Modularity Measurement and Evolution in Object-Oriented Open-Source Projects

Mamdouh Alenezi
College of Computer & Information Sciences
Prince Sultan University
Riyadh 11586, Saudi Arabia
malenezi@psu.edu.sa

Mohammad Zarour
College of Computer & Information Sciences
Prince Sultan University
Riyadh 11586, Saudi Arabia
mzarour@psu.edu.sa

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. In this work, we study the modularity evolution of two 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.

CCS Concepts

•General and reference → Empirical studies; Measurement; *Experimentation;* **•Software and its engineering → Object oriented architectures; Modules / packages;**

Keywords

Software Quality; Software Modularity; Software Evolution; Software measures; Open source.

1. INTRODUCTION

Software evolution is the dynamic behavior of software systems while they are maintained and improved over their lifetimes [12]. Software systems usually evolve to fix problems, to accommodate new features, and to improve their quality. Hence, the changes that software undergo 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. Most software evolution studies highlight the changes in statistical techniques by analyzing its evolution measures [12], little effort has been carried to comprehend how exactly the structure of these systems evolve [17]. For that reason, we focus in this paper on studying the structure of open source software systems by considering various object oriented structural software measures.

Controlling and monitoring the evolution quality is very essential for efficient software maintenance. The maintenance activities might negatively or positively affect software quality including modularity, enhancing software modularity will improve the flexibility and understandability of software systems. Modularity is a key concept that developers exercise to decrease the complexity of software systems. A recurrent issue in software products is that, as their size and functionality increases, they become harder to code, test and maintain. Even the smallest change would be a very hard task in time and cost due to the system's complexity. To tackle this problem and keep the system simple, software should be modularized properly during their design.

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 [5, 6]. Modularity is one of the maintainability characteristics of the ISO/IEC SQuaRe quality standard series [14]. 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 [14]. 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 [1].

In practice, modularization parallels to finding the correct decomposition of a problem. Organizing the modules of the source code with favoring cohesion (within-module connections) over coupling (between-module connections) is considered a good practice. Modularity is an essential property of quality software. High modularity improves the flexibility and understandability of the software system [13], whereas low modularity causes costly refactorings and software bugs [27]. Therefore, modularity is usually utilized as an essential criterion for evaluating the software design quality [14]. In this paper, the modularity evolution two object-oriented open software systems has been studied through a thorough empirical study.

The remainder of this paper is organized as follows. Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICEMIS '15, September 24-26, 2015, Istanbul, Turkey

© 2015 ACM. ISBN 978-1-4503-3418-1/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2832987.2833013

tion 2 discusses related work. Section 3 discusses the research methodology adopted in this paper. Section 4 states the measures used in this study. The data collection mechanism is given in Section 5. Data analysis and results are presented in Section 6. Conclusions are presented in Section 7.

2. RELATED WORK

One of the metrics' categories developed by [11] to assess object oriented software maintainability is the license type, e.g. open source and proprietary software. 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 [1]. This new structure is totally unlike the common software engineering practices during the times of Lehman's software evolution laws [12]. Lehman et al. have built the well-known research on the evolution of large software systems. LehmanâĂŹs 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.

Koch found differences in the evolution of open-source software projects of different sizes [16]. 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. [7] 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 [21] 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. [19] 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 [12]. Xie et al. [28] conducted an empirical analysis on the evolution of seven open source systems and investigated LehmanâĂŹs evolution laws.

MacCormack et al. [20] employed Design Structure Matrix (DSM) [24] 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. [18] 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.

In this paper, we study the modularity evolution of 2 open-source systems. The focus of this study is not the Lehman's Law but the modularity using coupling, cohesion, and complexity measures.

3. RESEARCH METHODOLOGY

In this research work we are applying various modularity measures to empirical data related to open source software. The data are collected from PROMISE[1], 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?
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)

4. 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 [11], we decided to consider coupling, cohesion and complexity as measures to be considered in this study. According to [11] 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 intramodular functional relatedness which describes how tightly bound the internal elements of a module are to one another [6]. An excessive coupling between a system modules affects its modularity but promoting encapsulation and reducing coupling improve modularity [4]. 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 [10, 2].

[1]https://code.google.com/p/promisedata/

This section presents the definition of the measures used in the study. For more detailed definition about these measures refer to [15]. 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 [9], we have also considered other relevant measures related to coupling and cohesions as shown in the following sub-sections.

4.1 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.

4.2 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.

4.3 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:

- 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.

5. DATA COLLECTION

We conducted the empirical study on two 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 [12]. Characteristics of the selected software systems are listed in Table 1. An overview of each system is provided in the following paragraphs.

Camel (http://camel.apache.org/). Apache Camel is a powerful open source integration framework based on known Enterprise Integration Patterns with powerful Bean Integration. POI (http://poi.apache.org/). The POI project

Table 1: Selected Software Systems

System	Versions	LOC
Camel	1.0-1.6	3594-113055
POI	1.5-3.0	55428-129327

consists of APIs for manipulating various file formats based upon Microsoft's OLE 2 Compound Document format, and Office OpenXML format, using pure Java.

The data for this study were collected by [15] and are available online at the PROMISE repository. This data was widely used in the software engineering literature for different purposes [22, 26, 3]. The collected measures' data for the two 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: Descriptive Statistics of the Measures

Measures	Min	Max	Med	σ
CBO	0	185	9.7	15.20
RFC	0	494	27.64	39.8
Ca	0	184	5.33	13.79
Ce	0	93	5.24	6.75
CBM	0	25	1.59	3
LCOM	0	41713	116.3	933.3
LCOM3	0	2	1.14	0.67
CAM	0	1	0.47	0.25
WMC	0	407	10	18.8
Avg(CC)	0	25.14	1.28	1.3

Table 2 shows descriptive statistics about the selected measures.

6. 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). The principal component analysis (PCA) 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 Rs Procomp[2] 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.

[2]https://stat.ethz.ch/R-manual/R-patched/library/stats/html/prcomp.html

6.1 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	38%	22%	20%	20%
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 (38%): 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 (20%): RFC counts the number of accessible methods.

6.2 Cohesion Evolution Analysis & Results

We also conducted PCA 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.

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
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

6.3 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 two 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) [23, 25]. 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 [5]. Excessive inter-module dependencies has been acknowledged to be an indicator of poor design and decrease the comprehending of components in isolation [8].

Figure 1 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 1 a and d, 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. Hence, coupling is slightly increasing while the software is evolving, this indicates that the modularity is not improving over time.

As can be seen from figure 1 b, we can see that the LCOM and LCOM3 are increasing over the various releases which means that the lack of cohesion in methods increase, while the cohesion among methods is increasing. Hence, two of the cohesion measures shows that the Camel software is not improving while evolving. Figure 1 e 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.

With regard to the complexity, 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. 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. Accordingly, the various measures of coupling, cohesion and complexity of Camel and POI software show that the modularity of the two software is not improving overall! This means that restructuring is needed in the coming releases. This answers the second research question.

7. CONCLUSION

Enhancing our ability to understand and capture software evolution is essential for better software quality and easier software maintenance process. One of the software characteristics that helps in achieving this is software modularity. Modularity is one of the sub-characteristics of maintainability which is one of the software product quality factors. In this research work, we have used empirical data related to two 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 improves 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 two systems is not improving over time.

8. REFERENCES

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

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

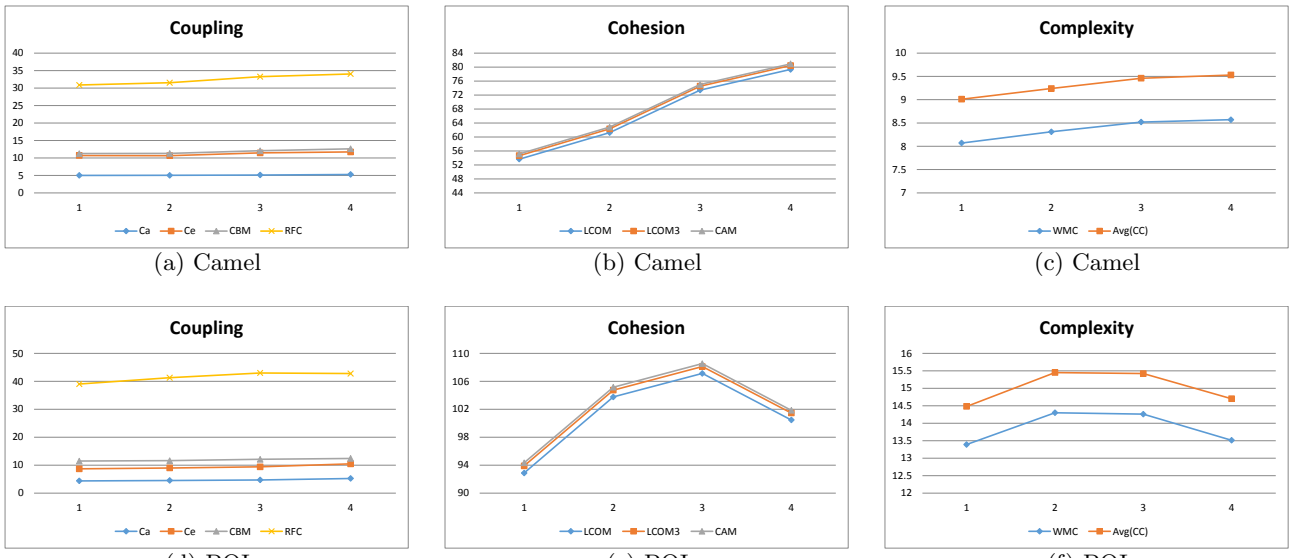
[3] M. Alenezi, S. Banitaan, and Q. Obeidat. Fault-proneness of open source systems: An empirical analysis. In *International Arab Conference on Information Technology (ACIT2014)*, pages 164–169, 2014.

[4] M. Badri, L. Badri, and F. 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.

[5] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[6] G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. *Object-oriented analysis and design with applications*, volume 3. Addison-Wesley, 2008.

[7] H. P. Breivold, M. A. Chauhan, and M. A. 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.



(a) Camel (b) Camel (c) Camel

(d) POI (e) POI (f) POI

Figure 1: Modularity Evolution of the Selected Systems.

[8] H. P. Breivold, I. Crnkovic, and M. Larsson. Software architecture evolution through evolvability analysis. *Journal of Systems and Software*, 85(11):2574–2592, 2012.

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

[10] D. P. Darcy, S. L. Daniel, and K. J. Stewart. Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–11. IEEE, 2010.

[11] J. de AG Saraiva, M. S. de França, S. C. Soares, J. Fernando Filho, and R. M. de Souza. Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs. *Journal of Systems and Software*, 103:85–101, 2015.

[12] M. W. Godfrey and D. M. German. On the evolution of lehman's laws. *Journal of Software: Evolution and Process*, 2013.

[13] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08.*, pages 411–420. IEEE, 2008.

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

[15] M. Jureczko and D. Spinellis. Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pages 69–81, 2010.

[16] S. Koch. Software evolution in open source projectsï£¡a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(6):361–382, 2007.

[17] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. 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.

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

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

[20] A. MacCormack, J. Rusnak, and C. 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.

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

[22] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies. Class level fault prediction using software clustering. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 640–645. IEEE, 2013.

[23] I. Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2010.

[24] D. V. Steward. The design structure system: a method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (EM-28), 1981.

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

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

[27] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[28] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *IEEE International Conference on Software Maintenance, 2009. ICSM 2009*, pages 51–60. IEEE, 2009.