Does Software Structures Quality Improve over Software Evolution? Evidences from Open-Source Projects

Mamdouh Alenezi and Mohammad Zarour
College of Computer & Information Sciences
Prince Sultan University, Riyadh 11586
Saudi Arabia

Abstract

Throughout the software evolution, several maintenance actions such as adding new features, fixing problems, improving the design might negatively or positively affect the software design quality. Quality degradation, if not handled in the right time, can accumulate and cause serious problems for future maintenance effort. Several researchers considered modularity as one of the success factors of Open Source Software (OSS) Projects. The modularity of these systems is influenced by some software metrics such as size, complexity, cohesion, and coupling. In this work, we study the modularity evolution of four open-source systems by answering two main research questions namely: what measures can be used to measure the modularity level of software and secondly, did the modularity level for the selected open source software improves over time. By investigating the modularity measures, we have identified the main measures that can be used to measure software modularity. Based on our analysis, the modularity of these two systems is not improving over time. However, the defect density is improving overtime.

1 Introduction

Software evolve for many reasons that include continuing change, increasing complexity, continuing growth and etc. This means that software need to fix problems, to accommodate new features, and to improve their quality. All these maintenance activities lie within corrective, preventive, adaptive and perfective maintenance that lead to software evolution. In order for the software to survive for a long period, it needs to evolve. This paper is an extended version of our previous work [1]. In this paper we study the software structures quality and investigate more their improvement opportunities over the evolution of four different open source projects.

Software end-users are usually concerned about the external software quality factors depicted as efficiency, usability, and reliability while developers and software engineers are also concerned with the internal quality factors such as evolution and reusability [2]. Software keeps evolving after it has been set in use for the first time. The cost associated with software maintenance and evolution is estimated to be 60% to 80% of total costs associated with a software system [3]. Software evolution is correlated with software structures and complexity [4]; Software structures can be altered via maintenance activities which usually introduce new source code changes that may introduces new dependencies among software

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

elements e.g. packages, methods and classes. Most of the software evolution studies highlight the changes in statistical techniques by analyzing its evolution measures [5], little effort has been carried to comprehend how exactly the structure of these systems evolve [6]. For that reason, we focus in this paper on studying software structures' quality and investigate their improvements during software evolution. Our investigation is based on open source software systems by considering various object oriented structural software measures.

Software structures refer to the various program elements (modules)that make up certain software. the way these elements are organized in the program defines its structural complexity [7]. Modularity has great effect on software development and evolution [8][9][10]. It plays a central role in the design and production of software artifacts, mainly when developing large and complex software [11]. Modularity is one of the maintainability characteristics of the ISO/IEC SQuaRe quality standard series [12]. According to this standard, modularity is defined as a degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components [12]. modularization is the process of decomposing a system into logically cohesive and loosely-coupled modules that hide their implementation from each other and offer functionalities to the outside world through a well-defined interface [13, 14]. Maintenance activities during software evolution might negatively or positively affect software quality including modularity, enhancing software modularity will improve the flexibility and understandability of software systems. As software Modularity increase, its complexity decrease. High modularity in open source allows multiple developers to work on the same software entity, usually in competition, which increases the probability of timely, high-quality solutions [15].

Modularity is an essential property of quality software. High modularity improves the flexibility and understandability of the software system [8], whereas low modularity causes costly refactorings and software bugs [10]. Therefore, modularity is usually utilized as an essential criterion for evaluating the software design quality [12]. In this paper modularity measures are used as means to study the software structures quality and their evolution over projects' releases.

The remainder of this paper is organized as follows: Section 2 discusses the research methodology adopted in this paper. Section 3 states the measures used in this study. The data collection mechanism is given in Section 4. Data analysis and results are presented in Section 5. Threat to validity are discussed in Section 6. Section 7 discusses related work. Conclusions are presented in Section 8.

2 Research Methodology

In this research work we are applying various modularity measures to empirical data taken from open source software. The data are collected from PROMISE, the software engineering repository. Nowadays, open source software repositories provide researchers with the possibility to access large amount of publicly available data for analysis to produce new studies and results. In our study, we will investigate the relationship among various design measures and software modularity. Modularity forms our dependent variable to be studied while the various design measures form the independent variables. Our empirical study focuses on the following research questions:

1. What measure(s) can be used to measure the modularity of OO software programs?

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

2. Did the modularity of the OO programs studied in this research work improve over the various versions?

To answer these questions, we will follow the following steps:

1. Identify the applicable set of measures related to the modularity (Section 4)
2. Identify the set of open source software systems to be used in this research work and collect necessary data pieces from PROMISE repository needed to calculate the specified measures (Section 5)
3. Analyze and report findings (Section 6)

3 Measures in This Study

Various measures are used to measure the quality of modularization. Although deciding which measures can be adopted in experiments on object oriented software modularity is a hard task [16], we decided to consider coupling, cohesion and complexity as measures to be considered in this study. According to [16] measures related to these three internal attributes are among the most adopted measures by experts in the domain. Coupling is the degree of interdependence between modules, whereas cohesion is the intra-modular functional relatedness which describes how tightly bound the internal elements of a module are to one another [14]. An excessive coupling between a system modules affects its modularity but promoting encapsulation and reducing coupling improve modularity [17]. Complexity is also revealed by both cohesion and coupling. Higher cohesion indicates lower complexity, when coupling increases, the complexity also increases. Coupling, cohesion, and complexity relate strongly to the maintenance effort [18]. Moreover, Defect Density is used as a measure of software product quality to investigate if the defect level is improving over successive releases.

This section presents the definition of the measures used in the study. For more detailed definition about these measures refer to [19]. Modularity measures assess the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. SQuaRE standard defined two basic modularity measures:

- Coupling of components: How strong is the coupling between the components in a system or computer programs? basically two measures are used for coupling measure: Coupling between object classes (CBO) and Response for a Class (RFC).
- Cyclomatic complexity: How many software modules have the acceptable cyclomatic complexity? The cyclomatic complexity is measured by two main measures namely: Weighted Methods per Class (WMC), and McCabe's Cyclomatic Complexity (CC).

We observe the modularity of open source software systems by measuring coupling, cohesion, and complexity measures. While major emphasis has been on object oriented measures proposed by Chidamber and Kemerer [20], we have also considered other relevant measures related to coupling and cohesions as shown in the following sub-sections.

3.1 Coupling

Beside the two basic coupling measures given above, we have also chosen other measures that measure the interconnection of software modules. this includes: Afferent couplings

(Ca), Efferent couplings (Ce), Coupling Between Methods (CBM). These coupling measures are well-known and were excessively studied in the literature. Accordingly the selected coupling measures include:

- Coupling between object classes (CBO): It represents the number of other classes that are coupled to the current class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
- Response for a Class (RFC): RFC is the measure of number of methods that can be invoked in response to a message received by an object of the class. Ideally, RFC should measure the transitive closure of the call graph for each method.
- Afferent couplings (Ca): It represents the number of classes from other packages depending on classes in this package. This describes the packages responsibility. Ca is the number of other packages depending on one package. A high number indicates bad design, or that the package is used for crosscutting concerns.
- Efferent Couplings (Ce): It represents the number of packages the classes of this package depend upon. This describes the packages independence. This can be used to point out non-adherence to the design if certain packages have an unreasonable high number of efferent couplings.
- Coupling Between Methods (CBM): It represents the total number of new/redefined methods to which all the inherited methods are coupled. An inherited method is coupled to a new/redefined method if it is functionally dependent on a new/redefined method in the class. Therefore, the number of new/redefined methods to which an inherited method is coupled can be measured.

3.2 Cohesion

To study software systems cohesion, we have chosen different measures that measure the cohesion of software modules. These cohesion measures are well-known and were excessively studied in the literature. We selected the following cohesion measures:

- Lack of cohesion in methods (LCOM): It counts the sets of methods in a class that are not related through the sharing of some of the class fields. It is calculated by subtracting from the number of method pairs that do not share a field access the number of method pairs that do.
- Lack of cohesion in methods (LCOM3): It is an improved variation of the LCOM measure. It calculates the cohesion of the class by considering the effective usage of the class attributes.
- Cohesion Among Methods of Class (CAM): It computes the relatedness among methods of a class based upon the parameter list of the methods. It sums the number of different types of method parameters in every method and divides it by a multiplication of number of different method parameter types in whole class and number of methods.

3.3 Complexity

To study software system's cohesion, we used different complexity measures which are well-known and excessively studied in the literature. These measures include:

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

- Weighted Methods per Class (WMC): It is the sum of the complexities of all class methods.
- McCabe' s Cyclomatic Complexity (CC): It is equal to the number of different paths (decision points) in a method plus one. We report Avg(CC) which is the arithmetic mean of the CC value in the investigated class.

3.4 Defect Density Evolution

Defect Density is post-release defects per thousand lines of delivered code [21]. Defect Density is used here to measure the quality of the software product. It gives an indication of quality improvement achievements in successive releases of certain software. The lower the number of defect density, the better the software quality is. Defect density can be computed using equation 1 as follows:

$$\text{Defect Density} = \frac{\text{Number of Defects}}{\text{KLOC}} \quad (1)$$

Defect density is correlated with number of developers and software size jointly [22]. similar results are obtained in [21], where projects size is found to be an affecting factor (large projects are found to have lower defect density). Development mode is found to be another factor that affects defect density rate (open source projects are found to have a lower defect density).

4 Data Collection

We conducted the empirical study on four open source systems. In selecting the subjected systems, we used several criteria. First, we want well-known systems that are used very widely. Second, systems had to be sizable, so we can understand the issues that appear in the evolution of realistic, multi-developer software. Third, the systems had to be actively maintained. Finally, the data of these systems had to be publicly available. Public availability of the data used for empirical studies is crucial. A theory of software evolution must be based on empirical results, verifiable and repeatable [5]. Characteristics of the selected software systems are listed in Table 1. An overview of each system is provided in the following paragraphs.

Table 1. Selected Software Systems

System	Versions	LOC
Camel	1.0-1.6	3594-113055
jEdit	4.0-4.3	144803-202363
POI	1.5-3.0	55428-129327
Xerces	1.0-1.4	90718-141180

Apache Camel is a powerful open source integration framework based on known Enterprise Integration Patterns with powerful Bean Integration. jEdit is a mature programmer' s text editor with hundreds (counting the time developing plugins) of person-years of development behind it. It is written in Java and runs on any operating system with Java support, including Windows, Linux, Mac OS X, and BSD. The POI project consists of APIs for

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

manipulating various file formats based upon Microsoft's OLE 2 Compound Document format, and Office OpenXML format, using pure Java. Xerces is a parser that supports the XML 1.0 recommendation and contains advanced parser functionality, such as support for XML Schema 1.0, DOM level 2 and SAX version 2.

The data for this study were collected by [19] and are available online at the PROMISE repository. This data was widely used in the software engineering literature for different purposes [23, 24, 25]. The collected measures' data for the four systems are added up correspondingly into one data set along with the relevant values for coupling, cohesion, and complexity measures. Descriptive statistics (Min, Max, Median, Std. dev.) defined the minimum, maximum, median, and standard deviation measures. Table 2 shows descriptive statistics about the selected measures.

Table 2. Descriptive Statistics of the Measures

Measures	Min	Max	Med	σ
CBO	0	187	9.8	15.20
RFC	0	498	27.72	39.8
Ca	0	184	5.33	13.79
Ce	0	95	5.24	6.75
CBM	0	25	1.59	3
LCOM	0	41719	116.3	933.3
LCOM3	0	2	1.14	0.67
CAM	0	1	0.47	0.25
WMC	0	409	10	18.8
Avg(CC)	0	25.14	1.28	1.3

5 Data Analysis and Results

In order to answer the first question, we need to know which aspects of coupling is measured by any of the chosen coupling measures. Same thing holds for cohesion. To achieve that, we use the well-known Principal Component Analysis (PCA) which is a standard statistical procedure that uses orthogonal transformation to identify the underlying, orthogonal dimensions that explain relations between the variables in the data set. We conducted the experiments using the R statistical software (version 3.1.1) and we used R's Procomp procedure to our data to produce principal components. The analysis is done on the entire data set of the considered measures.

The objectives of principal component analysis are to discover or reduce the dimensionality of the data set and identify new meaningful underlying variables. PCA is a de facto technique for uncovering the underlying orthogonal dimension that explains variables relations in a dataset. PCA is used in our case to identify measures (i.e, groups of independent variables) that measure the same underlying dimension (i.e., mechanism that defines coupling and cohesion among classes). Principal Components (PCs) are linear combinations of independent variables. The number of PCs is less than or equal to the number of original variables. PCs are interpreted as follows. Each new PC is orthogonal to all previously calculated PCs and captures a maximum variance under these conditions.

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

5.1 Coupling Evolution Analysis & Results

In this section we apply the PCA approach to the coupling measures to specify any correlations among them. If a group of coupling measures are strongly correlated, these measure are likely to measure the same underlying dimension (i.e., class property) of the object to be measured.

Table 3. Rotated Components of Coupling measures

	PC1	PC2	PC3	PC4
Proportion	39%	22%	20%	19%
Cumulative	38%	60%	80%	100%
CBO	0.92	0.34	-0.02	0.20
RFC	0.21	0.39	0.11	0.89
Ca	0.99	0.00	-0.03	0.10
Ce	0.18	0.91	0.04	0.37
CBM	-0.03	0.04	1.00	0.08

By analyzing the coefficients associated with every coupling measure within each rotated component given in Table 3, we interpret the identified PCs as the following:

- PC1 (39%): CBO and Ca measures count inbound coupling through method invocations. The correlation betweeen the two measures is high. We can use one of them rather than using both. Apparently, the afferent couplings measure is the contributing measure as it has higher PC value.
- PC2 (22%): Ce captures outbound coupling through method invocations.
- PC3 (20%): CBM captures coupling between inherited and redefined methods.
- PC4 (19%): RFC counts the number of accessible methods.

5.2 Cohesion Evolution Analysis & Results

We also conducted PCA analysis on the selected cohesion measures. We want to see if any correlations exists between these measures.

Table 4. Rotated Components of Cohesion measures

	PC1	PC2	PC3
Proportion	33%	33%	33%
Cumulative	33%	67%	100%
LCOM	1	-0.01	-0.07
LCOM3	-0.01	0.98	0.20
CAM	-0.08	0.21	0.98

By analyzing the coefficients associated with every cohesion measure within each rotated component given in Table 4, we found that the identified PCs as each on of these cohesion measures is unique and does not overlap with the others.

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

5.3 Defect Density Evolution

We measured the defect density of each version of the adopted open source software in order to investigate if the evolution of each software reduces the defect density or increases it over the different releases. Figure 1 shows how defect density evolved in the selected systems. The defect density is shown to improve for each of the tested software over the successive releases.

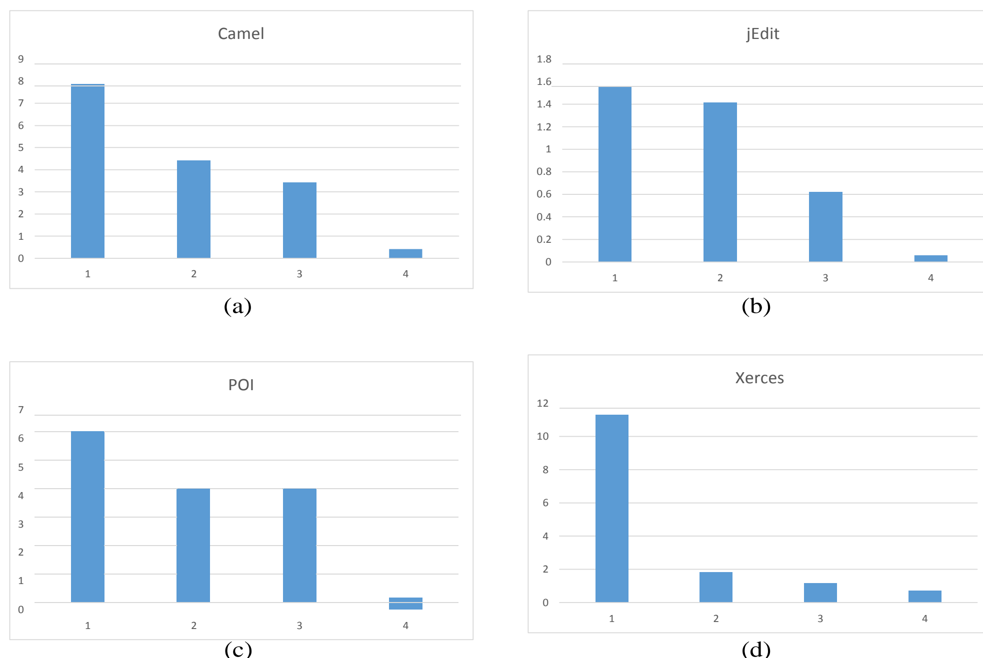

Figure 1. The Evolution of Defect Density in the Selected Systems.

5.4 Discussion

According to our PCA analysis, the coupling measures that can be used to measure the system's modularity are Ca, Ce, CBM and RFC. The CBO measure has been excluded as the Ca measures the same dimension. The cohesion measures that can be used to measure the system's modularity are LCOM, LCOM3, and CAM. These measures along with the complexity measures WMC and CC measures altogether are the set of measures that measure the Modularity of a system or software program. This answers the first research question.

Table 5 shows the coupling, cohesion, and complexity evolution of the four selected systems. There are three notions which characterize good and bad things about modules, coupling (we want low coupling between modules), cohesion (we want highly cohesive modules), and complexity (we want modules that have low complexity) [2]. Modularity is a concept in which a software is decomposed of several distinct and logically cohesive sub-units, offering services through a well-defined interface [13]. Excessive inter-module

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

Table 5. Modularity Evolution of the Selected Systems

			Ver. 1	Ver. 2	Ver. 3	Ver. 4
Camel	Coupling	Ca	4.99	5.02	5.11	5.27
		Ce	5.69	5.62	6.33	6.43
		CBM	0.56	0.64	0.61	0.91
		RFC	19.63	20.23	21.2	21.42
	Cohesion	LCOM	53.65	61.24	73.42	79.33
		LCOM3	0.99	1.08	1.11	1.1
		CAM	0.48	0.5	0.49	0.49
	Complexity	WMC	8.07	8.31	8.52	8.57
		Avg(CC)	0.94	0.93	0.94	0.96
jEdit	Coupling	Ca	7.51	7.93	8.62	8.74
		Ce	6.43	6.63	7.16	7.1
		CBM	1.61	1.59	1.55	1.5
		RFC	38.24	39.87	40.98	39.85
	Cohesion	LCOM	197.38	187.89	310.76	259.91
		LCOM3	1.05	1	0.99	1.09
		CAM	0.47	0.45	0.44	0.46
	Complexity	WMC	12.88	13.13	13.16	12.35
		Avg(CC)	1.79	1.87	1.92	1.83
POI	Coupling	Ca	4.36	4.51	4.7	5.23
		Ce	4.31	4.48	4.68	5.22
		CBM	2.78	2.62	2.7	1.95
		RFC	27.56	29.65	30.9	30.35
	Cohesion	LCOM	92.87	103.76	107.12	100.46
		LCOM3	1.02	0.97	0.98	1
		CAM	0.44	0.42	0.43	0.38
	Complexity	WMC	13.39	14.3	14.26	13.51
		Avg(CC)	1.09	1.15	1.16	1.19
Xerces	Coupling	Ca	3.33	2.52	2.67	3.35
		Ce	3.38	2.68	2.75	3.27
		CBM	1.93	1.41	1.38	1.43
		RFC	23.33	21.23	21.7	19.24
	Cohesion	LCOM	139.48	91	94.52	75.49
		LCOM3	1.22	1.49	1.47	1.47
		CAM	0.52	0.51	0.5	0.52
	Complexity	WMC	11.43	11.28	11.38	9.94
		Avg(CC)	1.26	1.22	1.24	1.4

dependencies has been acknowledged to be an indicator of poor design and decrease the comprehending of components in isolation [26].

Figure 2 shows the evolution of coupling, cohesion, complexity measures of the selected systems over four different releases for each system. The X-axis represents the release number while the Y-axis represents the measures data. As can be seen from figure 2 a, d and g, we can see that there is a minor change in the Ca, Ce and CBM coupling measures. But there is slightly more increase in the RFC measure in Camel, jEdit and POI while the RFC slightly decrease for Xerces. Hence, coupling is slightly increasing while the software is evolving, this indicates that the modularity is not improving over time.

Figure 2 b show that the LCOM and LCOM3 are increasing over the various releases of Camel. Figure 2 e shows that for jEdit, the cohesion level indicated some improvement in the second release but lost it in the third release then again made some progress in the forth release but still not as good as in the first release which meanins that overall the

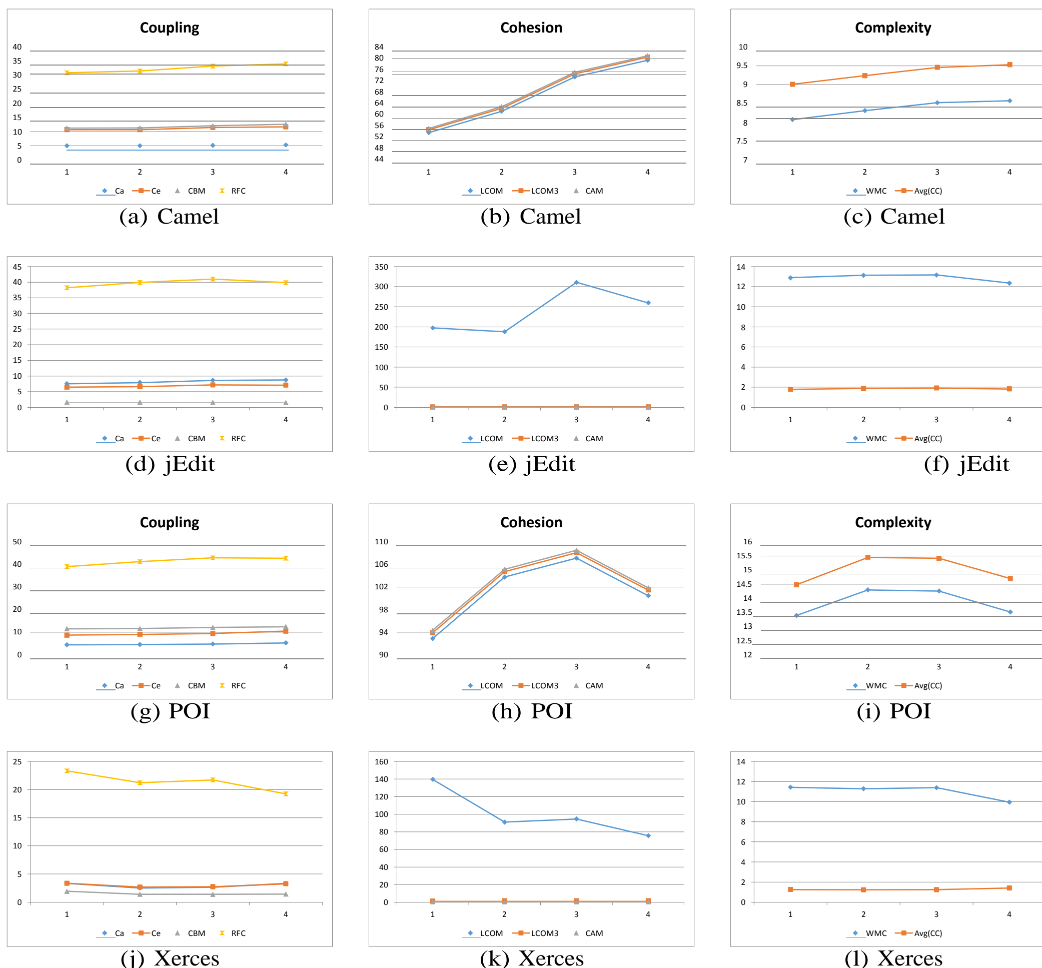

Figure 2. Modularity Evolution of the Selected Systems.

cohesion is not improving significantly in jEdit. Figure 2 h shows cohesion of the POI software. overall the cohesion measures indicate that the cohesion is not improving till the third release, then the cohesion started improving in the fourth release but still not good as in the first release. Xerces is in better situation that the other software where the LCOM is improving over the various successive releases, LCOM3 and CAM are kept in a steady level. Accordingly, we can notice that the cohesion measures shows that the Camel jEdit and POI software are not improving while evolving.

Regarding the complexity measure, Figure 1 c shows that the Camel software complexity is increasing over the various releases. This indicates that there is no restructuring activities is done in these four versions. jEdit software shows some improvement over its successive releases. This means that some restructuring activities have took place but did not significantly improve the jEdit complexity. POI software complexity has increased in the second release, but started to decrease in the following releases but still the complexity is slightly more than that of the first release. For Xerces software complexity shows a noticeably improvement in WMC measure and some improvement in the avg(CC) measure in the fourth

Table 6. Spearman correlation coefficient of Modularity measures and Bugs

	Ca	Ce	CBM	RFC	LCOM	LCOM3	CAM	WMC	avg(CC)
Bugs	0.14	.19	.12	.38	0.40	0.22	0.15	0.32	0.36

release.

Accordingly, the various measures of coupling, cohesion and complexity of Camel jEdit and POI software show that the modularity of three of the software is not improving overall! Xerces software is in a better situation where its measures showed some improvements. This means that modularity is not improving significantly, hence we can not say that an effective restructuring has took place, although defect density is improved. This means that designers and developers where concerned with solving bugs and problems without paying enough attention to restructuring that aims to improve software structures quality. Hence, we believe that restructuring is needed in the coming releases to improve software quality. This answers the second research question.

Moreover, to test the relationship between the modularity measures and the number of bugs in a software version, we have conducted a correlation analysis. Correlation analysis studies the degree to which changes in the value of an attribute (one of the modularity measures) are associated with changes in another attribute (number of faults in a version). The Spearman correlation is preferred instead of Pearson correlation because the former ignores any assumptions about the data distribution [27].

Ifthe measuretends to increase whenthe number of bugs increases, the Spearman correlation coefficient is positive. Ifthe measuretends to decrease whenthe number of faultsincreases, the Spearman correlation coefficient is negative. Table 6 shows that RFC, LCOM, WMC, and Avg CC have a moderate correlation with number of faults. These results are very similar to Johari and Kaur study [28]. Accordingly, our data shows that there is a moderate relation between modularity measures and number of faults in our software sample.

6 Threats to Validity

The conducted research in this paper is exposed to possible validity threats which are defined and discussed in [29]:

- Construct Validity: The various measures we used (coupling measures, cohesion measures, complexity measures, defect measures and correlation measures) are well documented in literature. The data are collected for four open source software which are public ally available.
- External Validity: Our data set is collected from the software engineering repository PROMISE. We have collected data for four open source software over four successive releases for each. Results obtained based on this data set should be relevant and valid for other releases of the studied software as well as other ones.
- Internal Validity: All the needed data pieces in this study have been collected by the researchers from the mentioned data repository. missing data could be there but has minimal effect of the conducted analysis and conclusion.
- Conclusion Validity: Our analysis have been conducted based on the collected data

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

set. A threat to the conclusion validity is can be related to how the data is reported in the repository; e.g what is considered a fault or bug for example when a certain incident occur in the system would it be considered as a fault or change request. As we are talking about open source software, the developers and designers skills participating in the four project from different locations with different experience skills may form a threat. The number of projects used in this research may also form a kind of threats. We have used data sets for four software projects. Although we believe the results can be generalized for other open source projects, enlarging the data set by adding more projects may provide more reliable results.

7 Related Work

Open-source systems are usually developed by distributed teams, without frequently meeting face-to-face, and communicating only by electronic means. Achieving high modularity in open source allows multiple developers to work on the same software entity without issues [15]. This new structure is totally unlike the common software engineering practices during the times of Lehman's software evolution laws [5]. Lehman et al. have built the well-known research on the evolution of large software systems. Lehman's laws are based on case studies of several large software systems, suggest that as systems grow in size, it becomes increasingly difficult to add new code unless clear steps are taken to restructure the overall design.

MacCormack et al. [30] employed Design Structure Matrix (DSM) to compare and contrast the design structures of two software systems, Linux kernel and Mozilla web browser. They used a clustering algorithm to measure dependencies by different parts of the system and calculated marginal changes in cost rather than the total cost of the matrix. However, the comparison between these two systems critically depends on selecting versions of the systems that are comparable in terms of number of source files. One motivation of our work was to remove this restriction, and to allow the comparison of code bases of different size. LaMantia et al. [31] examined the evolution over time of two software systems, Apache Tomcat and another closed source server product. They introduced a rough measure that mimics the change ratio between the consecutive versions in the software evolution. The authors concluded that DSM could, to some extent, explain how modularization allow for different rates of evolution to occur in different modules.

Koch found differences in the evolution of open-source software projects of different sizes [32]. He found that small open-source software projects fulfill some of the laws. However, large software projects do not follow them at all. These projects have a large number of participants and an unbalanced workload among participants. One of the essential characteristics of software systems is evolution. Several research studies aimed at explaining and understanding the evolution in open source software projects. Breivold et al. [33] conducted a systematic literature review of enormous studies, which investigated the evolution of open source software systems. Another direction has emphasized how software measures can be applied to software evolution [34] where they provided ways in how software measures have been and can be used to analyze software evolution. They suggested that measures are good candidates to understand the quality evolution of a software system by considering its successive releases. Particularly, measures can be used to measure whether the quality of a software has improved or degraded between two releases. Lee et al. [35]

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

provided a case study of one open source software, JFreeChart, evolution with software measures. They studied the evolution in terms of size, coupling and cohesion, and discussed its quality change based on the Lehman’ s laws of evolution [5]. Neamtiu et al. [36] conducted an empirical analysis on the evolution of nine popular open-source programs and investigated Lehman’ s evolution laws where their study confirmed that continuing change and continuing growth are still applicable to the evolution of today’ s open-source software.

Neamtiu et al. [36] used source code measures with project defect information to analyze software growth, characterize software changes, and assess software quality. Murgia et al. [37] focused their study on software quality evolution in open source projects using agile practices. They used several OO measures to study how bug distribution relates to software evolution. They found that there is no a single metric that is able to explain the bug distribution during the systems evolution. Eski et al. [38] investigated the relationship between OO measures and changes in open source software systems and proposed a metric-based approach to predict change-prone classes.

Other researchers have conducted similar empirical studies and prosed some new metrics, for instance Li et. al. [39] studied the evolution of an object-oriented system using the OO metrics suggested by Chidamber and Kemerer to measure the class-level design and proposed three new metrics to study OO system evolution (System Design Instability (SDI), Class Implementation Instability (CII), and System Implementation Instability (SII)). Drowin [40] analyzed empirically the quality evolution of an open source software using a control flow based metric (Quality Assurance Indicator - Qi) which they claimed that Qi metric reflects properly the quality evolution of the system.

In this paper, we study the modularity evolution of four open-source systems. The focus of this study is not the Lehman’ s Law but the modularity using coupling, cohesion, and complexity measures.

8 Conclusion

Enhancing our ability to understand and capture software evolution is essential for better software quality and easier software maintenance process. one of the vital features that reflects the software quality is its structures quality. structures quality has relationship with software modularity. We have used modularity measures to give indications about software structures quality. In this research work, we have used empirical data related to four OO open source programs to answer two main research questions namely: what measures can be used to measure the modularity level of software and secondly, did the modularity level for the selected open source software and their structures quality improve over time? By investigating the modularity measures as mentioned in the SQuaRE standard and various other coupling and cohesion measures, we have identified the main measures that can be used to measure software modularity. Based on our analysis, the modularity of these four systems did not show a significant improvement in their modularity and structures quality over time. However, the defect density is improving over time.

References

[1] Mamdouh Alenezi and Mohammad Zarour. Modularity measurement and evolution in object-oriented open-source projects. In *International Conference on Engineering & MIS 2015 (ICEMIS’15)*. ACM, 2015.

[2]Frank F Tsui. *Essentials of software engineering*. Jones & Bartlett Publishers, 2013.

[3]Edward E. Ogheneovo. Development of a software maintenance cost estimation model: 4th gl perspective. *Journal of Computer and Communications*, 2:1–16, 2014.

[4]Ana Filipa Nogueira. Predicting software complexity by means of evolutionary testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 402–405. ACM, 2012.

[5]Michael W Godfrey and Daniel M German. On the evolution of lehman's laws. *Journal of Software: Evolution and Process*, 26:613619, 2013.

[6]Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software: Evolution and Process*, 25(2):139–163, 2013.

[7]Narasimhaiah Gorla and Ravi Ramakrishnan. Effect of software structure attributes on software development productivity. *Journal of Systems and Software*, 36(2):191–199, 1997.

[8]Sunny Huynh, Yuanfang Cai, Yuanyuan Song, and Kevin Sullivan. Automatic modularity conformance checking. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08.*, pages 411–420. IEEE, 2008.

[9]David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[10]Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[11]Alessandro Rossi and Alessandro Narduzzo. Modular design and the development of complex artifact lesson fron free open source software. Technical report, Department of Computer and Management Sciences, University of Trento, Italy, 2003.

[12]ISO/IEC. Systems and software engineering - systems and software quality requirements and evaluation (square). *ISO/IEC 25010 - System and software quality models*, 2011.

[13]Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[14]Grady Booch, Robert A Maksimchuk, Michael W Engel, Bobbi J Young, Jim Conallen, and Kelli A Houston. *Object-oriented analysis and design with applications*, volume 3. Addison-Wesley, 2008.

[15]Mark Aberdour. Achieving quality in open-source software. *IEEE Software*, 24(1):58–64, 2007.

[16]Juliana de AG Saraiva, Micael S de França, Sérgio CB Soares, JCL Fernando Filho, and Renata MCR de Souza. Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs. *Journal of Systems and Software*, 103:85–101, 2015.

[17]Mourad Badri, Linda Badri, and Fadel Touré. Empirical analysis of object-oriented design metrics: Towards a new metric using control flow paths and probabilities. *Journal of Object Technology*, 8(6):123–142, 2009.

[18]Mamdouh Alenezi and Khaled Almustafa. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8(2):257–266, 2015.

[19]Marian Jureczko and Diomidis Spinellis. Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wroc~lawskiej*, pages 69–81, 2010.

[20]Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[21]Syed Muhammad Ali Shah, Maurizio Morisio, and Marco Torchiano. An overview of software defect density: A scoping study. In *19th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 406–415. IEEE, 2012.

[22]Cobra Rahmani and Deepak Khazanchi. A study on defect density of open source software. In *IEEE/ACIS 9th International Conference on Computer and Information Science (ICIS)*, pages 679–683. IEEE, 2010.

[23]Giuseppe Scanniello, Carmine Gravino, Andrian Marcus, and Tim Menzies. Class level fault prediction using software clustering. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 640–645. IEEE, 2013.

[24]Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101–1118, 2013.

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

[25]Mamdouh Alenezi, Shadi Banitaan, and Qasem Obeidat. Fault-proneness of open source systems: An empirical analysis. In *International Arab Conference on Information Technology (ACIT2014)*, pages 164–169, 2014.

[26]Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. Software architecture evolution through evolvability analysis. *Journal of Systems and Software*, 85(11):2574–2592, 2012.

[27]Jay Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2015.

[28]Kalpana Johari and Arvinder Kaur. Validation of object oriented metrics using open source software system: an empirical study. *ACM SIGSOFT Software Engineering Notes*, 37(1):1–4, 2012.

[29]Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[30]Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[31]Matthew J LaMantia, Yuanfang Cai, Alan D MacCormack, and John Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *Seventh Working IEEE/IFIP Conference on Software Architecture*, pages 83–92. IEEE, 2008.

[32]Stefan Koch. Software evolution in open source projectsa large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(6):361–382, 2007.

[33]Hongyu Pei Breivold, Muhammad Aufeef Chauhan, and Muhammad Ali Babar. A systematic review of studies of open source software evolution. In *17th Asia Pacific Software Engineering Conference (APSEC), 2010*, pages 356–365. IEEE, 2010.

[34]Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 83–86. ACM, 2001.

[35]Young Lee, Jeong Yang, and Kai H Chang. Metrics and evolution in open source software. In *Seventh International Conference on Quality Software, 2007. QSIC'07.*, pages 191–197. IEEE, 2007.

[36]Iuan Neamtiu, Guowu Xie, and Jianbo Chen. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process*, 25(3):193–218, 2013.

[37]Alessandro Murgia, Giulio Concas, Roberto Tonelli, and Ivana Turnu. Empirical study of software quality evolution in open source projects using agile practices. In *Proc. of the 1st International Symposium on Emerging Trends in Software Metrics*, page 11, 2009.

[38]Sinan Eski and Feza Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 566–571. IEEE, 2011.

[39]Wei Li, L Etzkorn, C Davis, and J Talburt. An empirical study of object-oriented system evolution. *Information and Software Technology*, 42(6):373–381, 2000.

[40]Nicholas Drouin, Mourad Badri, and Fadel Tour´e. Metrics and software quality evolution: A case study on open source software. In *Proceedings of the 5th International Conference on Computer Science and Information Technology, Hong Kong*, 2012.