Automatic checking for elegance

Boris Arkenaar

May 10, 2015
v1.2

1 Introduction

Our goal is to make life easier for a tutor of the JavaScript programming language. In our bachelor project we will develop a JavaScript–examiner which will relieve the tutor from checking all students' solutions to exercises and give them appropriate feedback in an online course. At the same time, the JavaScript–examiner will benefit the students as well. The tutor simply doesn't have time to check the solutions of all the students. And therefore the tutor cannot give individual feedback to each student. The tutor will pick the biggest and most common pitfalls he sees in the solutions he picks. Then he will give the students general feedback on those pitfalls in the next online group session.

The reason why the JavaScript–examiner will benefit students — besides relieving the tutor of too much of a workload — is threefold. First off the JavaScript–examiner is able to check all the exercises of all students. Secondly, the JavaScript–examiner can give the students immediate feedback on their exercise — instead of them having to wait for the next online group session with the tutor. And finally the JavaScript–examiner gives the student individual feedback. Instead of some general explanation about common pitfalls the student might not even have had trouble with, the JavaScript–examiner gives each student feedback on the specific problems encountered in his or her solution.

This article will be an analysis of the quality aspects of JavaScript code on which the JavaScript–examiner can focus. What kind of quality aspects exist for programming languages in general? To make sure the scope of this article won't get to wide I will only pick a few quality aspects. I will base my choice on what we have learned at the Open University and on well documented quality aspects found in literature.

How easy is it to measure those quality aspects for a given code? and how precise is the feedback of those measurements? For measuring quality aspects I will look at literature about available measurements, how well they are documented and how difficult it would be to implement such a measurement. The results of the measurements of different quality aspects can vary from a general idea about the quality aspect of a given code to more specific indication of how a code might be improved. I will compare the measurements found in this article on the impact of their results. More on how feedback is given to the student can be read in the domain research of Bram Nieuwenhuize[1].

2 Code quality

Code quality can be divided into two areas. In one area you look at the quality of the code itself, without executing it. You look at aspects like how the code is laid out, how easy it is to read the code and how well you can understand what the code is trying to do. These aspects have no effect on the execution of the code however. The second area looks at the quality of execution. How well does the code execute? Does it use efficient language constructs?

We will start in this section by looking at the first area: the quality of the code. First we will see how we can determine the quality of the code by looking at its layout. Secondly we examine metrics for expressing the maintainability of a given code and how we can use that to give feedback to students.

2.1 Natural Language

Although a programming language is an artificial language, natural language is also used often in code in the form of names (of variables, routines, etc.), headers (documentation blocks above routines, classes, etc.) and comments (explanation of the code between lines). These are three important aspects for giving feedback as explained by Stegeman et al. [2014]. Determining the quality of natural language is much more difficult than for an artificial language. Therefor it would be best not to start with this aspect when development of the JavaScript–examiner starts. It is a very interesting subject though, and it would be very valuable if the JavaScript–examiner could give feedback on this area as well. But this subject would need a lot of research and can be looked in to in a later stage of the development process of the JavaScript–examiner, probably for another team.

2.2 Code layout

A good Code layout can help you maintain an overview of the various sections of a code. This is another important aspect for giving feedback by Stegeman et al. [2014]. It says something about the order in which routines are placed and if routines are located close to the routines that call them. Because this becomes an important aspect when your code grows it won't be important for us to implement into the JavaScript–examiner. We will be looking at small exercises. When the JavaScript–examiner will be expanded and also be used for bigger exercises it might become interesting to take a look at this aspect again.

[1]domain-research-feedback.pdf

2.3 Code formatting

For readability it is important to structure your code in a way that conveys clearly the intention of the code, this is called code formatting [Stegeman et al., 2014]. There are many different ways you could format your code and there is not one formatting that is better than all the others. Most important is consistency.

The easiest way to check for a consistent formatting would be if one particular format would be required from the students. Then the JavaScript–examiner can simply check for that formatting. Otherwise it would have to determine the formatting of a particular exercise on various aspects and see if that formatting is used consistently throughout the entire assignment.

Besides being consistent in using one formatting there are still formatting aspects that would be considered bad practice and should be avoided at all times. There are many tools for examining code to make sure none of these bad practices are used in code. Even specific for JavaScript code such tools already exist.[2]

2.4 Maintainability

Various metrics can be used to give an idea about the maintainability of code. We are not looking for measuring maintainability specifically, but an indication of it can help us determine the quality of the code, for maintainability is an aspect of good quality code. Rakic and Budimac [2013] list the following metrics used in tools for analyzing code:

- Lines of code;
- Cyclomatic complexity;
- Halstead complexity;
- Object oriented metrics.

We will look at the first three metrics mentioned. They touch the subjects of flow, expressions, and decomposition as described by Stegeman et al. [2014] The last one — Object oriented metrics — might also be an interesting metric when object oriented aspects are introduced in the exercises. But it would lead too far for this research to go into; it can be the subject of a later research.

First off you can look at the amount of code in terms of lines of code. The more lines of code, the more complex it gets. There are different ways to determine the lines of code though. You can simply count all the lines in the source files, which we will call lines of code (LOC). But you could also skip the lines which contain comments and empty lines, the remaining lines we will call source lines of code (SLOC). Going one step further you can count the actual statements in the code instead of the physical lines. That will give you the

[2]For instance JSLint (http://www.jslint.com/) and the less strict variant JSHint (http://www.jshint.com/)

logical lines of code (LLOC). A disadvantage of LLOC is that it is more difficult to calculate. Instead of simply counting the lines of a text file. You would need serious knowledge of the programming language in question.

Only looking at the lines of code is not very useful. We can compare the LOC of the student with the solution of the tutor, but that will give you only a rough idea about whether the student used far too much code or extraordinarily few.

A more precise measurement of the complexity of code might be cyclomatic complexity. This basically tells you in how many unique paths your code could be executed [Booth and Harlock, 2014]. When the cyclomatic complexity increases, so does the complexity of your code. The more possible execution paths, the more difficult it is to reason about your code. But because the cyclomatic complexity increases when the code base increases it is more useful to express the cyclomatic complexity relative to the lines of code. This gives us the cyclomatic complexity density [Gill and Kemerer, 1991]. Also important is to make sure that the bodies of conditional statements don't get to large. And routines should as well contain only a limited set of tasks and variables [Stegeman et al., 2014].

The third metric we will be discussing is the Halstead complexity which looks at operators and its operands. It determines the amount of operators and operands in a given code. It also determines the amount of distinct operators and operands. With that data some calculations are performed to determine the complexity of the code.

While it is not necessary to fully understand the programming language for performing the Halstead metric, that method has some shortcomings as explained by Yu and Zhou [2010]. Where the cyclomatic complexity looks at the control flow of the code, the Halstead method only looks at the operations performed, but ignores the control flow completely. When used together however you can look at your code from both sides and express the maintainability in a combination of the results.

3 Execution quality

The quality of a given code can be determined by looking at the language constructs that have been used, and by looking at the manner in which the language constructs have been combined.

There is an interesting way to look at a new programming language which can also help us here to look at JavaScript code, of what structures it is composed and how they are combined. These words from Abelson and Sussman[3] give us that way of thinking: "What are the primitives, what are their means of combination, and what are their means of abstraction?"

To follow these words we would first have to look at the language constructs of JavaScript. Then we can determine in what ways these language constructs

[3]Abelson, H., Sussman, G.J. 1984 *Structure and Interpretation of Computer Programs*. MIT Press, [https://mitpress.mit.edu/sicp/]

can be combined. Finally we could look at the means of abstraction JavaScript offers us, but unfortunately there is not enough time to handle that aspect in this research.

3.1 Relative execution time

The relative execution time of a given code can be measured mathematically. As explained by Goodrich et al. [2008, Chapter 4] the relative execution time can be expressed using seven functions (constant, logarithm, linear, n-log n, etc.). To determine with which function the glsrel-exe-time of a given code can be expressed one has to look at the language constructs used in that code. For each language construct the relative execution time has to be determined. Then you can calculate the relative execution time of the combination of those functions (i.e. the combination of the language constructs that make op the code).

This method would be a good way to see if the student has used an algorithm that is too complex for the given exercise. If, for example, the solution of the tutor executes in n-log n time while the code of the student executes in quadratic time, feedback could be given to the student that he should use a more efficient algorithm for that exercise. The question would be: how to give the student more concrete feedback? Instead of just telling him he can do better it would be constructive to point him in the right direction. However, this would require more knowledge of the particular exercise and the difference between the solution of the tutor and the solution of the student.

An implementation of this method would require more than determining the language constructs used in a given code and how they are combined. When a loop is used, for instance, you would need information about run time variables to determine how many times the loop would be iterated over. In other words: looking at the language constructs is not enough, you would need a good understanding of the algorithm used and the problem that is being solved. This understanding might be provided by letting the tutor specify information about the algorithm used when creating an exercise in the JavaScript–examiner. However further study will be required to figure out how this would best be implemented, and what information would be required exactly from the tutor.

3.2 Language constructs

When looking at the meaning of the code you can distinguish different language constructs. When you look at an exercise, it might be the case that a good solution should contain one or two specific language constructs. That means that we can look for those structures in a student's code. If they are not there, or if there are too much of them, the student can be given feedback mentioning this fact. In order to be able to give the student a more profound explanation of why his code is not optimal we would need input from the tutor. Because our goal is to relieve the tutor from too much work we could try and implement this feature in a similar way Watson et al. [2011, Section 3.2] proposed in their article. They describe a way to provide feedback on how to correct a given code that generates an error (halting on execution). The first time the system encounters a specific error it has to request the tutor for feedback. The system saves that feedback into a database along with the error. On any subsequent occurrence of that same error in any of the students' code the system retrieves the tutor's feedback from the database. It does not need to bother the tutor any more, while the student receives valuable feedback.

We would need to determine first what language constructs are important for a specific exercise. When a student uses a for loop where the tutor does not, that would be an interesting case. But when a student uses one more simple assignment statement than the tutor does, that will probably not be very important for instance. We can determine the set of language constructs used in the code, by type and by quantity. And compare that with the solution of the tutor. This gives us a set of language constructs the student did not use while the tutor did, and a set of language constructs of the other way around. After filtering these two sets of language constructs for important ones we have something to give feedback on. Now we can use the same mechanics as Watson et al. [2011] described. We look in the database for the feedback on a particular language construct that was used — or not used — and show that to the student. In case no feedback was found in the database, the tutor is requested for input. In that case the student will not receive immediate feedback. But after the tutor has provided feedback to the system all other students who run into the same problem will receive the feedback immediately.

Interesting use of this method for later implementation can be the comparison — not only of the a student's solution with the tutor's solution — of a student's code with the code of previously submitted code of other students. This might give the student feedback on how he is doing relative to his fellow students.

Research would be needed to see if this would work the way I describe it here. Would it be feasible to determine how to rate the language constructs of the JavaScript language? And would the system find all the problem cases or will it miss some, or find cases that are of no importance? Besides that, this functionality would have to be created from the ground up because it is quite specific and I have not found something like it for JavaScript.

4 Comparing code

The code layout is an absolute check to see if the student's code adheres to a well defined set of rules of best practices in coding. The code either fails on some points or it is completely laid out according to the layout definitions used. How you go about giving feedback when some rules are broken is another question. The JavaScript–examiner could say the code is incorrect or more gently tell the student that it could be better. It could clearly point out where the shortcomings are or simply tell the student to do better. The point is that the rules of code layout can be well defined, how strict you hold the student to it is a matter of feedback.

A different matter are the other metrics for code and execution quality. They do not simply give you an answer whether the code is good or not. It is more of a gray area and you would need to determine when code is good, when it is acceptable and when it would need improvement. These acceptability margins will be different for each exercise (e.g. a more complex assignment should allow a higher cyclomatic complexity). The tutor could be asked to specify these margins when creating an exercise, but the tutor might struggle to determine good margins for the exercise at hand.

The great thing is: we do not need to determine the quality of a given code on its own. We can compare it to a well defined base code that is assumed to be correct and efficient (i.e. an acceptable solution), by asking the tutor to write a solution to the exercise himself. The JavaScript–examiner could determine a baseline for the different metrics by calculating the metrics for the tutor's solution. That means that we do not need to decide how many lines of code would be too much, or what cyclomatic complexity would make the code too complex. We can just compare the lines of code, and the Halstead complexity of a student's code with the solution of the tutor. That way the system would have a good base for which it can say: when I get these results (or better results) for the metrics on a given code, the code is good, otherwise the code could be better. Preferably you would still want some form of margin, where the results of the metrics only slightly below those of the tutor's are as acceptable as the tutor's. Determining how those margins can best be calculated is subject to further research. We do have to determine however: how bad is the code if it is not as good as the solution of the tutor? There should probably be some margin of acceptance.

5 Conclusion

While no one method is perfect it would be best to combine several methods to determine the quality of the code and of its execution. Looking at code layout is a good starting point and will be easy to implement because many tools exist for performing these checks for JavaScript code. The other methods will require a baseline which can be obtained by analyzing the tutor's solution. This won't be a problem to implement as well but would require further research and development for determining the right margins of acceptance of these metrics.

The maintainability metrics are well defined and have been implemented in different tools. Therefore these are also a good candidate to implement in the JavaScript–examiner. Some extra research might be necessary though to find the right implementation, use and combination of these metrics for best representing the quality of the students' code.

Examining execution quality of a given code would be an interesting area to look into, but will take more time and effort to implement. To calculate the relative execution time, good understanding of the code and the algorithm are needed. Also — because of its complexity — no ready to be used tools exist for executing this specific method. Serious research and development will be needed to be able to implement it.

Building up a database with specific usages of language constructs for an exercise along with helpful feedback for a student seems an interesting solution. The JavaScript–examiner will be able to generate valuable feedback, simply and effectively by asking the tutor. At the same time the tutor will be relieved from too much work because the system would save that feedback into a database. Further research is needed to determine the best way of extracting the important language constructs from code. Development is needed to implement this functionality and make sure the feedback from the database will be supplied at the right times.

Bibliography

P. Booth and P. Harlock. About complexity — JSComplexity.org, 2014. URL http://jscomplexity.org/complexity.

G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. 17(12):1284–1288, 1991. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=106988.

M. T. Goodrich, R. Tamassia, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Data Structures and Algorithms in Java*. John Wiley & Sons, 2008.

G. Rakic and Z. Budimac. Problems in systematic application of software metrics and possible solution. 2013. URL http://arxiv.org/abs/1311.3852.

M. Stegeman, E. Barendsen, and S. Smetsers. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 99–108. ACM, 2014. URL http://dl.acm.org/citation.cfm?id=2674702.

C. Watson, F. W. B. Li, and R. W. H. Lau. *Learning Programming Languages through Corrective Feedback and Concept Visualisation*, pages 11–20. Number 7048 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25812-1, 978-3-642-25813-8. URL http://link.springer.com.ezproxy.elib10.ub.unimaas.nl/chapter/10.1007/978-3-642-25813-8_2.

S. Yu and S. Zhou. A survey on metric of software complexity. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 352–356. IEEE, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5477581.