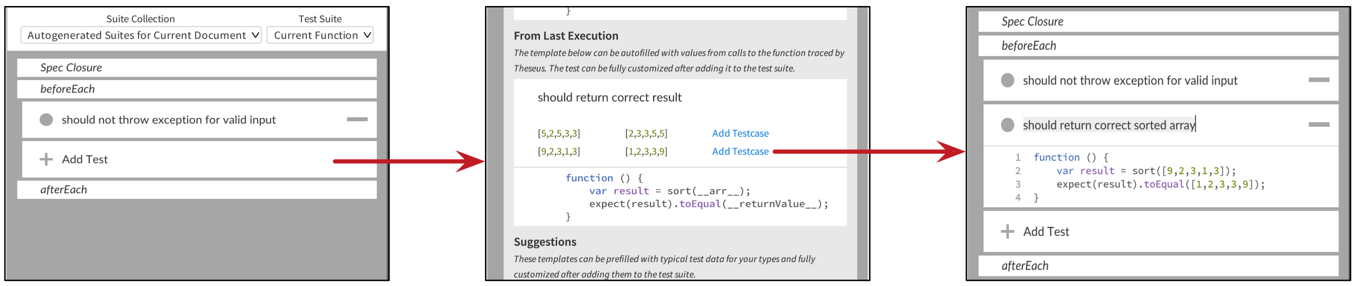
Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages

Jan-Peter Krämer[1,2], **Joel Brandt**[2], **Jan Borchers**[1]
[1] RWTH Aachen University
{kraemer, borchers}@cs.rwth-aachen.de
[2] Creative Technologies Lab, Adobe Research
joel.brandt@adobe.com


Suite Collection
Autogenerated Suites for Current Document
Test Suite
Current Function
Spec Closure
beforeEach
should not throw exception for valid input
Add Test
afterEach
From Last Execution
should return correct result
Add Testcase
Add Testcase
Suggestions
Spec Closure
beforeEach
should not throw exception for valid input
should return correct sorted array
Add Test
afterEach

The template below can be autofilled with values from calls to the function traced by Theseus. The test can be fully customized after adding it to the test suite.

[5,2,5,3,3]	[2,3,3,5,5]	Add Testcase
[9,2,3,1,3]	[1,2,3,3,9]	Add Testcase

```
function () {
    var result = sort(__arr__);
    expect(result).toEqual(__returnValue__);
}
```

These templates can be prefilled with typical test data for your types and fully customized after adding them to the test suite.

```
function () {
    var result = sort([9,2,3,1,3]);
    expect(result).toEqual([1,2,3,3,9]);
}
```

Add new Test — Choose a correct execution to start from — Manually refine the test

Figure 1. To improve the workflow of writing unit tests, Vesta offers test templates that are instantiated with runtime information.

ABSTRACT

Documentation and unit tests increase software maintainability, but real world software projects rarely have adequate coverage. We hypothesize that, in part, this is because existing authoring tools require developers to adjust their workflows significantly. To study whether improved interaction design could affect unit testing and documentation practice, we created an authoring support tool called Vesta. The main insight guiding Vesta's interaction design is that developers frequently manually test the software they are building. We propose leveraging runtime information from these manual executions. Because developers naturally exercise the part of the code on which they are currently working, this information will be highly relevant to appropriate documentation and testing tasks. In a complex coding task, nearly all documentation created using Vesta was accurate, compared to only 60% of documentation created without Vesta, and Vesta was able to generate significant portions of all tests, even those written manually by developers without Vesta.

Author Keywords

IDEs; software development; authoring tools; documentation; unit tests; dynamic languages

ACM Classification Keywords

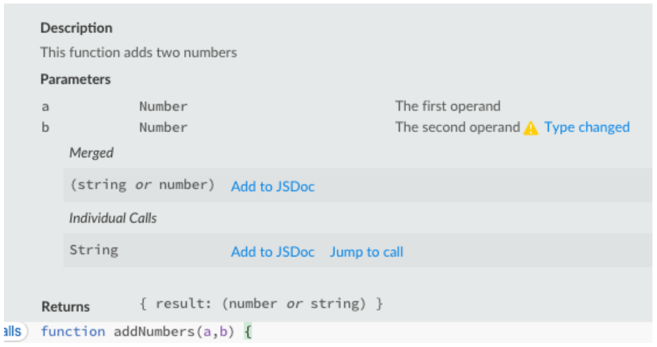
D.2.6. Software Engineering: Programming Environments

INTRODUCTION

Documentation and unit tests increase software maintainability [19]. They describe expectations about a part of the source code, e.g., a parameter is expected to always have the documented type. Our goal is to support creating documentation and unit tests in JavaScript. JavaScript is dynamic: it uses dynamic and weak typing, polymorphic call sites, and runtime type modifications [26]. Hence, few expectations about a program are represented explicitly in JavaScript code. This makes documentation and unit tests especially valuable. However, creating them requires extra effort, so they are often missing or out of date [16, 30].

To specify expectations about JavaScript programs, static and dynamic analysis tools have been proposed. Static tools often focus on inferring type information [14, 23, 12]. They usually require initial manual source code annotations provided through either a JavaScript dialect that syntactically allows variables to have types [14, 23], or JSDoc-formatted [8] documentation [12]. Even with these annotations, though, JavaScript's dynamic features make static analysis difficult [26]. Dynamic tools can often suggest other expectations beyond types, and create corresponding unit tests [2, 22].

Despite the variety of powerful tools available, real-world projects often lack documentation and unit tests [30]. We believe one reason for this to be that existing tools require developers to change their workflows, either to manually create annotations, or to handle the setup and regular execution of a dynamic analysis tool. Imposing these workflow requirements can impair flexibility, which is one of the most important reasons developers choose to use dynamic languages [25]. To explore how an improved interaction design affects the documentation and test authoring process, we created a prototype tool called Vesta. It is comprised of two authoring components, one for documentation and one for unit tests. Our key idea is to leverage the fact that developers frequently execute their program manually to verify their work – in one


Description

This function adds two numbers

Parameters

a	Number	The first operand
b	Number	The second operand ⚠ Type changed

Merged

(string *or* number) Add to JSDoc

Individual Calls

String Add to JSDoc Jump to call

Returns { result: (number *or* string) }

```
alls) function addNumbers(a,b) {
```

Figure 2. Vesta's documentation interface shows that a recorded type conflicts with the documentation.

study [4], 80% of edit-test-edit cycles were shorter than 5 minutes. Vesta is a dynamic analysis tool that uses runtime traces of these manual executions to suggest updates to documentation and unit tests. Because developers already perform these manual executions, Vesta does not change their existing workflow.

In the following section, we present design challenges related to leveraging traces from manual execution, and show how Vesta's interface is designed to address these challenges. In evaluating our prototype, we learned that (a) nearly all method documentation created with Vesta was completely accurate compared to only 60% of documentation created without Vesta, and (b) Vesta was able to generate all unit tests, even those written manually by developers without access to Vesta. Vesta also allowed developers to use the collected runtime data for other tasks, e.g., locating bugs.

THE VESTA PROTOTYPE

To prototype the interaction we propose, we used a dynamic analysis technique that only analyzes method parameters and return values, i.e., it only supports authoring function documentation. We considered this small subset of documentation to be useful for developers, while being reasonably efficient to capture. As we explored the design of Vesta, we found that working with information obtained only from runtime traces of manual executions posed four design challenges: (1) Manual executions rarely execute all possible code paths, causing the information to be **incomplete**. (2) Information can only be updated when the developer chooses to perform a manual execution, potentially causing information to become **outdated**. (3) Many manual executions are performed while the code is not yet working as expected, causing the information to be **incorrect**. (4) Information gathered during executions is **impermanent** – information that was correct at some point is likely to change when code evolves.

We addressed the problems of **incomplete** and **outdated** information by designing the interaction with Vesta around current development practice: Vesta encourages writing documentation and unit tests as part of the rapid edit-test-edit cycles that developers already perform routinely [4]. Immediately after new code has been manually tested, Vesta suggests updates to documentation and unit tests. Because the purpose of a developer's manual testing is to exercise the code she is currently writing (and thus documenting), information obtained by Vesta is likely relevant even though it might be **incomplete**. Further, the developer has just performed the manual execution Vesta analyzes, so the information is not yet **outdated**.

After a new runtime trace is recorded, Vesta compares information gathered during function invocations with the documentation. If no documentation exists for a function, Vesta inserts documentation for types of function parameters and return values. Otherwise, Vesta checks if the values recorded at runtime match the documented types and presents type conflicts (Fig. 2). Developers can navigate to the call site, which is useful when the conflict may constitute a bug. They can also instruct Vesta to update the documentation either by replacing existing type information, or by merging existing and new type information to indicate polymorphism.

To support the user in assembling a test suite (a group of unit tests), Vesta offers two test templates that can be instantiated with information gathered at runtime. The first template (Fig. 1) allows programmers to automatically recreate one recorded invocation of a function and test that it returns the recorded return value. Using this template, developers can already achieve good test coverage by picking correct combinations of parameter and return values from their edit-test-edit cycles. The second template helps programmers create test cases for common edge cases for parameters of a function, based on the parameter types stored in the documentation. This template is designed to encourage writing tests that cover erroneous behavior. Because all templates require runtime information, this workflow provides little benefit during test-first development. This is a limitation common among all test authoring systems that rely on program analysis. Users can also write custom test cases using Vesta, using all features of the Jasmine [1] unit test framework.

We address the problems of **incorrect** and **impermanent** information by applying design guidelines: Visualizations need to be easy to parse, because a developer needs to spot **incorrect** information quickly. At the same time, they should not draw attention after every manual execution, since these often occur during debugging phases that are already cognitively demanding. Faced with **impermanent** information, developers need to be able to change or extend the information quickly. To implement these guidelines, we build on previous design ideas to combine visual representations of source code with keyboard-based text editing [3, 17].

When documentation is not being edited, it is rendered in an easy to parse, visually appealing non-code format (see Fig. 2). For editing, Vesta switches to a JSDoc-formatted textual representation as soon as the cursor is moved into the documentation block. Cursor movement is identical to standard text editors: the cursor can be positioned using either the arrow keys or the mouse pointer.

To provide developers with a starting point for test organization, Vesta maintains one associated test suite for every function. Vesta's unit test interface adds visual elements to the source code to provide an easy to parse overview of tests: A separate area to the right of the source code shows one test suite at a time. By default, this is the associated test suite for the function that is currently being edited. Every test is shown in a box that shows the test title and current failure status in its header. The source code of each test can be collapsed to view a concise list of test case names. As for documentation, text-editor-like cursor navigation is available throughout the interface.

Vesta is implemented as an extension for the Brackets editor [13] and uses a modified version of Theseus [21] to record runtime traces. The video figure shows a complete walk-through of the interaction with Vesta.

EVALUATION: LAB STUDY

To learn about how developers use Vesta, we compared developers using Vesta with those using an unmodified version of Brackets [13] in a lab study. We tested whether the quality of documentation and unit tests produced by participants using Vesta is higher. Quality was assessed along the following axes: (1) amount, accuracy, and completeness of documentation; (2) number of test cases; (3) number of test cases testing failure cases; and (4) source code lines covered by all tests.

Setup

Participants implemented a web-based API to access menus of restaurants on a college campus. The API had to provide a list of restaurants, opening hours, menu languages, and daily menus for the current week. To obtain the information, developers had to fetch and parse the official websites listing the menus. Participants had 6 hours to work on this task. Having this large open-ended task gave us rich and ecologically valid qualitative results—though sometimes at the expense of statistical significance in low-level quantitative measures. If the task had been too simple, effects of a tool on developer productivity could have been over-exaggerated [9].

We asked participants to write complete documentation and a thorough test harness along with the code. Participants had to use JSDoc [8] for documentation and Jasmine [1] for unit tests. The task description included an introduction to JSDoc and rules on the required accuracy for type information.

Participants had to use Node.js [31] and the libraries express [15], lodash [29], and cheerio [24]. This restricted set of libraries was chosen to give participants a helpful starting point, and to reduce the variety in solutions to make comparison between participants more meaningful. All libraries, required software, and a minimal project template were pre-installed on identical 27-inch iMacs with screen recording.

Participants in the study were randomly assigned to one of two conditions: In the treatment condition, participants worked with Brackets and the Vesta extension; in the control condition, participants worked with Brackets and the FuncDocr [20] extension, which analyses the header of a function to provide templates for all required `@param` and `@return` statements. The features of FuncDocr match the documentation support in many IDEs.

Participants filled out a questionnaire about their current documentation and unit testing practices before the trial. Those in the Vesta condition also filled out a questionnaire about Vesta after the trial. Two to four participants were invited to each trial to work in parallel. They were allowed to talk to each other to share implementation ideas and help when problems arose. The study also included a lunch break in which participants were allowed to talk. Allowing communication made the setup more ecologically valid, as during a regular work day interruptions from co-workers are common for software developers [11]. The experimenter was present as a silent observer at all times. Participants received a 100€ gift certificate as compensation for their time.

Results

14 computer science students (two female) participated in our study (average age: 24 years, *SD*: 2.6). One participant in the control condition was excluded afterwards due to technical problems during the trial. On average, participants reported to have 3.0 years of experience with JavaScript (*SD*: 1.8), and to spend 13.1 h/week programming (*SD*: 7.3).

Participants did not consistently report to be required to write documentation (*Mdn*: 3)[1] or unit tests (*Mdn*: 2) in their daily work. Still, they reported to know what good tests should look like (*Mdn*: 4). Despite strongly disagreeing that it is not worth it to write tests (*Mdn*: 1) or documentation (*Mdn*: 1), participants often skip writing tests (*Mdn*: 4) and documentation (*Mdn*: 4). Most agreed that they should write tests (*Mdn*: 4) and documentation (*Mdn*: 4) more often.

Documentation

We first analyzed the number of documentation lines created per function. We found no significant difference between conditions, i.e., the number of documentation lines was correlated roughly linearly with the number of functions in both conditions. This result shows that most participants complied to our requirement to write documentation as part of the task. Variance in length of documentation blocks is mostly introduced because the number of parameters and return values that need to be documented differs for each function.

A type specification was considered *accurate* if it was syntactically and semantically correct (i.e., conformed to the rules in the task description). This metric favors Vesta, as a type specification is always accurate unless it was edited manually in a way that impaired accuracy. This happened for one participant who generalized type specifications at the end of the study, introduced three errors, and time ran out before she was able to check her changes using Vesta. More surprisingly, 38.9% (*SD*: 37.2%) of type specifications created in the control condition were not accurate, showing that Vesta solves a real problem.

We considered a function *completely* documented if type specifications were present for every parameter and the return value (if applicable), and a description was present for the function and each parameter. The latter had to be added manually in both conditions. In the control condition, 51.9% (*SD*: 30.9%) of functions were completely documented, vs. 68.4% (*SD*: 32.7%) in the Vesta condition. This difference is encouraging, though not strongly statistically significant (one-sided t-test $p = 0.093$, $t(10.3) = 1.42$).

[1]5-point Likert scale, 1 - “strongly disagree”, 5 - “strongly agree”.

Unit Tests

Both groups wrote fewer tests than expected, with four of six participants in the control condition and one in the Vesta condition writing no tests at all (control: *M*: 1.50 tests, *SD*: 2.81, Vesta: *M*: 4.43 tests, *SD*: 3.31).

85% of all test cases tested the default behavior of a function, i.e., they checked whether the function returned the correct result when called with a correct parameter. In the Vesta condition, these tests were exclusively authored using Vesta, and all but one participant used the test suites Vesta associates with each function. Only 2 participants (one in each condition) wrote at least one test checking the behavior in failure cases, e.g., invalid requests or network failures. No participant tested every function.

Discussion

Documentation was more accurate and more complete when authored with Vesta, and developers strongly agreed (*Mdn*: 5) that Vesta made writing documentation more enjoyable. We observed that, despite the availability of dynamic typing, developers usually assume a fixed type. Type information that is stored in the documentation without being a language feature can represent the developer's model accurately, e.g., by allowing two alternative types to be specified at the same time. The type information authored using Vesta is sufficient to transition to more advanced analysis tools such as Tern [12].

Qualitative observations indicate that Vesta's interaction design is crucial for its success: (1) Developers often forgot to update documentation when their code evolved. Vesta reminded participants to update documentation, increasing completeness and accuracy. In the control condition, some participants wrote all documentation at the end of the trial, which led to errors when documenting methods implemented long ago. (2) Developers often neglected to either create any documentation or to include all required parts. When using Vesta, developers regularly checked if Vesta generated the correct types to verify their source code. On this occasion, they opportunistically filled in descriptions, leading to higher completeness. (3) Developers often had insufficient knowledge about third party APIs. E.g., participants confused whether an API method returned an array of strings or DOM elements. In such cases, Vesta helped to create correct documentation and, additionally, prevented programming errors. These observations show that Vesta's interaction design enables additional uses of runtime information beyond creating documentation. Developers regularly identified bugs using the information generated by Vesta.

Characterizing all developers, we found a striking underprioritization of unit test authoring. This was reported in the pre-study questionnaire, and confirmed in the experiment: Several participants in the control condition did not write a single test, even though we asked to prioritize writing tests for partial functionality over finishing the task. Prioritization of authoring tests improved slightly when using Vesta.

All unit tests authored during our study – even those written manually when Vesta was not available – could be generated by only analyzing developer-initiated manual executions. Developers appreciated the template-based assembly of test suites, especially to test methods that require complex parameters like HTML strings. We identified two reasons why, regardless of these encouraging results, even in the Vesta condition no sufficiently thorough test suites were created: First, developers rarely tested for potential failures when manually executing their application, yielding tests suites that did not cover these cases either. Applying more advanced test suggestion algorithms could address this. Second, participants frequently found Vesta's test organization cumbersome, and thought they would write better tests without it (*Mdn*: 4). A future iteration of Vesta's interface should use a more familiar organizational concept for test suites.

RELATED WORK

Rothermel et al. [27] presented an interaction similar to Vesta to generate tests for spreadsheets. Users could mark cells as correct, and the system determined relevant input cells to generate a test case. Spreadsheets automatically update after every change; in contrast, Vesta needs to rely on when and what developers manually execute.

Previous tools used runtime traces of developers' manual program executions mostly to support debugging. These tools allow, e.g., recreating failure states [5], retrospectively analyzing which sequence of method calls caused errors [18], or observing which methods are called while the program is running [21]. In contrast to these debugging tools, Vesta provides the most benefit with error-free test runs.

Many techniques to generate parts of documentation and unit tests can be complemented by our interaction design and could extend Vesta's technical capabilities in the future. Existing tools using static analysis and symbolic execution for static languages can describe, e.g., important function calls in a part of the source code [28], the effect of a change [7], or possible exceptions thrown by a function [6]. Daikon [10] can detect multiple kinds of potential invariants, e.g., static types or values, in a runtime trace. These can be used, e.g., to generate suggestions for new unit tests [32].

SUMMARY AND FUTURE WORK

We presented an interaction for the creation of documentation and unit tests that is designed around current development workflows. Our key idea is to provide updates to these documents immediately after manual executions, which are already performed regularly during development. We showed that Vesta does improve the quality of documentation and unit tests created. Documentation accuracy increased significantly because maintaining type information allowed for automatic type checking as an additional immediate benefit. In the future, we plan to integrate existing static analysis tools into Vesta, and use its recorded type data to bootstrap static analysis. Similarly, we plan to explore other test generation algorithms with the hope of improving code coverage and quality of authored test suites. Finally, to further strengthen the external validity of our results, we plan to conduct a field study.

REFERENCES

1. Christopher Amavisca, JR Boyens, Gregg Van Hove, and Vinson Chuong. 2015. Jasmine. http://jasmine.github.io/. (2015). [Online; accessed 17-September-2015].

2. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of Javascript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 571–580. DOI: http://dx.doi.org/10.1145/1985793.1985871

3. D Asenov and P Muller. 2014. Envision: A fast and flexible visual code editor with fluid interactions (Overview). *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2014), 9–12. DOI: http://dx.doi.org/10.1109/VLHCC.2014.6883014

4. Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (Sept. 2009), 18–24.

5. Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 473–484. DOI: http://dx.doi.org/10.1145/2501988.2502050

6. Raymond P.L. Buse and Westley R. Weimer. 2008. Automatic Documentation Inference for Exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 273–282. DOI: http://dx.doi.org/10.1145/1390630.1390664

7. Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically Documenting Program Changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 33–42. DOI: http://dx.doi.org/10.1145/1858996.1859005

8. The contributors to the JSDoc 3 documentation project. 2011. @use JSDoc. http://usejsdoc.org/. (2011). [Online; accessed 17-September-2015].

9. B de Alwis, G C Murphy, and M P Robillard. 2007. A Comparative Study of Three Program Exploration Tools. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 103–112. DOI: http://dx.doi.org/10.1109/ICPC.2007.6

10. Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (Dec. 2007), 35–45.

11. Victor M. González and Gloria Mark. 2004. "Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 113–120. DOI: http://dx.doi.org/10.1145/985692.985707

12. Marijn Haverbeke. 2015. Tern. http://ternjs.net/. (2015). [Online; accessed 17-September-2015].

13. Adobe Inc. 2015a. Brackets. http://brackets.io/. (2015). [Online; accessed 17-September-2015].

14. Facebook Inc. 2015b. Flow. http://flowtype.org/. (2015). [Online; accessed 17-September-2015].

15. StrongLoop Inc. 2015c. Express. http://expressjs.com/. (2015). [Online; accessed 17-September-2015].

16. Mira Kajko-Mattsson. 2001. The state of documentation practice within corrective maintenance. *IEEE International Conference on Software Maintenance. ICSM 2001* (2001), 354–363.

17. Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 387–396. DOI: http://dx.doi.org/10.1145/1124772.1124831

18. Andrew J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1569–1578. DOI: http://dx.doi.org/10.1145/1518701.1518942

19. Jeffrey Kotula. 2000. Source code documentation: an engineering deliverable. *Technology of Object-Oriented Languages* (2000).

20. Ole Kröger. 2015. FuncDocr. https://github.com/wikunia/brackets-funcdocr. (2015). [Online; accessed 17-September-2015].

21. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

22. Ali Mesbah and Arie Van Deursen. 2009. Invariant-based automatic testing of AJAX user interfaces. *IEEE 31st International Conference on Software Engineering* (May 2009), 210–220. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070522

23. Microsoft. 2015. TypeScript. http://www.typescriptlang.org/. (2015). [Online; accessed 17-September-2015].

24. Matthew Mueller. 2015. Cheerio. http://cheeriojs.github.io/cheerio/. (2015). [Online; accessed 17-September-2015].

25. Linda Dailey Paulson. 2007. Developers shift to dynamic programming languages. *Computer* 40, 2 (Jan. 2007), 12–15. DOI: http://dx.doi.org/10.1109/MC.2007.53

26. Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. *SIGPLAN Not.* 45, 6 (June 2010), 1–12. DOI: http://dx.doi.org/10.1145/1809028.1806598

27. Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. 2000. WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation. In *Proceedings of the 22Nd International Conference on Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 230–239. DOI: http://dx.doi.org/10.1145/337180.337206

28. Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 43–52. DOI: http://dx.doi.org/10.1145/1858996.1859006

29. Benjamin Tan, John-David Dalton, Kit Cambridge, Mathias Bynens, and Blaine Bublitz. 2015. lodash. https://lodash.com/. (2015). [Online; accessed 17-September-2015].

30. M J Taylor, J McWilliam, H Forsyth, and S Wade. 2002. Methodologies and website development: a survey of practice. *Information and Software Technology* 44, 6 (April 2002), 381–391.

31. S Tilkov and S Vinoski. 2010. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83. DOI: http://dx.doi.org/10.1109/MIC.2010.145

32. Tao Xie and David Notkin. 2006. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering* 13, 3 (2006), 345–371. DOI: http://dx.doi.org/10.1007/s10851-006-8530-6