warnings Group B JSHint	#Warnings Group B JSLint	#Total JSHint	#Total JSLint
masonry	3.1.5	197	16.47	1.11	0	12	12	165	12	177
gulp	3.7.0	282	4.27	3.66	17	18	49	59	66	77
randomColor	0.1.1	361	30.08	7.22	7	11	5	39	12	50
respond	1.4.2	460	0	3.40	0	2	0	133	0	135
mustache.js	0.8.2	571	71.37	12.14	8	5	0	42	8	47
clumsy-Bird	0.1.0	628	11.48	4.21	3	8	52	141	55	149
deck.js	1.1.0	732	183	14.93	1	14	3	35	4	49
impress.js	0.5.3	769	0	40.47	0	0	0	19	0	19
isomer	0.2.4	770	13.27	1.75	3	26	55	414	58	440
fastClick	1.0.2	798	0	16.62	0	0	0	48	0	48
parallax	2.1.3	1007	0	968	0	7	0	97	0	104
intro.js	0.9.0	1026	20.52	21.37	32	5	18	43	50	48
alertify.js	0.5.0	1036	20.72	21.37	28	3	22	146	50	149
async	0.9.0	1117	21.48	15.30	3	5	49	68	52	73
socket.io	1.0.4	1223	101.91	13.89	7	5	5	83	12	88
qunit	1.14.0	1379	197	26.01	5	3	2	50	7	53
underscore	1.6.0	1390	43.43	46.33	30	6	2	24	32	30
slick	1.3.6	1684	842	42.41	2	1	0	39	2	40
turn.js	3.0.0	1914	34.17	20.80	26	2	30	90	56	92
numbers.js	0.4.0	2454	175	15.24	5	27	9	134	14	161
cubism.js	1.6.0	2456	16.26	5.62	61	81	90	356	151	437
zepto	1.1.6	2456	4.10	4.54	54	32	545	508	599	540
typeahead.js	0.10.2	2468	32.90	2.87	55	14	20	844	75	858
rickshaw	1.5.1	2752	12.41	1.83	31	65	190	1433	221	1498
express	4.4.1	2942	50.72	22.98	14	7	44	121	58	128
knockout	3.3.0	3157	4.23	3.35	96	81	649	861	745	942
jasmine	2.3.4	3956	44.44	13.18	21	28	68	272	89	300
mocha	2.2.5	4225	3.76	3.30	98	30	1025	1248	1123	1278
slickgrid	2.1.0	5345	15.44	3.56	195	44	151	1456	346	1500
babel	5.4.7	5893	4.31	6.11	167	150	1198	813	1365	963
wysihtml5	0.3.0	5913	20.11	2.94	149	88	145	1921	294	2009

of code. JSLint finds a warning for each 3.66 lines of code. The low number of Group A warnings (warnings that can cause the system to do not function properly) indicates that programmers are cautious and want to ensure code quality and they may be making use of static analyzers in their systems. In `impress.js` system, only 19 warnings of Group B were found by JSLint. Another system which showed only warnings of Group B was `fastclick` (48 warnings found by JSLint). In all systems, JSHint detected less warnings that JSLint. On the other hand, JSHint detected more warnings of group A that JSLint (20.12% versus 6.24% respectively).

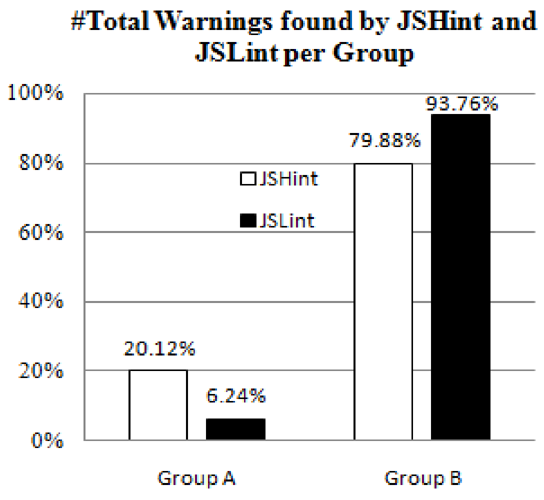

Figure 4. Percentage of warnings detected by JSHint and JSLint.

Table 2. Type of warnings found (ordered by #Systems column).

Warning	#Systems		Warning	#Systems		Warning	#Systems	
	JSHint	JSLint		JSHint	JSLint		JSHint	JSLint
{a} is already defined ∗	12	0	{a} is out of scope ∗	7	1	missing semicolon †	18	1
use '!==' to compare with '0', 'null', or 'undefined' ∗	9	10	unnecessary semicolon †	6	1	use '===' to compare with '0', 'null', or 'undefined' ∗	8	9
use 'isNaN' function instead compare with 'NaN' ∗	1	0	eval is evil ∗	4	3	missing Break Statement before 'case' ∗	2	1
expected a conditional expression and instead i saw an assingment ∗	8	0	expected an assignment and instead i saw an expression ∗	17	2	don't make functions within a loop ∗	11	4
missing '()' invoking a constructor ∗	11	5	{a} was used before it was defined ∗	3	3	do not use 'new' for side effects ∗	0	5
missing 'new' prefix when invoking a constructor ∗	1	2	expected {a} at column x, not column y †	0	20	possible Strict Violation ∗	2	0
missing name in function declaration ∗	1	0	unexpected space after †{a}	0	2	unnecessary Semicolon †	6	1
do not declare variables in a loop †	0	4	bad line breaking before {a} †	10	0	read only ∗	1	2
variable was not declared correctly ∗	3	2	combine this with the previous var statement ∗	0	8	empty block †	0	9
expected an identifier and instead saw 'undefined' (a reserved word) ∗	1	8	missing "use strict" statement †	0	22	use the function form of 'use strict' †	2	3
move "var" declarations to the top of the function †	0	14	unexpected dangling '_' in {a} †	0	17	weird relation ∗	0	1
weird condition ∗	0	1	Script URL ∗	2	0	[{a}] is better written in dot notation †	6	7
did you mean to return a conditional instead of an assignment? ∗	2	0	do not use {a} as a constructor ∗	1	0	don't use 'with' ∗	1	0
the '_proto_' property is deprecated †	4	2	unexpected ++ †	0	12	constructor name should be start with a uppercase letter †	0	3
unreacheable {a} after 'throw', 'return' ∗	3	0	comma warnings can be turned off with 'laxcomma' †	3	0	unexpected sync method ∗	0	6
unexpected TODO Comment †	0	3	missing name in function statement ∗	0	1	move the invocation into the parens that contain the function ∗	0	5
missing space after {a} †	0	9	unexpected 'typeof'. Use '===' to compare directly with undefined ∗	0	6			
use spaces not tabs †	0	6	Unexpected 'in'. Compare with undefined, or use the hasOwnProperty method instead †	0	3			

Table 2 presents the number of systems who have occurrences of a particular type of warning. For example, the warning (`{a}is already defined`) appears in twelve systems and was detected only by JSHint. Some warnings displayed by JSHint and JSLint have the same meaning, but have different names. For example, warnings about use of `eval` function. JSHint shows a warning with the text: `eval can be harmful` and JSLint shows the text: `eval is evil`. In this study, we consider the terminology used by JSLint. In the analyzed systems 52 types of warnings were found. JSHint and JSLint are able to detect more than 150 types of warnings [Kovalyov 2015]. Warnings marked with ∗ belong to Group A and warnings marked with † belong to Group B.

The most common warnings shown in Table 2 are:

- `missing "use strict" statement` (occurred in 22 systems), this warning is raised to highlight a deviation from the strict form of JavaScript coding. This warning can be helpful as it should highlight areas of code that may not work as expected, or may even cause fatal JavaScript errors.
- `Expected {a} at column x, not column y` (occurred in 20 systems), this code style warning (group B) warns programmers about lack of indentation.
- `Unexpected dangling '_' in {a}` (occurred in 17 systems), this warning reported by JSLint warns about use of underscore character in variables name.
- `Move "var" declarations to the top of the function` (occurred in 14 systems), this warning is thrown when JSLint encounters a variable declaration in a `for` or `for-in` statement initializer.
- `{a} is already defined` (occurred in 12 systems), only this warning belong to group A and and was detected only by JSHint.

The less common warnings shown in Table 2 are:

- `do not use {a} as a constructor` (1 ocurrence), this warning is thrown when JSHint or JSHint encounters a call to `String`, `Number`, `Boolean`, `Math` or `JSON` preceded by the `new` operator. This kind of bad practice can become a fatal error in JavaScript.

- `Weird relation` (1 occurrence), this warning is raised when JSLint encounters a comparison in which the left hand side and right hand side are the same, e.g. `if (x === x) or if ("10" === 10).`
- `missing name in function declaration` (1 ocurrence).
- `use 'isNaN'function instead compare with 'NaN'` (1 occurrence).
- `do not use {a} as constructor` (1 occurrence).

We also investigated whether warnings detected by two static analyzers increase or decrease along the different versions of the analyzed systems. We selected five systems with more number of warnings of all groups and analyze an earlier version with at least one year of difference between the versions. Figure 5 shows the total number of warnings reported by JSHint and JSLint for each system in an earlier and previous version. For example, the wysihtml5 system in version 0.2.0 presents 335 warnings detected by JSHint and 2290 warnings detected by JSLint. The version 0.3.0 of wysihtml system which has a year of difference to version 0.2.0 presents 294 and 2009 warnings detected by JSHint and JSLint respectively, a reduction of 14% of warnings. In the five systems analyzed in Figure 5 there was reduction of warnings. On average the reduction was 14% for JSHint and 12% for JSLint. These results may indicate that developers are using static checkers, as JavaScript IDE's have JSHint and JSLint embedded.

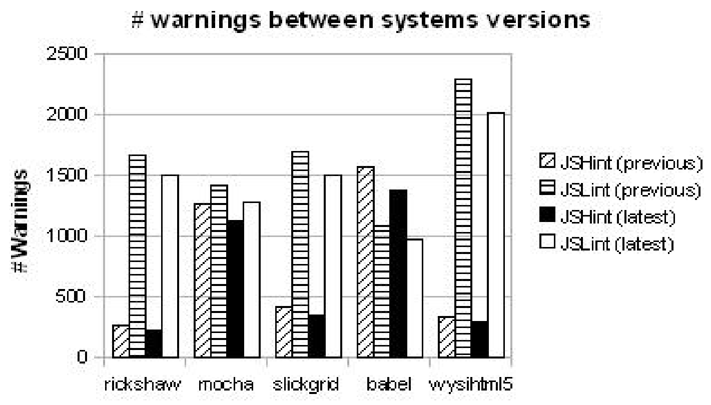

Figure 5. Warnings detected by JSHint and JSLint in different systems versions.

Threats to Validity: Our systems sample can be small and not be representative. We minimize this threat by selecting projects of different sizes of LOC and different domains. Static analyzers may have incorrectly considered some code snippets as bad coding practices. During the analysis of systems, we minimize this kind of threat, by manual inspection of some code snippets and making sure there were no bad coding practices.

4. Evaluation with Developers

We chose fifteen systems with more warnings and report the results to the team of developers through pull-requests on GitHub. We obtained responses from 11 development teams. For the SlickGrid system, the development team said that use JavaScript code editing tools that already has an integrated static analyzer and that many of the errors found by our study are issues related to style and organization of code (Group B) and they modify the analyzer to not detect such problems. In two other systems, Jasmine and Knockout developers use JSHint with added specific rules for their projects. In addition, developers of Jasmine and Knockout systems will analyze the version tested by our study and examine the problems founded. In seven systems, developers said they are aware the warnings can become potential bugs, but they will not fix these problems right now because there are other priorities in the project. In addition, the developers of these seven systems said that many errors can be related to environment variables of third-party libraries that can be used in the project code without being declared, for example, variables and functions of the jQuery library. The developers of randomColor system not using said static code analyzers because the system has fewer lines of code (LOC 361).

According to the answers of the developers we realize that for large projects static analyzers are used. Futhermore, developers are more concerned with warnings of Group A (which can become potential bugs) than with warnings of Group B (stylistic problems). One developer answered that static code analyzers do not understand more advanced syntax of JavaScript properly. In addition, the developer suggested that static analyzers should have an option for users to be able to inform their level of knowledge in JavaScript.

"I use WebStorm for JavaScript editing, which has the lint/hint tools built in. I checked it out and there were indeed about 10 reasonably serious errors it picked up. However a lot of warnings these tools flag are the result of them not understanding the syntax of more advanced JavaScript properly. So they are useful to a point, but can be quite annoying when you have to trawl through a bunch of incorrectly flagged errors to find the one true error. This is what tends to put people off. I notice also that some errors being flagged in my project are in the jQuery and other libraries. This is a sign in itself. These libraries tend to use nonstandard but highly optimised ('guru level') code structures. It would perhaps be good if the Lint user could flag their level of expertise (newbie, competent, guru) to tell the tool that suspicious but correct looking code is probably not a mistake but the work of a guru, or to at least allow different types of structure in this case."

5. Related Work

Some works in the literature use JSHint and JSLint as auxiliary tools to check bad practices of JavaScript coding, but so we could not find any work comparing static analyzers of JavaScript code. DLint [Gong et al. 2015] presents a dynamic tool for checking bad coding practices in JavaScript and concludes that JSHint is a good checker and must be used in addiction with dynamic checkers. In this study they found that static tools report more warnings of Group B than of Group A and group B warnings are easier to solve. JSNOSE [Fard and Mesbah 2013] presents a technique for detecting code smells that combines static and dynamic analysis in order to find patterns in the code that could result in understanding and maintenance problems. [Politz et al. 2011] use a novel type system for JavaScript to encode and verify sandboxing properties, where JSLint is used in the verification process. FLIP Learning [Karkalas and Gutierrez-Santos 2014] present an Exploratory Learning Environment (ELE) for teaching elementary programming to beginners using JavaScript. They used rules addressed by JSLint to help programmers. [Horváth and Menyhárt 2014] also use static code analyzers to teach introductory programming with JavaScript in higher education.

6. Conclusion

In this study, we found that even in notably known JavaScript systems bad coding practices are found by JSHint and JSLint. We noticed that systems have a low amount of warnings due to the use of static code analyzers, since these tools are able to detect more than 150 different types of pitfalls. Besides, our study shows that both tools detected problems in groups A and B, and JSHint detected a greater number of warnings from Group A (20.12%) in the systems and JSLint a greater number of warnings of group B (93.76%). But that is not enough to say which tool has best rate to find bad coding practices since their results are similar in this study. Furthermore, according to the response of developers we proved that 10 of the 31 analyzed systems static analyzers are used. For future works the field of study for static analyzers for JavaScript code is large and there are several approaches that can be exploited as an analysis of which bugs that are fixed on JavaScript systems are related to the warnings pointed out by JSHint and JSLint tools. In addition, we can improve the static analyzers according to our results, improving other lint-like analyzers to find a larger number of warnings that can become potential bugs and that are more subtle to find in a naive inspection of source code. We also plan to evaluate the false positives raised by JSHint and JSLint following a methodology used in the evaluation of Java static checkers [Araujo et al. 2011].

Acknowledgments: This research is supported by FAPEMIG and CNPq.

References

Araujo, J. E., Souza, S., and Valente, M. T. (2011). A study on the relevance of the warnings reported by Java bug finding tools. *IET Software*, 5(4):366–374.

Cantelon, M., Harter, M., Holowaychuk, T., and Rajlich, N. (2013). *Node.Js in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Crockford, D. (2008). *JavaScript - the good parts: unearthing the excellence in JavaScript.* O'Reilly.

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

Gong, L., Pradel, M., Sridharan, M., and Sen, K. (2015). Dlint: Dynamically checking bad coding practices in javascript. In *ISSTA - International Symposium on Software Testing and Analysis*, pages 94–105.

Horváth, G. and Menyhárt, L. (2014). Teaching introductory programming with javascript in higher education. In *Proceedings of the 9th International Conference on Applied Informatics*, pages 339–350.

Karkalas, S. and Gutierrez-Santos, S. (2014). Enhanced javascript learning using code quality tools and a rule-based system in the flip exploratory learning environment. In *Advanced Learning Technologies (ICALT), 2014 IEEE 14th International Conference on*, pages 84–88.

Kovalyov, A. (2015). *Beautiful JavaScript: Leading Programmers Explain How They Think.* O'Reilly Media, Inc., 1st edition.

Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. (2011). Adsafety: Type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA. USENIX Association.

Silva, L., Ramos, M., Valente, M. T., Anquetil, N., and Bergel, A. (2015). Does JavaScript software embrace classes? In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 73–82.