Analyzing the Evolution of Javascript Applications

Angelos Chatzimparmpas[1], Stamatia Bibi[2], Ioannis Zozas[2] and Andreas Kerren[1]
[1]*Department of Computer Science and Media Technology, Linnaeus University, Växjö, Sweden*
[2]*Department of Informatics & Telecommunications Engineering, University of Western Macedonia, Kozani, Greece*

Keywords: Software Evolution, Lehman's Laws, JavaScript, Maintenance, Software Quality.

Abstract: Software evolution analysis can shed light on various aspects of software development and maintenance. Up to date, there is little empirical evidence on the evolution of JavaScript (JS) applications in terms of maintainability and changeability, even though JavaScript is among the most popular scripting languages for front-end web applications, including IoT applications. In this study, we investigate JS applications' quality and changeability trends over time by examining the relevant Laws of Lehman. We analyzed over 7,500 releases of JS applications and reached some interesting conclusions. The results show that JS applications continuously change and grow, there are no clear signs of quality degradation while the complexity remains the same over time, despite the fact that the understandability of the code deteriorates.

1 INTRODUCTION

In the past decade, the developers' interest in dynamic languages, such as JavaScript, Python, and Ruby has resurged, a fact confirmed by the growth and penetration of the languages (Amanatidis and Chatzigeorgiou, 2016). Undoubtedly, JavaScript (JS) is today among the most popular programming languages as according to GitHub statistics[1], it was the most active language in 2017.

JS is a high-level dynamic, object-based, multi-paradigm, interpreted, and weakly typed programming language (Diakopoulos and Cass, 2017). The majority of websites are written in JS, and all current web browsers have a built-in JS engine to support without needing plug-ins (Diakopoulos and Cass, 2017). Additionally JS has already started to support the emerging needs of new types of applications for controlling elements in the physical world within the context of IoT. Despite this, there is little knowledge today regarding the maintenance and evolution of JS applications in terms of quality and changeability.

The objective of this study is to explore the evolution of JS applications over time, in terms of quality and changeability. Software evolution refers to maintaining both software performance and usefulness across time and occurs through software development and maintenance processes (Belady and Lehman, 1976). The motivation behind the need to analyze the evolution in terms of quality and changeability of applications developed in JS is the fact that this language is considered to be weakly typed (Diakopoulos and Kass, 2017). This means that it has looser type rules, which may generate unpredictable results during an application's life-cycle. In this context, we want to explore (a) whether this fact may cause problems to the ***quality*** of projects through time and (b) the level to which ***changes*** are performed during the evolution of JS applications helping towards their maintenance.

In order to assess the evolution of JS applications, we performed an empirical study on twenty (20) popular Open Source Software (OSS) projects downloaded from GitHub repository and examined the five laws of evolution as introduced by Lehman (1996) regarding the *quality* and the level of *changes* performed in an application. We considered Lehman's Laws for assessing the evolution of JS projects, as being representative for traditional studies of software evolution (Belady and Lehman, 1976). We followed the analysis steps firstly demonstrated in earlier similar studies, experimenting with different languages (Amanatidis and Chatzigeorgiou, 2016), (Godfrey and Tu, 2000). Thus, we also enable the comparison with other

[1] https://madnight.GitHub.io/githut/#/pull_requests/2017/ [Accessed: 15 February 2019]

Chatzimparmpas, A., Bibi, S., Zozas, I. and Kerren, A.
Analyzing the Evolution of Javascript Applications.
DOI: 10.5220/0007727603590366
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 359-366
ISBN: 978-989-758-375-9

popular programming languages such as C, C++, Java, and PHP.

The rest of the paper is organized as follows: Section 2 provides related work overview. In Section 3, we describe the case study design. Section 4 presents the results on twenty JS project analysis. In Sections 5 and 6, we discuss the results and conclude the paper.

2 RELATED WORK

Over forty years ago, Lehman's laws have been considered as a reference to software evolution. Belady and Lehman (1976) were the first that on an empirical basis studied the changes in both complexity and size of several large programs. They conducted a quantitative study over release versions that summarized their three initial qualitative laws of program evolution dynamics. This initiated a large number of research efforts, which were introduced by adopting a variety of metrics to test the validity of each law.

Lawrence (1982) tested statistically the first five of Lehman's laws before law revisions twenty years later that included the role of process feedback (Lehman, 1996). Gall et al. (1997) were the first to provide a confirmation based on software release data. Other researchers proposed different metrics and frameworks over software evolution. Kemerer and Slaughter (1999) conducted a longitudinal empirical business case study, while Godfrey and Tu (2000) analyzed data from the growth of the Linux kernel finding linear growth in evolution. Paulson et al. (2004) analyzed empirical data for various systems to confirm previous software evolution assumptions.

To this time, a growing interest in whether the laws apply to OSS emerged (Oliveira and Almeida, 2016). Wu and Holt (2004) measured the size evolution of two large systems confirming the first four laws. German (2004) analyzed software trails over a single system, demonstrating a methodology to recover software evolution information. Neamtiu, Foster, and Hicks (2005) mined software repositories of popular C systems focusing on code function names and revealed trends, present semantic differences, and software evolution traces. Later Neamtiu, Xie, and Chen (2013) analyzed official software releases to confirm Lehman's first five laws and indicated violations. Gyimothy, Ferenc, and Siket (2005) already proposed and applied metrics on one project to detect fault-proneness as a software evolution derivation and Kim, Whitehead, and Bevan (2006) investigated signature changes in seven projects to detect evolution patterns using statistical correlations.

The growing interest over OSS projects continued to emerge through a growing number of research efforts. Herraiz et al. (2007) studied the evolution in size of a project over time by applying time series analysis. Fernandez-Ramil (2008) studied the growth trend on popular libre operating systems to contradict three of Lehman's laws, confirming the latter five. Antoniol et al. (2007) focused more on the role of the identifier lexicon on overall software evolution, while Businge et al. (2010) investigate 5 out of 8 laws confirming the results of previous research. Grechanik et al. (2010) investigated Java applications and expanded the law validity by practice-at-large on Java development. Kaur et al. (2014) researched the law applicability on two prominent OSS C++ applications. Amanatidis and Chatzigeorgiou (2016) analyzed data acquired from successive versions of PHP projects to evaluate the validity of each law by applying primarily trend tests.

In this paper, our goal is to extend current research efforts to evaluate the validity of Lehman's laws concerning JS applications.

3 CASE STUDY

The case study performed was designed following Runeson and Höst's (2008) guidelines. We examined JS applications evolution from the perspective of Lehman's laws, which characterize trends in quality and changes of the evolving software systems.

Concerning the *Research questions* of this study, the main goal is to explore the trends in changes and quality of JS applications, over time, from the perspective of Lehman's laws of evolution. Since our main focus is on the changes performed between successive versions and their impact on quality, the remaining three laws relevant to size remained out of our scope. Therefore, we examined the following research questions:

RQ1: *Is Law I: "**Continuous Change**" confirmed by JavaScript applications, as an indicator of a trend in **changes?***

In this research question, we aim to see whether JS applications actually support continuous change and if this change is more intense or loose over the successive releases.

RQ2: *Is Law II: "**Increasing complexity**" confirmed by JavaScript applications, as an indicator of a trend in **quality?***

In this question, we will explore whether the complexity of JS applications is constantly increasing or whether the maintenance actions performed are sufficient to keep complexity levels stable.

RQ3: *Is Law IV regarding "**Conservation of Organizational Stability**" confirmed by JavaScript applications, as an indicator of a trend in **changes?***

In this question, we will explore whether the work produced between successive releases of JS applications is constant.

RQ4: *Is Law V regarding "**Conservation of Familiarity**" confirmed by JavaScript applications, as an indicator of a trend in **changes**?*

In this question, we will explore whether the new content added between successive releases of JS is stable or present large deviations.

RQ5: *Is Law VII regarding "**Declining Quality**" confirmed by JavaScript applications, as an indicator of a trend in **quality?***

In this research question, we aim to check if the quality of a software system deteriorates during the lifecycle of a software system.

Table 1: JS projects examined in the study.

Name	Releases	First Release		Last Release	
		Date	Size	Date	Size
Lodash	380	2012	78,559	2017	3,541
Material-UI	161	2014	2,649	2017	3,627
Dropzone	97	2012	947	2017	3,751
Bower	102	2012	5,663	2017	4,559
WebTorrent	257	2013	592	2017	4,639
Yarn	110	2016	1,455	2017	5,372
Q	65	2010	568	2014	6,768
Cropper	52	2013	7,076	2017	19,477
Video.js	327	2014	5,791	2017	19,552
Jasmine	58	2009	9,940	2017	20,045
Medium-Editor	150	2013	2,116	2017	20,709
Hexo	120	2012	55,186	2017	24,826
Webpack	253	2013	15,300	2017	42,404
Chart.js	37	2013	4,787	2017	44,925
JSHint	66	2011	199,130	2017	66,579
PDF.js	44	2011	40,996	2017	76,238
Vue	207	2013	12,840	2017	89,095
Hyper	42	2016	5,222	2017	92,988
OpenLayers	161	2006	7,354	2017	102,097
ESLint	171	2013	15,264	2017	234,324

Concerning the *Case Study Design* to explore the evolution of JS applications, several criteria were employed to select the JS applications, which were included in the analysis. Initially, all JS applications hosted in GitHub were ranked according to their popularity. Then, we filtered the applications and selected the ones ***with at least 95% of JS*** code, ***two or more years of lifespan***, and ***more than thirty releases***. Afterward, we chose s***mall, medium, and large-sized*** applications depending on the lines of code (LoC) for their last release.

Table 1 presents the application complete lifespan until August 2017, their number of releases, the scope of each, as well as certain metrics referring to the first and last release of each. Moreover, Table 2 presents the metrics adopted to assess the validity of each law.

Table 2: Metrics used for each law.

Law	Property	Metric	Description
I	Changes	Days Between Releases	Recent release date – Previous release date.
		Lines of Code	Lines of Code of the release (excluding comments).
II	Quality	Cyclomatic Complexity	Cyclomatic Complexity Number (total code paths or splits in flow)/Lines of Code.
		Cognitive Complexity	A measure of the relative understandability of methods, calculated by SonarQube[2].
IV	Changes	Maintenance Effort	Incremental Changes/Days Between Releases.
		# of Commits	Project commits/releases.
V	Changes	Incremental Changes	Number of functions added/ /modified/removed.
		Number of new Functions	Number of new functions/releases.
VII	Quality	Comment Rate	Comments/(Lines of Code + Comments) %.
		Maintainability	The ratio between the cost to develop and cost to fix potential bugs found in a release. It is calculated by SonarQube based on the Lehman's (1996) technical debt concept.

The final set of metrics was collected with the following process (a) by initially mining the webpage of GitHub projects to get general project information using a *parser tool* developed by the first author and (b) downloading all successive releases from git for each project in order to derive source code metrics with the help of *JSClassFinder* or *SonarQube*.

Concerning the *Data Analysis*, we employed statistical hypothesis testing to check whether the five laws of Lehman are confirmed or not. We applied the *Mann-Kendall (M-K test)* trend test, which is a nonparametric test used to identify a trend

[2] https://blog.sonarsource.com/cognitive-complexity-because-testability-understandability [Accessed: 15 February 2019]

in a time series. The Mann-Kendall test explores the following hypotheses in the context of this study:

H_0: The null hypothesis H_0 is that there is no trend supported by the software data analyzed, so the relevant law cannot either be confirmed or contradicted.

H_1: The non-null hypothesis H_1 refers to the alternative hypotheses that there is a negative, non-null, or positive trend regarding the relevant Law.

The "p-value" is automatically generated to distinguish the two hypotheses. A value less than 0.05 indicates that there is a trend exhibiting the dependent variable and vice versa for a value greater than 0.05. The threshold of 0.05 is common practice (Garg et al., 1998) when deciding upon a hypothesis (Sen, 1968).

In the case where the null hypothesis is rejected, we calculated the *Sen's estimator* (Sen, 1968) value to assess the slope of the fitted trendline. Based upon ***Sen's slope*** estimator, we can draw a conclusion related to the trend that a variable exhibits and *statistically confirm* or not the relevant law. In the case where the null hypothesis is not rejected for the majority of projects, we plotted the relevant metrics in subsequent releases of projects that do not present a trend, so as to allow the *visual inspection* of their evolution through time.

4 RESULTS

In this section, we present the results of the trend analysis performed to confirm or contradict the Lehman's law hypothesis on software evolution. The results are presented in Tables 3 to 7. For each examined metric, we can see the results of Mann Kendall trend test in the form of the p-value and also the slope value (in the case of a trend), which is accompanied by a trend arrow sign that indicates either a positive trend (meaning increase over time) or a negative trend (indicates decrease over time).

4.1 Law I: Continuing Change

To obtain insights on the 1st law of Lehman, we have statistically tested two metrics the **Lines of Code (LoC**) and the **Days Between Releases (DBR)**. *LoC* is an indicator of the changes performed between successive releases. In Table 3, we observe that LoC presents a positive trend in almost every application, which leads to the *confirmation of the law*. Also, the positive trend implies that the changes performed in successive releases are increased over time may be due to the need to add new functionalities. The second metric we tested is DBR. A positive trend in DBR implies that the number of days elapsed between successive releases tends to increase as time passes by, leading to the conclusion that new software releases are published more rarely. As we can observe in Table 3, in most of the applications there is no statistical evidence for the presence or the absence of a distinct trend. To visually investigate the evolution of DBR, we plotted a chart for the projects with the p-value greater than 0.05 (Figure 1) that presents the DBR metric in successive releases. The chart has lots of fluctuations, a fact that strengthens the assumptions that almost all JS projects change over time, presenting though, an unknown rate of change. In conclusion, **Law I is confirmed statistically and visually confirmed**, by taking into consideration the LoC metric results and the DBR metric plots. We can say that JS applications continuously change, but the rate of change is unknown.

Table 3: RQ1 - trend analysis results.

Law I	**Continuing Change**			
Program	**LoC**		**DBR**	
	p	**Slope**	**p**	**Slope**
Lodash	0.116		$<10^{-4}$	-
Material-UI	$< 10^{-4}$	28.27↑	$<10^{-4}$	-0.03↓
Dropzone	$< 10^{-4}$	7.397↑	0.001	0.05↑
Bower	$< 10^{-4}$	7.824↑	0.051	
WebTorrent	$< 10^{-4}$	2.169↑	0.105	
Yarn	$< 10^{-4}$	2.268↑	0.141	
Q	$< 10^{-4}$	31.72↑	0.001	0.7↑
Cropper	$< 10^{-4}$	6.417↑	0.074	
Video.js	$< 10^{-4}$	22.97↑	0.377	
Jasmine	$< 10^{-4}$	31.72↑	0.375	
Medium-Editor	$< 10^{-4}$	20.63↑	0.088	
Hexo	$< 10^{-4}$	12.88↑	$<10^{-4}$	0.09↑
Webpack	$< 10^{-4}$	13.33↑	0.001	0.01↑
Chart.js	$< 10^{-4}$	38.63↑	0.990	
JSHint	$< 10^{-4}$	6.000↑	0.022	0.4↑
PDF.js	$< 10^{-4}$	12.09↑	0.328	
Vue	$< 10^{-4}$	31.72↑	0.848	
Hyper	$< 10^{-4}$	104.9↑	0.297	
OpenLayers	$< 10^{-4}$	21.49↑	0.246	
ESLint	$< 10^{-4}$	122.4↑	0.665	

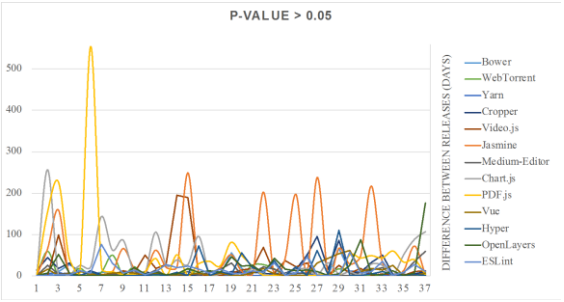

Figure 1: DBR for successive releases of JS applications with no trend.

4.2 Law II: Increasing Complexity

To check the 2nd law, we statistically test two metrics the **Cyclomatic Complexity** metric and the **Cognitive Complexity** metric. The *Cyclomatic Complexity* is an indicator of the source code complexity and is mainly used for measuring the testability of a software method. As we can observe in Table 4, no statistical trend is found in seventeen applications. The absence of a trend is a strong indicator that *Cyclomatic Complexity* remains at the same levels as time passes by, a fact that weakens the validity of Law II. The *Cognitive Complexity* is an indicator of the effort required to understand a method and is a measure of the understandability of a software method. As we can observe in Table 5, in the majority of applications there is a positive trend. That means Cognitive Complexity *increases* as time passes by, a fact that strengthens the validity of Law II.

Table 4: RQ2 - trend analysis results.

Law II	**Increasing Complexity**			
Program	**Cyclomatic Complexity**		**Cognitive Complexity**	
	p	**Slope**	**p**	**Slope**
Lodash	$< 10^{-4}$		0.941	
Material-UI	0.0		0.107	
Dropzone	$< 10^{-4}$		$<10^{-4}$	3.957↑
Bower	$< 10^{-4}$	0.01↑	$<10^{-4}$	0.681↑
Web Torrent	0.621		0.426	
Yarn	0.034	-0.01↓	$<10^{-4}$	0.268↑
Q	$< 10^{-4}$		$<10^{-4}$	1.233↑
Cropper	0.008		$<10^{-4}$	2.724↑
Video.js	0.724		$<10^{-4}$	6.712↑
Jasmine	$< 10^{-4}$		$<10^{-4}$	0.270↑
Medium-Editor	$< 10^{-4}$		$<10^{-4}$	3.548↑
Hexo	$< 10^{-4}$		$< 10^{-4}$	5.109↑
Webpack	$< 10^{-4}$		$< 10^{-4}$	2.603↑
Chart.js	0.002		$< 10^{-4}$	7.096↑
JSHint	0.048		$< 10^{-4}$	3.292↑
PDF.js	$< 10^{-4}$	0.01↑	$< 10^{-4}$	4.123↑
Vue	$< 10^{-4}$		$< 10^{-4}$	11.62↑
Hyper	0.174		$< 10^{-4}$	7.727↑
OpenLayers	$< 10^{-4}$		$< 10^{-4}$	2.488↑
ESLint	0.021		$< 10^{-4}$	6.500↑

In conclusion, we can say that the Complexity metric of JS applications remains the same in terms of testability, but it increases in terms of understandability. Maybe this could be explained due to the maintenance effort from the developers to keep low the complexity level of the JS software from the perspective of code control flow, a fact that on the other hand reduces the understandability of the code. **Therefore, Law II is statistically confirmed with respect to the understandability of the application but not confirmed in terms of testability.**

4.3 Law IV: Conservation of Organizational Stability

To check the 4th law, the **Maintenance Effort** and the **Number of Commits (NoC)** metrics are statistically tested. In Table 5 for the *Maintenance Effort*, we can observe that in seventeen applications we have not any slope, so there is no evidence of a statistical trend. Figure 2 presents the Maintenance Effort metric evolution in subsequent releases for the applications with p-values greater than 0.05.

Table 5: RQ3 - trend analysis results.

Law IV	**Conservation of Organizational Stability**			
Program	**Maintenance Effort**		**Number of Commits**	
	p	**Slope**	**p**	**Slope**
Lodash	0.669		$<10^{-4}$	-0.03↓
Material-UI	0.401		0.790	
Dropzone	0.775		$<10^{-4}$	-8.86↓
Bower	0.173		$<10^{-4}$	-21.91↓
WebTorrent	0.076		$<10^{-4}$	-0.2↓
Yarn	0.925		$<10^{-4}$	-15.14↓
Q	0.020	-0.07↓	$<10^{-4}$	-12.7↓
Cropper	0.825		$<10^{-4}$	-18.9↓
Video.js	1.000		0.015	-4.2↓
Jasmine	0.035	0.14↑	$<10^{-4}$	-4.6↓
Medium-Editor	0.594		$<10^{-4}$	-0.54↓
Hexo	0.379		0.935	
Webpack	0.018	-0.05↓	$<10^{-4}$	-8.03↓
Chart.js	0.958		0.564	
JSHint	0.108		$< 10^{-4}$	-3.5↓
PDF.js	0.476		$<10^{-4}$	-94.61
Vue	0.163		$<10^{-4}$	-
Hyper	0.634		$<10^{-4}$	-5.25↓
OpenLayers	0.268		$<10^{-4}$	-69.36↓
ESLint	0.384		$<10^{-4}$	-38.0↓

In Figure 2, we can observe that the Maintenance Effort is stable apart from a few exceptions that present extreme effort values during their evolution that deviate from the usual values. Regarding the *NoC* variable, we observe that the majority of applications present a negative slope, indicating that the number of commits is decreased as developers publish new releases. This law combines the activity and the work rate in a fraction (activity/work rate).

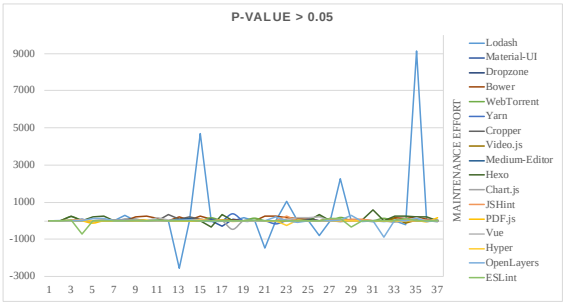

Figure 2: Maintenance effort for successive releases of JS applications with no trend.

As a proxy for an *activity*, we consider the Number of Commits metric and as a proxy for *work rate*, we consider the maintenance effort metric. Maintenance effort metric shows the amount of effort contributed to a particular release. In JS applications, we observe that in the majority of cases the activity declines while the work rate remains stable. This causes the decrease of the entire fraction, a fact that is in contrast to what the law proposes. So, the maintenance effort remains the same in general despite the fact that the commits are reduced over time. **In conclusion, Law IV can be confirmed with respect to the work rate but not concerning the global activity.**

4.4 Law V: Conservation of Familiarity

To check the 5th law, the **Number of Functions (NoF)** and the **Incremental Changes (IC)** metrics are statistically tested. *NoF* represents the number of new functions added in a release and is an indicator of the cumulative changes performed between successive releases. In Table 6, we observe that NoF presents a positive trend in almost every application, which leads to the conclusion that during the evolution of a JS application the rate in which new functions are inserted tend to increase. Taking into consideration this fact, we cannot confirm the law. Regarding IC metric we see in Table 6, that for eighteen applications we have not any slope, so no clue that proves the existence of a statistical trend. To visually examine the evolution of IC metric we plotted the projects with the p-value greater than 0.05 for further investigation in Figure 3.

Table 6: RQ4 - trend analysis results.

Law V	**Conservation of Familiarity**			
Program	**NoF**		**Incremental Changes**	
	p	**Slope**	**p**	**Slope**
Lodash	$< 10^{-4}$	-2.01↓	0.114	
Material-UI	$< 10^{-4}$	17.96↑	0.041	0.16↑
Dropzone	$< 10^{-4}$	7.98↑	0.677	
Bower	$< 10^{-4}$	15.96↑	0.774	
WebTorrent	$< 10^{-4}$	2.65↑	0.067	
Yarn	$< 10^{-4}$	0.95↑	0.867	
Q	$< 10^{-4}$	24.21↑	0.042	-0.94↓
Cropper	$< 10^{-4}$	7.28↑	0.705	
Video.js	$< 10^{-4}$	12.42↑	0.390	
Jasmine	$< 10^{-4}$	30.0↑	0.872	
Medium-Editor	$< 10^{-4}$	12.46↑	0.978	
Hexo	$< 10^{-4}$	30.33↑	0.184	
Webpack	$< 10^{-4}$	8.2↑	0.185	
Chart.js	$< 10^{-4}$	74.44↑	0.340	
Jhint	$< 10^{-4}$	11.58↑	0.438	
PDF.js	$< 10^{-4}$	55.45↑	0.085	
Vue	$< 10^{-4}$	15.83↑	0.577	
Hyper	$< 10^{-4}$	6.43↑	0.392	
OpenLayers	$< 10^{-4}$	61.96↑	0.287	
ESLint	$< 10^{-4}$	34.77↑	0.729	

The fluctuations of the plot indicate that some projects have a positive trend which implies that the number of functions added/modified/removed increases and for the others, this is not the case. It seems that there are breaking points in the lifecycle of JS applications were a great amount of functionality is added, or the existing code base is refactored as we can observe from various releases that present large deviations in terms of changes. In conclusion, **Law V is not confirmed**.

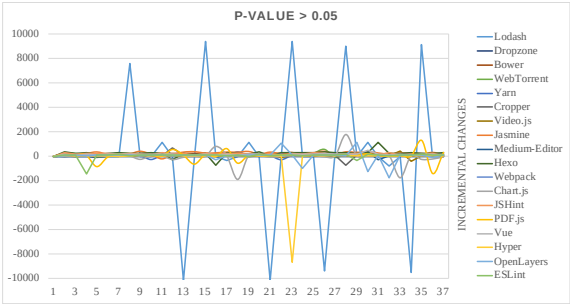

Figure 3: Incremental changes for successive releases of JS projects with no trend.

4.5 Law VII: Declining Quality

To check the 7th law, the **Comment Rate (CR)** and **Maintainability** metrics are statistically tested. For the *CR* variable, as we observe in Table 7, approximately half of the projects have a negative trend. The CR metric seems to gradually decrease in new releases but still based on the slope values the level of decrease is too small.

Table 7: RQ5 - trend analysis results.

Law VII	**Declining Quality**			
Program	**Comment Rate**		**Maintainability**	
	p	**Slope**	**p**	**Slope**
Lodash	0.003	-0.04↓	0.0	-0.03↓
Material-UI	$<10^{-4}$	0.01↑	$<10^{-4}$	-0.01↓
Dropzone	$<10^{-4}$	-0.12↓	0.623	
Bower	0.024	0.02↑	$<10^{-4}$	-0.01↓
WebTorrent	$<10^{-4}$	0.09↑	$<10^{-4}$	0.02↑
Yarn	$<10^{-4}$	-0.02↓	$<10^{-4}$	0.01↑
Q	0.385		0.001	-0.11↓
Cropper	0.121		0.071	
Video.js	0.186		0.724	
Jasmine	0.003	0.05↑	0.571	
MediumEditor	$<10^{-4}$	-0.02↓	0.0	
Hexo	$<10^{-4}$	0.02↑	$<10^{-4}$	-0.01↓
Webpack	$<10^{-4}$	-0.04↓	0.118	
Chart.js	0.012	-0.08↓	$<10^{-4}$	-0.01↓
Jhint	0.531		0.249	
PDF.js	0.513		0.368	
Vue	$<10^{-4}$	-0.05↓	$<10^{-4}$	0.01↑
Hyper	0.001	-0.07↓	0.252	
OpenLayers	$<10^{-4}$	-0.06↓	0.583	
ESLint	0.602		0.196	

For the *Maintainability* variable in eleven samples, we have not any slope which means that there is no indication of a statistical trend. By examining each project separately, we can identify a small decrease in Maintainability and CR metrics, but it is important to note that the actual slopes are pretty low. In other words, the quality remains stable in most of the cases. **In conclusion, the results cannot support the confirmation or not of Law VII.**

5 DISCUSSION

5.1 Implications to Researchers and Practitioners

The results of this study can be used both by researchers and practitioners to invest their efforts in the following areas:
Since the JS applications participating in this study do not increase their cyclomatic complexity over time or present signs of quality degradation, it will be interesting for **researchers** *to study which coding conventions (e.g., writing small methods, using design patterns or using micro-templates) help to reserve or decrease complexity over time*. A comparison between applications that show signs of increased complexity with more stable ones may lead to some conclusions. For example, bad smells or anti-patterns might be found to the later ones.

Additionally, **researchers** can also work on *cost and quality models for estimating the effort required to maintain JS* applications and *assessing* the *quality level of* subsequent releases. In that context, certain language-specific quality metrics (e.g., null pointer dereferences, deprecated functions) along with usage metrics (e.g., number of users of the application, types of browsers, types of devices) can help towards quantifying maintenance activities and effectively managing subsequent releases.

The results show to **practitioners** that even though JS applications continuously change their complexity remains constant. JS applications present several points during their lifecycle at which a severe amount of new functionality is introduced. This fact demonstrates that *software managers should often take large-scale maintenance actions*. At the moment in most cases, we observed the big-bang approach, where releases are offering many new requirements at once, an approach that can be risky. Instead of this, the appropriate flexible software development model should be selected to allow the introduction of new, small scale functionalities in short development lifecycles launched as many small releases. Also, continuous end-user involvement could help in that direction.

5.2 Threats to Validity

In this section, we discuss the threats to validity for this study, based on the categorization of Runeson and Höst (2008). Regarding *Construct Validity*, we should mention that the set of metrics used to assess the evolution of JS applications may affect the findings. Our rationale behind selecting these metrics was based on content and scope similarities with other studies adopting them (Amanatidis and Chatzigeorgiou, 2016) without denying the evaluation of non-selected alternative metrics as future work. Regarding *Internal Validity* we do not claim that the produced results form a causality between the metrics and the various evolution aspects, but we argue that our results indicate current trends. Concerning *reliability*, we believe that the replication of our research is safe and the overall reliability is ensured. The process that has been followed in this study has been thoroughly documented in the relevant section, so as to be easily reproduced by any interested researcher. The structural metrics calculation and the overall extraction of the defined data set were performed with the use of a widely used research tool (SonarQube). Concerning *the external validity* and in particular the generalizability supposition, changes in the findings might occur if we altered samples of OSS projects or closed source JS projects were studied. A future replication of this study, on larger JS project data sets and closed source projects, would be valuable to verify these findings.

6 CONCLUSIONS

In this study, we have explored the evolution of twenty popular OSS JS applications in terms of changes and quality by examining whether the relevant laws of Lehman can be confirmed. In total, we have recorded evolution metrics of more than 7,500 releases of JS projects and performed trend tests to verify the applicability of the laws. The results show that JS applications continuously change and grow, there are no clear signs of quality degradation, while the complexity remains the same over time, despite the fact that the understandability of the code deteriorates.

ACKNOWLEDGEMENTS

This research was co-funded by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship, and Innovation, grant number T1EDK-04873, project "Drone innovation in Saffron Agriculture," DIAS.

REFERENCES

Amanatidis, T., Chatzigeorgiou, A., 2016. Studying the evolution of PHP web applications. *In Information and Software Technology*, *72*, 48-67.

Antoniol, G., Gueheneuc, Y. G., Merlo, E., Tonella, P., 2007. Mining the lexicon used by programmers during software evolution. *In IEEE International Conference on Software Maintenance, Paris*, 14-23. IEEE.

Belady, L. A., Lehman, M. M., 1976. A model of large program development. *IBM Systems journal*, *15*(3), 225-252.

Businge, J., Serebrenik, A., van den Brand, M., 2010. An empirical study of the evolution of Eclipse third-party plug-ins. *In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 63-72. ACM.

Diakopoulos N., Cass S., 2017. IEEE Spectrum Interactive: the top programming languages 2017. https://bit.ly/2wWgUaB [Accessed: 15 February 2019]

Fernandez-Ramil, J., Lozano, A., Wermelinger, M., and Capiluppi, A., 2008. Empirical studies of open source evolution. In *Software evolution*, Berlin, 263-288. SPRINGER.

Gall, H., Jazayeri, M., Klosch, R. R., Trausmuth, G., 1997, October. Software evolution observations based on product release history. *In Proceedings International Conference on Software Maintenance*, 160-166. IEEE.

Garg, S., Van Moorsel, A., Vaidyanathan, K., Trivedi, K. S., 1998. A methodology for detection and estimation of software aging. *In Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 283-292. IEEE.

German, D. M., 2004. Using software trails to reconstruct the evolution of software. *In Journal of Software Maintenance and Evolution: Research and Practice*, *16*(6), 367-384.

Godfrey M. W., Tu Q., 2000. Evolution in open source software: a case study. *In Proceedings of the International Conference on Software Maintenance*, San Jose, 131-142.

Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Qing, Ghezzi, C., 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 International Symposium on Empirical Software Engineering and Measurement*, 11. ACM.

Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897-910.

Herraiz, I., Gonzalez-Barahona, J. M., Robles, G., German, D. M., 2007. On the prediction of the evolution of libre software projects. In *IEEE International Conference on Software Maintenance*, 405-414. IEEE.

Kaur, T., Ratti, N., Kaur, P., 2014. Applicability of Lehman laws on open source evolution: a case study. *In International Journal of Computer Applications*, *93*(18), 0975-8887.

Kemerer, C. F., Slaughter, S., 1999. An empirical approach to studying software evolution. *In IEEE Transactions on Software Engineering*, *25*(4), 493-509.

Kim, S., Whitehead, E. J., 2006. Properties of signature change patterns. *In IEEE International Conference on Software Maintenance*, 4-13. IEEE.

Lawrence, M. J., 1982. An examination of evolution dynamics. *In Proceedings of the 6th International Conference on Software Engineering*, 188-196. IEEE Computer Society Press.

Lehman, M. M., 1996. Laws of software evolution revisited. *In European Workshop on Software Process Technology* (pp. 108-124), Berlin. SPRINGER.

Neamtiu, I., Foster, J. S., Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. *In ACM SIGSOFT Software Engineering Notes*, *30*(4), 1-5. ACM.

Neamtiu, I., Xie, G., Chen, J., 2013. Towards a better understanding of software evolution: an empirical study on open source software. *In Journal of Software: Evolution and Process*, *25*(3), 193-218.

Oliveira R. P., Almeida E. S., 2016. Evaluating Lehman's laws of software evolution for software product Lines. *In IEEE Software*, 33(3), 90-93.

Paulson, J. W., Succi, G., Eberlein, A., 2004. An empirical study of open-source and closed-source software products. *In IEEE Transactions on Software Engineering*, (4), 246-256.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. *8th International Workshop on Principles of Software Evolution (IWPSE'05)*, Lisbon, *14*(2), 131.

Sen, P. K., 1968. Estimates of the regression coefficient based on Kendall's tau. *In Journal of the American Statistical Association*, *63*(324), 1379-1389.

Wu, J., Holt, R. C., 2004. Linker-based program extraction and its uses in studying software evolution. *In Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*, 1-15.