EVALUATING THE QUALITY OF TEST CODE AGAINST FEATURE CODE ON POPULAR JAVASCRIPT LIBRARIES

Vidroha Debroy  
Dept. of Computer Science, Southern Methodist University  
6425 Boaz Lane, Dallas TX 75205, USA  
vdebroy@smu.edu

Ranran Feng  
Dept. of Computer Science, University of Texas at Dallas  
800 W. Campbell Road, Richardson, Texas 75080, USA  
rxf090020@utdallas.edu

ABSTRACT
A large number of web applications today leverage freely available third-party JavaScript libraries. However, in doing so, there is an increased reliance on not just the core library or feature code, but also any accompanying test code – so that regressions can be found when the feature code is modified, new tests can be written for proposed extensions code using the existing tests as a sample, etc. Thus, it is crucial that both the feature and the test code, that comprise a distributable library, be of high quality. We empirically investigate whether this is the case in practice by virtue of a case study, conducted on 10 large, popular and freely available JavaScript libraries, that compares the quality of test code against that of feature code, using several static quality metrics (size in terms of Source/Compiled Lines of Code, Commenting Ratio, Cyclomatic Complexity, Maintainability Index and Difficulty Measure). Our results are intriguing in that while they support some expectations: we find that feature code is more extensively commented than test code; we also find that more test code is actually written (in terms of compiled lines of code) than the feature code; and yet on average the test code is of a higher quality based on its complexity, maintainability and difficulty.

KEY WORDS
Software Quality, Maintainability, Test Code, JavaScript.

1. Introduction
JavaScript is a very popular choice among web developers to create client-side scripts that facilitate: user interaction, response to events, loading of content asynchronously, etc. Indeed, web applications have become one of the fastest growing types of software systems today [1]. When writing JavaScript code for web applications, it is common for developers to leverage existing libraries to accomplish common tasks (for example, date parsing, manipulation and formatting using the Moment.js library [22]). Doing so allows developers to focus on their own core product code and not re-invent the wheel. Furthermore, it is easy to bring the library code into a web application and to extend the library code (by defining/re-defining functions in the JavaScript code that is processed after the library code). This however, introduces a number of concerns, such as:

- What assurances are available that the library code functions correctly to begin with?

- If the library code is to be modified to meet some specific requirement, how can developers know that they have not accidentally introduced regressions?

- If the library code is to be extended by developers, how can the correctness of the extensions be assessed?

While the first concern can partially be addressed by examining reviews from other users of the library code (when available), the only practical way to address all other concerns is for the library to not just include core functional code, hereafter referred to as feature code, but to also include tests (test code) that can be run against the feature code. The tests could therefore, potentially serve as a regression suite to detect undesired effects when modifying the library code. The tests could also serve as a template for creating new tests when extending the library code, in an automated or semi-automatic fashion [7]. These claims are in turn however, contingent on the test code of being of a sufficient quality – such that it is understandable and maintainable by developers that may leverage the library, even though they had nothing to do with its creation. Thus, a good JavaScript library must not just comprise of feature code of a high quality, but must also contain test code of a high quality. We note that this is desirable for all software regardless of which programming language is used – JavaScript is used for the purposes of this study due to its growing popularity and ease of incorporation into projects as further discussed in Section 2.

This paper evaluates whether the above claim is applicable in practice by analyzing 10 popular, open-source, freely available JavaScript libraries and statistically contrasting test code against feature code with respect to metrics that indicate software quality. Our results are intriguing because while we find evidence that supports typical expectations – feature code is found to be better commented than test code, we also find evidence to support the claim that often test code is of a higher quality than feature code in terms of its complexity, maintainability and difficulty, while in fact being larger in size than feature code in terms of compiled lines of code. To the best of our knowledge, we are the first to undertake such a study, especially in the context of JavaScript and web applications.

The remainder of this paper is organized as follows: Section 2 describes the background and motivation for the work presented herein. Section 3 then discusses the subject code, data collection and analysis techniques for our experiments,
followed by Section 4 which presents the results. Related work is discussed in Section 5 and finally, we present our conclusions and ideas for future work in Section 6.

2. Background and Motivation

We begin by illustrating the ease with which external JavaScript libraries may be brought into one’s own web page or application, as shown in Figure 1. For our example we make use of AngularJS [1], which is a very popular and freely available JavaScript library that allows developers to extend HTML vocabulary for their applications. In the figure, the HTML code snippets on the left and the right differ in only two lines: line 4 where a reference to the minified version of version 1.2.26 of AngularJS is added, and line 7 where the ngApp directive is applied to the div element (by adding the ng-app attribute). However, the difference in the way the two HTML snippets are rendered is very stark: in the case of the snippet on the left, the rendered output contains just the text of the paragraph tag on line 8; whereas in the case of the snippet on the right, the rendered output contains text that corresponds to the logical addition of 3 and 1 to produce 4. How did this come to be?

The explanation is that the application of the ngApp directive on line 8 auto- bootstraps any code within the corresponding div tag as an AngularJS application (AngularJS being brought in via the script reference on line 4). AngularJS in turn knows to parse and evaluate the code within the double curly braces (also referred to in various contexts as handlebars) as a logical expression such that the rendered output contains a 4 instead of {{3 + 1}}. Thus, just by adding a script reference and adding a single attribute to a DOM element, we are able to evaluate expressions at run-time such that HTML can be rendered very differently than would traditionally be done. Also note that in this example since the script reference is to a publicly hosted AngularJS source file, no actual source code needs to be imported into our own project (though that is always an option), allowing us to cleanly separate our own product code from library code, and also simplifying deployments by ensuring that library code need not be included as part of the deployment.

Continuing our discussion – the complete AngularJS library contains an extensive set of tests which can be downloaded and run locally, which instills trust in the framework in that it is likely to be well tested. Now one aspect of AngularJS that makes it very popular is that it is fully extensible and every feature can be modified or replaced to suit unique development workflow and feature needs [1]. However, if developers were to do so, they would need to know how to write new tests and possibly modify existing tests, as well as understand how to interpret and debug any failing tests. Thus, if the test code that is bundled with the distributed library were easily understandable, that would go a long way towards making the overall process simpler and more straightforward for developers.

Figure 2 presents 3 unit test cases for the ngController directive, written using Jasmine – a framework for testing JavaScript code [13] – that are included with the AngularJS distribution (with common set-up code omitted for brevity). Jasmine tests (referred to as specs) are defined by calling the global Jasmine function describe which contains a function with a string as an argument. The test is the title of the spec and the function the actual test. A spec contains one or more expectations (which evaluate either to true or false) that test the state of the code [3]. Looking at the first two unit tests we see that in both cases the expectation is that an exception will be thrown (toThrowMinErr is a custom matcher that is implemented in one of the helper files included with the AngularJS distribution). To begin with, we note the seemingly interchangeable usage of ‘exception’ and ‘error’ in the titles of the tests. While this is relatively minor (though per the ECMAScript specification ‘exception’ would be more accurate [6]) it creates confusion on which convention developers writing new unit tests should follow.

More seriously however, the third unit test is confusing in its intent and/or its implementation. While the test name is suggestive of the controller (a term meaningful in the context of AngularJS) constructor function not successfully producing an instance of the controller, in reality the test expects that an exception will be thrown. No message or type checking is actually performed on the resulting exception. Arguably, if developers were to do so, they would need to know how to extend the AngularJS library and write further unit tests using the third unit test as an example, it would constitute bad practice (regardless of programming language – the evaluation of whether an operation is successful or not should not always be determined by whether an exception is thrown or not). Also it is relevant to note that comments are not provided for any of the above unit tests.
Figure 2. Three sample unit tests for the ngController directive included with AngularJS distribution

Thus, we see that while it is relatively easy to bring external JavaScript libraries into our own web applications, our ability to extend and modify these libraries, or at the very least test our extensions and modifications, can be hampered if the test code that accompanies the feature code of the library, is not of high quality. Prior to ending this section it is also important to point out that when the documentation of a library is incomplete or incorrect, examining the tests is a practical way to gain more knowledge of the library in automated or semi-automated ways [7], which offers another reason why developers should aim to write well-commented and understandable test code.

3. Experimental Design

In this section we discuss the design of our experiment(s) – the subject JavaScript libraries used; the metrics and tools utilized as well as the research questions evaluated, and the data-collection steps followed.

3.1 Subject JavaScript Libraries

To ensure a comprehensive study, a total of 10 JavaScript libraries were utilized in this study as detailed in Table 1. Each of the libraries is publicly available on Github [10] and consists of both feature and test code (the bulk of the tests that are included with such libraries are unit tests, and sometimes integration and end-to-end tests) and in all cases, the master branch of each repository was used. The libraries are popular not just from a usage but also a development perspective – for example, at the time of writing of this paper each of the repositories had received a commit from a contributor no earlier than 4 days ago. Each of the libraries (i.e., the repositories for the libraries) also has multiple contributors and many commits. At the time of writing of this paper the lowest number of commits for a library was 474 with the lowest number of contributors being 14 – both numbers correspond to Formula.js; while the highest number of commits is 6327 and the highest number of contributors is 1146 – both numbers correspond to AngularJS (such meta-data is provided by Github and is also publicly accessible at each of the repositoryUrls listed in Table 1). This high degree of activity indicates a lot of interest in these libraries, and coupled with the fact that they are freely available and include tests, make them ideal for such a study. Thus, to summarize – these libraries were chosen because they are: highly popular; open-source and freely available; actively maintained by multiple contributors; and come bundled with tests.

3.2 Tools, Metrics and Research Questions

Since the purpose of this study is to focus on the qualitative aspects of test code vs feature code, as opposed to the quantitative aspects, we focus on static metrics (as opposed to dynamic or metrics collected at runtime) in this paper. Indeed for a paper such as this, dynamic metrics do not make much sense because while they can be collected for the feature code by executing it directly or via the test cases, there are no tests of the test cases (so as to speak) to provide dynamic metrics for the test code. Thus, static metrics represent a suitable choice to fairly compare feature code with test code.

Moving forward, the most basic and intuitive way to compare feature code and test code is in terms of the number of Source Lines of Code (inclusive of comments and referred to as SLOC) written in each case. Source code size is considered to be an indicator of scale in many other research studies focusing on many aspects of Software Engineering, such as [9], [15], [23]. We thus ask ourselves the following research question (RQ1):

- Do the popular JavaScript libraries have as much test code (SLOC) as feature code?

The SLOC count however, is inclusive of comments and having too many lines of comments in either test code or

<table>
<thead>
<tr>
<th>JavaScript Library</th>
<th>Description</th>
<th>Version</th>
<th>Github Repository Url</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backbone [2]</td>
<td>Supplies structure to JavaScript-heavy applications</td>
<td>1.2.1</td>
<td><a href="https://github.com/jashkenas/backbone">https://github.com/jashkenas/backbone</a></td>
</tr>
<tr>
<td>jQuery [14]</td>
<td>Simplifies HTML operations, event handling, animation, and Ajax</td>
<td>3.0.0-pre</td>
<td><a href="https://github.com/jquery/jquery">https://github.com/jquery/jquery</a></td>
</tr>
<tr>
<td>Moment.js [22]</td>
<td>Parse, validate, manipulate, and display dates in JavaScript</td>
<td>2.8.4</td>
<td><a href="https://github.com/moment/moment">https://github.com/moment/moment</a></td>
</tr>
<tr>
<td>RequireJS [26]</td>
<td>Loads plain JavaScript files as well as more defined modules</td>
<td>2.1.15</td>
<td><a href="https://github.com/jrburke/requirejs">https://github.com/jrburke/requirejs</a></td>
</tr>
<tr>
<td>Marionette [17]</td>
<td>A composite application library for Backbone.js</td>
<td>2.3.0</td>
<td><a href="https://github.com/marionettejs/backbone.marionette">https://github.com/marionettejs/backbone.marionette</a></td>
</tr>
<tr>
<td>Underscore [31]</td>
<td>Supports functional helpers such as map, reduce, filter, etc.</td>
<td>1.7.0</td>
<td><a href="https://github.com/jashkenas/underscore">https://github.com/jashkenas/underscore</a></td>
</tr>
<tr>
<td>Formula.js [8]</td>
<td>Implementation of most Microsoft Excel formula functions</td>
<td>1.0.6</td>
<td><a href="https://github.com/sutoiku/formula.js">https://github.com/sutoiku/formula.js</a></td>
</tr>
</tbody>
</table>
feature code can skew results based on just the SLOC count. Therefore, we also consider the number of Compiled Lines of Code, referring to this metric as CLOC, which is the source code except with all of the comments removed. This data was collected using Google’s Closure Compiler application [4] discussed further in Section 3.3. Thus, RQ1 can again be assessed with respect to just compiled lines of code (i.e., CLOC).

Additionally we investigate with respect to the Commenting Ratio which corresponds to the fraction of source code that is commented, and can be computed using the SLOC and the CLOC (given by: SLOC – CLOC)/SLOC. The higher the Commenting Ratio, the more the source has been commented (a value of 0 indicates the source has not been commented at all, a value of 0.5 indicates that there as many comments as there are compiled lines of code, and a value of 1 trivially indicates that the source consists only of comments). Based on the Commenting Ratio we can ask (RQ2):

- Do the popular JavaScript libraries have their test code commented as much as the feature code?

However, relying on the SLOC or the CLOC count as any indicator of quality by itself can lead to misleading or biased results. Similarly having more comments on a piece of code does not necessarily make the comments more meaningful and does not necessarily imply that that piece of code is better commented relative to others (though it is nevertheless a valuable indicator). Taking this into account we also leverage Plato [25] which is an open-source tool to visualize JavaScript code complexity. In particular, in addition to the number of source lines of code (inclusive of comments), Plato reports the average Cyclomatic Complexity (CC) [19] which measures the structural complexity of the code by computing the number of different paths in the flow of the code. The more complex a unit of code is, the harder it is to maintain the code and in turn, the lower its overall quality is expected to be. Thus, to continue the qualitative comparison between test code and feature code with respect to Cyclomatic Complexity, we can also ask (RQ3):

- Is the feature code more (or less) complex than the test code in the case of popular JavaScript libraries?

Along with the Cyclomatic Complexity, Plato also reports the Maintainability Index (MI) for the code being analyzed which corresponds to a value between 0 and 100 such that a higher value indicates better maintainability – which in turn represents the ease with which a software system can be modified, and is an important software quality attribute [20]. The formula used to compute the MI is presented in Equation (1) with further details available at [16].

\[
\text{MI} = \max(0, 171 - 5.2 \cdot \ln(\text{Halstead Volume}) - 0.23 \cdot (\text{CC}) - 16.2 \cdot \ln(\text{CLOC})) + 100 / 171
\]  

We note that while many variants of the MI exist [32], the metric used by Plato is also used by Microsoft as part of Visual Studio’s code analysis reporting [16] and builds on top of the Cyclomatic Complexity metric (which does not reduce the value of reporting on the complexity metric by itself). Based on MI we now ask ourselves (RQ4):

- Is the feature code more (or less) maintainable than the test code in the case of popular JavaScript libraries?

Finally, Plato also reports on the Difficulty Measure (DM) [12], of the code being analyzed, the measure being representative of the difficulty of the code to write or understand, e.g. when doing code review. The Difficulty Measure computation is shown in Equation (2) where \( n_1 \) refers to the number of distinct operators, \( n_2 \) refers to the number of distinct operands and \( N_2 \) refers to the total number of operands, with respect to the code under study.

\[
\text{DM} = \frac{n_1 \times N_2}{n_2} 
\]

The higher the difficulty measure, the harder it is for a developer to comprehend the code. Thus, in terms of comparing the understanding of test code relative to feature code, we can ask ourselves (RQ5):

- Is the feature code more (or less) difficult (to understand) than the test code in the case of popular JavaScript libraries?

Having described the quality metrics used for analysis and the tools utilized to produce the metrics, as well are the research questions that are to be answered based on values of the metrics, we are now ready to discuss data collection.

### 3.3 Data Collection

The steps followed for the data collection are described below and illustrated in Figure 3.

**Figure 3. Data collection**

1. The first step in the overall data collection process involves identifying popular JavaScript libraries that are freely available and consist of not just feature code, but also test code. Since for the purposes of our study the tests are not to be executed, no restrictions were made on the testing framework that the tests are required to run under, assertion libraries required by the tests, etc.

2. After suitable libraries are identified, the source is downloaded and refined such that the test code is manually separated from the feature code and all external dependencies that are not part of the feature or test code (for example, vendor code) are removed from consideration with respect to data collection.

3. Both the feature code and feature code is processed using Google’s Closure Compiler [4] (freely available) with the WHITESPACE_ONLY and pretty_print flags set such that commented lines of code are removed from the source, but line breaks are retained.
4. The feature code and test code (with and without comments) for each library is processed via Plato and the quality metrics discussed in Section 3.2 are collected for analysis. Note that with respect to a good tool, the removal of comments should not affect the metrics such as the Cyclomatic Complexity, Maintainability Index and the Difficulty Measure (as they are strictly based on compiled lines of code) and we verify that this is the case with respect to Plato by comparing these metrics for test/feature code before and after processing by Closure Compiler, for each of the studied JavaScript libraries.

4. Results
The results of the comparisons between test code and feature code across each of the studied JavaScript libraries with respect to the various research questions and metrics discussed in Section 3 are presented here.

4.1 Source/Compiled Lines of Code
In terms of the Source Lines of Code (SLOC) and Compiled Lines of Code (CLOC) across the various subject JavaScript libraries, the results are as shown in Figure 4 (the feature code results are annotated by the name of the library while the corresponding test code is annotated with the name of the library suffixed with ‘-test’). For example, as per the figure we see that the AngularJS feature code has an SLOC count of 117022 and a CLOC count of 30497, whereas the AngularJS test code has an SLOC count of 61127 and a CLOC count of 42031.

Based on the results we see that with respect to the SLOC, there are only three cases where the feature code out-sizes the test code (i.e., there is more feature code than test code), namely in the case of AngularJS, MathJS, and Formula.js. Such results generally suggest that more (when one also takes comments into consideration) test code is written than feature code, for most popular, open-access JavaScript libraries (7/10 as per Figure 4). The results however, do become more interesting when we consider only compiled lines of code (i.e., source code with all the commented lines removed) where we observe that with the exception of the Formula.js library, the test code always out-sizes the feature code which is highly suggestive of the fact that more (compiled) test code is actually written than feature code for such libraries, even when comments are ignored.

Additionally, to evaluate based on sound statistics, we make use of the Wilcoxon signed rank test (an alternative to the paired Student’s t-test when a normal distribution of the population cannot be assumed) [24]. For example, with respect to size we evaluate the one-tailed alternative hypothesis that the test code is greater than the feature code, in terms of SLOC and CLOC respectively. The null hypothesis here corresponds to the case when the feature code is greater than or at least equal to the test code in terms of size). Stated differently,

\[ H_0: \text{Size of feature code} \geq \text{Size of test code} \]

Based on the statistical test, we find that while the results are indicative that the SLOC of test code is higher than that of feature code (i.e., test code out-sizes feature code) across the various JavaScript libraries, such a claim cannot be made with an acceptable level of significance (only 78.42% based on the \( p \)-value of the test, which is 0.2158). The

![Figure 4. Comparison with respect to Source/Compiled Lines of Code across various JavaScript libraries under study](image-url)
results, are however, quite different when one considers just the compiled lines of code (CLOC). Ignoring comments and examining just the CLOC, we find that the confidence with which it can be claimed that test code outsizes feature code is 99.32% which is statistically significant. Thus, based on the popular JavaScript libraries under study, the answer to RQ1 is that while it cannot be concluded that test code outsizes feature code in terms of SLOC (inclusive of comments), such a claim can however be made in terms of the CLOC (i.e., when one disregards comments) with a very high degree of certainty (over 99%). Thus, the test code is found to be more voluminous than the feature code in terms of the actual (compiled) lines of code.

The Commenting Ratio can be computed based on the data presented in Figure 4 as discussed in Section 3.2. For example, in the case of the feature code of AngularJS the Commenting Ratio is \( \sim 0.74 \) (117022 – 30497/117022). For convenience, the exact Commenting Ratios are presented in Table 2. From the table we see that the Commenting Ratio is consistently higher in the case of feature code (the one exception being the Underscore library). With respect to the Commenting Ratio, we find that it can be said with 99.70% confidence that feature code is more commented than test code (the answer to RQ2). This is therefore, an area of improvement for the test code that is bundled with JavaScript libraries in that the test code is not observed to be commented nearly enough as much as the feature code is observed to be.

Table 2. Commenting Ratios for the JavaScript Libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Commenting Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Feature</td>
</tr>
<tr>
<td>AngularJS</td>
<td>0.7394</td>
</tr>
<tr>
<td>Backbone</td>
<td>0.3222</td>
</tr>
<tr>
<td>Grunt</td>
<td>0.1827</td>
</tr>
<tr>
<td>jQuery</td>
<td>0.4860</td>
</tr>
<tr>
<td>MathJS</td>
<td>0.5703</td>
</tr>
<tr>
<td>Moment.js</td>
<td>0.2570</td>
</tr>
<tr>
<td>RequireJS</td>
<td>0.4880</td>
</tr>
<tr>
<td>Underscore</td>
<td>0.1667</td>
</tr>
<tr>
<td>Formula.js</td>
<td>0.2222</td>
</tr>
<tr>
<td>Marionette</td>
<td>0.5191</td>
</tr>
</tbody>
</table>

4.2 Cyclomatic Complexity (CC)
Focusing now on the Cyclomatic Complexity metric, the results are presented in Figure 5.

Based on the figure we see that for all of the JavaScript libraries studied, the Cyclomatic Complexity of the feature code is greater than that of the test code, i.e., the feature code is more complex than the test code. The Wilcoxon statistical test too confirms that it can be said with 99.9% confidence that the feature code is more complex than the test code which is the answer to RQ3. This is an important result, but is perhaps not as surprising because arguably the complexity of feature code is expected to be higher than that of test code because feature code typically involves argument checking, pre-condition checking etc. which involves more branching (i.e., more conditions exist), whereas on average each test is specific in its intent and is expected to run under specific conditions and thus, is much less conditional.

4.3 Maintainability Index (MI)
We now consider the Maintainability Index (MI) as an indicator of quality, and provide our analysis of the test code relative to the feature code, with respect to the JavaScript libraries under study. The data is depicted in Figure 6.

Figure 6. Comparison based on Maintainability Index (MI)
Based on Figure 6 we see that the test code appears to be significantly more maintainable than the feature code on average. In fact, statistically speaking we can affirm with 99.70% confidence that the test code is more maintainable than, the feature code. This (the answer to RQ4) is a very intriguing and surprising result for a number of reasons.

First and foremost, the primary factor that drives the adoption of such JavaScript libraries is the ease with which the feature code can be leveraged, modified and extended. It is therefore, surprising to see that it is actually harder (as indicated by the MI) to modify and maintain the feature code than to do so with the test code. Secondly, feature code is often subjected to multiple re-factorings, abstractions to separate concerns, the use of basic coding patterns, formatting guidelines and peer reviews, and even complexity analysis etc.

In contrast, it is not required for test code to adhere to such strict requirements. In fact, it is arguably much easier to (give in to the temptation to) write code of an inferior quality in the case of test code. For example, consider the case where a poorly written test method may be copied many times over, changing just the inputs to the
functionality under test to make each test different from the other and verifying the right behavior in each test. While re-factoring code to promote re-use and reduce complexity, for example by using common setup and teardown functions, is undoubtedly beneficial, the value may not always be evident to developers in a practical setting, especially given that spending time to improve the quality of test code takes away time that could instead be used to implement more features. Thus, our expectation was for the test code to be of an inferior quality, relative to the feature code, in terms of maintainability. However, not just do we find that not to be the case, rather we find that the test code is conclusively more maintainable than the feature code under test.

We conclude this section by noting that while the test code is statistically more maintainable than the feature code for the libraries under study, this does not necessarily imply that the feature code is poor in terms of its maintainability. Consider that Visual Studio 2013 by Microsoft provides color coded ratings to quickly identify trouble spots in code, based on the Maintainability Index. A green rating between 20 and 100 and indicates that the code has good maintainability; a yellow rating is between 10 and 19 and indicates that the code is moderately maintainable; while a red rating is assigned to code that has an index between 0 and 9, which indicates low maintainability [3]. Based on this scale, all of the subject libraries studied, feature and test code alike, can be said to have good maintainability.

4.4 Difficulty Measure (DM)
As mentioned in Section 3.2, the Difficulty Measure represents the difficulty of the code being analyzed, to be understood by developers [12] such that a relatively higher difficulty indicates relatively lower quality [21]. The Difficulty Measures for the various JavaScript libraries (feature and test code) are presented in Figure 7. From the figure we see that in general the difficulty of the feature code is much higher than that of the test code (the two exceptions being in the case of AngularJS and MathJS). In the case of the RequireJS library the feature code is almost 20 (~19.74) times as difficult as the test code. Also, based on the Wilcoxon statistical test, the answer to RQ5 is that it can be said with 97.31% confidence that the feature code is more difficult, i.e., harder to understand, than the test code.

![Figure 7. Comparison based on Difficulty Measure](image)
This is a surprising result, because similar to the discussion with respect to the Maintainability Index (in Section 4.3), feature code is expected to undergo re-factorings, reviews, etc. more frequently than test code; and thus, it is strange to see that the test code is much easier to understand than the feature code (with respect to the Difficulty Measure).

5. Related Work
In [27] Rosenberg et al. show that even better results in terms of software quality estimation can be obtained by combining size (lines of code) along with other metrics such as Cyclomatic Complexity, as has been done in this paper. In [32], a close look is taken at the Maintainability Index (MI) and it is determined that it provides valuable insight into software maintainability issues and that using it to assess source code and thereby, identify and quantify maintainability is an effective approach. This supports the use of the Maintainability Index in our own study.

In [29] Steidl et al. state that comments in code represent a main source for system documentation and are hence key for source code understanding with respect to development and maintenance. Furthermore, they state that existing approaches for software quality analysis ignore commenting or make only quantitative claims. The relevance of commenting source code is also examined in studies such as [28] where it is stated that comments are important software artifacts. In [21], Misra and Bhavsar study the difficulty of software (i.e., code) and utilize the Halstead Difficulty Measure, as has been done in this paper. In [34], it is investigated whether production code and the accompanying tests coevolve, by exploring a project’s versioning system, code coverage reports and size-metrics.

Finally, test code quality has often been assessed in terms of its ability to cover code [33] or detect faults [5]. In contrast this study assesses the quality of test code using the same static metrics that are used by many studies to assess the quality of feature code. Indeed the fact that tests themselves are often used to ratify claims on the quality of the feature code using code coverage in studies such as [30], without assessing the quality of the tests themselves, indicates the novelty of our work. This paper thus, paves the way for new research work into how the quality of tests need not necessarily be assessed by how much they cover feature code, or how many feature code bugs they detect; but rather how test code quality can also be assessed on a standalone-basis, using the same metrics as would be normally be used to assess the quality of feature code.

6. Conclusions and Future Work
In the context of library code, it is important that not just feature code, but also test code, be of a high quality because the existing tests are crucial to developers using the libraries – in terms of a regression suite and template for creating new tests, especially when the developers using the libraries modify or extend them. The tests (and comments in the tests) can also provide useful information on the libraries themselves. This study compares the quality of test code against that of feature code using 10 popular and freely available JavaScript libraries based on a number of quality metrics. Our study indicates that the test code provided with such libraries out-sizes the feature code (in
terms of the number of compiled lines of code) and also the average function size is relatively less in the case of test code, indicating higher quality relative to feature code. Furthermore, we also find that test code is less complex, more maintainable and less difficult (i.e., easier to understand) than feature code based on the metrics. Additionally, we present subjective arguments to explain some of the observations we have found, i.e., put them in perspective in order to better assess quality. Since all of the JavaScript libraries studied are freely available and popular, our results are useful to not just other researchers, but also practitioners who may currently use, or plan to use, the libraries. Additionally, we provide new insights as to how coding patterns used by developers may contribute to feature code being perceived as more difficult than test code (when using measures such as the Difficulty Measure).

Future work includes expanding our study across even more libraries and leveraging other quality analysis tools; and studying how the dynamic aspects of the test code (i.e., how tests cover the feature code, etc.) correlate with the static metrics. Additionally, JavaScript was a suitable choice for such a study; though there is no reason why the experiments and results so-derived should be applicable to only JavaScript libraries and we will investigate this hypothesis in the future.

References

[26] RequireJS. http://requirejs.org/