Why and How JavaScript Developers Use Linters

Fjóla Tómasdóttir, Kristín; Aniche, Maurício; van Deursen, Arie

Publication date
2017

Document Version
Accepted author manuscript

Published in
32nd IEEE/ACM International Conference on Automated Software Engineering

Citation (APA)
Fjóla Tómasdóttir, K., Finavaro Aniche, M., & van Deursen, A. (2017). Why and How JavaScript Developers Use Linters. In 32nd IEEE/ACM International Conference on Automated Software Engineering.

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Why and How JavaScript Developers Use Linters

Kristín Fjóla Tómasdóttir, Maurício Aniche, Arie van Deursen
Delft University of Technology - The Netherlands
k.f.tomasdottir@student.tudelft.nl, {m.f.aniche, arie.vandeursen}@tudelft.nl

***Abstract*—Automatic static analysis tools help developers to automatically spot code issues in their software. They can be of extreme value in languages with dynamic characteristics, such as JavaScript, where developers can easily introduce mistakes which can go unnoticed for a long time, *e.g.* a simple syntactic or spelling mistake. Although research has already shown how developers perceive such tools for strongly-typed languages such as Java, little is known about their perceptions when it comes to dynamic languages. In this paper, we investigate what motivates and how developers make use of such tools in JavaScript projects. To that goal, we apply a qualitative research method to conduct and analyze a series of 15 interviews with developers responsible for the linter configuration in reputable OSS JavaScript projects that apply the most commonly used linter, ESLint. The results describe the benefits that developers obtain when using ESLint, the different ways one can configure the tool and prioritize its rules, and the existing challenges in applying linters in the real world. These results have direct implications for developers, tool makers, and researchers, such as tool improvements, and a research agenda that aims to increase our knowledge about the usefulness of such analyzers.**

I. Introduction

Automatic static analysis tools (ASATs) are used to automatically examine code and look for defects or any issues related to best practices or code style. These tools aid in finding issues and refactoring opportunities early in the software development process, when they require less effort and are cheaper to fix [1], [2]. Therefore, ASATs have become commonly used by software development teams, and particularly in JavaScript projects [3].

JavaScript indeed has become a very popular language in the last years and in fact has been the most commonly used language on GitHub since 2013 [4]. One of its most remarkable characteristics is its dynamicity. As an example, the language allows for generating new code during runtime execution and for dynamic typing where variables do not need to be declared before they are used. Partly due to its dynamic nature, the language is considered to be error-prone where it can *e.g.* be easy to introduce unexpected program behavior with simple syntactic or spelling mistakes, which can go unnoticed for a long time [5], [6], [7]. A linter is a a type of tool that performs static analysis of source code, which can be especially useful for JavaScript to detect these types of mistakes.

Several studies have focused on static analysis in JavaScript since ASATs have different kinds of requirements for dynamic languages than for static languages [8], [9], [10]. Others have studied how developers use general ASATs and how they perceive them [11], [12], [13], [3]. It is important for both researchers and industrial tool makers to know how developers use these tools to provide the appropriate context and requirements for future development and research. Until now, no studies have concentrated on how developers perceive ASATs specifically for a dynamic language such as JavaScript. As JavaScript is very different from other commonly studied programming languages, such as Java, we expect different motivation and behavior from developers.

In this study, we aimed at understanding how and why developers use linters in real world applications, where we examine the usage of ESLint [14], the most commonly used linter for JavaScript[1]. We investigate the benefits of using such a tool and what could be done in a better way. Furthermore, linters need to be incorporated to the development process and configured appropriately for a project to reduce the amount of perceived false positives out of the large volume of warnings that are often reported [2], [16], [12]. We investigate what methods developers use to configure this tool and how they maintain those configurations. To that goal, we used a qualitative research method [17], [18] to conduct and analyze interviews with 15 developers from reputable open source software (OSS) projects that have been actively involved with enabling and configuring ESLint in those projects.

Our main results from this study consist of in-depth knowledge and actual use cases that can benefit other developers, tool makers and researchers. More specifically, the contributions of this study are the following:

1) We describe the different benefits JavaScript developers obtain when using a linter. With that information, we present to other developers motivation on why using a linter can be valuable for them and provide tool makers with information regarding what type of linter functionality is the most important to its users.
2) We gather various methods regarding how developers prioritize rules and how they decide which rules to include in configurations for a project. This information can be used to help developers in using a linter and to guide them in creating a suitable configuration.
3) We collect several challenges that developers face when using linters in the real world, providing tool makers with advice for future tool improvements and researchers with opportunities for further research.

The remainder of the paper is organized as follows. We start by providing some background information on linters for JavaScript in Section II. The methodology of the study

[1]According to the number of npm downloads [15].

is described in Section III with the derived results in Section IV. We examine related work in Section VI and provide a discussion with implications in Section V. Finally we go over the limitations of the study in Section V-C and conclude the paper in Section VII.

II. Background: Linters for JavaScript

Well known and much researched static analysis tools include FindBugs [19], CheckStyle [20] and PMD [21]. These tools all have different focus. For example, FindBugs finds numerous defects, such as infinite recursive loops and references to null pointers, CheckStyle is focused towards following coding standards, and PMD detects both code style and defects. JavaScript linters work in a similar fashion where the best known and most popular ones include ESLint, JSHint [22], JSCS[2] [23] and, the first linter created for JavaScript, JSLint [24].

ESLint is the newest of these and has gained much popularity in the last two years in the JavaScript community. ESLint was designed to be an extremely flexible linter, where it is both easily customizable and pluggable. ESLint provides a number of 236 base rules[3], grouped in the following categories: *Possible Errors, Best Practices, Strict Mode, Variables, Node.js & CommonJS, Stylistic Issues and ECMAScript 6*. Example rules include *no-eval*, which disallows the use of the notorious eval function in JavaScript [25], and *indent*, which enforces consistent use of indentation. The description of each rule and category can be found at the tool manual [14]. Developers are required to specify which of these 236 rules should be turned on or use publicly available presets, as none of these rules are enabled by default. A preset is a set of rules that are made available to the public, such as the ones from Airbnb [26], Standard [27], or even the ESLint's recommended settings.

ESLint was chosen as the linter to be analyzed in this study as it is the most commonly used linter in the JavaScript community and offers the greatest amount of functionality and flexibility out of all well known linters, thus not excluding nor focusing on any specific type of linting such as only analyzing styling issues or solely identifying possible errors.

III. Methodology

We followed a qualitative research approach in our study [17], inspired by many concepts of classic Grounded Theory [18], [28] where the aim is to discover new ideas emerging from data instead of testing any preconceived research questions. With an open mind we wanted to understand how and why developers use a linter for JavaScript. For that purpose we collected data by conducting 15 interviews with developers from reputable JavaScript projects on GitHub. We explain the process of conducting these interviews in Section III-A. The interview recordings were manually transcribed and analyzed with continuous *memoing* and *coding*, which is further described in Section III-B. Finally, we detail our participants in Section III-C.

[2]JSCS is no longer supported and the maintainers have joined forces with ESLint as of April 2016.

[3]As of release v3.13.0 in January 2017.

A. Interview Procedure and Design

The interviews were conducted in a semi-structured fashion as it is commonly done in software engineering research [29]. With this method specific questions are combined with open-ended questions to also allow for unexpected information in responses. Hove and Anda [29] describe the experience of performing interviews in software engineering research and encourage interviewers to talk freely, to ask relevant and insightful questions and to follow up and explore interesting topics. These guidelines were followed while performing the interviews. Each interview was built upon a list of 13 standard questions but were also dynamic where follow up questions were asked based on the participants' replies.

The questions were designed to get as much information from the participants as possible in a relatively short time. To begin with they were asked broad questions which often provided an opportunity for further discussion. Example questions include: *Why do you use a linter in your project?* and *How do you create your configuration file and maintain it?*. Other questions were more specific, such as: *Do you experience false positives? if so, which?*. The complete list of questions is available in the technical report [30].

Interviewees were asked to participate in an online video call. The interviews were recorded with permission and lasted from 16 to 60 minutes, with an average duration of 35 minutes. Three out of the 15 participants were not able to participate in the call and instead received a list of questions via email and gave written responses.

B. Analysis

Continuously after each interview, *memoing* was done to note down ideas and to identify possible categories in the data. The interview recordings were then ultimately manually transcribed. As parts of the interviews included casual chat about the topic, some irrelevant information along with repetition was left out and some parts were summarized. First, we performed *open coding* where the transcripts were broken up into related sentences and grouped together into the two main topics that drove our interviews (why and how developers use linters). During this process, we observed developers addressing many of the challenges they face when using linters, and therefore we decided to promote this topic. Secondly, we performed *selective coding* where more detailed categories were identified which became the topics we present in the Results section. In this process we took advantage of the memos that had been written over the course of conducting the interviews. The complete list of codes can be found in the technical report [30].

C. Participants

In order to find potential participants for the interviews we examined the most popular JavaScript projects on GitHub, according to their number of stars in December, 2016. We

conjecture that by observing the top projects on GitHub we can obtain an insight into active and reputable projects with many contributors, providing more interesting and relevant information about the usage of linters. We detected those who 1) use ESLint, 2) have some custom configuration (*e.g.* not only using a single preset) and 3) where one or two contributors could be identified that have been more involved than others in editing the configuration file. We then sent an e-mail to one or two main contributors of the ESLint configuration file of the corresponding project, briefly explaining the purpose of the study and asking for a short interview. These requests were sent out in batches of 5-10 emails (starting with the most popular projects) as it was difficult to predict how many positive replies we would receive. Batches were sent out until we had received a sufficient number of positive replies back, where the goal was to perform at least 10 interviews, or until we were satisfied with the amount of information we had collected.

A number of 120 projects were eventually examined where 37 requests were sent out. These resulted in 15 interviews being performed, thus with a response rate of 40%. The information from these 15 interviews was considered enough to provide us with *theoretical saturation* [18]. Table I shows the developers who participated in the interviews where, in order to keep the participants' anonymity, they are given names starting with the letter P and followed with a number from 1 to 15. The amount of months for which each corresponding project had used ESLint is also displayed[4], where most projects had migrated from using another linter such as JSHint. The table furthermore shows the placement of the projects in the top 120 JavaScript projects on GitHub within a range of 10 projects each, also to maintain the participants' anonymity. A summary of the participants' experience is shown in Table II where the average experience as a professional software developer was 11.8 years. Among the 15 participants, four are founders of the project, seven identified themselves as lead or core developers and four are project maintainers.

TABLE I
ALL PARTICIPANTS' CODENAME, AMOUNT OF MONTHS USING ESLINT IN THE CORRESPONDING OSS PROJECT AND THE RANGE FOR THE PROJECT PLACEMENT IN THE TOP 120 JAVASCRIPT PROJECTS ON GITHUB

Code	Months	Placement
P1	25	11-20
P2	22	11-20
P3	5	21-30
P4	14	21-30
P5	8	31-40
P6	7	41-50
P7	1	61-70
P8	23	71-80
P9	5	81-90
P10	3	81-90
P11	4	91-100
P12	16	91-100
P13	15	111-120
P14	24	111-120
P15	22	111-120

TABLE II
EXPERIENCE OF PARTICIPANTS, SHOWING THE LOWEST AND HIGHEST ANSWERS ALONG WITH THE AVERAGE OF ALL ANSWERS

	Low	High	Average
Years as developer	3.5	27	11.8
Years as JS developer	1.3	20	8.9
Years in project	0.6	5	2.7
Project age	1	8	5.1

IV. RESULTS

In the following we present our results on why and how JavaScript developers use linters, along with the challenges that they face.

A. Why do JavaScript developers use linters?

1) Prevent Errors: Using a dynamic language such as JavaScript is not free of risks: "*Without a linter [JavaScript] is a very dangerous language. It's very easy to make a very big problem and then spend 30 minutes to find it.*" (P7). The majority of the participants reported that the number one reason as to why they use a linter is to catch possible errors in their code: "*There are things which are easy mistakes to make and are obvious errors and I think those provide the highest value. Because you have a one to one correspondence between times that a rule catches something and bugs that you've prevented.*" (P4). More explicitly, when asked about the most important category of warnings in ESLint, 10 participants answered that Possible Errors was the most important one (P1, P2, P3, P4, P5, P7, P8, P9, P11, P12): "*Possible Errors is the #1 most important, the biggest reason to use a linter is to catch errors the programmer missed, before they become a runtime bug.*" (P9). So not only does it catch many bugs but it also does so early in the development process: "*If you can get some bugs away from your code so early as when you write it, it's great.*" (P12). These rules can also be especially useful for bugs that are hard to find and to debug (P15). An example of this situation is using two equal marks when three should have been applied, which can cause substantial unpredicted problems but the cause can be very difficult to find (P8).

A special category of bugs in JavaScript has to do with the declaration of variables because of the dynamic nature of the language. Indeed, there is a specific category in ESLint that only contains rules that have to do with variable declarations and errors regarding variables (appropriately named Variables): "*Possible Errors will catch a lot of unintended behavior, and Variables will catch a good deal more.*" (P5). Two participants reported that Variables was the most important category (P10, P15). When a developer *e.g.* mistypes a variable or uses the wrong variable name, the linter can catch it and warn the developer: "*It's very easy to write JavaScript code that has errors, you might use a variable that hasn't*

[4]As of February 2017.

been declared or you might have a typo in your variable name and because JavaScript is often not compiled, you'll only discover that much later when you run the code." (P1). More specifically, nine participants (P1, P3, P4, P5, P8, P10, P12, P13, P15) mentioned the importance of the rule *no-unused-vars* (identifies variables that have been declared but never used) and five (P1, P3, P5, P13, P15) mentioned *no-undef* (identifies variables that have not been defined) which are both useful to identify mistyped variables.

2) Augment Test Suites: While linters are being used to catch errors in code, there is another popular and widely accepted method to catch bugs which is to write unit tests. It is therefore interesting to know how these two methods are combined for this purpose. Some participants mentioned that they use a linter on top of unit testing as a complementary approach to the regular tests (P1, P3, P8). P1 and P8 pointed out that unit tests commonly do not cover all code which can result in problems being easily missed: "*You need to seek all possible cases for unit tests, but sometimes it's very hard, and of course in all projects, unit tests don't cover all possible cases. So this is why a linter is a second protection line.*" (P8). In some cases a developer might have written tests for new code but then makes a final refactoring or clean up and forgets to update the tests as well: "*Definitely I've done that before and the linter has helped me catch those errors beforehand.*" (P3). Furthermore, the tests can also take substantial time to run and thus the linter can be seen as a much faster version of smaller subtests (P1).

On the other hand, participant P4 believes that unit tests and manual tests can usually cover all errors, so even though ESLint would not be used, the errors would eventually be caught by the various tests that are applied. However he says that the linter can catch them earlier in the process and is also better at identifying code that is ambiguous.

3) Avoid Ambiguous and Complex Code: It can be difficult to understand code correctly where the intention is not perfectly clear. The category Best Practices tries to tackle this problem where, according to its documentation [14], it contains rules that relate to better ways of doing things to help developers avoid problems. While only one participant recognized the category to be the most important one (P6), others identified it as the second most important (P4, P8, P13, P15). Some of these rules try to prevent code from being misunderstood: "*It helps enforce code which says what it does, so that it's easy to understand.*" (P4). In some cases code is actually doing something else than it appears to and a linter can help to detect these situations (P2, P4, P6). One example of this is restricting the usage of switch statements by forbidding the use of "fall throughs"[5]. This is done because the intention of switch statements can be easily misunderstood when that feature is used (P4). Another example is where the linter identifies code that is unreachable (rule *no-unreachable*). According to P2, not only can it catch possibles mistakes by a developer but it can also help with removing a lot of unnecessary code that otherwise makes the codebase more difficult to understand and where the intent of the code can be unclear. Moreover, four other participants mentioned that this is indeed an important rule to use (P1, P3, P6, P15): "*Having code in your app that's never going to run sounds like the worst idea ever.*" (P1).

4) Maintain Code Consistency: Every single participant mentioned that one of the reasons why they use a linter is to maintain code consistency. Having a consistent code style in a project is beneficial for many reasons, one being that it improves the readability and understandability of the code. As an example, P10 reported that inconsistent code, such as having different spaces and semi colons, makes the code very difficult to read and understand since in those cases these inconsistencies consume all his attention. This might even be especially relevant in the case of JavaScript since it is a language where the developer has substantial freedom in how to write the code (P12, P14): "*With JavaScript you can write code in many ways, and it can be hard to read other people's code if you write it in a different way.*" (P12).

This topic relates mostly to the category Stylistic Issues where there are many different rules available to enforce specific code styles. Even though every participant mentioned this matter in the interviews, it does not seem to be of high priority for them. When participants were asked which category of rules they thought were the most and least important ones, two considered Stylistic Issues to be the most important category while 10 thought it to be the least important one.

Some participants were bothered by the fact that choosing which style to follow is a very subjective decision and developers generally have very different opinions on how code should be written (P1, P3, P5): "*Stylistic Issues - they're all opinion based.*" (P5). On the other hand, four participants explained that Stylistic Issues was indeed the least important category simply because other categories can catch bugs which is far more important (P2, P4, P9, P13). This category thus still provides a lot of value and they would not want to omit it: "*They make it [the code] harder to read but they just don't cause issues as much.*" (P13).

5) Faster Code Review: In order to uphold code consistency, project maintainers can monitor and review new code that is proposed. GitHub projects commonly make use of pull requests to submit new code where other developers can review the changes and write comments on them. There are several aspects to consider when reviewing pull requests, such as functionality and code style. Several participants mentioned that they use the linter to avoid having to manually review the code style in pull requests (P1, P2, P3, P4, P8, P11, P14, P15): "*Any software project wants to maintain some bar of quality and many of the ways that we assess that are going to need human intervention, but there are a subset of those problems that a piece of software has which can be done by a computer and as engineers I think we are apt to try to use computers to solve human problems where possible. You can free up human time to do more interesting things.*" (P4). Furthermore, it saves

[5]A "fall through" occurs when all statements after a matching case are executed until a break statement is reached, regardless if the subsequent cases match the expression or not.

time for the contributor of the pull request since he or she receives much faster feedback from a linter than from a person that would conduct a review (P4).

Maintaining code consistency with a linter can also make pull requests much easier to review. When there is a set of stylistic rules in a project to which everyone has to conform, all pull requests have minimal stylistic changes. If there are no rules, there can be multiple code changes of *e.g.* only whitespaces or line formatting which might be caused by different editors being used. This can make it difficult to see the actual changes that were done in the contribution since they are hidden by these formatting changes (P3, P12).

6) Spare Developers' Feelings: When receiving comments from a code review, developers can sometimes be sensitive to criticism (P2, P8, P11). This can particularly be the case for new developers: "*If you tell to a new developer that he or she made a mistake, it will be very sensitive. He may feel very uncomfortable because somebody found a mistake in his work. But if a linter tells you about a mistake, it is not a problem, because it's not personal.*" (P8). A new developer might also look up to the person that is conducting the code review which can make the criticism especially dispiriting (P2). Having a linter doing this job can also contribute to people feeling more equal in a project if there is no senior person telling others to do things differently (P11).

7) Save Discussion Time: Having a set of rules regarding code style can also save time that is spent on discussing different styles and opinions (P2, P4, P5, P6, P7, P10). In big projects with many contributors there can be many pull requests in circuit and discussions can occur where developers disagree on a certain style that is used. P2 explains that discussing code styles is not worth the effort when there are other more important things to discuss. He further describes that comments regarding code style on pull requests can be different depending on which developer is conducting the review. In some cases, contributors can therefore receive contradictory advice if no rules exist that everyone goes by.

The discussions about code style that can occur in pull requests or in issues can also even lead to arguments between people since developers have very different opinions on the matter (P1, P2, P3, P5). All this can be avoided by deciding upon a set of rules to begin with: "*It's almost like a code contract. There may be things that each of you have assumed and you don't know what your own assumptions are, and what could possibly lead to conflict down the road, so you have a written contract to try to address everything up front.*" (P7).

8) Learn About JavaScript: ESLint can be used to learn about new JavaScript features or new syntax. P12 used ESLint in helping him to learn the new syntax of ECMAScript 6 (ES6): "*When I switched to ES6, I used it as an educational tool. It's so easy to continue to use var for variable declarations. I used ESLint very strictly to enforce ES6 syntax, otherwise I would probably still use ES5 when I write code. But with the help of the linter it forces you to switch to ES6, which is a good habit.*" (P12). He used a custom rule set for React [31], a JavaScript library for building user interfaces, in a similar fashion when using it for the first time. He was notified by ESLint how the React code could be written in a better way, not just regarding formatting but also so that it would be better for execution. With that he learned how to write both correct and efficient React code.

Even though linters can be beneficial to all JavaScript developers, they can be especially helpful for new developers, either those who are new to a project or those who are new to programming in general (P6, P7, P9, P13, P14). Contributors in OSS projects usually have different levels of experience and using a linter can help with "*leveling the playing field and helping people to understand what's actually going on*" (P13). This particular example came from a developer that had been working with students who were accustomed to getting errors from the Java compiler, telling them what they can and can not do. However when using JavaScript, one can run code that includes various coding mistakes and not get notified about it (P13).

Some features of JavaScript can also be used in clever ways with enough knowledge of the language. However, P7 explains that, without sufficient knowledge, those features can be used in the wrong way and lead to unwanted behavior. He exemplifies: experienced developers can use a double equality with perfect intentions, but a newcomer in JavaScript might use it by accident, not knowing that a triple equality should have been used instead. ESLint indeed has a rule that warns developers about the usage of two equal marks (rule *eqeqeq*), and many other rules that deal with other possible mistakes.

Having non ambiguous code and a consistent style can also be even more helpful for new developers (P6, P9). Having consistent code can make it easier for new developers to read the codebase and get up to speed quickly. Using a linter "*makes sure it's the same throughout so that when other people come in and want to add a feature, they're not really confused for what they should be doing. It's easy to follow.*" (P6).

B. How do JavaScript developers configure linters?

1) Use an Existing Preset: There are many publicly available presets that anyone can use in a project instead of creating a custom configuration, or use as a part of a custom configuration. These presets have been carefully constructed by their creators and have been changed attentively over time: "*They thought about the code standard quite extensively and put a lot of thought in it.*" (P12). Several participants like to use a preset as a part of their configuration file (P1, P6, P10, P12, P13, P15) and one normally tries to solely use the preset as is (P8).

2) Project Fit: It is important that the stylistic and best practice rules fit the existing code when the rules are chosen (P3, P4, P6, P12): "*I wanted them to fit the code as it was, I wanted the linting in place with as little mess as possible.*" (P12). P4 explained that if something is already being done in a project, it is rather straightforward to enable a rule that ensures that it will continue to be done the same way in the future. Furthermore, if there is already some sense of style in the existing code, it is not very sensible to change it to

something else since it would create more work than necessary when setting up a linter. As an example, P4 mentioned that if a preset contains a rule that enforces either spaces or tabs and the setting is opposite to what has commonly been done in the project, the particular rule will be overwritten to fit better to the project. Otherwise, multiple stylistic changes would need to be done to conform to the rule.

For rules that relate to possible errors, P4 discussed a similar approach where he considers whether that particular rule will be useful for the project or if they will need to disable it in multiple locations in the code. If it needs to be disabled frequently, it is not worth it to enforce it.

3) Automatically Generated: Extending the last approach, ESLint provides an automatic method to insure that a configuration fits well to a specific project. The source code is inspected to detect a common code style and the user is asked a few questions, and from this a suitable configuration is created. Two interviewees used this method to create their configuration file (P11, P14). P11 then looked over the generated rules and the errors in the output to see if he agreed with them. He does not consider it to be wise to use a preset since linter configurations are generally very project dependant: "*I didn't even consider Airbnb or Google because I think every project is a little different.*" (P11).

4) Pull Request Discussions: Three participants (P2, P4, P10) reported that when something is discussed in a pull request that can be enforced with a rule, they use the opportunity to enable the corresponding rule. Since the topic surfaced in the pull request then there was obviously a need to make a decision on the matter being discussed, whether it is a code style or a best practice. Furthermore, that way the topic will not surface again and time will not be spent on discussing it, as it will be already decided on in the configuration file (P2).

5) Minimal Configurations: Some prefer to keep the configuration as simple and minimalistic as possible (P1, P5, P8, P15): "*We don't want people to feel like they have to jump through unnecessary hoops to get their PR's in, so turning on every single thing wouldn't be great.*" (P1). Furthermore, P8 thinks that if too many rules are enabled in a project, people will not trust the configurations: "*They will think that it is a bureaucracy and that it's not important.*" (P8). Both P1 and P8 prefer to only enable rules that can prevent errors.

6) Effort of Enforcing a Rule: Participant P15 described that he commonly enables a set of rules, *e.g.* a known preset, and then sees how it works out for the project. If some of the rules are starting to be bothersome for the project, *e.g.* needing to be disabled with inline comments or if too much refactoring is required to fit the rule, it is permanently disabled: "*Just start to use it and see how much pain it causes, where it's beneficial. But usually it's turning things off when it's apparent that it's creating more effort than it really helps.*" (P15). He describes the process as a feedback cycle where it is important that contributors agree on the rules that are used: "*The disagreement between people is very important, you have to get everyone on the same page.*" (P15).

7) Most Voted Style: Participant P7 reported that in a new project he would most likely go with the code style that is the most common one amongst the developers in the team, consequently adding rules that enforce that style. Generally when working with a new team, the first discussion he often has with them is which code style people are used to.

8) Consistent Rules: Lastly, some developers do not care very much about which rules are actually enabled (P2, P6, P10): "*I almost don't even care what the rules are, I have some opinions, but I'm much more interested in there just being consistent rules, than having a point of view about any particular rule.*" (P10). There just has to be some fixed set of rules to enforce consistency and to prevent unnecessary time being spent on discussions: "*Having a linter forces us to make choices, even if it's arbitrary choices in some situations.*" (P2).

C. What are the challenges faced by JavaScript developers when using linters?

1) Create and Maintain Configurations: Several participants mentioned that it was challenging to create the original configuration or to keep it updated (P1, P3, P5, P8, P12). Only two participants reported that they had read over all available rules when they originally created the configuration file (P3, P9). This involves evaluating a set of 236 rules that ESLint has available which can be a tedious process: "*Most of it was read through every rule, it was kind of a very painful process to set it up.*" (P3). Another participant used very similar wording when it came to setting up the tool: "*Sometimes a pain to set up in your editor with the right configuration.*" (P5). Meanwhile, the other participant that manually created the configuration file claimed that it was actually quite easy to set up (P9).

Others have been frustrated with keeping their configurations up to date after they have been created, especially when using presets (P1, P8, P12). When the presets are updated, there are often new rules that are enabled or older rules that are changed which can cause a high volume of new warnings or errors (P12). P12 explained that these changes are sometimes beneficial and he happily fixes the warnings, but at other times the change is not useful and they have to be overwritten. Moreover, P1 discussed that it can be frustrating when the presets update very frequently and the code has to be changed often because of it. However he also likes that the presets keep him updated of new rules regarding new JavaScript features.

2) Enable Rules in Existing Projects: Six participants (P1, P3, P4, P7, P10, P13) mentioned that it can be difficult to start using a linter or to enable new rules in an existing codebase. If the rules that are enabled cause many warnings or errors to occur, substantial effort is needed to go over all existing code to fix every reported instance. There can even be such a large volume of existing code that it would simply take too much time to review: "*The problem with [the project] is that it's really old and big, so for that we don't have the luxury of turning on whatever rules we want to because we're never going to be able to update a lot of our code to support them.*" (P1).

Even if it is possible to fix all warnings, it can be risky to change old code to conform to these rules since it is easy to introduce new bugs in the meantime (P3, P7, P10). Knowing the original intent of the code can be very difficult and can take considerable time to try to understand (P7). P7 discussed an example where there were many instances of two equal marks being used in the code instead of three, which resulted in many errors from the linter. In these cases it could have been intentionally written so or by accident, which would need to be carefully verified and tested. The risk and effort of going back and changing the old code might therefore not be worth it: "*Cleaning up for just clean up, just to enable it, I think is risky. I've been bitten by that a couple of times.*" (P3). Enabling a linter in a project to begin with can be a lot easier: "*A linter is great if you start with that and you enforce all those rules and the idea is that you will never run into ambiguous code.*" (P7). However with older projects it can be more difficult and even dangerous (P7).

In large existing projects with many collaborators it can also cause conflicts and frustration when new rules are enabled (P4, P13). P4 was working at a company with around 60-70 developers, all working on the same code every day. Usually there are multiple pull requests open at the same time and there will be many merge conflicts when a new rule is suddenly enabled, since it will likely affect code everywhere in the project. In P13's project it caused frustration with people to enable many rules all at once since the developers were not accustomed to them before. Instead, it was decided to do it slowly by enabling only one rule at a time to give developers a chance to get used to the new rules.

3) Agree on Rules in an Industrial Setting: In an open source project it is relatively easy to build consensus around what people want to do (P4, P15). If someone wants a new rule to be enabled, then only a few main contributors need to say yes and it is decided upon. Other maintainers usually follow what the project leaders propose (P4, P15). In a business environment it can be a very different case where everyone in the team has a say in the matter: "*There are 60 people with their own opinion about how code should be written.*" (P4). P4 further explains that it can be especially difficult to introduce rules on stylistic issues since it is not considered to be important and it can be hard to justify why you would inconvenience people to adhere to new rules: "*It may create more tension than it does actually solve problems*." (P4).

However, the power of the lead developers in open source projects can also be a negative factor. If a contributor wants to enable a new rule, it often only needs one reviewer to say no, so that it will not be accepted (P2). In that case the pull request might get rejected or it has to go through the core technical team for a discussion which might not be worth the trouble, only to enable a new rule (P2).

4) Enforce Developers to Follow the Rules: When a rule is configured in ESLint there are three settings that can be applied: *off*, *warn* and *error*. The rule is turned off when applying *off* and will have no effect whatsoever. When a rule is set to *warn*, each instance of the rule being broken will appear in the output when running the linter. Lastly, *error* will have the same effect as *warn* except it gives an exit code of 1, meaning it can break the build when ESLint is a part of the build process.

The majority of the participants used only or mostly errors and rarely warnings. These settings are used by the participants for different purposes, such as indicating the criticality of a rule (P15) or by using warnings as an adaptation period when enabling new rules (P4, P7, P13). Other participants liked to use warnings in the development process so that their build would not be interrupted when working on unfinished features (P9, P12, P15).

There is however a problem with using warnings: multiple participants claimed to use only errors since warnings would simply be ignored by developers (P2, P3, P4, P13). In addition, according to these participants, when warnings live for a long time in the codebase, people will start to devalue them and leave them behind. Developers do not feel any responsibility for the warnings and might think "*there were some warnings when I checked it out, not my job to fix it, I'm going to check it back in with warnings.*" (P4). P4 further explains that especially if there are many warnings, developers will not even read them and simply leave them behind. To enforce removal of the warnings, they therefore have to be set as errors that actually prevent a build to succeed.

5) Dynamic Features of JavaScript: JavaScript has been described as harsh terrain for static analysis because of its dynamic nature [32]. Static analysis for JavaScript has thus been critized and said to be limited since it can not account for runtime behavior [33]. The majority of the participants would indeed like ESLint to be able to do more, but they however think that the current version is very acceptable since they choose themselves to work with a dynamic language (P2, P5, P6, P7, P8, P9, P10, P11, P15). When P9 was asked whether he misses dynamic analysis from the linter he replied: "*Of course, but I don't particularly think that is ESLint's fault, so much as a language defect. Due to the dynamic nature of JavaScript, static analysis ranges from extremely difficult to impossible. I don't expect ESLint to be able to fix that.*" (P9). Regarding type checking, there are tradeoffs in choosing JavaScript as a programming language and there is a reason why some languages are not strictly typed (P2, P15). If static typing is something that a developer wants, he or she should rather use TypeScript or some statically typed language (P2, P10, P15): "*Each developer has to make a decision on if they want to work in a typed language or in an untyped language and the tradeoffs of that kind of lead you to the path of what tools can provide.*" (P15). The majority is therefore quite satisfied with what the linter can do and do not expect it to be able to detect more of the dynamic parts of the language.

Other participants were more bothered by the fact that a linter can not analyze the dynamic parts of the language, where the problem was mostly centered around the lack of variable types (P3, P4, P13, P14). P3 reports that he has spent substantial time on testing various mix-ups with strings and numbers, for which he would either like ESLint to warn about

types that change or would like to switch from using regular JavaScript to TypeScript. Many large companies have indeed exerted a lot of effort to try to solve this problem with projects such as TypeScript and Flow [34] (P4).

V. Discussion

A. Implications

The results have several implications for developers, linter creators and researchers, which we discuss in the following. While the findings are based on the usage of linters for JavaScript, they might also apply to other ASATs.

1) Developers: The results provide motivation for developers to use a linter and help them in using the linter for it to be the most beneficial.

i) Developers can find numerous benefits in using a linter that motivates future usage of such tools (IV-A). Using a linter: a) on top of tests detects more bugs (IV-A1, IV-A2), b) helps avoid ambiguous code (IV-A3), c) helps maintain consistent code (IV-A4), d) makes code review faster and easier (IV-A5), e) spares newcomers' feelings when making their first contribution (IV-A6), f) saves time that goes into discussing code styles (IV-A7) and g) helps developers to learn new language syntax and features and with onboarding newcomers to projects (IV-A8). The findings by Beller *et al.* [3] also suggest that developers need to be made aware of the benefits of using ASATs.

ii) Developers can find several ways to prioritize rules (IV-B). We provide many different ways to choose the appropriate rules for a project, such as using presets, fitting the rules to the existing style or using discussions in pull requests as input.

iii) Developers should include a linter from the beginning of a project (IV-C2). Enabling a linter at a later stage can involve substantial effort and risk. Additionally, Ayewah *et al.* [11] found that FindBugs users are less likely to fix warnings that apply to older code than to new code.

iv) In existing projects, developers should enable rules carefully and incrementally (IV-C2, IV-C4). This should be done to minimize the risk of introducing bugs and the risk of creating frustration within a development team. Furthermore, by incrementally introducing rules and continuously fixing the corresponding warnings, the amount of warnings can be kept to a minimum, making developers more likely to examine and fix them. Previous research has also reported that many ASATs output too many warnings which makes it more difficult to use these tools [12], [16], [35].

2) Tool Creators: The results help linter creators to improve future tool editions by understanding the needs and challenges of developers when using linters.

i) Tool creators should focus on detecting possible errors (IV-A1). This is the most important aspect to the participants, especially errors that have to do with variable declarations.

ii) Tool creators should offer functionality to maintain code consistency and have customizable rules (IV-A4). This was considered useful by all participants but as developers have different opinions on which code style to use, tools should to be flexible.

iii) Tool creators should focus on dynamic languages (IV-A1, IV-A4). Linters can be especially useful for this type of languages as errors can be easily introduced because of the dynamic typing in JavaScript. Furthermore, as JavaScript code can be written so freely and in many different ways, it can be even more valuable to maintain code consistency. Beller *et al.* [3] also found that dynamically typed languages seem to benefit more than static languages from the usage of ASATs.

iv) Preset maintainers should not update them too frequently (IV-C1). Developers that use the presets can get frustrated when they have to do frequent changes to their code due to the presets being updated.

v) Tool creators should find other ways to make developers feel responsible to fix warnings (IV-C4). As it can be difficult to enforce linting rules without making them break the build, other methods would be beneficial to encourage developers to fix warnings. This could be done *e.g.* by linking the output of the tool to the project repository, to make a somewhat more personal connection to the developers that are responsible. Sadowski *et al.* [36] had a similar experience when introducing ASATs at Google, where developers would ignore warnings that did not cause the build to fail.

3) Researchers: The results offer ample opportunities for further research into these findings, where various methods can be applied to verify them in larger scale. Some research directions include the following:

i) Balachandran [37] conducted a study where ASATs were applied on code changes that were subject to code review, and asked developers whether they agreed with the warnings reported by the tool. Similarly, Panichella *et al.* [38] studied whether warnings were being removed in code reviews, both in projects that use ASATs and for some that do not. Both results indicated that code reviews would benefit from using ASATs by automating some of the work that developers perform. Research needs to be conducted to understand the effects of linters during review of JavaScript code (IV-A5) and where which warnings developers more often remove from source code may provide us with hints on which rules are considered important for developers (IV-B).

ii) Steinmacher *et al.* [39] studied the barriers that newcomers face when making their first contribution in an OSS project. Social barriers that were discovered include late responses and negative or even impolite feedback. Technical barriers include bad code quality, code complexity and lack of code standards. One could do research to see whether newcomers face less of these barriers in projects that use a linter, as the linter performs

part of the code review with a non subjective feedback (IV-A5, IV-A6) and also helps with maintaining better code quality (IV-A3, IV-A8).

iii) Beller *et al.* [3] studied the usage of ASATs and in particular how they are configured. Since several participants claim that it is difficult to enable linters in existing code (IV-C2), it would be intriguing to further research how configuration files in projects that enable a linter early on differ from those in projects that enable a linter at a later stage. Furthermore, many participants explained that they try to create the configuration so that it fits the project in the best way possible (IV-B2). It would be valuable to know how thoroughly projects follow these configurations, providing insight into how well the configuration reflects the project style and how much developers care about upholding these code standards.

B. False Positives

A common problem with static analysis tools is the high volume of false positives [12], [40], [41], [42]. Studying the usage of ASATs, Johnson *et al.* [12] found that the presence of false positives is indeed one of the largest barriers in using such tools. Opposed to previous literature, the majority of the participants in our study reported that they do not in general experience false positives while using ESLint (P5, P6, P9, P10, P11, P12, P14, P15). Three participants mentioned that they had experienced some false positives in the past which have now all been fixed (P5, P12, P14). Moreover, P14 only experienced a false positive once where he then reported the issue which was fixed in a matter of a few hours. Two participants claimed to experience somewhat frequent false positives (P10, P13). However, in both cases they blame it on other elements than ESLint itself. In fact, P13 migrated from JSLint and JSHint to ESLint because he perceived much fewer false positives with ESLint.

This difference to previous literature might result from the type of analysis that is performed by ESLint and other JavaScript linters. Using more complex analysis methods to identify more intricate issues, the risk of detecting false positive increases, and additionally the risk of users not understanding the issue and mistaking true positives for false positives [40]. The issues detected by ESLint are of a more simplistic nature where warnings are typically reported on only a single line of code, *e.g.* a variable that is not initialized. This is *e.g.* different from code smell detection tools for Java; as many code smell definitions are somewhat abstract, tools rely on heuristics [43] which can lead to false positives, *e.g.* to detect a God Class, PMD's heuristic relies on the combination of *Weighted Method Count*, *Tight Class Cohesion* and *Access to Foreign Data* metrics [44], [45].

Furthermore, the term false positive can be understood in two different ways, either a wrongly reported warning or a true warning that is not considered by a developer to improve the software under analysis [46]. Some participants discussed this type of false positive, which is when the linter flags a certain instance but the developer thought it was appropriate to break the rule in that particular case (P3, P6, P7, P9, P11). In those circumstances, ESLint provides the functionality to write an inline comment to disable a rule for a piece of code. Participants P9 and P11 however considered it to be positive to receive a warning in these cases. P9 explained that it makes him think about whether breaking the rule is actually a good idea or not: "*That is OK, the linter is there to make sure we are being thoughtful about our choices.*" (P9). Furthermore, the presence of the disabling comments can be useful for other developers to know the intention of the code: "*That's helpful because now other people reading the code will know this is definitely intentional. So I think it's useful for the reader and ESLint is great that it's flexible like that.*" (P11). It can therefore serve as documentation of some sort to help others understand the intention of the code.

C. Threats to Validity

It is important to address the validity of this study which we will do in the context of qualitative research [47].

1) Transferability: The main limitation to this study is its possible lack of generalizability. The sample size is not large and it thus may not represent all OSS development. Moreover, as we only talk to developers from OSS projects, the results may not represent industry software. We tried to mitigate this fact by interviewing experienced developers from popular and reputable projects. However, that selection of the sample creates another bias in the study where the results might be different if smaller projects were examined along with projects having fewer configurations. Only projects were selected that had somewhat extensive configurations as they would have more input on how rules are selected for a project, thus being able to report on what methods they use for the task. Furthermore, future research needs to be conducted to understand why some developers do not use a linter or have ceased using one.

As we only looked at projects that use ESLint, the results might not reflect on usage of all JavaScript linters. Also examining other linters such as JSLint or JSHint might produce different results than presented in this study, which would be an interesting aspect to see in future studies. We chose to observe the usage of only one linter to make the interviews more consistent. As the available linters have different features, *e.g.* regarding configurability, we would not have been able to ask all participants the same questions, and thus possibly making the analysis less reliable. To minimize the effects of this, we chose to address the most popular and most flexible linter, also to not restrict the results with more limited use cases. Additionally, we verified that ESLint is indeed the most popular linter among the top 120 JavaScript projects by manually evaluating each project to see whether they use any linting tool.

2) Credibility: Possible variables that effect the results of this study relate to the previous knowledge of the participants. It is likely that we interviewed people who already feel strongly about linters as they are frequent users of the tool. We can not know for sure if their opinions are based on

their own experiences with using the tool or if it is based on external literature that they have read. Because of this concern, we tried to address our questions to relate specifically to the participants' own opinions and experience working on the particular project. However, in some cases, the participants had other and even more experience in working with a linter on other projects, for which they also based their answers on.

3) Confirmability: Another possible limitation concerns the coding of the interviews. It was conducted by one person in order to obtain consistency in the results. It is however possible that someone else would have coded them differently, perhaps resulting in different conclusions [48]. This was mitigated to some extent by focusing on not having any preconceived ideas before processing the data and thus not trying to fit the data to any existing theories. Furthermore, the derived codes are available online for inspection [30].

VI. Related Work

Johnson *et al.* [12] researched the usage of ASATs, in particular why some developers do not use these tools to find bugs despite of their proven benefits. Similar to our study, they examined one tool, FindBugs, and interviewed 20 developers to find that the main reason why developers choose to use an ASAT to find bugs is to reduce time and effort that goes into manually performing the task. Reasons to not use these tools include poorly presented output where there are too many false positives or too many warnings outputted in general, in addition to tools not being integrated conveniently in the workflow. Christakis and Bird [13] also investigated how developers perceive ASATs and specifically which barriers they face in the adoption of these tools. By surveying 375 developers at Microsoft they found that the largest obstacle in using these tools was the fact that some unwanted rules are turned on by default in the tools' configurations, and proposed having only a subset of rules enabled by default instead. This is indeed what ESLint does, where no rules are enabled by default but it is easy to enable a preset. Other frequently experienced challenges were bad warning messages, too many false positives and for the analysis to be too slow. Interestingly, our findings do not reflect the results of these two studies. This could be explained by the different nature of ESLint compared to the wide range of ASATs that they examine, or perhaps by the fact that ESLint is actively maintained by a large community where frequent improvements are made to the tool. Our work builds on these studies as we also research why static analysis tools are used and how they can be improved, but focusing solely on JavaScript.

Several studies have focused on the effectiveness of ASATs to find bugs in software where the studied tools were not successful in identifying reported errors [49], [50], [16]. However, Wedyan *et al.* [50] found ASATs to be more effective in finding refactoring opportunities. In this study, though without any empirical verification, ESLint seems to be successful at reporting both defects and refactoring opportunities.

Ayewah *et al.* [11], [51] studied the usage of FindBugs where they saw that users are generally interested in fixing warnings from the tool, especially the high priority ones, but which types of warnings depends on the user's context. It is therefore valuable to have configuration options for these different groups of users. Similarly, Jaspan *et al.* [2] recognized the importance of customizing and prioritizing rules to make developers more willing to use an ASAT as it reduces the amount of perceived false positives for a project. Regarding how much configurations are applied, Beller *et al.* [3] performed a large-scale study on the prevalence of ASAT usage along with how they are configured. They found that these tools are commonly used in OSS software and in particular for JavaScript projects. The configurations for these projects are most often changed from the default settings, but typically only one rule is added, removed or modified. This research included ESLint but we suspect that these results would be different if the study would be conducted again today, as ESLint has since then removed any default configurations in the tool and all rules are now off by default. A developer using the tool thus now needs to create some configuration, which in the simplest case could just include the recommended settings from ESLint.

Several studies have examined how to prioritize warnings from static analysis tools [52], [53], [54], [42]. Kim *et al.* [52] propose a history-based method to improve the prioritization of ASAT warnings using past bug fixes from software change history. A different history-based approach is proposed by Heckman [53] where the developers' feedback to the warnings is used. In our research we examine how the participants prioritize rules to create the configuration files for their projects.

Techniques for static analysis of JavaScript code has been a popular research topic in recent years [55], [56] where static analysis is said to be a difficult task due to the dynamic nature of the language and the frequent use of libraries [32], [57], [8]. Tools have therefore been developed that apply dynamic analysis to handle these challenging language features [58], [10], such as DLint from Gong *et al.* [33]. Furthermore, Pradel *et al.* [7] created a tool to counter against errors that have to do with inconsistent types in JavaScript. Gao *et al.* [59] also found that applying static type systems such as TypeScript and Flow can reduce errors in JavaScript applications. We address this problem by observing the challenges that developers face with the dynamic features (including dynamic typing) of the language and how the linter fits into that setting.

VII. Conclusion

In this study we examine why and how JavaScript developers use linters in their projects. To that goal, we use a qualitative research method to conduct and analyze interviews with developers from popular and reputable OSS projects on GitHub. The derived results explain why and how developers use such tools, as well as the challenges they face. Our results have direct implications to developers, tool makers, and researchers.

As a final message, we encourage the JavaScript community to take advantage of the many benefits that linters provide.

REFERENCES

[1] B. W. Boehm *et al.*, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[2] C. Jaspan, I. Chen, A. Sharma *et al.*, "Understanding the value of program analysis tools," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 963–970.

[3] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 470–481.

[4] (2015) Language trends on github. [Online]. Available: https://github.com/blog/2047-language-trends-on-github

[5] T. Mikkonen and A. Taivalsaari, "Using javascript as a real programming language," 2007.

[6] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, "Javascript errors in the wild: An empirical study," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 100–109.

[7] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 314–324.

[8] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 499–509.

[9] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: A static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 121–132.

[10] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 116–125.

[11] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, 2008.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[13] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 332–343.

[14] Eslint. [Online]. Available: http://eslint.org

[15] Npm stats on jshint, jslint, eslint and jscs. [Online]. Available: https://npm-stat.com/charts.html?package=eslint&package=jshint&package=jslint&package=jscs&from=2015-01-01&to=2017-04-30

[16] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

[17] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[18] B. G. Glaser and J. Holton, "Remodeling grounded theory," in *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, vol. 5, no. 2, 2004.

[19] Findbugs. [Online]. Available: http://findbugs.sourceforge.net

[20] Checkstyle. [Online]. Available: http://checkstyle.sourceforge.net

[21] Pmd. [Online]. Available: https://pmd.github.io

[22] Jshint. [Online]. Available: http://www.jshint.com

[23] Jscs. [Online]. Available: http://jscs.info

[24] Jslint. [Online]. Available: http://www.jslint.com

[25] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 52–78.

[26] Airbnb eslint preset. [Online]. Available: https://github.com/airbnb/javascript

[27] Standard preset and linter. [Online]. Available: https://github.com/feross/standard

[28] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487–513, 2011.

[29] S. E. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *Software metrics, 2005. 11th ieee international symposium*. IEEE, 2005, pp. 10–pp.

[30] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, *Why and How JavaScript Developers Use Linters: Technical Report*, Jul. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.835658

[31] React. [Online]. Available: https://facebook.github.io/react

[32] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 1–12.

[33] 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*. ACM, 2015, pp. 94–105.

[34] Flow. [Online]. Available: https://flow.org

[35] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004, pp. 245–256.

[36] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 598–608.

[37] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 931–940.

[38] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 161–170.

[39] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles, "Social barriers faced by newcomers placing their first contribution in open source software projects," in *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. ACM, 2015, pp. 1379–1392.

[40] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[41] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[42] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *International Static Analysis Symposium*. Springer, 2003, pp. 295–315.

[43] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[44] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[45] Pmd, description of god class metric. [Online]. Available: http://pmd.sourceforge.net/pmd-5.0.1/rules/java/design.html

[46] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.

[47] E. G. Guba, "Criteria for assessing the trustworthiness of naturalistic inquiries," *Educational Technology research and development*, vol. 29, no. 2, pp. 75–91, 1981.

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[49] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[50] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 2009, pp. 141–150.

[51] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008, pp. 1–5.

[52] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 45–54.

[53] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *Crossroads*, vol. 14, no. 1, p. 7, 2007.

[54] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 341–350.

[55] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript." in *SAS*, vol. 9. Springer, 2009, pp. 238–255.

[56] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," *ECOOP 2012–Object-Oriented Programming*, pp. 435–458, 2012.

[57] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the html dom and browser api in static analysis of javascript web applications," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 59–69.

[58] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 165–174.

[59] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in javascript," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 758–769.