# Evaluating Dependency based Package-level Metrics for Multi-objective Maintenance Tasks

Mohsin Shaikh
Qaid-e-Awam University
of Engineering Science
and Technology, Nawabshah

Adil Ansari
Qaid-e-Awam University
of Engineering Science
and Technology, Nawabshah

Kashif Memon
UCET, The Islamia University
of Bahwalpur

Akhtar Hussain Jalbani
Qaid-e-Awam University
of Engineering Science
and Technology, Nawabshah

Ahmed Ali
Qaid-e-Awam University
of Engineering Science
and Technology, Nawabshah

***Abstract*—Role of packages in organization and maintenance of software systems has acquired vital importance in recent research of software quality. With an advancement in modularization approaches of object oriented software, packages are widely considered as re-usable and maintainable entities of object-oriented software architectures, specially to avoid complicated dependencies and insure software design of well identified services. In this context, recently research study of H. Abdeen on automatic optimization of package dependencies provide composite frame of metrics for package quality and overall source code modularization. There is an opportunity to conduct comprehensive empirical analysis over proposed metrics for assessing their usefulness and application for fault-prediction, design flaw detection, identification of source code anomalies and architectural erosion. In this paper, we examine impact of these dependency optimization based metrics in wide spectrum of software quality for single package and entire software modularization. Our experimental work is conducted over open source software systems through statistical methodology based on cross validation fault-prediction and correlation. We conclude with empirical evidence that dependency based package modularization metrics provide more accurate view for predicting fault-prone packages and improvement of overall software structure. Thus, application of these metrics can help the developers and software practitioners to insure proactive management of the source code dependencies and avoid design flaws during software development.**

***Keywords*—Software quality; package-level metrics; software modularization; fault-prediction***

## I. Introduction

Software engineering is aimed at developing mechanism and tools that automates the manual operation. For the assessment of software quality, functional stability and maintainability of its design are prime objectives. Recently, packages have acquired core interest for proper organization of source code entities due to growing complexity of classes in object oriented (OO) source code paradigms. A package is relatively easier to re-use, re-factor and test, eventually reducing maintenance cost [1], [2]. There have been increasing efforts to analyze packages and their architecture in object oriented systems to determine quality attributes of object oriented source code [3], [4]. Conventionally, software evolution process has been subject to structural and architectural changes in the source code, targeting suitable and organized placement of classes in particular. However, such re-factoring practices can cause drift and deterioration in modularization quality of software [5]. Consequently, to insure flexible software modularization, optimization of package structure and their connectivity can be a vital maintenance task. In practice, if quality of package dependencies is evaluated quantitatively, then modifying its structural components to avoid potential flaws becomes easier task.

Although, there have been attempts to improve the modularization of software through heuristic search methods using decomposition techniques and deterministic procedures. These are frequently based on clustering approaches which do not address the source code design issues at precise level of granularity like, classes or packages [6], [7]. Furthermore, the existing approaches of software modularization for changing the structure of package entail costly maintenance overhead, complicating its understandability and comprehension [8], [9]. Abdeen *et al.* recently proposed a package level metrics suite which support the modularization of source code architecture using existing package structure, thus without affecting prevalent software design adversely [10].

Despite different existing efforts of proposing metrics to characterize the packages in object oriented systems, there is a need to evaluate capability of these metrics to measure intended quality attributes of software [11], [12]. In this paper, we examine the usefulness of package quality metrics proposed by Abdeen *et al.* in wide spectrum of software quality, i.e., fault-proneness, vulnerability detection, coding standard violation. We also determine the correlation of modularization metrics presented by Abdeen *et al.* with already studied modularity metrics of different application domains. In this context, First, we develop prediction model for fault-proneness of packages with logistic regression in comparison to traditional Martin package level metrics suite and linear correlation models

with post-release faults (reported in standard bug repositories), vulnerabilities and coding standards are examined. Secondly, we form correlation model among different modularization metrics to understand and evaluate the their relationships and its impact over entire modularization design. Experimental results on open source software systems show that: 1) Package Quality metrics by Abdeen et al. are better candidates of fault-proneness when used in combination with Martin's metric suite in inter as well as intra releases of software systems; 2) Most of Package Quality metrics have shown reasonable association with actual post-release faults, vulnerabilities and coding standards violations detected through open source tools like *FindBugs* and *PMD*; 3) Package level Modularization metrics by Abdeen *et al.* can be used to evaluate the external strength of overall structural design from the perspective of cohesion, coupling and cyclic/acyclic dependencies. It asserts that improper handling of dependencies at package level can further deteriorate source code causing operational complexity and difficult re-usability.

The rest of the paper is organized as follows. Section 2 explains the motivation of research study in the context of Software Quality. Section 3 describes investigated metrics with their formal definitions. Section 4 provides illustrative example for comprehension of metrics computation. Section 5 presents detailed empirical study with stated research objectives, methodology and obtained results using graphical and tabular representation. Section 6 illustrates most significant related work in the literature of package level fault-prediction. Implication of research study is discussed in Section 7. Threats to validity are explained in Section 8 followed by Conclusion as Section 9.

## II. Motivation

Over the years, a variety of quality models (QMs) has been proposed objectively to support the software development, through description, assessment and prediction of software quality [13]. These models evaluate quality of software systems using defined metrics. Examples of metrics-based models are Maintainability Index(MI) that determines quantitative value of maintainability [14], Modularization Quality (MQ) that evaluates cluster based cohesion or software architecture [15]. However, there is rare study over the package based design of source code for maintenance objectives. In this section, we briefly present the motivational context of our research. To achieve this, we discuss domains of these QMs relevant to our paper.

### A. Static Analysis based Quality Assessment

Static analysis is carried out to analyze the source code using mainly using open source tools to discover security vulnerabilities. These vulnerabilities can cause critical system malfunction are economically harmful as well. Clearly, techniques that can reduce occurrence of bugs would be beneficial. To achieve this goal, static analysis tools have been designed to report early warnings and design anomalies. Although, their effectiveness is realized some settings, however, usefulness of warnings generated by them is still unclear. Recently, two static analysis tools FindBugs and PMD have been empirically reported to have less false positives [16] which are briefly introduced below:

*1) FindBugs:* FindBugs, developed by Hovemeyer *et al.*, is a tool that analyzes the java byte-code against various families of warnings characterizing common bugs in many system [17]. The main warnings provided by Findbugs are: null pointer de-reference, method not checking for null argument, close() invoked value that is always null. It actually checks the correctness, bad practice, malicious code vulnerability and performance, etc.

*2) PMD:* PMD is static analysis tool that finds defects, deadcode, duplicate code, sub-optimal code and overcomplicated expression, was first developed by Copeland *et al.* [18]. PMD operates over Java source code unlike FindBugs which analyzes byte-code. PMD statically warns many patterns, such as, jumbled incremented, return finally block, class cast exception, etc.

### B. Prediction based Quality Assessment

These models are usually based on source code metrics or defection detection to estimate number of systems faults, failures chances and maintenance effort. Mostly, software fault proneness prediction is taken as good example of these models. Other examples include Software Reliability growth (SRGM) and modeling of processes associated with software failures [19].

## III. Description of Studied Metrics

As a matter of best programming practices, software code should adhere to basic principle of high cohesion and low coupling. However, package optimization process should facilitate any structural change within current design of modularization, otherwise, subsequent decision of modification shall make software system vulnerability prone. Metrics proposed by Abdeen et al. provide an approach for automatic optimization of software modularization by minimizing the cyclic connections (direct cyclic-connectivity) among the packages. There are mainly two suites of proposed package measure, i.e., 1) for quality of single package; 2) evaluation of modularization quality on based cyclic/acyclic dependencies which are described in Tables I and II.

TABLE I. Description of Investigated Package Quality Metrics

| Metric | Definition |
|---|---|
| Package Cohesion | $CohesionQ(p) = \frac{\lvert p_{Int.D}\rvert}{\lvert p_D\rvert}$ |
| Package Coupling | $CouplingQ(p) = 1 - \frac{\lvert p_{Pro.P} \cup p_{Cli.P}\rvert}{\lvert p_D\rvert}$ |
| Package Cyclic Dependencies | $CyclicDQ(p) = 1 - \frac{\lvert p_{Cyc.D}\rvert}{\lvert p_D\rvert}$ |
| Package Cyclic Connections | $CyclicCQ(p) = 1 - \frac{\lvert p_{Cyc.Con}\rvert}{\lvert p_D\rvert}$ |

- *CohesionQ(p)*: Measures ratio of Internal Dependencies of package to all dependencies among and within a package.
- *CouplingQ(p)*: Measures ratio of package providers and clients to all dependencies among and within a package.
- *CylicDQ(p)* : Measures ratio of class cyclic dependencies within the package to all dependencies of package.

- *CylicCQ(p)* : Measures ratio package cyclic connections among the packages to all dependencies of package.

TABLE II. DESCRIPTION OF INVESTIGATED PACKAGE MODULARIZATION METRICS

| Metric | Definition |
|---|---|
| Inter-Package Dependencies | $IPD = \sum_{i=1}^{\lvert\mathcal{M}_p\rvert} \lvert p_{i_{E.xt.Out.D}} \rvert$<br>$ICD = \sum_{j=1}^{\lvert\mathcal{M}_c\rvert} \lvert c_{j_{Out.D}} \rvert$<br>$CCQ(\mathcal{M}) = 1 - \frac{IPD}{ICD}$ |
| Inter-Package Connections | $IPC = \sum_{i=1}^{\lvert\mathcal{M}_p\rvert} \lvert p_{i_{Out.Con}} \rvert$<br>$CRQ(\mathcal{M}) = 1 - \frac{IPC}{ICD}$ |
| Inter-Package Cyclic Dependencies | $IPCD = \sum_{i=1}^{\lvert\mathcal{M}_p\rvert} \lvert p_{i_{Out.Cyc.D}} \rvert$<br>$ADQM(\mathcal{M}) = 1 - \frac{IPCD}{ICD}$ |
| Inter-Package Cyclic Connections | $IPCC = \sum_{i=1}^{\lvert\mathcal{M}_p\rvert} \lvert p_{i_{Out.Cyc.Con}} \rvert$<br>$ACQM(\mathcal{M}) = 1 - \frac{IPCC}{ICD}$ |

- $CCQ(\mathcal{M})$: measures common closure of modularization for the classes that change together among the packages.
- $CRQ(\mathcal{M})$: measures common reuse of modularization for package that are are reused together.
- $ADQ(\mathcal{M})$: measures the extent of modularization to which cyclic dependencies between the classes are minimized.
- $ACQ(\mathcal{M})$: measures the extent of modularization to which cyclic connections among the package are minimized .

Table I summarizes formal definitions of above mentioned metrics. All theses metrics are based on relationships established among the software entities through direct/indirect and cyclic/acyclic dependencies established through classes and packages. Abdeen et al. has described utility of these metrics in addressing following optimization challenges in packages.

1) Inter-connections among the classes of large application create complex design and also increase inter-package connectivity.
2) Inadequate distribution of classes may result in highly complex afferent coupling on particular package, hence, violating the basic design rules of package, i.e, domain, size and coding practice.
3) Minimization of package dependencies may also degrade other packages.

Table II describes modularization metrics proposed by Abdeen *et al.* These modularization metrics were specifically formulated to address the prevalent modularization limitations which mainly focused on changing the structural shape of software from the scratch. Where as, metrics presented in Table II have goal of multi-objective optimization for improving existing package structure in accordance with well-known design principles. Thus, these metrics bear unique importance in automatic software re-modularization while respecting original design decisions.

Martin metrics suite has been defined in Table IV which have been already used as *Baseline* in many package level bug prediction studies [20], [21]. Table V presents the definitions of modularization metrics studied by Lee et al. [22]. The main objective of their research was to analyze and compare the various modularity metrics that have been studied in different domains. We set this research work as baseline to further investigate the applicability of package based modularization proposed by Abdeen *et al*. Below we summarize the definitions, notations used in definitions and interpretation of these metrics in particular context of study by Lee *et al.*. [22]. They have conducted an experimental evaluation of these metrics on evolutionary software and reported correlation of different modularity metrics and their sensitivities towards particular modular factors.

TABLE III. DESCRIPTION OF NOTATIONS USED TABLES I AND II

| Metric | Definition |
|---|---|
| $p_{Int.D}$ | Unique internal dependencies present within the package. |
| $p_{Pro.P}$ | Unique provider dependencies of package $p$. |
| $p_{Cli.P}$ | Unique client dependencies of package $p$. |
| $p_{Cyc.D}$ | Unique cyclic dependencies produced through classes of modularization. |
| $p_{Cyc.Con}$ | Unique cyclic dependencies produced using packages of modularization. |
| $p_{i_{E.xt.Out.D}}$ | Unique inter-package dependencies going out from package $p$. |
| $p_{i_{Out.Con}}$ | Unique inter-package external connections from package $p$. |
| $c_{j_{Out.D}}$ | Inter-class dependencies going outside the package $p$. |
| $p_{i_{Out.Cyc.D}}$ | out-going cyclic dependencies produced through classes from package $p$. |
| $p_{i_{Out.Cyc.Con}}$ | out-going cyclic dependencies produced through packages from package $p$. |

TABLE IV. DESCRIPTION OF MARTIN METRIC SUITE

| Metric | Definition |
|---|---|
| N | Class entities: The number of concrete, abstract classes and interfaces in the package. |
| Ca | Afferent Coupling: The number of other packages that depend upon classes within a package. |
| Ce | Efferent Coupling: The number of other packages that other class in a package depend upon. |
| A | Abstractness: The ratio of abstract classes in package to total number classes in a package. |
| I | Instability: The ratio of Efferent Coupling to total Coupling, $I = \frac{Ce}{(Ce+Ca)}$. |
| D | Distance: The distance from the main sequence: $D = \lvert A + I - 1 \rvert$. |

TABLE V. BASELINE MODULARIZATION METRICS STUDIED IN DIFFERENT DOMAINS

| Reference | Definition |
|---|---|
| Modularity and community structure in network[23] | $M_{newm} = \frac{1}{2m} \sum_i \sum_j \left( A_{ij} - \frac{k_i k_j}{2m} \right) \delta(g_i, g_j)$ |
| Modularity of software based on clustering[15] | $MQ = \sum_{i=1}^{k} \frac{2\mu_i}{k}, \quad M_{bunch} = \frac{MQ}{k}$ |
| Modularity of mechanical products. [24] | $M_{g\&g} = \frac{\sum_{k=1}^{M} \sum_{i=n_k}^{m_k} \sum_{j=n_k}^{m_k} R_{ij}}{(m_k - n_k + 1)^2} - \frac{\sum_{k=1}^{M} \sum_{i=n_k}^{m_k} \left( \sum_{j=1}^{n_k - 1} R_{ij} + \sum_{j=m_k+1}^{N} R_{ij} \right)}{(m_k - n_k + 1)(N - m_k + n_k - 1)}}{M}$ |
| Modularity based on dependency cost[25] | $M_{rcc} = 1 - \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{DependencyCost(i,j)}{N^{2\lambda}}$ |

- $\boldsymbol{M_{newm}}$: This metric is well known approach for quantifying modularity of social network represented in graphical structures. Recently, there has been extensive focus on application of this metric into studies pertaining different scientific domains, specially, social network, metabolic network, neural network and the World Wide Web. Computation of metric is based on theoretical heuristic that edges (links between nodes) within a module (community) are greater than expected ones. Further, in the definition, $i$ and $j$ are nodes, $A_{ij}$ represents edges between nodes. $m$ is the number of total edges and $k_i$ indicates expected number of edges in node $i$. $\delta$ is a comparator function that it outputs 1 where its two parameters are same, 0 otherwise. $g_i$, parameter of $\delta$, represents the module containing node $i$. This metric ranges between 1 as best value and 0 as worst value.
- $\boldsymbol{M_{bunch}}$: This metric is normalized version of clustering factor ($MQ$) introduced by Mancoridis textitet al.. [15]. $MQ$ is the most frequently used method for evaluation of a software modularity. $\mu_i$ is representation of

intra-edges of module $i$, while $\epsilon_{i,j}$ denotes inter-edges between modules $i$ and $j$ in total number of modules $k$.

- $\boldsymbol{M_{g\&g}}$: This metric was formulated to measure the modularity of complex mechanical products. However, their application in software systems can be interesting towards incorporating mechanical engineering principles and software design theories. Basically, metric quantifies modularity of physical entities using difference of inter and intra edge densities. In the definition, $M$ is number of modules and $N$ is number of mechanical components (total software nodes in our context of study). The numerator of the fraction consists of two part, the sum of intra-edge density of modules and the sum of inter-edge density of the modules. Symbols, i.e., $n_k$ and $m_k$ are the indexes of first node and last node respectively in module $k$. $R_{ij}$ denotes row and column in dependency between node $i$ and $j$ (software nodes).
- $\boldsymbol{M_{rcc}}$: This metric has its basic application in measuring the modularity of evolving software systems. Computing the Relative Clustered Cost of software systems is key idea of this metric. In perspective of software architectures, software systems having no dependencies shall bear the value 0 and 1 in case of all inter-dependent nodes. *DependencyCost* function returns a weighted dependency between node $i$ and $j$. $N$ is the number total nodes, $n$ is the size of module, $\lambda$ is a user defined parameter for the metric. The weight varies along with the dependency types. If a dependency between $i$ and $j$ is an intra-dependency of a single module, the weight is $n^{\lambda}$, where $n$ indicates the number of nodes in the module. On the other hand, if it is an inter-dependency between separate modules, the weight becomes $N^{\lambda}$ to have considerable penalty in terms of poor coupling.

## IV. Working Example

Consider an example of four packages connected through different dependencies as shown in Fig. 1. To simplify the working mechanism of Abdeen et al.'s metrics, we illustrate dependencies by package $p$ with other packages in modularized design.

In Fig. 1, package $p$ forms four kinds of dependencies with four other packages; Internal dependencies which is within the classes of package $p$, external dependencies which connect classes of package with other four packages, cyclic dependencies which goes out of classes of package $p$ and return to same classes of package $p$, cyclic connections which form cycle between package $p$ and other package in the modularization design.

It can be seen from Fig. 1, package $p$ contains five internal dependencies within classes $C1, C2, C3, C4$. Also, it can be visualized that there are 7 external incoming dependencies of package $p$ from the classes, i.e., $(C31, C32, C41, C23, C21, C52)$ and it contains five external outgoing dependencies towards the classes, i.e., *(C31, C41, C42, C21, C23, C52)*. Among external dependencies, four dependencies produce the cycle between package $p$ and other packages, i.e., *(C3, C31), (C3, C31, C32), (C4, C41), (C1, C52, C2)*. Similarly, these external dependencies create three cyclic connections $(p, p1), (p, p3), (p, p4)$. On the basis of definitions of Abdeen *et al.*'s metrics, Table VI presents metrics computation of modularization design of packages $p$ shown in Fig. 1.

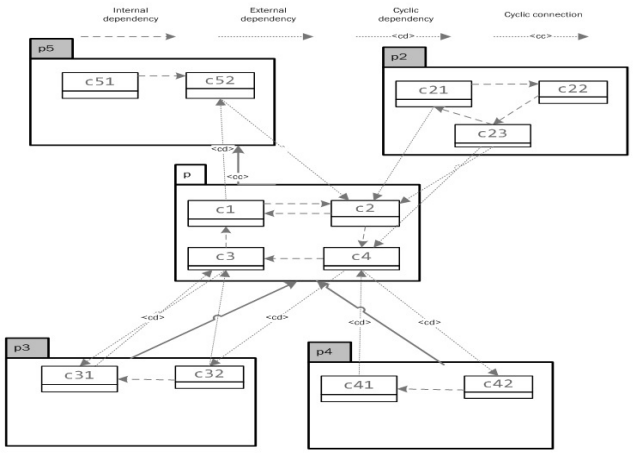



Fig. 1. Example of packages having inter and intra dependencies.

TABLE VI. Computation of Abdeen *et al.'s* Metrics

| Metric | Computation |
|---|---|
| $CohesionQ(p)$ | $\frac{\lvert C1+C2+C3+C4\rvert}{\lvert C1+C2+C3+C4+C31+C32+C41+C23+C21+C52+C42\rvert} = (4/11)$ |
| $CouplingQ(p)$ | $1 - \frac{\lvert (C52+C41+C31+C42)\cup(C52+C21+C23+C41+C32+C31)\rvert}{\lvert C1+C2+C3+C4+C31+C32+C41+C23+C21+C52+C42\rvert} = (7/11)$ |
| $CyclicDQ(p)$ | $1 - \frac{\lvert (C3+C31+C32+C1+C2+C52+C4+C41+C42)\rvert}{\lvert C1+C2+C3+C4+C31+C32+C41+C23+C21+C52+C42\rvert} = (9/11)$ |
| $CyclicCQ(p)$ | $= 1 - \frac{\lvert (P3+P5+P4)\rvert}{\lvert C1+C2+C3+C4+C31+C32+C41+C23+C21+C52+C42\rvert} = (3/11)$ |

## V. Empirical Study

In this section, we describe the methodology used to analyze the metrics on open source software systems. The analysis procedures for empirical evaluation involve descriptive structural information of subject systems, statistical analysis, linear correlation and logistic regression analysis. An overview of data processing steps are introduced in Fig. 2. The first step is to mine the source code of subject systems from repositories and archives. Second step is to apply three pronged static analysis of source code files through: 1) Understand [1], commercial static analysis tool for re-engineering and maintenance the software that provides metrics information of parsed code; 2) FindBugs [2] that identifies high priority warnings in the source code; 3) PMD [3] an extensive cross language static code analyzer that reports high priority coding rule violations. Step three is mapping of post-release faults from bug repositories, dependency based metrics and high priority source code violation warnings against package entities of corresponding source code file to compose the data-sets. In fourth step, data analysis techniques are used to build prediction models and determine the relationships among the modularization metrics. Finally, fifth step reports performance of models.

[1] https://scitools.com/
[2] http://findbugs.sourceforge.net/
[3] https://pmd.github.io/

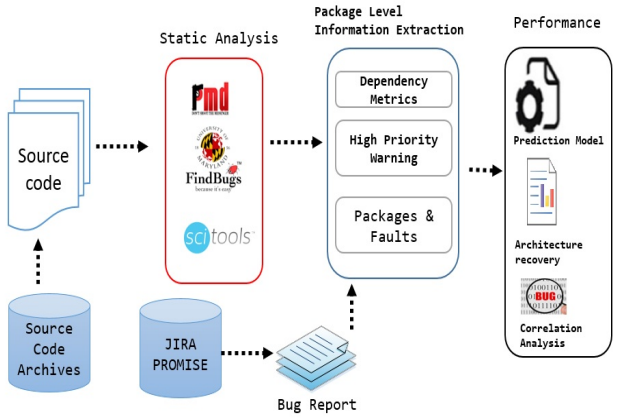



Fig. 2. Data processing mechanism.

### A. Research Objectives

In particular, our research is directed to explore following implications of inter and intra-package dependencies as salient objectives.

1) Cyclic dependencies among the packages are anti-patterns and may cause design flaws or make packages fault-prone.
2) Directed dependencies can provide composite view of package coupling and package cohesion which are yet evolving concepts for package entities of source code.
3) Automatic package optimization improve overall modularization quality of software.

Above mentioned challenges require evaluation of metrics through rigorous process of software quality. This can be achieved, if described metrics may succeed to provide complementary view of source code, consequently strengthening structural and functional validity of software design through proactive decision making. Another aspect of this study is to investigate relationship among Abdeen's modularization metrics solution based on package optimization and *Baseline* modularization approaches described in Table III. Our assessment criterion are formed on the basis of notion that proposed modularization can provide better illustration of *Baseline* approaches through different dimensions (e.g., Cyclic dependencies minimization, intra-package dependencies maximization), eventually, help developer's decisions to adhere with design principles during software development.

### B. Experimental Methodology

In this section, we provide a brief overview of statistical techniques and mechanism of their application in our study.

*1) Correlation Analysis:* The correlation analysis aims to determine relationship among variables. The correlation coefficient is measure of linear association between two variables. For this purpose, Spearman's rank correlation is widely performed over nonparametric nature of software metrics. The significance of correlation is tested at different levels of confidence interval 95%, i.e., $p - value < 0.05$.

*2) Multivariate Logistic Regression::* Logistic regression is standard statistical modeling technique in which the dependent variable can take on one of two different values: 0 and 1. A multivariate logistic regression model is based on the following relationship equation:

$$Pr\,(Y = 1|X_1, X_2, ...X_n) = \frac{e^{\alpha+\beta_1 X_1+\beta_2 X_2+...\beta_n X_n}}{1+e^{\alpha+\beta_1 X_1+\beta_2 X_2+...\beta_n X_n}}$$

Where, $X_1, X2..., X_n$ are independent variables, i.e., characteristics describing the source code(package level metrics), $Pr\,(Y = 1|X_1, X_2, ...X_n)$ represents the probability that the dependent variable $Y = 1$, i.e., the extent of package predicted as faulty.

*3) Classification:* Classification methodology is applied to predict weather a package is faulty or not. Various studies have set confusion matrix as benchmark to evaluate the performance of models and analyze the prediction capability of independent variables. From the confusion matrix, following two popular accuracy measures are computed to conduct the evaluation. All our prediction models output probabilities of fault-proneness of package entities. To classify a package as faulty, varying thresholds on probability are utilized. Thus, different choices of threshold will produce varying rates of *false* positives/negatives (FP/FN) and *true* positives/negatives (TP/TN).

- *Accuracy (Acc.)*: Measures the proportion of correct predictions. Accuracy is defined as: $Acc = \frac{TP+TN}{TP+TN+FP+FN}$.
- *Precision (Pr.)*: Measure of exactness, defines probabilities of true faulty packages to the number of package predicted as faulty. Precision is defined as $Pr = \frac{TP}{TP+FP}$.
- *Recall (Rec.)*: Measure of completeness, defines the probabilities of true faulty packages in comparison to total number of faulty packages. Recall is defined as $Rec = \frac{TP}{TP+FN}$.
- *F-measure (F1.)*: Measures harmonic mean of precision and recall of predicted model. $F1 = \frac{2*Pr.*Rec.}{Pr.+Rec.}$.

We build Logistic Regression (LR) model to predict the fault-proneness of Abdeen's metrics (*AbdeenMod*) and comparative analysis is carried out against the Martin's package level metrics (*RM*). Further, statistical association is used to discover the impact of dependencies over design anomalies (null pointer de-references, infinite recursive loops, bad uses of java libraries) and common programming flaws (unused variable, unnecessary object creation). The correlation analysis aims to determine significant relationship of each of metric described in Table I with quality attributes of source code. According to recent survey, logistic regression is most commonly used and productive technique for fault-prediction performance in software engineering [26].

### C. Data Sets

Basically, there are two different types of data-sets formed for experimental work for each category of metrics. We tend to follow the research based perspectives of taking into account subjects (software systems ) of varying nature, i.e., with large

TABLE VII. STRUCTURAL INFORMATION OF SUBJECT SYSTEMS

| System | Versions | Number of Packages | Total Number of faults | Faulty Packages | Percentage of Faulty packages | FB-Warnings Priority-1 | PMD-Warnings Priority-1 |
|---|---|---|---|---|---|---|---|
| Eclipse | 2.0 | 377 | 917 | 190 | 50% | 551 | 422 |
| Eclipse | 2.1 | 428 | 657 | 193 | 45% | 720 | 5365 |
| Eclipse | 3.0 | 642 | 1523 | 307 | 47% | 356 | 7002 |
| POI | 2.5 | 74 | 107 | 15 | 20% | 96 | 2601 |
| POI | 3.0 | 80 | 316 | 30 | 38% | 175 | 3743 |
| Camel | 1.6 | 210 | 144 | 23 | 62% | 27 | 148 |
| JDTCore | 3.4 | 46 | 47 | 32 | 47% | 159 | 931 |
| Lucene | 2.4 | 42 | 318 | 34 | 80% | 82 | 879 |
| jEdit | 4.2 | 28 | 14 | 12 | 42% | 115 | 363 |
| jEdit | 4.3 | 37 | 19 | 6 | 16% | 109 | 485 |

and small size of packages, with diverse domain of application and already utilized in research literature of software quality. For package quality metrics, seven releases of 3 different open source software systems and four other open source software releases were considered. Eclipse[4]: An Integrated Development Environment (IDE) for software development in collaborative working groups. jEdit[5]: A mature programmer's text editor, written in java and an extensible plug-in architecture. POI [6] a powerful tool to read and write MS Excel files in Java. Lucene [7] provides java based indexing and search technology. JDT-Core[8] is an infrastructure of eclipse IDE. In total 10 data-sets are formed. Each data-set comprises of 6 traditional package-level metric described by R.C Martin [2],4 package quality metrics defined by Abdeen et al. [10] for fault-proneness of packages, source code bug warnings of priority 1 identified by FindBugs and code rules violation warnings of priority 1 reported by PMD.

It can be well inferred that meaningful statistical conclusions can be drawn, as data-sets encompass diverse domains of architectural composition. Post-release fault data of subjects was obtained from public repositories, i.e., Eclipse Bug Data [9] and PROMISE[10]. For package modularization metrics, experiment study consists of 23 versions of two different open source software systems. JHotDraw [11]: is a Java GUI framework for technical and structured Graphics. Ant[12]: is a Java library and command-line tool whose mission is to drive processes described in build files. These data-sets have been already studied in comparative modularity analysis by Lee et al. [22]. Therefore, we utilized same data for 4 well-known clustering based modularity metrics in our study part of modularity analysis. However, all the *AbdeenMod* metrics (package quality and package modularization) were computed by our own scripts developed through Understand-Perl API[13] with utmost reliance and incremental testing (Made available public ally at site[14]).

Table VII provides a summarized description of the data-sets in our experimental study. It mainly represents system name with release version, number of total packages, total Number of faults, number of highest priority design flaws detected through FindBugs[15] and source code violations having highest priority identified through PMD [16] in corresponding software. Both of these tools are extensively used in source code analysis with recognition in software industry and research pertaining to software quality [27].

[4]https://eclipse.org/
[5]http://www.jedit.org/
[6]https://poi.apache.org/
[7]https://lucene.apache.org/
[8]https://eclipse.org/jdt/core/
[9]https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/
[10]http://openscience.us/repo/issues/bugfiles.html
[11]http://www.jhotdraw.org/
[12]http://ant.apache.org/
[13]https://scitools.com/feature/api/
[14]https://github.com/Analyzer2210cau/Cyc-Deps-Maintainence-Objectives
[15]http://findbugs.sourceforge.net/

TABLE VIII. INTRA-RELEASE PREDICTION MODELS: COMPARISON

| **System** | *RM* | | *AbdeenMod+RM* | | Improved |
|---|---|---|---|---|---|
| | **Acc.** (mean) | **F1.** (mean) | **Acc.** (mean) | **F1.** (mean) | **%age** |
| Camel-1.6 | 0.88 | 0.40 | 0.90✓ | 0.50 | 1.1% |
| Eclipse-2.0 | 0.60 | 0.46 | 0.68✓ | 0.64 | 13.3% |
| Eclipse-2.1 | 0.59 | 0.38 | 0.62✓ | 0.40 | 5% |
| Eclipse-3.0 | 0.64 | 0.50 | 0.66✓ | 0.54 | 3% |
| JDTCore-3.4 | 0.77 | 0.84 | 0.76 | 0.77 | – |
| Lucene-2.4 | 0.83 | 0.56 | 0.84 | 0.53 | –% |
| POI-2.5 | 0.67 | 0.65 | 0.62 | 0.42 | -7% |
| POI-3.0 | 0.75 | 0.5 | 0.77✓ | 0.6 | 2.6% |
| jEdit-4.2 | 0.68 | 0.77 | 0.78✓ | 0.83 | 14.7% |
| jEdit-4.3 | 0.82 | 0.4 | 0.80 | 0.3 | –% |

TABLE IX. INTER-RELEASE PREDICTION MODELS: COMPARISON

| **System** | *RM* (Acc) | *AbdeenMod+RM* (Acc) | %Improvoed |
|---|---|---|---|
| Eclipse-2.0(Train), Eclipse-2.1(Test) | 0.65 | 0.67✓ | 3% |
| POI-2.5(Train), POI-3.0(Test) | 0.57 | 0.75✓ | 31% |
| jEdit-4.2(Train), jEdit-4.3(Test) | 0.60 | 0.59 | -2% |

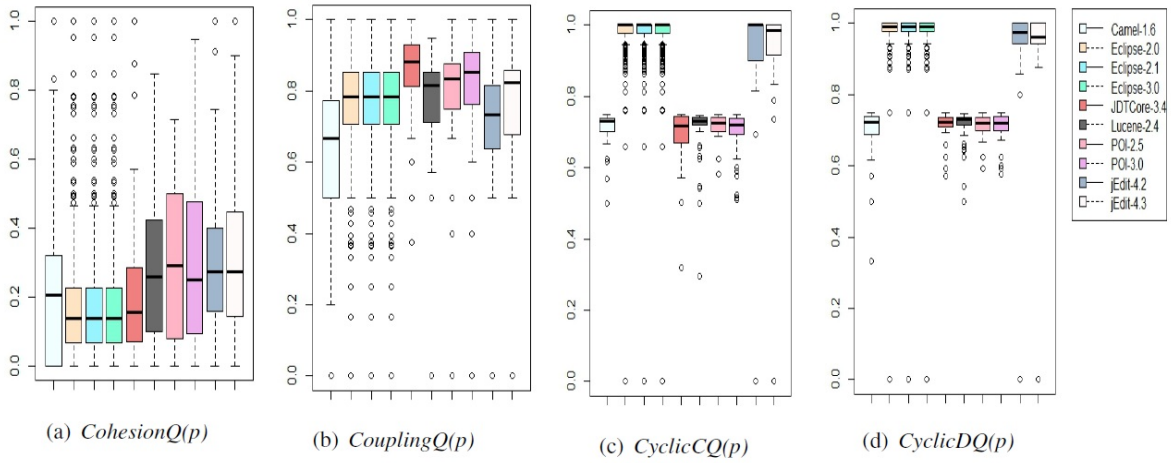
### D. Experimental Results

This section briefly illustrates experimental results and analysis on both categories of metrics. In our empirical evaluation, our focus is to obtain maximum possible statistical significance for the research objectives defined in earlier section.

*1) Package Quality Metrics:* Fig. 3 depict the box-plot to describe the distribution of four package-quality metrics in each data-set. Box-plot range distribution of *CohesionQ* and *CouplingQ* lies within 25 to 75 percentile as shown in Fig. 3. It can be observed that *CohesionQ* metric has low median ($value \approx 0.15$) for three versions of eclipse, however, it is almost in even distribution for all versions of jEdit ($value \approx 0.3$). On the contrary, *CouplingQ* values seem to be distributed with quite high median ($value \approx 0.75$) for all the versions of data-sets. It depicts the fact that high coupling and low cohesion trend is found in almost all data-sets of our study, employing potential threat of packages being faulty quite high. Similarly, values of *CyclicCQ* and *CyclicDQ* are relatively with very high median ($value \approx 0.95$) compared to other two metrics. Whereas, distribution of *CylicCQ* and *CyclicDQ* range within 25 to 75 percentile as shown in Fig. 3, showing presence of high inter-package cyclic dependencies and connections in all the data-sets.

Table VIII summarizes detailed information of fault-prediction model built on the basis of intra-release test and train data-sets. Results obtained from 10 times 10-fold cross-validation for two models *RM* and *AbdeenMod+RM* using LR are presented in Table VIII. It can be clearly observed that there is substantial accuracy improvement while classifying the *AbdeenMod+RM* model against *RM*(Baseline) model in most of cases (indicated with ✓). From Table VIII, we make following observations. First, *AbdeenMod+RM* models for data-set with higher number of packages have easily

[16]https://pmd.github.io/




(a) *CohesionQ(p)* (b) *CouplingQ(p)* (c) *CyclicCQ(p)* (d) *CyclicDQ(p)*

Fig. 3. Box plot representation of metrics for all data-sets.

outperform traditional approaches of fault-prediction which is case with three versions of Eclipse (2.0, 2.1, 3.0) and Camel-1.6. Second, if data-set contains small number of packages then prediction may result adversely which is the case with jEdit-4.3 and JDTCore-3.4 where accuracy is not improved with *AbdeenMod+RM* model. Third, *AbdeenMod+RM* model's highest accuracy improvement was reported as 14.7%, making the application of cyclic dependency metrics quite significant for fault-proneness prediction. F1-score is another dimension of performance measure that takes int account both precision and recall. Mean accuracy of successful models in Table VIII has been seen in range of $0.83 \approx 0.40$ (indicated with ✓). Despite the accuracy improvement in some of cases, F1-score is still unsatisfactory in some cases, e.g., Lucene-2.4, implying lesser precision of *AbdeenMod+RM*. Generally, prediction result has been improved by 13.3% as maximum value in Eclipse (Larger data-set) and 14.7% as maximum value in jEdit (Relatively smaller data-set).

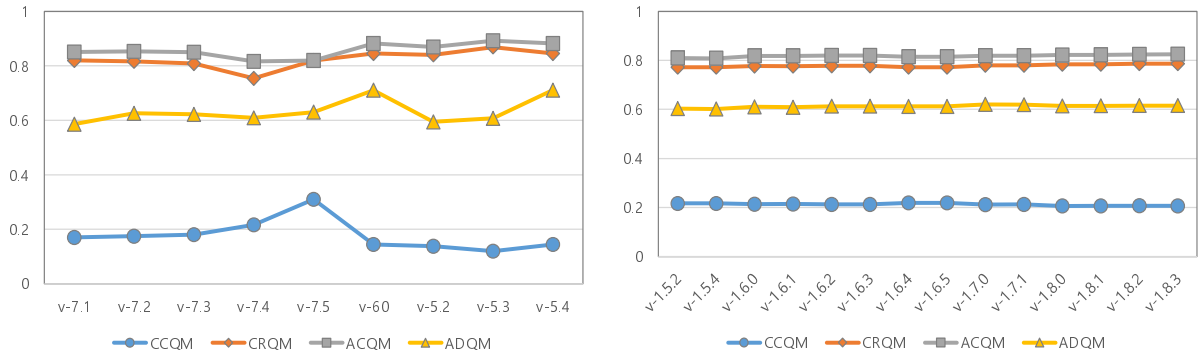
Table IX presents summary of fault-prediction model using inter-release frame which is more rigorous approach of developing train and test data-sets. Table IX mainly shows values of accuracy measure for fault-prediction model across releases of Eclipse, POI and jEdit. Clearly, prediction results with *AbdeenMod+RM* achieved competitive accuracy in two cases (indicated with ✓) as shown in Table IX and is recorded with 31% of maximum improvement. Table X shows analysis carried out over the linear correlation of each Abdeen's Package Quality metrics with number of post-release faults, coding violation and anomalies warnings of priority-1 using FindBugs and source code vulnerabilities arnings of priority-1 using PMD in corresponding package for each dataset. Significant magnitude of association is indicated with ($^*$) evaluated at $p - value < 0.05$ as thresh-hold. Findings of Table X can lead following inferences: First, *Cohesion* metrics exhibits significantly negative correlation as observed in most of the cases except Lucene and JDTCore. Second, statistical significance of all the package quality metrics was observed frequent in data-sets with larger number of packages, e.g., (Eclipse, POI, Camel). Third, coupling metric *CouplingQ* exhibits statistically significant magnitude of correlation with post-release faults in most of the data-sets. Fourth, statistical significant correlation of *CyclicDQ(p)* and *CyclicDQ(p)* metrics is not witnessed frequently with the exception of Eclipse-2.0. Another objective was to determine the relationship of all *AbdeenMod* metrics with design flaws and vulnerabilities detected *PMD* or *FindBugs*. Interestingly all the *AbdeenMod* metrics show the positive or negative association, but, their statistical significance is rarely observed in all cases. However, *Cohesion* and *CouplingQ* metrics have managed to develop correlation significance to considerable extent with data-sets, like (Eclipse, POI, jEdit), adding an evidence to soundness of study. On the contrary *CyclicCQ* and *CyclicDQ(p)* are not found correlated with with FB and PMD. Such findings lead to implication that design anomalies detected through open source tools like *FindBugs*and *PMD* have relatively substantial influence over package quality metrics, however, lack impact in comparison to actual post release faults.

*2) Modularization Metrics:* In Table XI, degree of modularization correlation between Abdeen's modularization metrics solutions and *Baseline* modularization metrics is presented. Values inside the visualization table show the magnitude of association in corresponding cell of metric at the significant level denoted as: $p - values(0.001, 0.01, 0.05, 0.1) \Leftrightarrow$ symbols("***", "**", "*", "."). For further illustration, Fig. 4(a) and (b) show overall correlation among the modularity metrics through visualization table representation for JhotDraw and Ant data-sets respectively. Distribution of each variable is shown on the diagonal with bi-variate scatter plot. As described earlier, impact of cyclic or direct dependencies is relatively considered as an anti-pattern, hence, negative correlation can be expected in our analysis.

There are some unique implications which can be formed from modularity analysis of each data-set. For JHotDraw, $CCQ(\mathcal{M})$ $ADQ(\mathcal{M})$ and $CRQ(\mathcal{M})$ exhibit complementary view of strong association with modularity metrics except $\mathbf{M}_{g\&g}$. On the contrary, $ACQ(\mathcal{M})$ is found with less association in

TABLE X. MAGNITUDE OF CORRELATION OF PACKAGE QUALITY METRICS IN ECLIPSE

| Metric | *CohesionQ(p)* | | | *CouplingQ(p)* | | | *CyclicCQ(p)* | | | *CyclicDQ(p)* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Faults | FB | PMD | Faults | FB | PMD | Faults | FB | PMD | Faults | FB | PMD |
| Eclipse-2.0 | **-0.03** | **0.27**$^{*}$ | **0.14**$^{**}$ | **0.18**$^{*}$ | **-0.39**$^{*}$ | 0.23 | **-0.25**$^{*}$ | -0.038 | 0.47 | **-0.25**$^{*}$ | -0.038 | 0.47 |
| Eclipse-2.1 | –0.03 | 0.03 | **0.14**$^{**}$ | **0.18**$^{***}$ | **0.15**$^{**}$ | **0.24**$^{***}$ | 0.06 | 0.045 | 0.04 | 0.076 | 0.052 | 0.068 |
| Eclipse-3.0 | -0.024 | **0.14**$^{***}$ | 0.015 | **0.22**$^{***}$ | **0.17**$^{***}$ | **0.14**$^{***}$ | 0.06 | 0.05 | 0.03 | **0.089**$^{*}$ | 0.063 | 0.052 |
| JDTCore-2.4 | -0.012 | - 0.016 | -0.15 | **0.30**$^{*}$ | **0.33**$^{*}$ | 0.083 | -0.20 | -0.13 | 0.024 | 0.17 | 0.20 | 0.081 |
| Lucene-3.0 | 0.034 | - 0.049 | 0.000092 | 0.17$^{*}$ | -0.11 | 0.083 | 0.047 | -0.044 | 0.0016 | 0.092$^{*}$ | -0.094 | 0.083 |
| POI-2.5 | **0.33**$^{*}$ | **0.35**$^{*}$ | **0.34**$^{*}$ | 0.19 | 0.21 | 0.23 | **0.36**$^{*}$ | 0.074 | 0.098 | 0.095 | 0.011 | 0.24 |
| POI-3.0 | 0.20 | **0.31**$^{*}$ | 0.20 | 0.20 | **0.23**$^{*}$ | **0.27**$^{**}$ | 0.053 | 0.067 | 0.12 | 0.087 | 0.11 | 0.14 |
| Camel-1.6 | **0.43**$^{**}$ | **0.23**$^{*}$ | 0.33 | **0.39**$^{**}$ | **0.41**$^{**}$ | 0.058 | 0.0550.036 | 0.052 | 0.054 | 0.04 | 0.059 | 0.13 |
| JEdit-2.0 | **-0.23**$^{*}$ | **0.49**$^{*}$ | 0.33 | **0.34**$^{*}$ | **0.48**$^{*}$ | **0.40**$^{*}$ | -0.020 | 0.047 | -0.12 | 0.086 | 0.19 | 0.12 |
| JEdit-2.5 | **-0.25**$^{*}$ | 0.14 | 0.095 | **0.29**$^{*}$ | **0.35**$^{*}$ | **0.31**$^{*}$ | 0.026 | 0.074 | 0.028 | 0.095 | 0.16 | 0.12 |




(a) *JHotDraw* (b) *Apache-Ant*

Fig. 4. Box-plot representation of Cohesion and Coupling metrics in each data-sets.

TABLE XI. CORRELATION COEFFICIENT OF MODULARITY METRICS

| Project | | $\mathbf{M}_{newm}$ | $\mathbf{M}_{bunch}$ | $\mathbf{M}_{g\&g}$ | $\mathbf{M}_{rcc}$ |
|---|---|---|---|---|---|
| *JHotDraw(9)* | $CCQ(\mathcal{M})$ | **0.77**$^{*}$ | **0.64**$^{\cdot}$ | 0.33 | **0.72**$^{*}$ |
| | $CRQ(\mathcal{M})$ | **-0.80**$^{**}$ | **-0.68**$^{*}$ | -0.25 | **0.69**$^{*}$ |
| | $ADQ(\mathcal{M})$ | **-0.89**$^{**}$ | **-0.69**$^{*}$ | -0.19 | **0.76**$^{*}$ |
| | $ACQ(\mathcal{M})$ | -0.25 | 0.14 | **0.67**$^{*}$ | – |
| *Apache-ant(14)* | $CCQ(\mathcal{M})$ | -0.08 | **-0.78**$^{***}$ | **-0.79**$^{***}$ | **-0.74**$^{**}$ |
| | $CRQ(\mathcal{M})$ | 0.18 | **0.84**$^{***}$ | **0.86**$^{***}$ | **0.84**$^{***}$ |
| | $ADQ(\mathcal{M})$ | 0.38 | **0.70**$^{**}$ | **0.74**$^{**}$ | **0.87**$^{***}$ |
| | $ACQ(\mathcal{M})$ | 0.21 | **0.74**$^{**}$ | **0.79**$^{***}$ | **0.89**$^{***}$ |

terms of statistical significance as shown in Table XI. These indications are evident for the presence of less cyclic dependencies and cohesiveness of packages. On the other hand, for Ant data-set, $CCQ(\mathcal{M})$ and $CRQ(\mathcal{M})$ are seen to have strong negative and positive correlation with our baseline modularity metrics respectively. Whereas $\mathbf{M}_{newm}$ is an exception to reveal any prominent significance. However, $ACQ(\mathcal{M})$ and $ADQ(\mathcal{M})$ exhibit strong positive correlation with our bench mark modularity metrics as shown in Table XI. Another perspective of this dependence among multiple variables reveal that all the Abdeen's modularization are significantly related with each other as shown in Fig. 4(a) and (b). More importantly, $CCQ(\mathcal{M})$, $CRQ(\mathcal{M})$ and $ADQ(\mathcal{M})$ depict a strong positive correlation with $M_{bunch}$ which is considered as most efficient modularity measure in software quality evaluation.

## VI. RELATED WORK

Some research studies have used package level metrics in evaluation of software quality and bug prediction. D'Ambros proposed coupling based technique for package understandability and their evolution [28]. Wilhelm et al. proposed package dependencies management and control using Martin and size metrics. Additionally, Reibing utilized these metrics to build Object-Oriented Design Model (ODEM) for formalization of design metrics [29]. It is worth mentioning that Abdeen *et al.* continued their effort to propose metrics for improving modularization, but, scope of their study was for entire modularization not for a single package [30]. However, exploring the relationship between package level metrics and external quality attributes is yet an interesting research subject. Gupta et al. presented empirical evaluation for package coupling as indicator of its understandability and maintenance [31]. Elish explored utility of Martin metrics as determinants of package understandability. He determined that almost all the Martin metrics have significant correlation with effort required to understand the package design [32]. Zimerman et al. collected fault data and complexity metrics for Eclipse releases 2.0, 2.1 and 3.0 at package level [33]. They successfully derived fault-proneness of packages by constructing logistic and linear regression models.

As a matter of observation, size and complexity metrics can not produce enough information on fault-prediction. There are certainly opportunities to explore other structural properties of packages to achieve an improved prediction accuracy. Taking this direction, Elish developed comparative inter and intra-release prediction models using CK, MOOD and Martin metrics suite [34]. Another notable study in recent time is by Zhao et al. [21], who has further endorsed the utility of package level metrics. He presented an empirical analysis for package modularization proposed by Sarkar et al. [35] metrics as having significant association with fault-proneness. All these efforts form the motivation of our study

to further investigate dependency based structural properties of packages. In particular, study extends the evidence for role f package based design in improvement of fault-prediction and effectiveness in describing software quality attributes [36], [37].

## VII. Discussion

Abdeen *et al.* extended their research work on the basis of package entities in proceeding years. Admittedly, each of their research effort was unique in terms of application and theoretical rationale. Similarly, modularization metrics described in Table V were studied for correlation among themselves, which required further exploration too. Therefore, our work differs in the context of application to determine modularity of Abdeen *et al.* in relation with these metrics. Following features makes our study distinctive and useful for research community.

- Adding an evidence of relationship between code metrics and design rules by static analysis tools.
- Empirical evaluation of package entities as effective de-bugging components.
- Providing the rationale that maintenance is multi-objective phase using techniques of modularity enhancement, fault-proneness prediction and source code design improvement.

Aforementioned aspects our this research study basically set novelty and technical significance for assuring quality software design and functioning. This study can help software quality engineers to prioritize their tasks.

## VIII. Threats to Validity

In empirical software engineering research, use of open source and limited software projects often account for external threats to validity. However, software systems used in our study are state-of-the-art in research literature of software quality bearing an abstract representation of software structure. Since, dependency information extracted in our study can be characteristics of any industrial based operational software, the reported results can be helpful in forming a generalized opinion to reasonable extent. Although, use of more industrial based software systems can mitigate the threat of external validity. Another aspect of computation which is likely to create threats to construct and external validity, is use of *FindBugs* and *PMD*. Unfortunately, the performance of these tools are dependent on particular experimental environmental setup and system configuration. Therefore, false positives in static analysis results can't be ruled out. Similarly, measurement of metrics and their experiment through use of tools for analysis and programming have their specific limitation like, language dynamics and precision of statistical computation. However, results were acquired with maximum possible programming reliance and analysis confidence to avoid any internal threats.

## IX. Conclusion

In this paper, we evaluated the impact of dependency based Package-Quality metrics towards software quality assurance. Our study validates the computation of these metrics on open-source software. Inter-release and intra-release of predictive models were constructed to conclude following: 1) Dependency based Package Quality metrics in combination with traditional package level metrics provide complementary view of fault-proneness of packages. It employs proper management and organization of packages by minimizing the dependencies can enhance the quality and ensure functional stability of software; 2) Comprehensive assessment of post-releases of software can be obtained with dependency based Package Quality metrics to avoid future potential bugs; 3) Dependency based Cohesion and Coupling metrics value improvement can help to keep the software systems up to coding standards and design rules.

The results indicate that prediction models developed with *AbdeenMod+RM* metrics achieved reasonable accuracy against *RM* metrics and traditional metrics are outperformed in fault-prediction modeling. Thus, recommendation of collecting dependency information can be significant towards software maintenance. In addition to obtained improved prediction accuracy compared to traditional model, *AbdeenMod* metrics show notable correlation with number of faults in the packages. These results also indicate that dependencies of source code at package level should be properly managed and resolved to depict the reliable software design. We also investigated the numerical relationship between Abdeen's modularization metrics and widely used modularization metrics. Our statistical computation through correlation matrix indicates that these metrics can be applied in quality assurance framework of object oriented software systems. This study has made vital contribution and provided an insights into early prediction of faults in packages through empirical evidences and statistical validations. In future, we aim to derive more metrics and explore the their relationships in other software quality attributes. In addition to this, comparative analysis using effort aware fault-prediction models is research domain to be explored in future.

## Acknowledgment

The authors would like to thank Prof. Chan-Gun Lee, Director, RTSE-Lab, Chung-ang University, Seoul, Korea and Dr. Kiseong Lee, Post-doctoral researcher, RTSE-Lab, Chung-ang University, Seoul, Korea for sharing the data of their research work.

## References

[1] D. F. D'souza and A. C. Wills, Objects, components, and frameworks with UML: the catalysis approach. Addison-Wesley Longman Publishing Co., Inc., 1998.

[2] R. C. Martin, "Design principles and design patterns," Object Mentor, vol. 1, pp. 1–34, 2000.

[3] S. Ducasse, M. Lanza, and L. Ponisio, "Butterflies: A visual approach to characterize packages," in Software Metrics, 2005. 11th IEEE International Symposium. IEEE, 2005, pp. 10–pp.

[4] R. S. Pressman, Software engineering: a practitioner's approach. Palgrave Macmillan, 2005.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," Software Engineering, IEEE Transactions on, vol. 27, no. 1, pp. 1–12, 2001.

[6] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 2, no. 3, pp. 228–269, 1993.

[7] B. S. Mitchell and S. Mancoridis, "On the evaluation of the bunch search-based software modularization algorithm," Soft Computing, vol. 12, no. 1, pp. 77–93, 2008.

[8] V. Tzerpos and R. C. Holt, "The orphan adoption problem in architecture maintenance," in Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on. IEEE, 1997, pp. 76–82.

[9] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-based improvement of subsystem decompositions," in Proceedings of the 7th annual conference on Genetic and evolutionary computation. ACM, 2005, pp. 1045–1051.

[10] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in Reverse Engineering, 2009. WCRE'09. 16th Working Conference on. IEEE, 2009, pp. 103–112.

[11] S. Mohsin and K. Zeeshan, "Program slicing based software metrics towards code restructuring," in Proc. 2nd International Conference onComputer Research and Development. IEEE, 2010, pp. 738–741.

[12] M. Saleem, R. Hussain, V. Ismail, and S. Mohsin, "Cost effective software engineering using program slicing techniques," in Proc. 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human. ACM, 2009, pp. 768–772.

[13] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements," in Software Quality, 2009. WOSQ'09. ICSE Workshop on. IEEE, 2009, pp. 9–14.

[14] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in Software Maintenance, 1992. Proceerdings., Conference on. IEEE, 1992, pp. 337–344.

[15] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. IEEE, 1999, pp. 50–59.

[16] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu et al., "To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools," Automated Software Engineering, vol. 22, no. 4, pp. 561–602, 2015.

[17] D. Hovemeyer and W. Pugh, "Finding bugs is easy," ACM Sigplan Notices, vol. 39, no. 12, pp. 92–106, 2004.

[18] T. Copeland, "Pmd applied," 2005.

[19] K. A. Mir, "A software reliability growth model," Journal of Modern Mathematics and Statistics, vol. 5, no. 1, pp. 13–16, 2011.

[20] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010, pp. 1–10.

[21] Y. Zhao, Y. Yang, H. Lu, Y. Zhou, Q. Song, and B. Xu, "An empirical analysis of package-modularization metrics: Implications for software fault-proneness," Information and Software Technology, vol. 57, pp. 186–203, 2015.

[22] K.-S. Lee and C.-G. Lee, "Comparative analysis of modularity metrics for evaluating evolutionary software," IEICE TRANSACTIONS on Information and Systems, vol. 98, no. 2, pp. 439–443, 2015.

[23] M. E. Newman, "Modularity and community structure in networks," Proceedings of the National Academy of Sciences, vol. 103, no. 23, pp. 8577–8582, 2006.

[24] F. Guo and J. K. Gershenson, "A comparison of modular product design methods based on improvement and iteration," in ASME 2004 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. American Society of Mechanical Engineers, 2004, pp. 261–269.

[25] 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, vol. 52, no. 7, pp. 1015–1030, 2006.

[26] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," Software Engineering, IEEE Transactions on, vol. 34, no. 4, pp. 485–496, 2008.

[27] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for java," in Software Testing, Verification, and Validation, 2008 1st International Conference on. IEEE, 2008, pp. 248–257.

[28] M. D. Ambros and M. Lanza, "Reverse engineering with logical coupling," in Reverse Engineering, 2006. WCRE'06. 13th Working Conference on. IEEE, 2006, pp. 189–198.

[29] R. Reißing, "Towards a model for object-oriented design measurement," in 5th International ECOOP workshop on quantitative approaches in object-oriented software engineering, 2001, pp. 71–84.

[30] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in Reverse Engineering (WCRE), 2013 20th Working Conference on. IEEE, 2013, pp. 212–221.

[31] V. Gupta and J. K. Chhabra, "Package coupling measurement in object-oriented software," Journal of computer science and technology, vol. 24, no. 2, pp. 273–283, 2009.

[32] M. O. Elish, "Exploring the relationships between design metrics and package understandability: A case study," in Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. IEEE, 2010, pp. 144–147.

[33] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," Empirical Software Engineering, vol. 17, no. 4-5, pp. 531–577, 2012.

[34] M. O. Elish, A. H. Al-Yafei, and M. Al-Mulhem, "Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of eclipse," Advances in Engineering Software, vol. 42, no. 10, pp. 852–859, 2011.

[35] S. Sarkar, A. C. Kak, and G. M. Rama, "Metrics for measuring the quality of modularization of large-scale object-oriented software," Software Engineering, IEEE Transactions on, vol. 34, no. 5, pp. 700–720, 2008.

[36] M. Shaikh, K.-S. Lee, and C.-G. Lee, "Assessing the bug-prediction with re-usability based package organization for object oriented software systems," IEICE TRANSACTIONS on Information and Systems, vol. 100, no. 1, pp. 107–117, 2017.

[37] M. Shaikh and C.-G. Lee, "Aspect oriented re-engineering of legacy software using cross-cutting concern characterization and significant code smells detection," International Journal of Software Engineering and Knowledge Engineering, vol. 26, no. 03, pp. 513–536, 2016.