Estimating the maintenance effort of JavaScript Applications

Ioannis Zozas
Department of Informatics & Telecommunications Engineering
University of Western Macedonia
Kozani, Greece
izozas@uowm.gr

Stamatia Bibi
Department of Informatics & Telecommunications Engineering
University of Western Macedonia
Kozani, Greece
sbibi@uowm.gr

Apostolos Ampatzoglou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
apostolos.ampatzoglou@gmail.com

Panagiotis Sarigiannidis
Department of Informatics & Telecommunications Engineering
University of Western Macedonia
Kozani, Greece
psarigiannidis@uowm.gr

***Abstract* – Successful software project survival and progress over time is highly dependent on effectively managing the maintenance process. Estimating accurately maintenance process factors like the maintenance effort and the level of changes required for a new release is considered a crucial task for allocating resources. In this work we examine the maintenance process factors of JavaScript applications, which at the moment are understudied despite the need of language specific maintenance models. Furthermore we propose two maintenance indices for estimating the changes and the effort required for maintaining JavaScript applications by considering a variety of maintenance drivers. We evaluated the proposed indices through a case study on 5,788 releases coming from 60 popular JavaScript applications. The results show that project activity factors (i.e., number of open bugs and number of corrective maintenance activities) are important maintenance drivers. The proposed indices are evaluated in terms of predictive and discriminative power and both achieve high accuracy.**

Index Terms - software maintenance effort; JavaScript; maintenance index; software development; open source software;

I. Introduction

Nowadays, mature software organizations collect a wealth of data regarding software development and maintenance, expecting to acquire knowledge for effectively monitoring the maintenance process. According to the IEEE 1219 (IEEE Std 1219, 1998) [13] software standards document, **software maintenance** is defined as the "*Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment*", while **maintenance effort** is subsequently defined as the "*effort required to reduce or eliminate maintenance problems"*. According to Lehner [17], the programming language adopted differentiates in a great degree the effort required for its maintenance and therefore it is important to generate language-specific software maintenance models.

In this work we focus solely on investigating the maintenance process drivers of JavaScript (JS) applications. Maintenance effort of JS applications is largely understudied [26] despite the fact that according to GitHub[1] JavaScript is among the most popular programming languages. The motivation behind the need to analyze JS applications resides upon the fact that a) JS is considered as a weakly typed programming language [26] that can generate unpredictable results that may cause problems to the maintenance of projects, b) many programmers rely upon popular JS frameworks for building their web applications so it is interesting to further explore the potentials of JS frameworks in terms of maintenance and adjustment to user demands.

In order to explore the maintainability factors that drive the maintenance process of JavaScript applications, we performed a case study on 5,788 releases coming from 60 popular open source JavaScript applications. We considered in our analysis a variety of metrics related to the internal source-code quality, size and complexity of software, metrics related to the end-user community and metrics relevant to the type of the maintenance activities performed.

In particular we investigated:

a) The maintenance activities (corrective, adaptive& perfective and preventive) that are more frequent in JS applications with respect to the size of the application.
b) The factors that are considered significant in estimating maintenance effort and changes of JS applications. Based on these factors we built two indices for estimating the maintenance changes and the maintenance effort of JS applications.
c) The validity of the two indices based on correlation, consistency, predictability, discriminative power, and reliability evaluation criteria.

In Section 2 we present related work and in Section 3 we describe the proposed indices. In Section 4, we present the study design that was used for evaluation purposes. The evaluation results are presented and discussed in Sections 5 and 6. We present threats to validity in Section 7, and conclude the paper in Section 8.

[1] https://github.com/search, https://octoverse.github.com/

II. Related work

Several models have been proposed so far that can help practitioners towards assessing the effort required to maintain a software project by quantifying a set of high level quality metrics [18], [10], [8]. Oman [24] and later Coleman [9] introduced a Maintainability Index (MI) based more on *complexity* and *size*, by utilizing the Halstead Volume, McCabe's Cyclomatic Complexity, Lines of code and Comments rate. In expansion, Thamburaj [31] proposed a maintenance effort prediction model based on the object-oriented cognitive complexity metrics through statistical techniques. As for complexity, Chandra [7] assessed maintainability by outlining the importance of *size* and the *complexity* described by the source code depth of inheritance tree. He used Support Vector Machine for the regression for forecasting of software maintenance effort with the Univariate and Multivariate approach. Alomari [2] used program slicing to estimate maintenance effort, by using three different granularities of slice (i.e., line, function, and file) analysed and compared the changes and complexity.

Milicic [20] apart from project size introduced the factor of the project life cycle by focusing on detecting useful patterns and interesting causalities in a simplistic approach. Ahn [1] introduced the factor of maintenance *activities* and the nature of the development team. He suggested an exponential function model which can show the relationships among the maintenance efforts and maintenance environment factors. Furthermore, Niessink [23] the type of each maintenance task, and furthermore Jorgensen [14] focused on the type of maintenance activities (i.e., whether corrective, adaptive, perfective or preventive). Later Chua [8] and Hayes [12] utilized determination of maintenance *changes* types and *duration*, by identifying factors (i.e., maintenance type activities, code size, changes and age of changes) that aligned in response to changes made by each maintenance task. Chua categorized maintenance effort data using regression analysis to evaluate adaptive and functional changes for efficacy determination. Hayes focused more on adaptive changes by performing regression models.

In an alternative direction, Yang [32] introduced *modularity* factors like data structures and field attributes, as well as defect as an *internal quality* factor. As for modularity, Sjoberg [30] focused on both the physical aspect of source code introducing factors like physical files and directories, as well as code smells for internal quality. Anda [4] introduced factors like code smells and source code vulnerabilities, while Mondal [22] performed an empirical study to compare the maintenance efforts required for cloned and non-cloned code.

The key focus of our research is to go beyond current literature by:

- Examining maintenance process factors related to JavaScript applications. The maintenance effort of JS applications is still understudied, despite the importance of language –specific models [17].
- Factors like complexity, size and modularity are extensively incorporated into research efforts, while others like internal quality and activity are less studied. In this study we examine all the aforementioned factors in the context of JS applications.

III. Proposed maintenance indices

In this section initially we describe the maintenance factors and the associated metrics that participate in the study and then we present the calculation of the indices for estimating a) the changes and b) the effort required to maintain JavaScript applications.

A. *Maintenance factors and metrics*

In order to assess the effort and the changes required to maintain JavaScript applications we considered a set of four high-level factors that are considered as important maintenance process drivers. Each set of factors can be assessed by the metrics presented in Table 1. We notice that all the metrics presented in Table 1 are calculated for each subsequent release separately.

Size and modularity metrics: These metrics are relevant to the source code size of an application and the modularity, i.e., logical partitioning of the application. Examples of these metrics are the Lines of Code and the Number of Functions correspondingly [5][15].

Complexity factors: These metrics refer to source code complexity and are calculated based on internal software entity interactions [16]. Such metrics are Cyclomatic Complexity and Cognitive Complexity among others.

Internal quality factors: As internal quality factors we considered cumulative metrics such as Code smells and Internal bugs [11] and also code duplications [4]. These metrics can be considered high-level indicators of the weaknesses in design and the reliability of the application.

Activity factors: Regarding the activity metrics we considered metrics that a) take into account the open source software nature of the applications under study and b) that can be accurately derived from GitHub repository from which the applications were retrieved. Therefore, we selected to include in the analysis the metrics like number of forks, developers, commits etc., that were directly available from GitHub.

Process Metrics: As process metrics we considered the *Days Between Releases, the Incremental Changes (IC)* metric that is calculated as the number of functions added or removed or modified in a particular release and the *Maintenance Effort (ME)* metric that is calculated as the ratio between the *Incremental Changes (IC)* metric to the *Days between Releases (DBR).* Also we considered the number of commits per different type of maintenance activity (Adaptive & Perfective, Corrective, and Preventive) as a proxy of the intensity of the different types of Maintenance activities performed per release [21].

Factor	Metric Name	Description and Values
Size and modularity	LOC	Lines Of Code (*SonarQube*)
	FILES	Total files analysed (*SonarQube*)
	DIRS	Total directories (*SonarQube*)
	NOF	Number of Functions (*JSClassFinder*)
	NOA	Number of Attributes (*JSClassFinder*
	NOC	Number of Classes (*JSClassFinder*)
	NOM	Number of Methods (*JSClassFinder*)
	COMMENTS	Lines of comments (*JSClassFinder*)
Complexity	DIT	Depth of inheritance tree (*JSClassFinder*)
	CMPLX	Complexity (*SonarQube*)
	CCN	McCabe's Cyclomatic complexity (*SonarQube*)
	CGCMPLX	Cognitive complexity (*SonarQube*)
Internal quality	D_LINES	Duplicate lines of code (*SonarQube*)
	D_BLOCKS	Duplicate blocks of code (*SonarQube*)
	CODE_SMELLS	Code smells (*SonarQube*)
	VULNERABIL	Code vulnerabilities (*SonarQube*)
	BUGS	Number of bugs (*SonarQube*)
Activity	ACT	Number of commits regardless of the type of task performed (*GitHub*). Represents the cumulative activity in a release.
	OP_BUGS	The number of bugs from Github's issues list (issue tracking).
	CONTRIBUTORS	Number of contributors (*GitHub*)
	POPULARITY	Number of forks and stars (*GitHub*)
Maintenance process metrics	ADP_ACT	Adaptive & perfective activities: Measured as the number of commits related to Adaptive/Perfective task commits (manual ranking based on vocabulary keywords e.g. add, improve, update etc.) [21]
	COR_ACT	Corrective activities: Measured as the number of commits related to Corrective task (manual ranking based on vocabulary keywords e.g. bug, fix, correct etc.) [21]
	PRV_ACT	Preventive activities: Measured as the number of commits related to Preventive tasks (manual parsing and ranking based on vocabulary keywords e.g. refactor, remove, replace etc.) [21]
	DBR	Days between releases (*GitHub)*
	IC	Incremental changes calculated as Number of functions added or removed or modified per release (*SonarQube*)
	MAINT_EFFORT	Incremental Changes / Days between each release

TABLE 1 - Proposed metrics

B. Calculation of indices

In order to quantify the changes and the effort required to maintain JS applications we calculated two indices, namely the *Maintenance Changes index* (*MCi*) and the *Maintenance Effort index (MEi)*. We performed stepwise regression with backward elimination [3] so as to produce two aggregated measures for the two indices. As dependent variables, we used a) *Incremental Changes (IC)* metric that is calculated as the number of functions added or removed or modified in a particular release (index *MCi*) [29] and b) the *Maintenance Effort (ME)* metric that is calculated as the ratio between the *Incremental Changes (IC)* metric to the *Days between Releases (DBR)* (index *MEi*). The two indices are calculated based on the values of the size, complexity, quality and activity metrics, which are the independent variables.

The rationale behind selecting stepwise regression with backward elimination was to include in the model the most relevant predictors ($p<0.05$) regarding the maintenance process metrics by isolating in each step the metrics that explain more effectively the proportion of variance of the two dependent variables. The outcome of the regression is an equation in the following form:

$$\textbf{Index} = \textbf{Constant} + \sum_{i=0}^{i<num\_metrics} \textbf{B(i)} * \textbf{metric(i)}$$

Where *Index* refers to the maintenance process index, and *B(i)* is the unstandardized Beta of each metric that shows the size and the sign of the corresponding coefficient. The index derived for the calculation of *MCi* and *MEi* along with the associated metrics and the B(i) coefficients are presented in Table 2, as well as the standardized *Beta* coefficient (by subtracting the mean from the variable and dividing by its standard deviation) which can be used to compare the strength of the effect of each individual metric to the dependent variable derived from the stepwise regression with backward elimination. The higher the absolute value of the beta coefficient, the stronger the effect. The sign of Beta shows whether this metric affects positively or negatively the index.

Maintenance Changes Index (MCi)			**Maintenance Effort Index (MEi)**		
	B(i)	**Beta**		**B(i)**	**Beta**
(Constant)	92.92		**(Constant)**	-23.4	
OP_BUGS	1.42	0.19	**OP_BUGS**	3.32	0.4
D_LINES	-0.008	-0.18	**FILES**	0.08	0.07
LOC	0.007	0.29	**CMPX**	263.2	0.43
NOA	0.03	0.07	**D_LINES**	-0.004	-0.15
COR_ACT	-8.17	-0.4	**LOC**	0.05	0.45
CMPX	991.7	0.17	**COR_ACT**	-2.99	-0.3
ACT	-0.034	-0.3	**ACT**	-0.05	-0.4

TABLE 2 – Indices calculation

We observe that the number of changes required for maintaining JS applications *(MCi)* depend on 7 factors. The three most contributing to *MCi* are the *Number of Corrective tasks (COR_ACT),* the *Cumulative Activity (ACT)* and the *Lines of Code (LOC)*. The Maintenance Effort Index also depends on 7 factors, with *LOC*, *number of Open Bugs* (*OP_BUGS)* and *ACT* being the most important ones. Also we observe that D_LINES, COR_ACT, ACT present negative signs in their coefficients which in our case means that they affect positively the indices. For example when there is intense activity in a project we expect that the number of changes in a final release will be minimized. This can be explained by the fact that intense activity usually includes a set of small, frequent changes of limited scope contrary to more rare activity that usually include extensive changes of wider scope.

IV. CASE STUDY DESIGN

In order to empirically investigate the validity of the proposed indices, we performed a case study on 5.788 releases from 60 open source JS applications following the guidelines of Runeson [27]. To investigate and compare the validity of the proposed indices we employ the properties of Correlation, Consistency, Predictability, Reliability described by 1061:1998 IEEE [13]. The sixth property of tracking was omitted from this study, since it requires heavy- weight process analysis, for each release separately that can form a standalone research effort, complementary to the current one.

A. *Objectives and Research Questions*

The overall goal of this case study is twofold a) to explore the types of maintenance actions that occur concerning JS applications and b) to evaluate the two maintenance indices MCi and MEi proposed for estimating the amount of changes and the effort required to maintain JS applications. According to this goal, three research questions are formulated:

[RQ1] ***What types of maintenance actions occur concerning JS OSS applications?***

In the first research question we want to explore the type of maintenance activities that are more frequent in the context of JavaScript applications maintenance. These activities can be adaptive/perfective, corrective or preventive maintenance activities [14][6].

[RQ2] ***What are the correlation, consistency and predictability of the proposed maintenance indices?***

In the second research question we investigate the validity of the proposed maintenance indices, with respect to the proposed [13] validity criteria (correlation, consistency, predictability and discriminative power).

[RQ3] ***What is the reliability of the proposed maintenance indices?***

In the third research question we investigate the validity of the reliability criterion [13]. To achieve this we test each of the other validation criteria on different types of projects based on their size.

B. *Case Selection & Unit of analysis*

The cases of the study are the 60 most popular JS applications (see Table 3) according to GitHub until October 2017 and all their releases, in total 5.788 releases. In order to select these applications we applied the following criteria a) the application should present more than 90% of JavaScript source code b) the application should have at least a two year period lifespan and should present more than 10 releases. Out of the 5,788 releases recorded in total we randomly selected 70% (4.040 releases) to serve as a training set so as to define the indices presented in Section 3. The rest 30% (1732 releases) was used as a test set to validate the proposed indices (see Section 5).

C. *Data collection*

For each JS project we have recorded the metrics presented in Section 3. The metrics have been collected in multiple ways for each project release: (a) the actual maintenance effort, project rating, open bugs, release information and commits has been recorded based on statistics provided by the GitHub platform; (b) commits and release notes provided by the platform were categorized for Adaptive/Perfective, Corrective and Preventive tasks [19][25] on word frequencies. Based on ranked vocabulary studies [21] each maintenance task was identified by parsing the comments accompanying each commit (e.g. for Adaptive/Perfective 25 keywords like "add", "create", etc., for Corrective 24 keywords like "correct", "fix", etc., and for Preventive 18 keywords like "refactor", "redesign", etc.), (c) structural metrics (like LOC) have been calculated using the *SonarQube* platform for static analysis; (d) JS specific metrics (like NOC) have been calculated using the *JSClassFinder* tool [28]; (e) more complex metrics like Incremental Change were calculated by comparing the relativeness of subsequent releases based on the names of functions included in each release and their respecting size.

#	Projects	Popularity	Releases	#	Projects	Popularity	Releases
1	React	88,009	67	**31**	datepicker	14,373	46
2	vue	73,870	207	**32**	swagger-ui	13,821	98
3	javascript	68,566	75	**33**	sequelize	13,274	227
4	jQuery	59,387	146	**34**	grunt	13,101	11
5	Three.js	47,600	79	**35**	vuex	12,658	34
6	Chart.js	39,927	37	**36**	medium-edit	12,492	150
7	Express	39,773	269	**37**	jsPDF	11,133	19
8	Moment	37,944	62	**38**	raphael	10,777	38
9	webpack	35,765	253	**39**	jquery-validat	10,442	17
10	material-u	33,675	161	**40**	karma	10,409	178
11	Ghost	29,278	116	**41**	eslint	10,310	171
12	yarn	29,032	110	**42**	fabric.js	10,235	62
13	axios	29,002	34	**43**	knockout	9,908	49
14	lodash	28,898	380	**44**	Parsley.js	9,347	89
15	fullPage.js	25,686	61	**45**	johnny-five	9,326	74
16	async	24,338	71	**46**	jshint	9,015	66
17	Modernizr	23,925	27	**47**	vue-router	8,867	51
18	Pdf.js	23,800	44	**48**	fine-uploader	8,848	99
19	video.js	22,406	327	**49**	marionette	8,521	143
20	hexo	20,791	120	**50**	vue-resource	7,895	46
21	clipboard	20,739	30	**51**	art-template	7,745	17
22	hyper	20,508	42	**52**	ui-grid	7,388	78
23	RxJS	19,349	104	**53**	cropper	7,307	52
24	pixi.js	18,378	79	**54**	angular-fstack	7,178	84
25	fetch	17,461	26	**55**	flot	6,875	17
26	bower	17,233	102	**56**	plupload	5,783	33
27	dropzone	15,909	97	**57**	form	5,683	16
28	wbtorrent	15,853	257	**58**	openlayers	4,313	161
29	q	14,809	65	**59**	jQuery-Mask-	3,954	136
30	jasmine	14,796	58	**60**	jquery-form	4,788	16

TABLE 3 - GitHub JavaScript project data set

D. *Data analysis*

To answer RQ1 we calculate the standard descriptive statistics (Min, Max, Mean, Median and Standard error of Mean [13]) of the target variables (commits categorized by each type of different maintenance tasks performed, either adaptive/perfective, corrective or preventive) [14][6]. Also we present the intensity of each type of maintenance activity in subsequent releases of six applications participating in the study that are considered as representatives of small-, medium and large sized applications.

To answer RQ2, we use the maintenance indices presented in Section 3, as assessors of the number of maintenance changes and the maintenance effort. Then we compare the output of the two indices to the actual values of the two maintenance process metrics under study, Incremental Changes *(IC)* and Maintenance Effort *(ME)* correspondingly. Concerning *Correlation* and *Consistency* we will use the Pearson correlation and the Spearman correlation coefficients respectively and the levels of statistical significance. Regarding *Predictability*, we will investigate the independent variable level of statistical significance of the effect over depended, and the mean standard error as the accuracy of the model [13]. Regarding the Discriminative power of the indices we evaluate them based on three metrics the *Precision* (positive predictive power), the *Recall* (sensitivity of the model), and *F-measure* (models accuracy).

To answer the third RQ, we will perform all the aforementioned tests of RQ2 separately, on two groups of the data set. The separation of the data set will be based on the project size (i.e., large and small groups respectively, measured in KLOC). The group of large-sized projects presents a median of 13 KLOC and contains 21 projects. The group of small-sized projects presents an average of 5.89 KLOC and contains 21 projects.

V. Results

In the current section we present the results of the case study performed to assess the level and the type of changes performed to maintain JS applications. In Section V.A we discuss RQ1 and present the frequency of the various types of maintenance tasks, as recorded in the successive versions of JS applications. In Section V.B we present the RQ2 results regarding the empirical validation of the proposed index in terms of correlation, consistency, predictive and discriminative power. In Section V.C we summarize the RQ3 results regarding the assessment of the reliability of the indices.

[RQ1] ***What types of maintenance actions occur concerning JS OSS applications?***

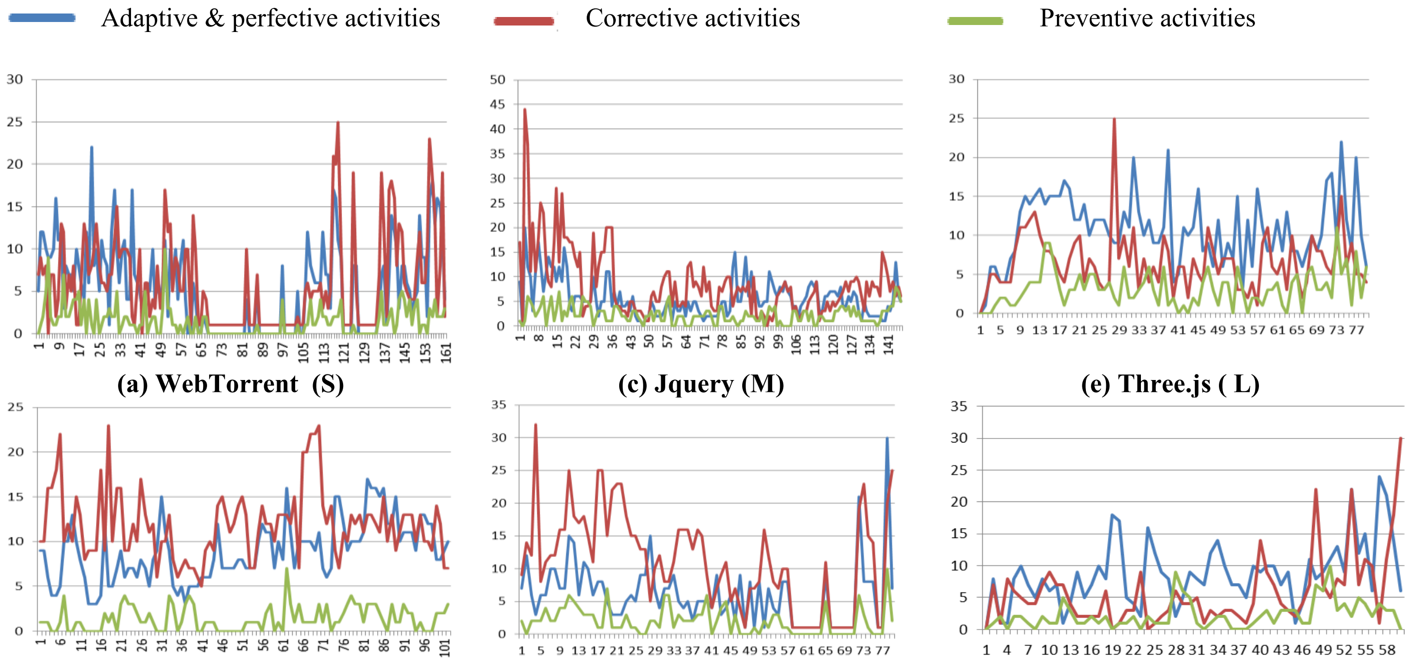
To investigate the type of maintenance tasks that are more frequently applied in JS applications we have recorded the number of commits related to Adaptive & Perfective tasks, Corrective tasks and Preventive tasks. Table 4 presents the descriptive statistics for the commits implementing the three types of maintenance tasks. The last two columns of Table 4 present in total the number of commits per task type for all 5.788 versions analyzed and their associated percentage. We observe that the majority of maintenance tasks performed focus on corrective actions (bug fixing), followed by adaptive &perfective tasks that usually include the addition of new functionalities, or the extension of existing ones. Preventive tasks are less frequent, a fact that reveals that code refactoring in JS applications are relatively rare.

Maintenance Tasks	Range (Min – Max)	Mean	Median	Standard Deviation	Total	%
Adaptive & Perfective	0 – 92	8.1	6	7.63	46,650	38.56
Corrective	0 – 78	10.1	7	10.2	58,259	48.17
Preventive	0 – 38	2.8	2	3.73	16,048	13.27
				Total	120,957	100.00

TABLE 4 - Maintenance tasks descriptive statistics

In figure 1 we present the number of commits implementing the three types of maintenaance activities between successive releases for six JS applications. We selected to observe the evolution of maintenance activities through time in these six projects beacause we believe that they are representative of small-sized (Webtorrent, Bower), medium-sized (Jquery, Pixi.js) and large-sized (Three.js, React) applications (small-sized projects with < 13 KLOC and large-sized projects with > 13 KLOC) and they are within the 25% and 75% quartiles of their associated group of projects in terms of the *Incremental Changes* variable. By observing Figure 1 we conclude that:

- In large projects (Fig1.e, f) maintenace actions concern mostly adaptive/ perfective tasks, while the corrective tasks are also performed between releases without though presenting a particular trend.
- In medium-sized projects (Fig1.c, d) maintenace actions concern mostly corrective tasks. In that case we observe that the intensity of the corrective actions is increased in the early releases compared to the subsequent releases that present lower intensity with respect to corrective actions. Also the intensity of adaptive & perferctive activities seems to remain stable throught the maintenance cycle.
- In small-sized projects (Fig1.a, b), after inspecting also the rest of these type of projects we did not identify any pattern regarding the intensity of tasks. The applications are split into two groups the ones that mostly present adaptive& perfective maintenance activities and the ones that present mostly corrective activities.
- Finally regarding preventive actions all three types of application, small, medioum and large-sized present a common pattern. Preventive actions a) are limited compared to the other two types of activities, b) their intensity remains stable throught the maintenance lifecycle and c) in all cases they range from 0 to ~ 7 related commits.



(a) WebTorrent (S)

(c) Jquery (M)

(e) Three.js (L)

(b) Bower (S)

(d) Pixi.js (M)

(f) React (L)

FIGURE 1. Changes performed during maintenance cycle for small (S) , medium (M) and large (L) sized projects.

[RQ2] ***What are the correlation, consistency and predictability of the proposed maintenance indices?***

In order to answer the second research question, we compare a) the estimations performed by *MCi* to the actual number of maintenance changes and b) the estimations performed by *MEi* to the actual maintenance effort. The comparison is made in terms of correlation, consistency and predictability [13]. The results are presented in Table 5 grouped by each validity criterion. For every criterion we present a set of success indicators. For correlation and consistency we present the the coefficient and significance indicators based on Pearson and Spearman correlations, while for Predictability the R^2, the standard error and the significance indicators. For the discriminative power of the indices we employ precision, recall and F-measure accuracy metrics. Statistically significant results are denoted in italics.

Based on the results presented in Table 5, both indices present very satisfying results in terms of the correlation, the consistence the predictive and discriminative power. MCi presents slightly improved results in both 4 criteria. This can be explained by the fact that it is safer to predict the level of changes required in a subsequent version instead of the effort required for them. Though, the results regarding MEi index are also very close and comparable to those of MCi. Concluding we should mention that both indices offer predictions significant at the 0.10 level, they are both strongly correlated to the actual values of maintenance changes, and effort (Pearson correlation coefficient > 0.5). Also the indices rank maintenance activities consistently with respect to the changes performed (Spearman correlation coefficient = 0.59) and the effort required (Spearman correlation coefficient = 0.53).

Validity Criteria	Success Indicator	MCi	MEi
Correlation	Coefficient	0.63	0.51
	Significance	*0.04*	*0.08*
Consistency	Coefficient	0.59	0.53
	Significance	*0.05*	*0.07*
Predictability	R-Square	45.7%	39.5%
	Std. Error	675.1	621.7
	Significance	*0.08*	*0.09*
Discriminative power	Precision	73%	61%
	Recall	76%	66%
	F-measure	74%	64%

TABLE 5 – Success criteria for MCi and MEi

For assessing the discriminative power of the model we classified the values of the dependent variables into 4 groups representing the small, average, high and very high number of changes and maintenance effort respectively. The cut-points of the four groups were defined by adopting equal-frequency binning. Then we classified the "point" estimates of the two indices into the aforementioned groups and derived an interval estimate. The accuracy of the interval estimate was then evaluated with precision, recall and f-measure metrics. The results show that the discriminative power of MCi is very strong (F-measure>70%) with MEi presenting very satisfying results (F-measure >60%).

[RQ3] ***What is the reliability of the proposed maintenance indices?***

In this section we evaluate the two indices in terms of reliability and we split our test set into two sets: small-sized projects (< 13 KLOC) and large-sized projects (> 13 KLOC). All the tests discussed in RQ2 are replicated for these two sets and the results are outlined in Table 6. With italics we denote statistically significant results. The results of Table 6 suggest that with respect to all criteria, the two indices are more accurate in the group of large-sized projects. Concerning reliability, MCi has been validated as a reliable metric regarding correlation, consistency, predictive and discriminative power. MEi, has been validated as a reliable metric regarding correlation, predictive and discriminative power but not regarding consistency. In particular, MEi was not able to accurately rank small-sized projects.

	Validity Criteria	**Success Indicator**	**MCi**	**MEi**
Small-sized JS applications	Correlation	Coefficient	0.55	0.46
		Significance	*0.06*	0.25
	Consistency	Coefficient	0.65	0. 45
		Significance	*0.10*	0.23
	Predictability	R-Square	32.7%	38.2%
		Std. Error	702.3	714.7
		Significance	*0.08*	*0.09*
	Discriminative power	Precision	64%	58%
		Recall	56%	48%
		F-measure	61%	53%
Large-sized JS applications	Correlation	Coefficient	0.73	0.61
		Significance	*0.01*	*0.04*
	Consistency	Coefficient	0.68	0.62
		Significance	*0.05*	*0.07*
	Predictability	R-Square	57.7%	52.5%
		Std. Error	598.7	582.3.7
		Significance	*0.10*	*0.02*
	Discriminative power	Precision	83%	78%
		Recall	77%	72%
		F-measure	79%	75%

TABLE 6 – MCi, MEi Reliability

VI. DISCUSSION

A. Interpretation of results

The results of the analysis of the maintenance process data of 60 JS applications show that JS maintenance process estimations need to take into consideration metrics related to the development team activity. Activity metrics like *Open Bugs*, *Corrective Activities* and *total Activity* participated in both indices. The number of Open Bugs that are related to the problems reported by the end-user community seem to be an important maintenance driver that increases the need of maintenance changes along with the amount of effort allocated to maintenance activities. Contrary to that increased developer activity, and increased number of corrective activities seem to limit the total number of changes and the effort required to maintain JS applications. Additionally regarding the types of maintenance activities performed we observe that *Adaptive & perfective tasks* are the most frequent activities during the maintenance of JS applications. *Corrective tasks* are also very frequent while *Preventive tasks* seem to be limited and stable through the maintenance cycle. This finding is in contrast to traditional estimation regarding maintenance activities that suggest that preventive activities sum up to 50% , while Adaptive& perfective sum up to 25% [6][14].

Based on the aforementioned:

Practitioners should keep in mind that *Adaptive & perfective tasks* are expected to occupy more than 40% of the maintenance activities. Therefore caution should be taken when designing JS applications so as to allow easy implementation of new functionalities. Additionally practitioners should take into consideration the activity metrics that seem to affect the changes and the effort required to maintain JS applications. It seems that is preferable to perform maintenance activities that include a set of small, frequent changes of limited scope contrary to more rare activities that usually include extensive changes of wider scope.

In this context **researchers** are also advised to further explore the maintenance activities performed through time, especially in the case of small-sized JS applications, for which we were not able to reach a safe conclusion. Additionally we encourage them to concentrate on maintenance activity metrics by introducing new metrics related to the activity of the development team and the end-user community.

B. Threads to validity

We will discuss the threats to validity identified for the current study, according to the guidelines of Runeson [27].

With respect to ***Construct validity*** we can identify one thread posed by the selection of factors and metrics participating in the calculation of the two maintenance indices. The estimation indices have been built from a variety of metrics, most of them appointed by relevant literature, describing both the activity of a project and its internal quality and structure. Though we should appoint that several object-oriented metrics were not included in the model due to the fact that JS language primary to 2017, did not support the clear definition of classes. Therefore a replication of the study in more recent JS projects can shed light regarding the effect of object-oriented metrics to the effort required to maintain JS applications. We acknowledge though that the maintainability indicators should be customized when the proposed methodology is applied in the context of proprietary software.

Internal Validity is not applicable in the scope of this study, since it is not our target to identify causal relationships between the maintenance effort and the associated factors or metrics. With respect to ***Reliability*** we believe that the followed research process ensures the reliability and the safe replication of our study. The data collection process was fully automated with the help of the tools presented in the Case Study Design Section while the data analysis methods adopted are also well-known, popular statistical methods. Therefore we believe that the re-production of the case study can be easily performed by any interested researcher.

Concerning the ***External validity*** and in particular the generalizability supposition, changes in the findings might occur if the applications for which the sample releases are analyzed are altered. The results certainly can be applied to projects implemented with JS programming language but their transferability to other non scripting languages is limited. Also since our data set is based on open source projects we acknowledge the fact that our results might need customization when applied to closed source software. Still since the majority of closed source JS applications is based on open source JS libraries, we believe that our results show a tendency when it comes to estimating the maintenance effort of JS applications. Still a future replication of this study, on maintenance data from other projects, even closed source, would be valuable to verify these findings.

VII. Conclusions

Estimating the maintenance effort of software applications is a challenging task as it depends on a variety of factors and aspects. Tin this study we performed a case study on 60 JavaScript applications and analyzed 5.788 releases in order to highlight the factors of significant importance on estimating maintenance changes and effort. We developed and evaluated two maintenance indices, namely Maintenance Changes index and Maintenance Effort index. The evaluation process showed that both indices present very satisfying results with respect to the validation criteria of IEEE standard.

Acknowledgment

This research was co-funded by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship, and Innovation, grant number T1EDK-04759

References

[1] Ahn, Y., Suh, J., Kim S., Kim, H., The software maintenance project effort estimation model based on function points. Journal of Software Maintenance 15, 2 (March 2003), 2003, 71-85

[2] Alomari, H. W., Collard, M. L., Maletic, J. I. 2014. A Slice-Based Estimation Approach for Maintenance Effort, IEEE Intern. Conference on Software Maintenance & Evolution, Victoria BC, pp. 81-90.

[3] Ampatzoglou A., Bibi S., Chatzigeorgiou A., Avgeriou P., Stamelos I. 2018. Reusability Index: A Measure for Assessing Software Assets Reusability. In: Capilla R., Gallina B., Cetina C. (eds) New Opportunities for Software Reuse. ICSR 2018. Lecture Notes in Computer Science, vol 10826. Springer, Cham

[4] Anda, B. C., Yamashita A., Sjoberg, D. I., Mockus A., Dyba, T., "Quantifying the Effect of Code Smells on Maintenance Effort," in IEEE Transactions on Software Engineering, vol. 39, no., pp. 1144-1156, 2013.

[5] Baggen, R., Correia, J. P., Schill, K., & Visser, J. 2011. Standardized code quality benchmarking for improving software maintainability. Software Quality Journal, 20(2), 287–307.

[6] Bennett, K.H. 1991. Automated support of software maintenance, Information and Software Technology, Volume 33, Issue 1, Pages 74-85

[7] Chandra, D., Choudhary M., Gupta D., 2017. Prophecy of software maintenance effort with univariate and multivariate approach, 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, pp. 876-880.

[8] Chua B.B., Verner J., 2011. Evaluating Software Maintenance Effort: The COME Matrix. In: Kim T. et al. (eds) Software Engineering, Business Continuity, and Education. ASEA 2011. Communications in Computer and Information Science, vol 257. Springer, Berlin, Heidelberg

[9] Coleman, D., Ash, D., Lowther, B., Oman, P. 1994. Using Metrics to Evaluate Software System Maintainability.IEEE Computer, pp. 44–49

[10] Glass, R.L., 2001. Frequently forgotten fundamental facts about software engineering. IEEE Software 18, 112–111.

[11] Harn, M., Berzins, V., Luqi, Mori, A., 1999. Software evolution process via a relational hypergraph model, in: Proceedings 199 IEEE/IEEJ/JSAI International Conference on Intelligent Transportation Systems (Cat. No.99TH8383). pp. 599–604.

[12] Hayes, J. H., Patel S. C., Zhao, L. 2004. "A metrics-based software maintenance effort model," Eighth European Conference on Software Maintenance and Reengineering, CSMR 2004. pp. 254-258.

[13] 1061-1998: IEEE Standard for a Software Quality Metrics Methodology, IEEE Standards, IEEE Computer Society, 31 December 1998 (reaffirmed 9 December 2009).

[14] Jørgensen, M. 1995. An empirical study of software maintenance tasks, J. Softw. Maint: Res. Pract., 7: 27-48

[15] Kyriakakis P., Chatzigeorgiou, A. 2014. "Maintenance Patterns of Large-Scale PHP Web Applications," IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 381-390.

[16] Lehman, M.M., Ramil, J.F., 2002. Software Evolution and Software Evolution Processes. Ann. Softw. Eng. 14, 275–309.

[17] Lehner, F., 1990. Cost comparison for the development and maintenance of application systems in 3rd and 4th generation languages. Information & Management 18, 131–141.

[18] Mehdi Hejazi Dehaghani, S., Hajrahimi, N., 2013. Which Factors Affect Software Projects Maintenance Cost More? Acta informatica medica : AIM : journal of the Society for Medical Informatics of Bosnia & Herzegovina : časopis Društva za medicinsku informatiku BiH 21, 63–6.

[19] Meqdadi, O., 2013, Understanding and Identifying Large-Scale Adaptive Changes from Version Histories, Kent State University, Computer Science Department, Phd Thesis, Adv.: Dr. Jonathan Maletic

[20] Milicic, D., Wohlin, C., 2004. Distribution Patterns of Effort Estimations. In: Proceedings of EUROMICRO conference, pp. 422–429

[21] Mockus, A., Votta, L. 2000. Identifying Reasons for Software Changes Using Historic Databases, In Proceedings of the International Conference on Software Maintenance (ICSM'00), IEEE Computer Society, Washington, DC, USA, pp. 120

[22] Mondal, M., Roy, C., Schneider, K., 2017. Does cloned code increase maintenance effort?, IEEE 11th International Workshop on Software Clones (IWSC), Klagenfurt, pp. 1-7.

[23] Niessink, F., van Vliet, H. 1998. "Two case studies in measuring software maintenance effort," Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), Bethesda, MD. pp. 76-85.

[24] Oman, P. W., Hagemeister, J. 1992. Metrics for assessing a software system's maintainability. In Proceedings of Conference on Software Maintenance, IEEE Computer Society, Los Alamitos, CA, pp. 337–344

[25] Ray, B., Posnett, D., Devanbu, P., & Filkov, V. 2017. A large-scale study of programming languages and code quality in GitHub. Communications of the ACM, 60(10), 91–100.

[26] Rostami S., Eshkevari L., Mazinanian D., Tsantalis N., "Detecting Function Constructors in JavaScript," 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME'2016), ERA Track, Raleigh, North Carolina, USA, October 2-10, 2016.

[27] Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir Software Eng 14, 131.

[28] Silva, L.H., Hovadick, D., Valente, M.T., Bergel, A., Anquetil, N., & Etien, A. (2016). JSClassFinder: A Tool to Detect Class-like Structures in JavaScript. CoRR, abs/1602.05891.

[29] Sjøberg, D., Anda, B., Mockus, A. 2012. Questioning software maintenance metrics: A comparative case study. International Symposium on Empirical Soft. Engineering and Measurement. 107-110.

[30] Sjøberg, D., Yamashita, A., Anda, B., Mockus A., Quantifying the Effect of Code Smells on Maintenance Effort, IEEE Transactions on Soft. Engineer., vol. 39, no. 8, pp. 1144-1156, Aug. 2013.

[31] Thamburaj, T. F., Aloysius, A. 2017. Models for Maintenance Effort Prediction with Object-Oriented Cognitive Complexity Metrics, 2017 WCCCT, Tiruchirappalli, pp. 191-194.

[32] Yang, Y., Li, Q., Li, M.S., Wang, Q. 2008. An Empirical Analysis on Distribution Patterns of Software Maintenance Effort. In: Proceedings of International Conference on Software Maintenance (ICSM), pp. 456–459.