Estimating Quality of JavaScript

Sanjay Misra[1] and Ferid Cafer[2]
[1]Department of Computer Engineering, Atilim University, Turkey
[2]Servus Bilgisayar, Turkey

Abstract: *This paper proposes a complexity metric for Java script since JavaScript is the most popular scripting language that can run in all of the major web browsers. The proposed metric "JavaScript Cognitive Complexity Measure (JCCM)" is intended to assess the design quality of scripts. The metrics has been evaluated theoretically and validated empirically through real test cases. The metric has also been compared with other similar metrics. The theoretical, empirical validation and comparative study prove the worth and robustness of the metric.*

Keywords: *Software engineering, software quality, software metrics, java script.*

Received July 21, 2010; accepted January 3, 2011

1. Introduction

It is well known that the maintainability is one of the important factors that affect the quality of any kind of software. JavaScript also requires modelling, measurement, and quantification for the ease of maintainability purpose. In addition software metrics play an important role since they provide useful feedback to the designers to impact the decisions that are made during design, coding, architecture, or specification phases. Without such feedback, many decisions are made in Ad-hoc manner.

Number of researchers have proposed variety of metrics [1, 5, 14] for different software, software languages [15], software products and related technologies [2, 3]. All the reported complexity measures are supposed to cover the correctness, effectiveness and clarity of a system and to provide good estimate of these parameters. With the emergence of the new technologies, also new measurement techniques evolve. There is an ongoing effort to find such a comprehensive measure, which addresses most of the parameters for evaluating quality of the system. In adition, the quality objectives may be listed as performance, reliability, availability and maintainability [17] that are all closely related with software complexity.

Among several software measurement techniques most of those measurements and metrics [6, 23] are developed covering generally programming languages. Although such an approach is efficient, the language-independent methods may not be suitable for some programming or scripting languages. More specifically, only a few researches have been done to produce new measurement techniques and metrics for JavaScript. The lack of researches has also been an initiative for us to start working on this issue. JavaScript has lots of skills such as providing a programming tool for HTML, making an HTML code dynamic, give response to events; validate data, and get client side information [19]. Because of not requiring a client-server interaction, it increases the efficiency in using a web page. All these issues imply that the design of the JavaScript plays an important role and needs to be quantified for the ease of maintainability. In this respect, specific metrics should be developed. In the present work, we develop a metric for JavaScript, which is capable to calculate the structural and cognitive complexity of the JavaScript.

The paper is organized in the following way. The discussion for the need of new metric is given in section 2. The metric is proposed and demonstrated in section 3. The metric is validated theoretically in section 4. The empirical validation with real test cases has been given in section 5 and conclusion drawn on this work is in section 6.

2. Need for A New Metric

There is no metric in the literature which specifically measures the complexity of JavaScript. One way to evaluate the complexity of JavaScript is going through the traditional metrics, but these metrics are under several criticisms. These criticisms are mainly based on lacking a theoretical basis [10, 18], lacking in desirable measurement properties [22] being insufficiently generalized or too implementation technology dependent [21], being too labor-intensive to collect [11] and only confined on the features of procedural languages. On the other hand, most of the available metrics do not consider the cognitive characteristics in calculating the complexity of a code, which directly affects the cognitive complexity. Complexity of a code directly affects understand ability. Understand ability of a code is known as program comprehension and is a cognitive process and related to cognitive complexity. The cognitive

complexity is defined as the mental burden on the user who deals with the code, for instance, the developers, the testers and the maintenance staff. In our proposal, we calculated cognitive complexity in terms of cognitive weights [20]. Cognitive weights are defined as the extent of difficulty or relative time and effort required for comprehending given software, and measure the complexity of logical structure of software. A higher weight indicates a higher level of effort required to understand the software. A high cognitive complexity is undesirable due to several reasons, such as increased fault-proneness and reduced maintainability. Moreover, one of the programmers may leave the project and another one may come to sustain the project. In such a case, the code should have a low complexity, that the latter programmer can easily grasp the code without wasting too much time. Additionally, cognitive complexity also provides valuable information for the design of systems. High cognitive complexity indicates poor design, which sometimes can be unmanageable [4]. In such cases, maintenance effort increases drastically.

Because of the current software metrics being subject to some general criticism, it seems appropriate to develop a new metric, which satisfy most of the parameters of software and also include all the factors responsible for complexity of JavaScript. Complexity measure based on cognitive weight has already been proved as a well-structured metric for procedural languages [15] and object oriented languages [14]. Here in this paper, we propose JavaScript Cognitive Complexity Measurement (JCCM) metric, which is intended to measure the architectural complexity of JavaScript in terms of cognitive weight. The metric follows the similar approaches that have taken by Misra and Ibrahim [15]. We improve the metric proposed by Misra and Ibrahim [15] by considering the contribution of arbitrary and meaningful named variables, which are also responsible for increasing the understand ability of the program. As such, the paper has the following agenda:

1. To propose a complexity measure that is constructed with a firm basis in theoretical concepts, and also take into account the architecture and features of JavaScript.
2. To evaluate the proposed metric theoretically and practically.
3. To empirically validate the proposed measure with real examples.

Even though JavaScript codes are usually not too complex, we propose a model to measure the cognitive complexity of a JavaScript code in order to be able to increase the coding efficiency in JavaScript.

3. The Proposed Metric: JavaScript Cognitive Complexity Measure

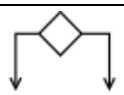
Complexity is defined as [9] "the degree to which a system or component has a design or implementation that is difficult to understand and verify". This definition implies that all the factors which make code difficult to understand are responsible for complexity. Accordingly, first we should identify these factors which are responsible for the complexity of JavaScript. When we analyse JavaScript codes we find that, the following factors are responsible for the complexity:

1. Size in terms of lines of code.
2. Number of Arbitrarily Named Distinct Variables (ANDV).
3. Number of Meaningfully Named Variables (MNV).
4. Cognitive weights of Basic Control Structures (BCS's).
5. Number of operators.

- *Number of Lines of Code:* Complexity of any program depends on the size of the code. In our formulation, we are including all the possible factors and identities which are assumed to increase the complexity of each line of a JavaScript code. Our method is sensitive to size factor, since it includes the complexity of only those lines which consist of variable(s) or operator(s). Naturally, this way of complexity calculation automatically includes the size of JavaScript, without including lines of code directly.
- *Number of Arbitrarily Named Distinct Variables:* Once we talk about the understand ability of a code, the names of variables used in the code play a very important role in increasing or decreasing the understand ability. If the name of a variable itself shows its meaning, it becomes easier to understand the code. Although, it is suggested that the name of the variables should be chosen in such a way which is meaningful in programming, most of the developers do not follow it very strictly. If the variable names are taken arbitrarily, there is no problem if the developer himself is evaluating the code. However, it is not the case in real life. After the system is developed, especially during maintenance time, arbitrarily named variables increase the difficulty in understanding four times more [12] than the meaningful names. In the formulation of JCCM, we are considering the weights of the arbitrarily named variables four times more than the meaningfully named variables.
- *Number of Meaningfully Named Variables:* From the discussion part taken in the above section, it is clear that meaningful named variables are more understandable than arbitrary named variables. We are assigning the weight of meaningful named variables as one unit.

- *Cognitive Weights of Basic Control Structures:* Complexity of a program is directly proportional to the cognitive weights of Basic Control Structures (BCS). Cognitive weight [20] of software is the extent of difficulty or relative time and effort for comprehending given software modelled by a number of BCS'. BCS' [20] are basic logics which build blocks of any software and their weights are one, two and three respectively. These weights are assigned on the classification of cognitive phenomenon as discussed by Wang [20]. He proved and assigned the weights for sub conscious function, meta cognitive function and higher cognitive function as 1, 2 and 3 respectively. Although, we followed the similar approach with Wang [20], we made some modifications in the weights of some BCS. For example, we include try-catch BCS (a special feature of JavaScript codes) in the list and assigned the weight 2, based on its structure. Further, the weight of nested loop is assigned to 3^n. The weight of nested loop is 3^n, because it depends on the number of nesting. As a result the identified BCS and their corresponding weights are given in Table 1. From the Table it is clear that sequence, condition and loops in JavaScript have similar structures with other programming languages. The differences lies in functional activity and exceptions, where alert/prompt/throw, event, and try...catch are new basic control structure. The new basic control structures are demonstrated with a graph in Table 1. The syntax exceptions and the structures with examples are given as following.

Table 1. Basic control structures and weights (adopted from Wang's paper [15] with some modifications).

Category	BCS	CWJ	Flow Graph
Sequence	sequence	1	
Condition	if-else	2	
	switch	2	
	go-to	2	
Loop	for	3	
	for...in	3	
	while/do...while	3	
	nested loop	3^n	-
Functional Activity	function-call	2	
	alert/prompt/throw	2	
	event	2	
	recursion	3	
Exception	try...catch	2	

Structures of new basic Control Structure
Loop
for...in:
for (variable in object)
{
code to be executed
}
Functional activity
Functional activities are used to bind several functionalities to the program. In case of occurrence of an event a function is called. Also, alert triggers a function. For this reason, their weights are reckoned as 2 similar to function calls.
function-call:
function functionname(var1,var2,...,varX)
{
some code
}
alert/prompt/throw:
alert("sometext");
prompt("sometext","defaultvalue");
throw "exception";
events:
<input type="text" size="30" id="email"
onchange="checkEmail()">
Exception
Exceptions are triggered in case of deviation from the ordinary flow of the program.
try...catch:
try
{
//Run some code here
}
catch(err)
{
//Handle errors here
}
Syntax Exceptions
break and continue are counted as 1.
Inside an HTML code if any event is called it is counted as 2. For example;
onclick="display()" is equal to 2.
onclick="display("Hello") is equal to 2x2=4.
Calculating arrays:
var mycars=new Array(); 3x2=6
mycars[0]= "Saab"; 5x1=5
Calculating for...in:
for (x in mycars) 3x3=9
So, here in is included to the calculation.
Calculating try...catch:
try is not counted. Only catch is counted. For example;
try{...}
catch(err) {...} 2x2=4
catch and the name of the error, which is err in this program, are counted.
document.write("The interest is "+i+" percent.");
Here + operators are not counted, but the variable, which is 'i' in this program, is counted. Of course, also semi-colon is counted.
If any function is called it is multiplied with 2. Even for creating an object Date() function is also included.
this and with are counted as 1.

- *Number of Operators:* The number of operators also increases the complexity of a JavaScript

program. Operators are responsible to increase the size. By keeping these points in our mind, in formulation of JCCM, the contribution of operators are considered.

Accordingly the total complexity of a JavaScript is given by the following formula:

- *JavaScript Cognitive Complexity Measure (JCCM).*

$$= \sum_{i=1}^{n} \sum_{j=1}^{m_i} ((4 * ANDV + MNV + Operators) * CWJ_{ij}) \quad (1)$$

Where complexity measure of a JavaScript code JCCM is defined as the sum of complexity of its n modules if any exists and module $\underline{i}$ consists of m_i line of code. In the context of formula 1, the concept of cognitive weights is used as an integer multiplier. Therefore, the unit of the JCCM is: JavaScript Cognitive Complexity UNIT-JCCM which is always a positive integer number. This implies achievement of scale compatibility.

3.1. Demonstration of the Metric

For demonstration of JCCM, we consider 3 different types of codes written in JavaScript taken from the web. These programs are differentiated from each other in their architecture. The calculations of JCCM for these examples are given in Tables 2, 3 and 4. The structures of all the three programs in tables are as follows: The second column of the tables shows the JavaScript codes. The sum of Arbitrarily Named Distinct Variables (ANDV), the Meaningfully Named Variables (MNV) and the operators in the line is given in the third column of the table. The cognitive weights of each JavaScript lines are presented in the forth column. The JavaScript complexity calculation measure for each line is shown in the last column of Tables 2, 3 and 4.

Table 2. Example 1.

Line No.	JavaScript Code	ANDV+MNV+ Operators	CWJ	JCCM
1	var i=0;	8	1	8
2	for (i=0;i<=5;i++)	10	3	30
3	{	0	1	0
4	document.write("T he number is " + i);	3	1	3
5	document.write("< br />");	1	1	1
6	}	0	1	0
	Total	-	-	42

The first example script, given in Table 2, consists of 6 lines of codes. The highest complexity value is 30 for line number 2. It is because; this line consists of a loop and, ten variables. In other words, this line is the most complex one in terms of structure and size. On the contrary, complexity value is zero for lines 3 and 6, since these lines do not contain any variable or operator and, therefore, have the simplest possible structure.

The second program demonstrated in Table 3, consists of 6 lines of code and is more complex than first one. It is because of two nested loops. The weight of second 'for loop' in line 3, is counted as 3^2 which is equal to 9, it is because of nested loops make program much more complex. As result, the overall complexity for this code is 132, which is approximately four times more than the first program, which is 42. It is worth to point out that, what the contribution of JCCM is, because it is obvious that the program with two nested loops will be more complex than program consisting of one loop.

Table 3. Example 2.

Line No.	JavaScript Code	ANDV+MNV+ Operators	CWJ	JCCM
1	var i, j;	10	1	10
2	for (i=0;i<=5;i++){	10	3	30
3	for (j=0; j<=i; j++)	10	9	90
4	document.write(" *");	1	1	1
5	document.write(" ");	1	1	1
6	}	0	1	0
	Total	-	-	132

Table 4. Example 3.

Line No.	JavaScript Code	ANDV+MNV+ Operators	CWJ	JCCM
1	var d = new Date();	8	2	12
2	var time = d.getHours();	4	2	8
3	if (time<10)	3	2	6
4	{	0	1	0
5	document.write(" Good morning");	1	1	1
6	}	0	1	0
7	else if (time>10 && time<16)	7	2	14
8	{	0	1	0
9	document.write(" Good day");	1	1	1
10	}	0	1	0
11	else	0	1	0
12	{	0	1	0
13	document.write(" Hello World!");	1	1	1
14	}	0	1	0
	Total	-	-	43

Third program given in Table 4, consists of 14 lines of code, have function calls in line 1 and 2. This is the reason; the cognitive weights for these lines are assigned as 2. It is worth to consider that, although this program is the most complex in terms of lines of code, i.e., 14, its JCCM value is 43, which is more realistic. Table 5, represents the summary of the complexity values for different complexity measures for three examples JavaScript. We have applied Logical Line of Code (LLOC), Cyclomatic

Complexity (CC) [13], Volume (V), Difficulty (D), Efforts (E) and Time (T) [8] to these JavaScript.

Table 5: Complexity values for different complexity measures of the three examples.

Program No.	JCCM	lLOC	CC	Halstead			
				V	D	E	T
1	42	4	3	79	6	474	26
2	132	5	5	93	9	837	46
3	43	8	4	148	4	592	32

In fact, the important thing is to calculate the most realistic value which really represents the complexity of the script. If we compare complexity values of all related complexity metrics given in Table 5, we find that JCCM values are higher to line of code, cyclomatic complexity, and T. Its reason is that, JCCM represents the complexity values due to all parameters responsible for complexity; however, all these parameters are independently evaluated by different metrics. The detailed comparison is demonstrated in section 5.

4. Evaluation of JCCM

The worth of any new measurement system should be proved by proper validation and testing process. A metric system should be theoretically evaluated for showing that a new system is developed based on strong theoretical base (scientific principles) and validated empirically to prove its practical usefulness. There are many validation criteria [7, 9, 16, 24] for theoretical validation of a metric system; however, most of them are based on principles of measurement theory. In the next section, we are evaluating JCCM against the principles of measurement theory [16].

4.1. JCCM and Measurement Theory

According to the measurement theory point of view, in a measurement system there should be an entity whose attributes are targeted for quantification, a property and measurement mapping. The metric is the mapping of entities to the values. For achieving to this definition, first we have to define empirical relation system, numerical relation system for our JCCM.

- *Definition 1:* Empirical Relational System-ERS for a given attribute, an Empirical Relational System is an ordered tuple $ERS=<E, R_1,...,R_n, o_1,..., o_m>$ where E is a set of entities, $R_1, ..., R_n$ denote n empirical relations such that each R_i has an arity n_i, and $R_i \subseteq E^{ni}$. $o_1, ..., o_m$ denote m empirical binary operations on the entities that produce new entities from the existing ones, so $o_j: E\times E\rightarrow E$ and the operations are represented with an infix notation, for example, $e_k= e_i\ o_j\ e_l$. According to this definition, the components of the quantification system are the values representing the decided quantities; the binary relations show the dependencies among them and the binary operations describe the production of new values from the existing ones. For JCCM, the entities are the JavaScript scripts. The empirical relation is assumed to be more_or_equal_complex and the only empirical binary operation is the concatenation of scripts. This can be explained by a solid example. Assume that a program body P is given and a new program body Q is obtained by simply duplicating P. One may easily establish the relation more_or_equal_complex between P and Q.
- *Definition 2:* Numerical Relational System-NRS is an ordered tuple $NRS=<V, S_1,...,S_n, p_1,..., p_m>$ where V is set of values, $S_1, ..., S_n$ denote n relations such that the arity of S_i is equal to the arity of R_i, and $S_i \subseteq V^{ni}$ and $p_1, ..., p_m$ denote m numerical binary operations on the values that produce new values from the existing ones, so $p_j: V\times V\rightarrow V$ and the operations are represented with an infix notation, for example, $v_k= v_i\ p_j\ v_l$. For JCCM, V is the set of positive integers, the binary relation is assumed to be $\geq$ and the numerical binary operation is the addition (i.e., +) of two positive integers.
- *Definition 3:* Measure- m is a mapping of entities to the values and it considers neither the empirical nor the numerical knowledge about systems, i.e., $m: E\rightarrow V$. The measure for JCCM complexity is defined by equation 1 in section 3.

Representation Condition

- *Definition 4:* A new metric system must satisfy the following two conditions known as representation condition:

1. $\forall i \in 1.....n\ \forall <e_1,...e_n> \in E_{ni}$ $(<e_1,...e_n> \in R_i \Leftrightarrow <m(e_1),...,m(e_n)> \in S_i)$
2. $\forall j \in 1.....m\ \forall <e_1,e_2> \in E\times E (m(e_1 o_j e_2)=m(e_1) p_j m(\ e_2))$.

The first part of the Representation Condition says that for a given empirically observed relation between entities, there must exist a numerical relation between corresponding measured values and vice versa. In other words, any empirical observation should be measurable and any measurement result should be empirically observable. The second part says a measured value of an entity which is obtained by the application of an empirical binary operation on two entities should be equal to the value obtained by corresponding numerical binary operation executed over individually measured values of entities. In other words, the complexity of the whole should be definable in terms of the complexities of its parts.

For JCCM, the representation condition requires that 1 if, for any scripts, e_1 and e_2 are in more_or_equal_complex relation (i.e., $<e_1, e_2> \in$ more_or_equal_complex) then the measured JCCM complexity value of entity e_1 should be greater than the measured complexity value of entity e_2 (i.e., $m(e_1) > m(e_2)$) and vice versa.

Considering the two JavaScript codes; if *Q* is the double of *P* then the number of BCSs, and variables for *Q* automatically becomes double. Consequently, for part (1) of the condition, it is possible to say that the empirically observed more_or_equal_complex relation between two program bodies leads to a numerical binary relation > among those entities or vice versa. However, part (1) is only satisfied if there are such clear empirically observable relations between program bodies; for example *P* and *Q*.

For part two of the representation condition, we can say that the JCCM complexity value of a program body which is obtained by concatenation (i.e., the empirical binary operation) of e_1 and e_2 is equal to the sum (i.e., the numerical binary operation) of their calculated complexity values. Therefore, JCCM satisfies the second part of the representation condition. Hence, we can say that JCCM satisfies the representation condition.

4.2. Evaluation of Measure Based on Scale

JCCM was evaluated through the measurement theory and found to be satisfied by the representation condition. Our next step is to find the scale of the JCCM. We can investigate the scale of JCCM through admissible transformation and extensive structures. Admissible transformation is the simplest way to find the scale of a measure. On the other hand, Zuse [24] has stressed the advantage of using extensive structure because it is one of the most important measurement structures which characterises empirical conditions of reality, hypothesis of reality and empirical conditions behind the software measure. However, both give the idea about the scale of metric. Therefore, we evaluated JCCM only by admissible transformation.

4.2.1. Admissible Transformation

- *Definition 5:* A scale is a triple <*ERS, NRS, m*>, where ERS is Empirical Relational System, NRS is Numerical Relational System, and m is the measure that satisfies the representation condition. For JCCM, we have already defined ERS, NRS and m in previous section.
- *Definition 6:* Given a scale <*ERS, NRS, m*>, the transformation of a scale f is admissible if m`= f o m (i.e. m` is the composition of *f* and *m*) and <*ERS, NRS, m'*> is a scale. Based on admissible transformation, four different types of scales can be considered as follows [24]:

  1. *Nominal Scale:* Each entity is labelled in categories and there is no ordering relation among them. An example for nominal scale is labelling given programs according to the name of their authors.
  2. *Ordinal Scale:* Entities are categorised in the form of total ordering. The associated values make entities comparable. As an example, program bodies can be assigned degrees from 1 to 5 with comparative meanings (e.g., 1 is the least reliable one and 5 is the most reliable one).
  3. *Interval Scale:* The difference among the assigned numerical values can be quantified in their amount. A new scale m` from *m* can only be obtained through transformations of the form $m`=a*m+b$ where $a>0$. An example for this can be the scale of Celsius that can be converted into Fahrenheit.
  4. *Ratio Scale:* The ratio among the numerical values associated with the entities is used for quantification. The form of transformation is: $m`=a*m$ where $a>0$. The main difference between interval and ratio scale is the existence of true zero-point in ratio scale. An example of ratio scale is the LLOC measure of a program body.

For case of JCCM, it can be very easily proved that <*ERS*, *NRS*, *m*> for JCCM is a ratio scale. Reconsidering the two program bodies *P* and *Q*, $Q/P=2$ and then $a=2$. This implies that $m`=2*m$. Therefore, it can be informally stated that the proposed measure JCCM is defined on ratio scale.

5. Empirical Validation of JCCM and Comparison

There is no specific metrics for calculating JavaScript codes. However, we are comparing our metrics with some popular metrics which are developed to be used for most of the programming languages. For not being developed specifically for JavaScript code, their deficiencies are obvious in comparison with JCCM.

For empirical validation of the JCCM metric, we have analysed thirty JavaScript files shown as Table 6. Most of the files of the analysed scripts were extracted from the web. The statistics that we have collected after analysing these JavaScript codes to evaluate the JCCM measures are shown in Table 7. Actually, our agenda of empirical validation is two-fold. First, we applied well known metrics like LLOC, CC [13], and volume, efforts, difficulty and time estimations from Halstead metrics [8]. Since, these metrics have not been tested in JavaScript; our studies evaluate the applicability of these metrics to JavaScript codes. Second, the statistics that we have collected from those metrics is compared with the values obtained from JCCM. It will prove the usefulness and effectiveness of our proposal.

All these complexity metricss under consideration have been applied on 30 JavaScript files. We believe that the selected 30 scripts are significant in number for comparison since they include different structures and, therefore, contain most of the characteristics of a system required for the validation of the proposed

measure. The complexity values of different measures for the cases are summarized in Table 7.

Table 6. References for Java script.

1.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_function1
2.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_switch
3.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_elseif
4.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_fornext
5.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_break
6.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_array_for_in
7.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_try_catch
8.	http://www.w3schools.com/js/tryit.asp?filename=tryjs_throw
9.	http://www.webdevelopersjournal.com/articles/jsintro3/js_begin3.html
10.	http://www.scriptci.com/modules.php?name=Downloads&d_op=getit&lid=287
11.	http://www.scriptci.com/modules.php?name=Downloads&d_op=getit&lid=885
12.	http://www.java2s.com/Tutorial/JavaScript/0500__Object-Oriented/Inheritance.htm
13.	http://javascript.internet.com/user-details/ip.html
14.	http://www.web-source.net/javascript_status_clock.htm
15.	http://www.buildwebsite4u.com/advanced/javascript-code.shtml
16.	http://www.sislands.com/coin70/week2/NestedLoops1.htm
17.	http://www.sislands.com/coin70/week3/slideshow.htm
18.	http://www.sislands.com/coin70/week7/counter.htm
19.	http://www.java2s.com/Tutorial/JavaScript/0080__Development/Trycatchexception.htm
20.	http://www.java2s.com/Tutorial/JavaScript/0300__Event/SeteventreturnvaluetofalseIE.htm
21.	http://www.java2s.com/Tutorial/JavaScript/0140__Function/NestedfunctioncallI.htm
22.	http://www.java2s.com/Tutorial/JavaScript/0140__Function/ReturnbooleanvaluefromfunctionI.htm
23.	http://www.java2s.com/Tutorial/JavaScript/0140__Function/PassintegertofunctionI.htm
24.	http://www.java2s.com/Tutorial/JavaScript/0180__Math/Mathlog.htm
25.	http://www.java2s.com/Tutorial/JavaScript/0120__String/ConvertStringtouppercase.htm
26.	http://examples.oreilly.com/jscript2/7.2.txt
27.	http://examples.oreilly.com/jscript2/14.3.txt
28.	http://examples.oreilly.com/jscript2/16.2.html
29.	http://examples.oreilly.com/jscript2/14.1.txt
30.	http://examples.oreilly.com/jscript2/6.2.txt

The lines of code, variables (arbitrary names and meaningful names), basic control structures, i.e., sequence, branch, iteration and function call are directly related to complexity of code in Table 3. The weight assigned to a line is 1 even if there is none of these factors in the code of that line since JCCM considers the sequential structure of each line. On the other hand, if this line does not contain any variable, its JCCM value becomes zero. All these prove that the proposed measure considers the size as a factor of complexity, which may not include every line, which are useless and do not contain any variable or operator. The graphs depicted in Figure 1 show the comparison results between the LLOC and the JCCM. It is clear from the graph that JCCM values are normally higher than LLOC. It is because that JCCM consists of complexity values due to other parameters/factors responsible for complexity also. Obviously, it includes the size as a factor, because, JCCM is intended to measure the complexity of each line of code CC can be estimated by $CC=e-n+2p$ for a flow graph having n vertices, e edges and p connected components and attempts to determine the number of execution paths in a program. CC neither considers size, (due to length of program and due to variables) nor the internal complexity of architecture of line. For this reason, for example, CC values for programs 1, 4, 13, 15, 20, 24 and 25, are equal $CC=2$ in Figure 2 and are minimum since these programs have no extra modules. However, JCCM does not only consider the size factor but also complexity due to number of modules and its internal structures. The JCCM values for the above mentioned programs are 7, 33, 27, 16, 9, 19, and 19 respectively, which indicate the complexity differences between programs better and therefore provide more information.

Table 7. Complexity values for different complexity measures.

Program	lLOC	CC	JCCM	Halstead			
				V	D	E	T
1	2	2	7	13	1	13	0
2	14	5	39	148	3	244	24
3	8	4	35	148	4	592	32
4	3	2	33	79	6	474	26
5	6	4	44	114	8	912	50
6	7	3	35	93	2	186	10
7	9	3	31	167	3	501	27
8	15	9	72	237	4	948	52
9	12	3	68	212	6	1274	70
10	32	3	163	212	12	2544	141
11	20	5	40	237	4	948	52
12	13	3	62	220	4	880	48
13	4	2	27	129	4	516	28
14	21	5	120	748	6	4488	249
15	4	2	16	148	3	444	24
16	10	7	153	152	16	2432	135
17	10	4	42	366	12	4392	244
18	8	3	34	93	2	186	10
19	7	3	10	63	1	63	3
20	3	2	9	48	2	96	5
21	5	3	10	23	1	13	1
22	6	3	24	76	2	152	8
23	5	4	30	93	2	186	10
24	4	2	19	76	2	152	8
25	4	2	19	48	1	48	2
26	5	4	28	171	3	513	28
27	6	5	54	259	4	1039	57
28	13	10	107	514	12	6168	342
29	6	4	38	192	4	768	42
30	20	31	128	696	4	2784	154

A graph which covers the comparison between CC, LLOC and JCCM is also plotted in Figure 3, to observe similarities and differences between them. A close inspection of this graph shown as Figure 3

shows that JCCM is closely related and CC and LLOC. This can easily be seen and observed in Figure 3, in which JCCM, CC and LLOC reflect similar trends. In other words, high JCCM values are due to large size, large number of variables, large number of iterations, branching structures, function calls or including all of them together in their content. For example, JCCM has the highest value for script 10 (163), which is due to having the maximum lines of code (32), variables and complex control structures.

We have also compared JCCM with the Halstead metrics. The graphs between JCCM, volume, difficulty and time have been demonstrated in Figure 4. It is observed that JCCM has similar trends with volume, difficulty and time. Further, JCCM values are less than volume measurement of Halstead but are almost similar with time measurement of Halstead in most of the scripts (except 14, 17 and 28). Actually, time measurement of Halstead is approximate time spent to understand a program and JCCM reflects the similar values to time measurement. This proves that JCCM is also a strong predictor of understandability.

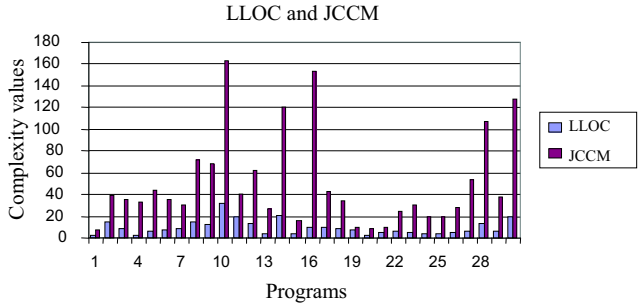


Figure 1. Comparison between logical line of code and JCCM.

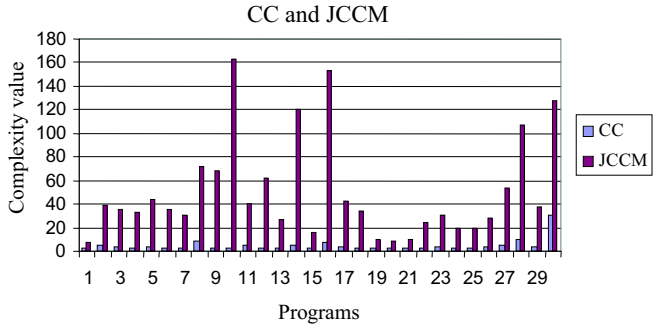


Figure 2. Comparison between cyclomatic complexity and JCCM.

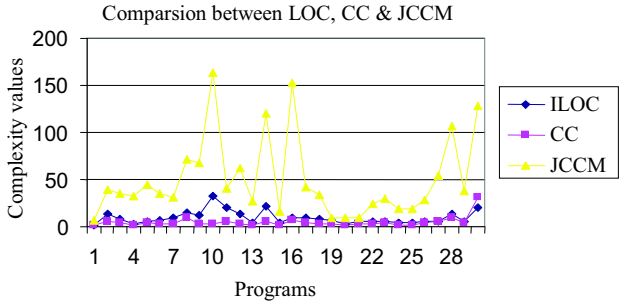


Figure 3. Relative graph between ILOC, CC and JCCM.

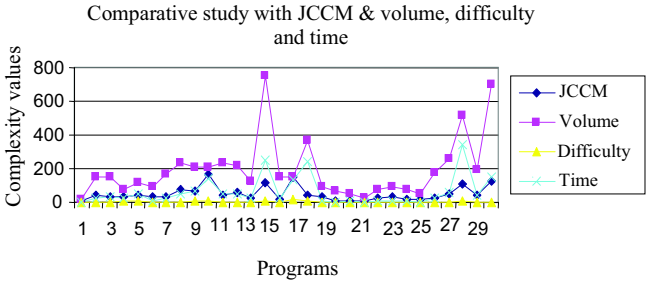


Figure 4. Relative graph between volume, difficulty, time and JCCM.

6. Conclusions

In the paper, we presented a measurement method for evaluating quality of JavaScript codes. The experiments show that none of the well known metrics/measures are as efficient to evaluate the quality of JavaScript, as JCCM. JCCM considers all the factors responsible for complexity, which is the major difference between the proposed measurement technique and others. The metric is theoretically evaluated and empirically validated. For future research, more experiments can be held and more details can be discussed in order to create a more stable metric.

References

[1] Basci D. and Misra S., "Entropy as a Measure of Complexity of XML Schema Documents," *The International Arab Journal of Information Technology*, vol. 8, no. 1, pp. 16-25, 2011.

[2] Basci D. and Misra S., "Data Complexity Metrics for Web-Services," *Advances in Electrical and Computer Engineering*, vol. 9, no. 2, pp.9-15, 2009.

[3] Basci D. and Misra S., "Measuring and Evaluating a Design Complexity Metric for XML Schema Documents," *Journal of Information Science and Engineering*, vol. 25, no. 3, pp. 1415-1425, 2009.

[4] Briand L., Bunse C., and Daly J., "A Controlled Experiments For Evaluating Quality Guidelines on The Maintainability of Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 27, no. 6, pp. 513-530, 2001.

[5] Costagliola G. and Tortora G., "Class Points: An Approach for the Size Estimation of Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 52-74, 2005.

[6] Fenton N. and Pfleeger S., *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing, 1997.

[7] Fenton N., "Software measurement: A Necessary Scientific Basis," *IEEE Transaction Software Engineering,* vol. 20, no. 3, pp. 199-206, 1994.

[8] Halstead M., *Elements of Software Science*, Elsevier North-Holland, 1997.
[9] IEEE Computer Society: *Standard for Software Quality Metrics Methodology*, Revision IEEE Standard, 1998.
[10] Kearney J., "Software Complexity Measurement," *Communications of the ACM*, vol. 29, no. 11, pp. 1044-1050, 1986.
[11] Kemerer C., "Reliability of Function Points Measurement: A Field Experiment," *Communications of the ACM*, vol. 36, no. 2, pp. 85-97, 1993.
[12] Kushwaha D. and Misra A., "Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 5, pp. 1-5, 2006.
[13] McCabe T., "A Complexity Measure," *IEEE Transactions Software Engineering*, vol. 2, no. 6, pp. 308-320, 1976.
[14] Misra S. and Ibrahim A., "Weighted Class Complexity: A Measure of Complexity for Object Oriented Systems," *Journal of Information Science and Engineering*, vol. 24, no. 1, pp. 1689-1708, 2008.
[15] Misra S. and Ibrahim A., "Unified Complexity Metric: A measure of Complexity," *in Proceedings of National Academy of Sciences Section A*, pp. 167-176, 2010.
[16] Morasca S., *Software Measurement, Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing, 2001.
[17] Pressman R., *Software Engineering: A Practitioner's approach,* McGraw Hill, 2001.
[18] Vessey I. and Weber R., "Research on Structured Programming: An Empiricist's Evaluation," *IEEE Transactions on Software Engineering,* vol. 10, no. 4, pp. 394-407, 1984.
[19] W3schools, available at: http://www.w3schools.com /JS/ js_intro.asp, last visited 2009.
[20] Wang Y. and Shao J., "A New Measure of Software Complexity Based on Cognitive Weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69-74, 2003.
[21] Wand Y. and Weber R., "Toward a Theory of the Deep Structure of Information Systems," *in Proceedings of International Conference on Information Systems*, Denmark, pp. 61-71, 1990.
[22] Weyuker E., "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357-1365, 1988.
[23] Zuse H., *Software Complexity Measures*, Walter de Gruyter, Berlin, 1998.
[24] Zuse H., *Framework of Software Measurement*, Walter de Gruyter, Berlin, 1998.

Sanjay Misra obtained M.Tech. degree in software engineering from Motilal Nehru National Natioanal Institue of Technology, India and D.Phil. from University of Allahabad, India. Presently, he is a Professor of Computer Engineering and chairing the Cyber Security Department of Federal University of Technology, Nigeria. He is a software engineer and previously held academic positions at Atilim University, Turkey, Subharati University, and UP Technical University India. His current researches cover the areas of: software quality, software measurement, software metrics, software process improvement, software project management, object oriented technologies, XML, SOA, web services, and cognitive informatics. He published more than 100 papers in these areas. He is the founder chair of several annual international workshops: Software Engineering Process and Applications, Software Quality and Tools and Techniques in Software Development Processes. The proceedings of these workshops are published by Springer and IEEE. Presently, he is chief editor of International Journal of Physical Sciences and serving as editor, associate editor and serving as editorial board member of several journals of international repute.

Ferid Cafer received his BSc degree from Bilkent University, Ankara and MSc in software engineering in 2010 from Atilim University, Turkey. Presently, he is working as software engineer in Servus Bilgisayar, Turkey. His area of interests are software measurements, object oriented design and programming, especially in python, web design and development, project management and software quality. He is a motivated researcher and has produced some very good results in recent research papers.