Assessing Architectural Sustainability during Software Evolution using Package-Modularization Metrics

Mohsin Shaikh[1*], Dilshod Ibarhimov[2], Baqir Zardari[3]
Westminster International University, Tashkent, Uzbekistan[1,2]
Quaid-e-Awam University of Engineering Science and Technology, Nawabshah, Pakistan[1,3]

***Abstract*—Sustainability of software architectures is largely dependent on cost-effective evolution and modular architecture. Careful modularization, characterizing proper design of complex system is cognitive and challenging task for insuring improved sustainability. Moreover, failure to modularize the software systems during its evolution phases often results in requiring extra effort towards managing design deterioration and solving unforeseen inter-dependencies. In this paper, we present an empirical perspective of package-level modularization metrics proposed by Sarkar, Kak and Rama to characterize modularization quality through packages. In particular, we explore impact of these design based modularization metrics on other well known modularity metrics and software quality metrics. Our experimental examination over open source java software systems illustrates that package-level modularization metrics significantly correlate with architectural sustainability measures and quality metrics of software systems.**

***Keywords*—Software architecture; software modularity; software quality; packages**

I. Introduction

In recent times, conventional conjectures of experimental research and theory have been joined by computational and data-intensive methodologies [1], [2]. These new research mechanisms are driven by software systems that are maintained to avoid complex functional hindrances and operated in distributed e-infrastructure. Integrating the hardware and software components is receiving increasing attention for development of sustainable computational systems. This leads us to understand concept of sustainability which accord with perspective of "capable of being endured or maintained". Secord *et al.* defines software sustainability related to development activities aimed to evolve and modify with changing requirements [3]. However, they also argue that sustainability is influenced by many other factors including the organization, developers, end-users, architecture and design documentation.

Software architecture provides abstract picture of fine-grained development details [4]. Architecture of software systems reflects implementation of major design decisions and their governed methodology. Software architectures postulate division of software into subsystems, components and other functional parts. Software architectures bear an influential importance in evaluating sustainable growth of software (i.e, cost-effectiveness and long lasting) and reliability [5], [6]. Additionally, software architectures metrics help developers to determine maintenance objectives, testing effort and overall design decisions. Typically, during life span of software, it undergoes many corrective and adaptive changes. Evolution of software process takes place through continuous addition, modification and re-organization of source code entities. As these activities are reflected into source code, there can be possible deterioration in software design and its architecture.

In this context, evaluating the sustainability of software architecture for insuring proper maintenance and evolution cost control becomes quite desirable. Architectural sustainability is influenced by many aspects that includes design decisions, evolutionary changes in the software, change-prone requirement and modularization practices. Thus, architecture-level metrics are required to quantify technical sustainability facets of software systems. However, setting a single metric for expressing software sustainability is difficult due to different complications involved in software development life cycle, such as, irrelevant requirement engineering, diverse technology choices, re-factoring of source code and implicit knowledge of software architect [7]. Sustainable architectures are mainly dependent on feasible software deign to insure compatibility with changing requirement.

Software modularization, object-oriented (OO) decomposition in particular, is an approach to ease the development and maintenance. In order to understand the OO software, flexible design with well-connected constituent components is highly demanded for accommodating future changes and requirements. Often, software maintenance costs are higher than its overall development budget [8]. Modularization essentially follows *divide and conquer* strategy for managing the complexity of large source code. Parnas *et al.* introduced concept of information hiding, which became a fundamental paradigm for modularizing the OO systems [9]. There has been significant advancement to reverse engineer the software systems for automatic extraction of its design depicting an aggregate view. Some of notable techniques in this regard are related to partitioning of software systems into subsystems(clusters) and recovering the architecture into module-view [10]. With increasing focus of building tools and methodologies for software maintenance, there has been significant research over mechanism of partitioning the software into subsystems taking into account source code abstractions like classes and packages [11]. In particular, package organization provides higher abstraction and easier way for comprehension, complexity reduction and understanding of software systems. However, due to frequent changes into software, decreasing

modularization quality is not an impossible occurrence.

Architectural sustainability can be obtained with implementation of best modularization practices during software development. Some of notable proposed practices include acyclic dependencies, layering organization,testability, encapsulation and concern dispersion [12], [11]. Sarkar *et al.* have proposed a new modularization metrics suite based on packages as its functional components [13]. They have further devised these modularization metrics into three categories, i.e., based on inheritance or association, method invocation and best programming practices. It is worth mentioning that Sarkar *et al.*'s study provided experimental validation of proposed architectural metrics to an extent. Further, they also introduced comparative analysis on modularity achievement between human based development effort and randomized modularization. However, there is still an opportunity to explore application of these metrics in broad spectrum of software quality and software sustainability, particularly during evolutionary phases of software development. As a matter of fundamental perspective of modularization, decay of architectural strength is often expected as the software longevity continues. Therefore, evaluation of architectural metrics during software evolution can provide comprehensive assessment of major sustainability concerns and quality oriented features.

In this paper, we explored Sarkar *et al.*'s package-level modularization metrics for automated optimization of module structure and determine their correlation strength with the metrics related to testing efforts, deficit produced in design of software and overall maintainability. First, we describe the theoretical framework to position our study into big picture of software sustainability. Then, an integrated and empirical approach is presented for evaluating various modularization metrics and software quality attributes during the process of software evolution. There are different determinants of architectural erosion or degradation, but, packages become important architectural subsystems in OO scenario [13]. Precisely, the ability of package components to manage and handle dependencies among classes is quite significant among OO design constituents. While characterizing software system, structural perspective are conventionally analyzed by the class level coupling and cohesion. It was indeed required to explore high level architectural dimension to identify design violations in subsequent releases of software systems. Hence, our motivation to report exploratory study with different architectural metrics becomes obvious with following intended contributions.

- Adding the evidence that package based modularization metrics describing cohesion, coupling and programming practices can be linked to modularization metrics of different engineering domains.
- Evidence that technical aspects of sustainability prescribed by package modularization metrics ultimately help in understanding the composition of software systems.
- Establishing statistical soundness to study as many of studied the modularization metrics show significant correlation from reasonable sample size of data-sets.

Our findings show that Sarkar's modularization metrics bear significant association with already existing modularity metrics. Additionally, significant statistical correlation was also witnessed with metrics quantifying maintainability, design deficit and testing effort. Consequently, these findings help to evaluate software sustainability in terms of architecture. Also, this research study can be utilized to assess the development effort and help taking measures to minimize the design flaws. This paper is organized in nine sections, starting from this introduction. Theoretical Framework is explained in Section 2. Related work is briefly described in Section 3. Section 4 describes the information on architectural level metrics and their summarized definitions. Section 5 illustrates the example for package level metrics. Section 6 presents detailed empirical study with analysis over obtained results. Different aspects of discussion over results obtained and design of study are illustrated in Section 7. Threats to validity are elaborated in Section 8 followed by Conclusion as Section 9.

II. Background

The Software Sustainability Institute relates sustainability[1] with concepts of availability, extensibility and maintainability of software. Despite numerous existing definitions of sustainability, there is an ongoing research to achieve consensus for the setting it's scope within field of software engineering. Taking this direction, we attempt to study system's maintainability and integrity as factors affecting the software sustainability.

In today's technologically motivated business world, evolution and maintenance of software systems are key processes carried out over the decades. Long lasting software systems are inevitable for automating industrial devices, as their longevity insures smoother and uninterrupted business operations. In theoretical terminology, software sustainability covers broad spectrum of measures needed to operate software system for longer time, i.e., stability of its infrastructure, adaptability to functional and environmental changes and inter-interoperability in competitive business strategies.

To define the sustainability in the context of software architecture, an explicit concept is required to confine its notion towards technical concerns of sustainability. Therefore, architectural sustainability of software primarily refers to long living software system that is maintained cost-effectively and evolved over its entire life cycle [14]. Thus, our intended connotation is to incorporate sense of cost-effective longevity and endurance towards sustainability of software systems, covering dimensions of maintainability, modifiablility and evolvability.

III. The Theoretical Framework

This framework aims to position our research into software quality, describing architectural strength during it's evolutionary period as key to software sustainability. Sustainability as quality objective has been targeted by many computer and management based systems [15]. There are various techniques and concepts that have been defined to evaluate architectural quality of software systems. Standard draft of quality models, i.e., ISO/IEC 42030 and ISO/9126[2] for architecture evaluation of software systems describe software, hardware, human and systems components as its major constituents.

[1] http://www.software.ac.uk/about
[2] http:// www.iso.org

However, it is required to incorporate characteristics of sustainability concerns with an existing model. Sustainability analysis framework with empirical evaluation are beginning to appear in research literature of software engineering [14]. Our work discussed here, is an effort to evaluate sustainability at architectural level and emphasize package view of architecture as its determinant.

A. Dimensions of Sustainability

As discussed earlier, sustainability is defined as the "capacity to endure and preserve the function of a system over an extend period of time". Recently, researchers and practitioners analyze the sustainability by its four dimensions, i.e., economic, social, environmental and technical. Social sustainability is concerned with application of software to help communities. Environmental sustainability means protection of natural resources using software based knowledge and application. Technical sustainability seeks to improve longevity and adequate evolution of software systems with changing technological requirements. While evaluating sustainability of software in holistic views a shown in Fig. 1, broader context of inter-dependence among the dimensions should also be considered. Economic sustainability aims at maintaining the business aspects of software systems. Applicability of sustainability analysis is shown through an example of Health Watcher System (WHS) in Fig. 1. Health Watcher System is a basically web-based information system for public health monitoring and complaint registration. This example has been frequently used to depict relevance functional and non-functional requirements of software systems with sustainability [16], [17]. Fig. 1 shows quality requirements of (WHS), sorted by sustainability dimensions and relations among them. This example was particularly selected to show how measurement of a software architectural strength influences sustainability in broader picture.

Sustainability portrays broad view of quality assured software systems that is to be achieved by all dimensions. From developers point of view, understanding the relationships among goals of all these dimensions is important to resolve conflicting aspects. However, setting up all these dimensions into one scope has been shortcoming in current software engineering practice. In particular, sub-characteristics of theses dimensions require quantitative evaluation and evidence of association among each other. The problem, we address here is that how architectural components influence technical sustainability, thereby supporting other dimensions as well.

B. Software Sustainability and Software Architectures

As a basis of discussion about sustainability, architectural evaluation method can provide potential mechanism for sustainability measurement. Architecture is foundation of any software system that expresses fundamental organization of system's components and relationships among them. It is design artifact or blue-print of developing software system [18]. Clements *et al.* argue that successful software development and evolution is highly dependent on design decisions [19]. This position is further endorsed by Koziolek *et al.* who describes quality induced software architecture as determinant of sustainability. They further suggest that methods and metrics should be integrated to evaluate architecture using scenario-based analysis. However, their own analysis highlights the limitations of existing method. A number of methods exist which provide evaluation mechanisms of software architecture using structured approach. The main focus in this regard has been analysis of candidate architecture and identification of potential risks involving non-functional requirements of software design. However, significant difference of approach and methodology is found in all these effort. Bowser *et al.* state that architecture decision represented by design metrics can be instrumental to achieve sustainability [20]. These architectural decisions are characterized by decomposition quality, best practices adherence, change scenario robustness and decision traceability. However, explicit expert knowledge for computing these metrics is challenge as acknowledged by them as well. One of hurdles to this approach is that architecture-level metrics are not yet integrated with evaluation approaches of software architectures. Similarly, appropriate context of applying any metric to evaluate implemented architecture is underpinning concept. Therefore, architectural representation of systems can be effective in understanding broader systems concerns. Deriving suitable measures and metrics towards architectural evaluations is key task.

Comprehension of architectural views of complex software systems entails different perspectives. Clements *et al.* describe three view types as most common and feasible to represent software architecture: *module view*, *Component-and-Connector View* and *Allocation View* [19]. Views mainly represent different units of implementation for an architecture that have composed software system. *Module View* determines construction and decomposition of source code (e.g., clusters, packages, files) at design level. Package view is an intuitive approximation of system's architecture that represent the system's architectural modules. Package structure is reasonably assumed comprehensive approximation of architecture as packages are created by developers of software system [21]. Therefore, package structure of java projects is studied as representative of *module view* architecture in which each module contains several classes and dependencies among them.

IV. Related Work

Different approaches have been proposed in recent research over modularity analysis of software design, which has explored many dimensions for characterizing the object oriented systems on distinct criteria [22], [9], [14]. There exists lot of work in the literature proposing metrics for OO software, the majority of these works centered on characterizing a single class as the criteria of high cohesion, low coupling and structural organization [23], [24], [25]. Evaluation of software architecture has emerged as an important software engineering practice which is evident from the efforts of designing tools and computing mechanisms [26]. Such evaluations are exercised to primarily to determine architectural strengths of software systems. Below we briefly explain the works which are closely related on their application for software sustainability measures.

There is prior empirical evidence in several studies that investigated the relationship between code dependency and software quality [27], [28], [29]. D' Ambrose *et al.* identified the relationships between change coupling and defects and

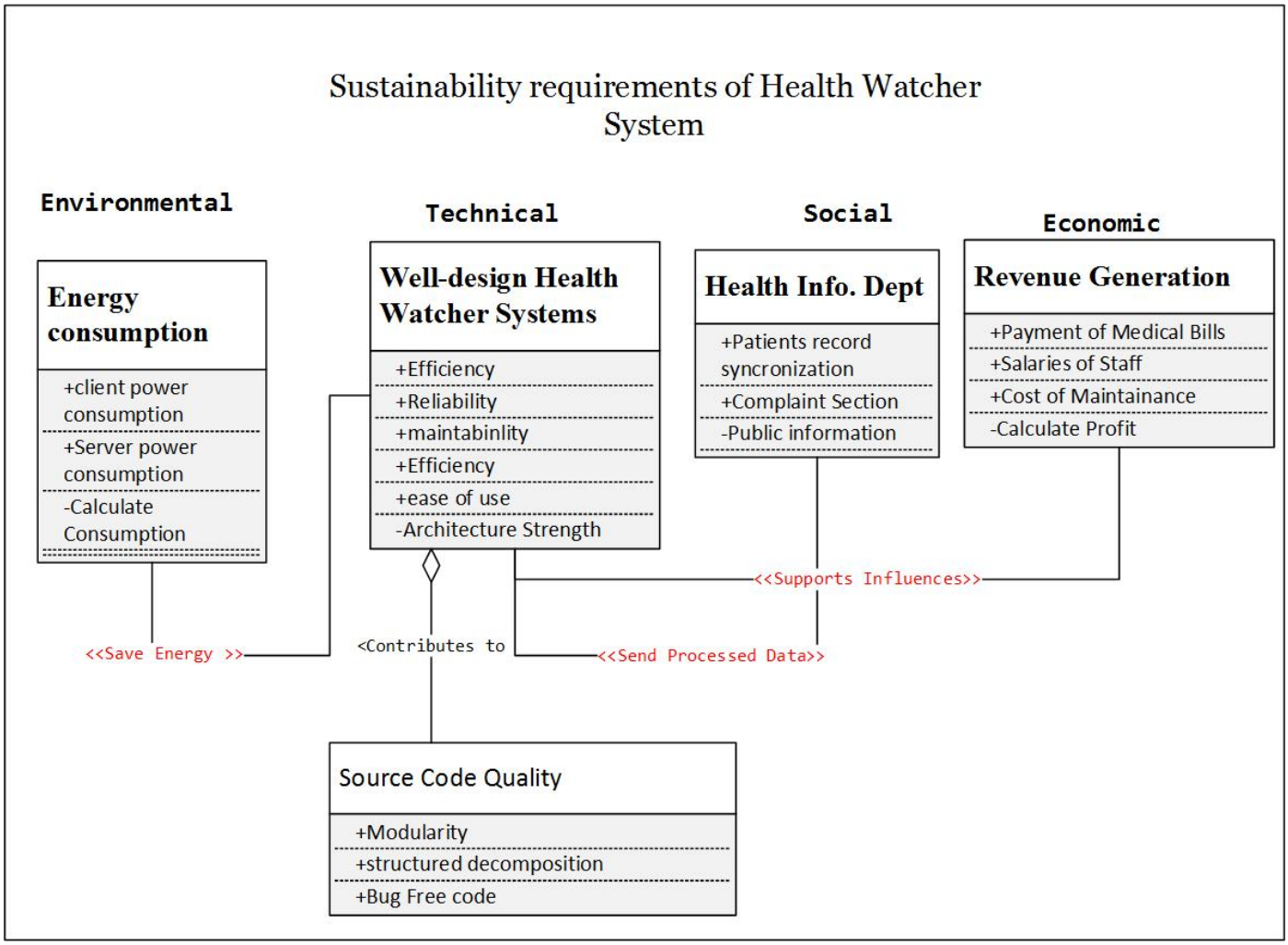


Fig. 1. Four dimensions of Software Sustainablity

class level [27]. Martin introduced Common Closure Principle (CCP) as a design principle about package cohesion, identifying CCP as guideline for decomposition of architecture [27]. Mockus *et al.* found that subsystems modified by a change can be predictor of fault, without defining exact structure(package, cluster, file) of subsystem [28]. Nagappan *et al.* use change-coupling effects to predict faults, however, source-code architecture is not publicly available[29]. In summary, majority of these studies have focused on examining effects of file level coupling on defects, while we approach architectural level evaluation of source code. Moreover, our research is an attempt to show that reverse engineered approximation of system's architecture at package level can be useful for technical sustainability.

According to a systematic review, there are more than 40 architectural-level metrics based on several design principles which assist sustainability evaluation of implemented architectures [14]. It has been reported in the survey that all these metrics are derived from certain design principles, whereas input for computing these metrics varies, e.g., concepts, module size design decisions. Lakos *et al.* proposed metrics known as Cumulative Component Dependency (CCD), which calculates summation of required dependencies in component or module of software systems [30]. CCD metric is numerical value describing strength of module coupling and help in determining maintainability and testability. Martin proposed several metrics, like, efferent and afferent coupling, abstractness, instability, dependency cycles for package entities of software systems [31]. Babar *et al.* carried out the survey to organize architecture-level metrics into framework [32]. These effort targeted the designed architecture and their impact on quality attributes, but, main focus was on differentiation of evaluation methods rather than in-depth quantification or analysis of software modularization and its evolution.

Sethi *et al.* provided an architectural evaluation methodology of software systems on the basis transforming UML component diagram into design structure matrix (DSM) [33]. Evaluation criteria proposed in this work require significant refinement of models and their diagrammatic representation with high precision. This may produce trade-off for large object oriented system and achieving high precision is not always guaranteed. Similarly, there were certain other notable efforts which had its prime focus to analyze architecture during design phases, but, on critical note, all these efforts require

complex procedures of collecting and updating relevant data or representing the software systems into graphs [34], [35].

With growing importance of evaluating software architecture, there has been research urge to propose and re-define the architecture-level metrics based on novel object oriented design principles. Sarkar *et al.* put-forward the determination methodology for evaluating well modularized system taking into account size of components and architectural operations [36]. However, proposed architectural measurement framework does not provide any explicit goodness or fitness of metrics, thus it required more rigorous analysis for its practical application. Therefore, Sarkar *et al.* extended architectural level metrics definitions which received notable recognition in research literature, specially for large object-oriented systems [12], [13]. To the best of our knowledge, Zhoa *et al.* studied the capability of Sarkar's package modularization metrics to predict packge level fault-proneness that remains relatively closer to our research direction [37]. Bouwers *et al.* applied research method of metric-based evaluation for software architectures which is also motivation of our study [38]. In simmilar effort, Yu *et al.* has studied the correlation between multiple package dependencies and evolution in correspondence [39]. However, this study does not address the impact of package's co-evolution on software quality. Koziolek *et al.* made remarkable effort for evaluating architectural sustainability, but their study is restricted on architectural tracking in temporary industrial experimental setup [7]. Our study, on the other hand, investigates effects of package's quality from an architectural perspective. Hence, it paves the research direction for application, assessment and evaluation of Sarkar's metrics in determining sustainability concerns and quality attributes of software systems.

V. Modularization and Quality Metrics

This section provides the description and summary of investigated metrics and software quality metrics. Table I summarizes the definitions of three categories of package-level modularization metrics produced with specific methodology of programming design, i.e., inheritance, association and polymorphism, method invocation and programming practices. Table II presents the summary of modularization metrics studied by Lee *et al.*[40]. The main objective of their research was to analyze and compare the various modularity metrics that have been proposed in different domains. Table III describes three quality metrics to evaluate software systems in terms of maintenance, design flaws and testing. These metrics are categorized as follows:

A. Inheritance and Association based Coupling Modularization Metrics

Within object-oriented (OO) design paradigm, inheritance and association are important dependence relationships between classes and packages in a software system. Inheritance is formed when a class extends another class, while association is formed when class uses (through attribute definition into methods) another class. In addition to this, such dependencies among the classes are often seen to be distributed in different modules (Packages in Sarkar's context of study). More specifically, if a class and its subclass exist in two different modules, modification to concrete base class may trigger the change in subclass. Such design phenomenon is known as fragile base-class problem which becomes prominent in particular when maintenance or ownership of packages is taking place across different development teams. In order to insure easier maintainability and re-usability of packages, minimization of inheritance or association based dependencies can be one of the best programming practices.

Sarkar *et al.* describe these metrics to measure the modularization quality of modules with respect to inter-module dependencies. Furthermore, illustration of each metric is given as under:

- **IC(S)**: is a composite metric that measures the extent to which inheritance-based dependencies among and within the packages are minimized. Whereas IC_1 measures extent to which a package extend other packages using inheritance relationship, IC_2 measures the extent to which classes of a package are extended by classes in other package and IC_3 measures the number of classes in a package that are derived in any of the other packages.
- **AC(S)**: is a composite metric that measures the extent to which association-based dependencies among and within the packages are minimized. Whereas AC_1 measures extent to package uses other packages attribute and parameter in method definition, AC_2 measures the extent to which classes of one package are used either as an attribute and parameter in method definition by classes in other packages and AC_3 measures the extent to which number of classes in a package that are used either as an attribute or parameter in method definition into any of the classes of other packages.
- **BCFI(S)**: is a composite metric that measures the extent to which polymorphic design of methods is restricted to the defining packages. Whereas $BCVSet(p)$ is set of classes in the package that contains defined or non-overridden inherited methods from their ancestor classes of other packages and $BCVMax(c)$ for class c measures maximum base-class violation by methods involved in base-class fragility problem.

B. Method Invocation based Coupling Modularization Metrics

Another suite of coupling metrics introduced by Sarkar *et al.* is related to inter-module coupling created by invocation of methods among the modules(packages in their study). In an ideal programming scenario, application programming interfaces (APIs) of a module should be used as service providers to other modules. This sort of design mechanism is established through particular programming pattern of invoking methods or calling methods in inter-module connections of software systems. In a well engineered code, software systems should adhere with principle of maximum segregation and similarity of purpose to avoid any structural decay. However, such modularization practices are violated at times, thus, their quantitative evaluation shall explore more dimension of application.

- **MII(S)**: It measures the extent to which all inter-module interactions are carried out through APIs (designated methods for providing services to other module) of module in entire software system. $MII(P)$ is ratio of $ExtCallRel(i)$ and $ExtCallRel(p)$. Whereas $ExtCallRel(i)$ is set that collects all external calls to API

TABLE I. DESCRIPTION OF SARKAR'S PACKAGE-LEVEL MODULARIZATION METRICS [13]

Metric	Definition
Inheritance based Inter-Module Coupling(IC)	$IC(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} IC(p), \quad IC(p) = \min(IC_1, IC_2, IC_3)$
Association Induced Inter-Module Coupling(AC)	$AC(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} AC(p), \quad AC(p) = \min(AC_1, AC_2, AC_3)$
Base-Class Fragility (BCF)	$BCFI(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} BCFI(p), \quad BCFI(p) = 1 - \frac{1}{BCVSet(p)} \sum_{c \in \mathcal{C}(p)} BCVMax(c)$
Module Interaction Index (MII)	$MII(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} MII(p), \quad MII(p) = \frac{\lvert\bigcup_{i \in I(p)} ExtCallRel(i)\rvert}{\lvert ExtCallRel(p)\rvert}$
non-API method closedness index (NC)	$NC(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} NC(p), \quad NC(p) = \frac{\lvert M_{na}(p)\rvert}{\lvert M_{pub}(p) - \{\bigcup_{i \in I(p)} M(i)\}\rvert}$
API Usage Index (APIU)	$APIU(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} APIU(p), \quad APIU(p) = \frac{APIUS(p) + APIUC(p)}{2}$
State access violation(SAVI)	$SAVI(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} SAVI(p), \quad SAVI(p) = \frac{1}{\lvert C(p)\rvert} \sum_{c \in \mathcal{C}(p)} SAVI(c)$
Population Pug-in Index (PPI)	$PPI(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} PPI(p), \quad PPI(p) = \frac{\lvert\bigcup_{m \in ImplExtn(p)} \{ModuleClosure(m,p)\}\rvert}{\lvert M(p)\rvert}$
Size Uniformity Index (CU_m)	$CU_m(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} \frac{\mu_m(p)}{\mu_m(p) + \sigma_m(p)}$
Size Uniformity Index (CU_l)	$CU_l(S) = \frac{1}{\lvert\mathcal{P}\rvert} \sum_{p \in \mathcal{P}} \frac{\mu_l(p)}{\mu_l(p) + \sigma_l(p)}$

Note: S represents entire software system, $\mathcal{P}$ denotes set of packages
$\mathcal{C}$ shows set of classes, M is set of methods, I is set of APIs
$C(p)$ is set of classes in a package p and $I(p)$ is set of API's in package p.
Module and package are used interchangeably in Sarkar's context of study.

methods of packge p, $ExtCallRel(p)$ is set that collects all external calls to public methods in package p.

- **NC(S)**: It measures the extent to which all inter-module interactions are carried out through Non-API methods in entire software system. More precisely, $NC(p)$ determines extent of coupling, a package p establishes through invocation of methods which are explicitly declared as non-abstract.
- **APIU(S)**: It is an average of segregation measure $APIUS(p)$ and cohesiveness measure $APIUC(p)$ of packages in entire software system.

C. Metrics based on Best Modularization Practices

In addition to inter-module coupling, Sarkar *et al.* have proposed third category of modularization metrics which can essentially be applied for best programming practices to insure enhanced software quality. These modularization mechanisms are based on measuring the extent to which inter-module interactions can be minimized. State access violation is phenomenon when communication among software design components is frequently carried out through attribute access. Similarly, third-party plug-ins are integrated within the software systems for their functional and operational extension which also require some design rule evaluation. Additionally, common reuse of software components and their size impact have been shaped into metrics form. These metrics are briefly explained as under:

- **SAVI(S)**: It measures the extent to which the attributes defined in the classes in a package are not directly accessed by other classes for entire software system. $SAVI(c)$ is computed using weighted average of different attribute access cases, i.e., inter/intra-classes and inter/intra-moduels.
- **PPI(S)**: It measures the extent to which the methods that are needed to define extension APIs exist in a plug-in package for entire software system. $ModuleClosure(m, p)$ is set of transitive closure of abstract method implementation in a package p and $M(p)$ is set of methods in package p.
- $\boldsymbol{CU_m}$(S): It measures the extent to which classes in packages of entire software system varies in their sizes taking into account number of methods, whereas $\mu_m(p)$ and $\sigma_m(p)$ are average and standard deviation of package class size in terms of number of methods .
- $\boldsymbol{CU_l}$(S): It measures the extent to which classes in packages of entire software system varies in their sizes taking into account lines of code, whereas $\mu_l(p)$ and $\sigma_l(p)$ are average and standard deviation of package classes size in terms of lines of code.

D. Baseline Modularization Metrics

Below we summarize the definitions, notations used in definitions and interpretation of these metrics in particular context of study by Lee *et al.* [40]. 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. Moreover, their study is another motivating factor to further examine the modularization metrics proposed with different design paradigms. We set this research work as *baseline* to further investigate the applicability of modularization metrics proposed by Sarkar *et al.* Their definitions with relevant references are mentioned in Table II.

- $\boldsymbol{M_{newm}}$: This metric is well known approach for quantifying modularity of social network represented in graphical structures. Recently, there has been extensive focus on

TABLE II. BASELINE MODULARIZATION METRICS STUDIED IN DIFFERENT DOMAINS

Reference	Definition
Newman *et al.* [41]	$M_{newm} = \frac{1}{2m}\sum_i\sum_j\left(A_{ij} - \frac{k_i k_j}{2m}\right)\delta(g_i, g_j)$
Mancoridis *et al.*[42]	$MQ = \sum_{i=1}^{k}\frac{2\mu_i}{2\mu_i + \sum_{j=1}^{k}(\epsilon_{i,j}+\epsilon_{j,i})}, \quad M_{bunch} = \frac{MQ}{k}$
Guo *et al.*[43]	$M_{g\&g} = \frac{\sum_{k=1}^{M}\frac{\sum_{i=n_k}^{m_k}\sum_{j=n_k}^{m_k}R_{ij}}{(m_k-n_k+1)^2} - \sum_{k=1}^{M}\frac{\sum_{i=n_k}^{m_k}\left(\sum_{j=1}^{n_k-1}R_{ij} + \sum_{j=m_k+1}^{N}R_{ij}\right)}{(m_k-n_k+1)(N-m_k+n_k-1)}}{M}$
MacCormack *et al.* [44]	$M_{rcc} = 1 - \sum_{i=1}^{N}\sum_{j=1}^{N}\frac{DependencyCost(i,j)}{N^{2\lambda}}$

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. δ is a comparator function that it outputs 1 where its two parameters are same, 0 otherwise. g_i, parameter of δ, 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 *et al.* [42]. MQ is the most frequently used method for evaluation of a software modularity. μ_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, λ 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^{λ}, 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^{λ} to have considerable penalty in terms of poor coupling.

E. Software Quality Metrics

All these quality metrics have been proposed to asses the quality of software systems. Although, there are diverse opinions for setting up any metric as standard for judging the quality of software. Nevertheless, it can be an interesting research direction, if their utility is realized with empirical evidences. Further, they are described below with their definitions in Table III.

- **MI**: HealStead Maintainability Index is composite metric that incorporates number of traditional source code metrics into single value that indicates relative maintainability. The equation presented in Table III for MI is its reformed version that considers *aveV* as Halstead Volume per Module, *aveV(g')* as extended cyclomatic complexity per module and *aveLOC* as average lines of code per module.
- **QDI**: Quality Deficit Index is a positive value aggregating the detected design flaws (i.e., code smells and architectural smells). In terms of computation, each flaw carries a specific weight which accumulates to form a score that determine extent of deficit in particular source code entity(class, package or entire software system).
- **TLOC**: Test Lines of Code metric determines effort required to test the software system. In this regard, TLOC is a size measure that counts physical lines of code within a test class or classes.

VI. AN ILLUSTRATING EXAMPLE

In this section, we use an example of simple architectural design to illustrate the working mechanism of Sarkar's metrics. In order to ease the comprehension, example shows evaluation of metrics for single package interactive modularization scenario. To simplify the presentation, only architectural representation of design for package p is described. Fig. 2 shows system of 6 packages to demonstrate Sarkar's metrics.

As it can be seen, package p is interacting with five other packages $p1, p2, ..., p5$ using *use*, *implement* and *extend* relationships through its classes and interfaces. From Fig. 2, we can see that p has 3 incoming *extend* relationships, i.e., from 2 classes of package $p2$ and 1 class of package $p1$. Further, package p has 1 outgoing *extend* and 1 outgoing *implement* relationship from its 2 classes to package $p5$ and $p2$. It is also visible that package p has 6 incoming *use* relationships from 1 class of package $p1$, 3 classes of package $p3$, 1 class

TABLE III. DESCRIPTION OF QUALITY METRICS.

Reference	Definition
Welker*et al.* [45]	$MI_1 = 171 - 5.2 \times \log(aveV) - 0.23 \times aveV(g') - 16.2 \times \log(aveLOC)$
Marinescu *et al.* [26]	$QDI = \frac{\sum_{all\text{-}flaw\text{-}instances} FIS_{flawinstance}}{KLOC}$
Tahir *et al.*[46]	$TLOC = \frac{\sum_{i=1}^{n} Loc(C_i)}{SLOC}$, C_i *is test class*

of package $p4$ and 1 class of package $p5$. Also, package p has 1 outgoing *use* relationship towards package $p2$. Thus, these *extend*, *implement* and *use* relationships form AC and IC coupling for package p respectively. Whereas, only 1 concrete method $m10$ is overridden outside the package p into package $p1$ and $p2$ causing the fragility in base class $C1$ of package p .

Fig. 2 also shows interaction among the packages through invocation of public methods, abstract methods and implementation of abstract methods. For a package p, 3 abstract methods ($m1$,$m2$ and $m7$) are called outside the package which form S-API's[3] of package p. Package p has 2 public methods ($m3$ and $m4$) that are called outside the package p which account non-API external calls for methods.

Fig. 2 further depicts access to attributes in intra-package and inter-package dependency scenarios. Attributes of classes $C6$ and $C7$ of package p are accessed into two packages: $p4$ and $p5$. Classes $C2$ and $C3$ of package p access attributes of $C6$ and $C7$ using intra-package dependency. This information of attribute access is utilized to evaluate the violation of state access to package p. Package p also provides implementation and calls through transitive closure of abstract methods ($m23$ and $m52$) declared outside the package p which serve as extension plug-in to package p using classes $C2$ and $C3$.

VII. EXPERIMENTAL STUDY

In this section, we describe our experimental evaluation of Sarkar's modularization metrics over open source software system. The measures described for architectural sustainability are pertaining to structural design and source code of software systems. Although, in larger picture there are certain additional indirect measures also discussed in the literature, like, documentation quality, technology choices, employment of human resources, volatile requirement, etc. [14]. However, our scope of our study is towards software design centric metrics.

A. Study Design

Selection of appropriate metric for evaluating the software product is a challenge. Therefore, we tend to adopt most efficient and already utilized methodology in our research with illustrative and insightful aspects [40]. To an examination level, our goal is to better understand the effect and impact of package level modularization metrics over different dimensions of software systems. i.e., modularization metrics, external quality attributes. In this regard following study features are set as research questions for our experiment.

RQ1: Do Sarkar's package modularization metrics correlate with Baseline modularizaton metrics and impact the software quality metrics?

RQ2: Do Sarkar's package modularization metrics provide any significant perspective for improving architectural quality during software evolution?

These research questions are important to understand the critical context of applying Sarkar's metrics. Conventionally, architectural quality of software systems is measured by certain quality metric and their application is not frequently witnessed in practice. Despite their availability comparison or association with existing metrics is rarely performed. Thus it is yet unclear to select particular architectural metrics to quantify the modularization of software system, which leads our motivation behind RQ1. Another objective of RQ1 is to expand the application and usage of Sarkar's metrics (exclusively designed on the basis of OO paradigm) in other dimensions of software quality assurance as restricting them to only modularization metrics may result worthless. The research context of RQ2 is to understand weather architectural facets of evolving software systems can be better explained by package modularization metrics proposed by Sarkar *et al.* This will help both software architects and researchers to understand practical value of applying of Sarkar's metrics to evaluate architectural longevity of evolutionary software systems.

B. Subject Systems

We selected 34 versions of three different open source systems in our experiment. JHotDraw [4]: a Java GUI framework for technical and structured Graphics. Ant[5]: a Java library and command-line tool whose mission is to drive processes described in build files. Google-Web Toolkit(GWT)[6]: an open source set of tools that allows web developers to create and maintain complex JavaScript front-end applications in Java. These systems have reasonable size, manageable degree of complexity, diverse application domain and easily accessible source code. Table IV provides descriptive information of subject systems used in our study, versions involved and their release period.

[3]Sarkar *et al.* described concept of abstract methods and their container entities(abstract classes and interfaces) serving as Application Programming Interfaces for a package. S-API refers to public abstract methods declared, implemented within the package and invoked outside the package.

[4]http://www.jhotdraw.org/

[5]http://ant.apache.org/

[6]http://www.gwtproject.org/overview.html

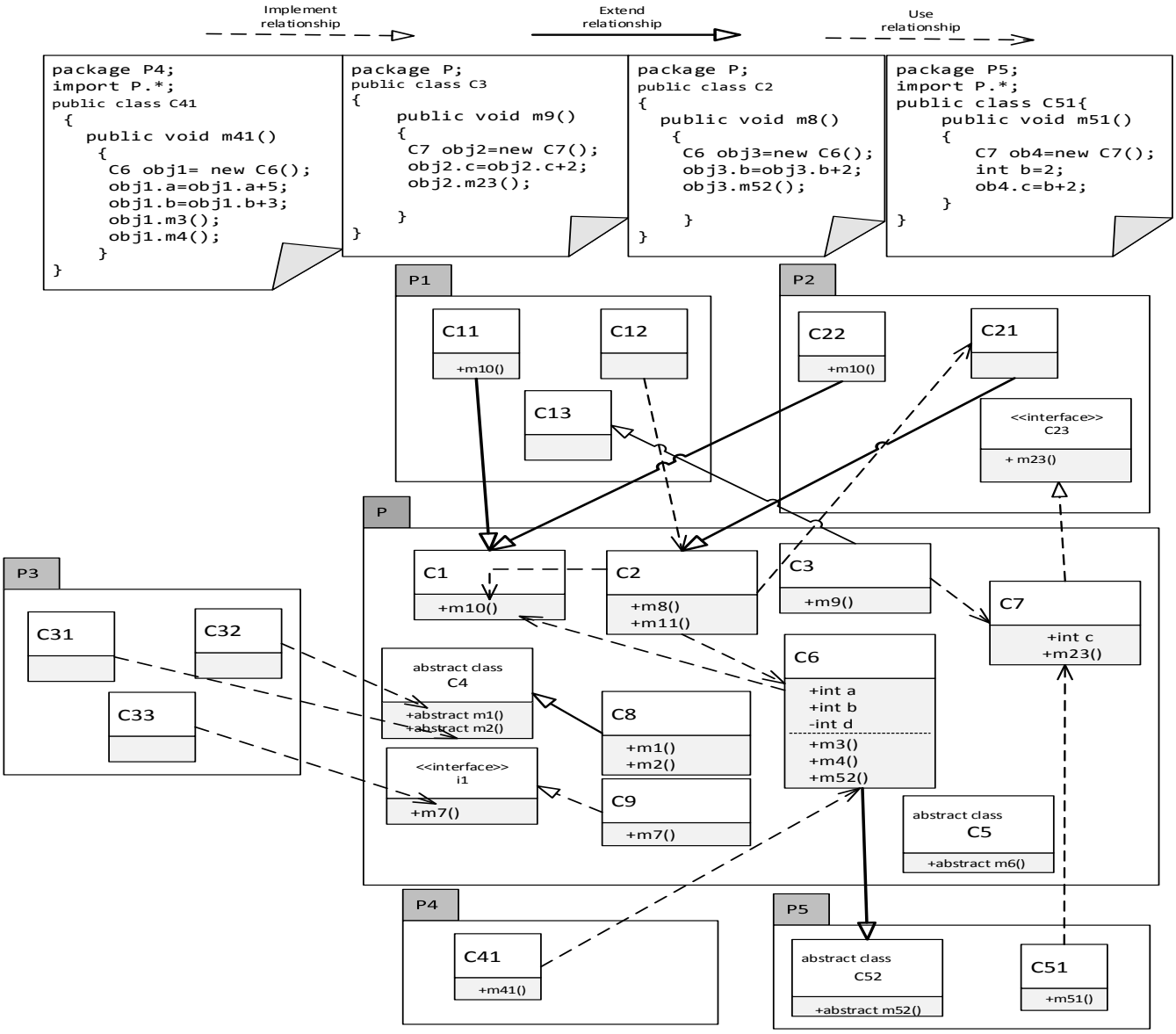


Fig. 2. Simple design for Sarkar's package level metrics

Structural information of subject systems in Table IV provides valuable information. The first to fifth columns report the systems name, versions involved in the study, revised number of entities (Packages, Classes, Interfaces) from initial to final release. The column sixth to nine, revision in number of methods and their invocation on diverse design methodologies are listed. It is evident that release versions experimented in our study have been under development process of over 7 years at-least and considerable changes have taken place in their structural components (Entities and dependencies among them). Since all these open source software systems are frequently utilized in research or industrial areas, it was expected that some of changes their code base would be in correspondence to architectural changes or modularization improvement. Rationale for this expectation was that developers continuously examine and improve structure of evolving software systems.

C. Data Processing

Our purpose of data processing is to compute defined metrics and conduct statistical analysis. To compute Sarkar's metrics, *TLOC* and *MI*, we parsed source code of applications from version archives database using *Understand*[7]: A commercial static analysis tool. We developed our own java code and Understand-Perl API scripts to derive all metric values. To calculate *QDI*, we used evaluation version of open source tool *Infusion*[8]: It parses the source code and provides an in-depth assessment of architecture and design quality. Moreover, the tool mainly detects code smells, architectural flaws and reports quality score for software systems. After collecting metrics information, data-sets for each subject system were

[7]https://scitools.com/
[8]https://www.intooitus.com/products/infusion

TABLE IV. STRUCTURAL INFORMATION OF SUBJECT SYSTEMS

System	Versions	Release Period	Revised packages	Revised classes	Revised interfaces	Revised methods	Revised S-API's	Revised non-API calls	Revised API calls
JHotDraw	(5.2-5.4, 6.0, 7.1-7.5)	2001-02-19 to 2010-08-01	11-62	168-997	23-60	1476-7334	157-291	101-961	127-228
Apache-Ant	(1.5.2,1.5.4, 1.6.0-1.6.5 , 1.7.0,1.71, 1.80-1.8.3)	2003-08-13 to 2012-03-13	62-79	902-1659	48-98	6759-13488	143-222	131-241	128-226
GWT	(1.3-1.7, 2.0-2.5)	2006-5-17 to 2014-11-20	106-498	1655-12390	213-1823	71598-10192	132-272	151-256	230-931

developed. On each data-set, statistical test of correlation is used to obtain correlation coefficient values using R [9] tool. The Spearman's correlation test is particularly applied in our experimental context due to non-parametric nature of data. Additionally, we have made our data sets public on our project site[10] for replication and reproduction. Fig. 3 depicts visual view of our data processing methodology.

Our method for examining the evolving modularity of subject systems builds on source code parsing and extracting the dependency information. Following are keys steps involved in our method.

1) Select the versions to be analyzed and obtain their binary distributions.
2) For each version, extract the dependency information from the parsed code.
3) Compute the required metrics and develop the data-set for each subject system.
4) Report graphical and statistical analysis.

D. Experimental Results

In this section, we report experimental results for modularity correlation between Sarkar's metrics and *baseline* modularization metrics described in Section 2. We have presented correlation between Sarkar's metrics and quality attributes. Further, graphical representation is shown to understand behavior of each category of Sarkar's modularization metrics in different evolving versions of subjects systems.

1) RQ1: Do Sakar's package modularization metrics correlate with Baseline modularizaton metrics and impact the software quality metrics?: To answer RQ1, we employed statistical methodology of correlation test. We briefly describe our results obtained after applying statistical test of Spearman's correlation. Also, we have tried to clarify theoretical and analytical perspectives of values indicated in all the tables. In order to best understand empirical relationship among the each category of Sarkar et al's modularization metrics and baseline metrics, magnitude of statistical association of is shown at significant level of 0.001 (indicated with ***), 0.01 (indicated with **) and 0.05 (indicated with *) in Table V, VI and VII. In particular, all the indicated values employ following rationale.

- Positive and statistically significant correlation with *Baseline* Modularization metrics is desired; meaning package modularization metrics improves architectural quality of software system, while negative and statistically significant correlation values are indication of deterioration in architectural quality.
- Since *QDI* and *TLOC* are measures of quality deficit and testing effort incurred over software system. Therefore, statistically significant and positive correlation values with package modularization metrics indicate that software system is subject to design restructuring or may incur exhaustive testing overhead. In other words, modularization of particular package design has not improved enough to compensate architectural debt or cause reduction in testing effort. This kind of occurrence in software design can be cause of weak modularization that eventually violate the design pattern rules.
- Similarly, *MI* describes strength of maintenance for software, hence, positive and statistically significant correlation of package modularization metrics with *MI* is an evidence of quality enhancement.
- Negative and positive statistically insignificant correlations are considered void of any impact in our scope of study.

Aforementioned implications essentially form experimental scope of our study to determine application of package modularization metrics for diverse objectives. It should be understandable that theoretical construct of each metric is different, thus, correlation analysis with Baseline modularization metrics and quality metrics would be distinctive.

Starting with Table V for Sarkar's coupling modularization metrics based on inheritance and association, some of important observations for different software systems are described: For *JHotDraw*, *BCF* is seen strongly correlated with M_{newm} and M_{rcc}. Whereas, *AC* has shown relatively significant relationship with modularity metrics of M_{bunch}, $M_{g\&g}$ and M_{rcc}. For *Apache-Ant*, strong association of correlation is established only for *IC* with M_{bunch}, $M_{g\&g}$ and M_{rcc}. Interestingly, M_{newm} has shown weak statistical significance with most of coupling metrics in all cases. While in case of *Google-Web ToolKit*, *AC* and *IC* have produced notable statistical relationship with modularity metrics. Such kind of variations in modularity correlation can be result of particular design paradigm of software systems where association based dependencies are minimized as in case of *JHotDraw* or inheritance based dependencies and fragile base classes are abundant as in case of *Apache-Ant*. Results for quality attributes can be elaborated with following specific findings: First, positive significant correlation of *BCF* with *QDI* and *TLOC* employs that fragility of base-class in software may cause design deterioration and testing effort. Thus, less maintenance work will be required as witnessed for correlation coefficients of *JHotDraw* and *Apache-Ant* with *MI*. Second, in most cases,

[9] http://www.r-project.org/
[10] https://github.com/Analyzer2210cau/Package-Sustain

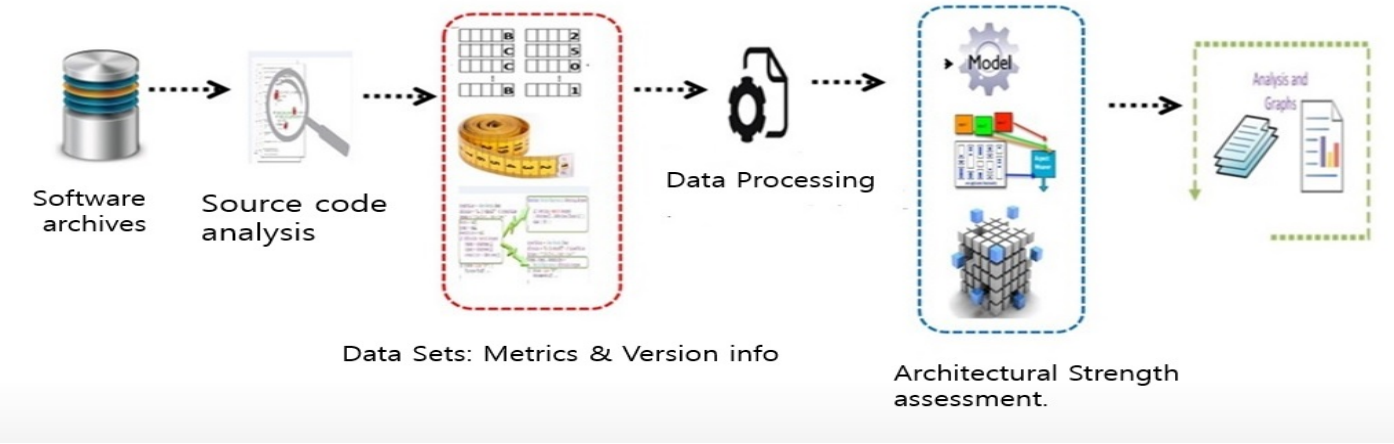


Fig. 3. Data processing Methodology

TABLE V. CORRELATION COEFFICIENT OF COUPLING MODULARIZATION METRICS BASED ON INHERITANCE AND ASSOCIATION

Project	Metric	Baseline Modularization Metrics				Quality Metrics		
		$\mathbf{M}_{newm}$	$\mathbf{M}_{bunch}$	$\mathbf{M}_{g\&g}$	$\mathbf{M}_{rcc}$	**MI**	**QDI**	**TLOC**
JHotDraw(9)	BCF	**0.94*****	**0.77***	0.59	**0.97*****	**-0.86****	**0.92*****	**-0.95***
	IC	**0.94*****	0.66	0.24	**0.84****	**-1.000*****	**0.99*****	**-0.78***
	AC	0.53	**0.72***	**0.78***	**0.70***	-0.18	0.32	-0.65
Apache-ant(14)	BCF	0.41	0.20	0.24	**0.60***	-0.072	**0.75****	**0.64***
	IC	0.27	**0.76****	**-0.79****	**-0.93*****	-0.52	-0.37	**-0.94****
	AC	0.13	**-0.54***	-0.54	0.31	-0.51	0.52	-0.26
GWT(11)	BCF	0.26	0.32	0.12	-0.45	-0.19	-0.49	-0.085
	IC	**0.83****	**0.80****	0.45	**-0.90*****	0.45	0.024	**-0.80****
	AC	**-0.94****	**-0.90***	-0.41	**0.98****	**-0.58***	0.01	**-0.87*****

IC and *AC* are negatively correlated to weak extent with *MI* regardless of their statistical significance that means reducing direct inheritance and association based inter-module coupling shall have influence over maintainability.

Correlation coefficients for Sarkar's coupling modularization metrics based on method invocation are reported in Table VI with following major observations. For *JHotDraw*, only *MII* has developed significant relationship with *Baseline* modularization metrics and quality attributes while *APIU* has shown no as such statistically significance. For *Apache-ant*, *MII*, *NC* and *APIU* are in negative correlation with *Baseline* modularization and quality attributes. Such findings employ that, during the development process, there has been frequent inter-module method call traffic which has in turn decreased the modularization, eventually quality attributes are adversely affected. While in case of *Google-Web ToolKit*, *MII* has depicted significant positive correlation with M_{newman}, M_{rcc} and significant negative correlation with M_{bunch}, however, *NC* and *APIU* do not show similar pattern in the table except for *MI* or *TLOC*. It can be inferred that either the cohesiveness and segregation properties are not followed properly or non-API public methods calls have caused excessive and unnecessary dependencies among the packages. Thus, there are wide variations on this category of modularization metrics which may not account for providing comprehensive view, but its application is yet useful to asses particular software design for inter-module method invocation. A closer examination of these values suggests that inadequate architectural and implementation changes can precipitate decrease in modularity.

Table VII shows correlation values of third category of Sarkar et al.'s metrics. For *JHotDraw*, weak correlation of *SAVI* with M_{bunch} and strong negative and positive correlation with *MI*, *QDI* and *TLOC* should not be surprising due to violation on accessing attributes rules. On the other hand, high trend of correlation in all metrics for CU_m and CU_l is quite visible. For *Apache-Ant*, all the category metrics have developed negative correlation with most of *baseline* modularization metrics and quality metrics, leading to conclude that there has been lack of adopting best modularization practices. For *Google-web-ToolKit*, statistically significant values are quite hard to realize except for *SAVI*, meaning rare violation of state access could be required.

In principle, modularization should reduce complexity of system, allow efficient comprehension of its structural paradigms and enable easier re-factoring process. As a matter of fact, each software system follows certain specific domain of design considering requirements and functional needs. Similarly, as the software evolves, changes in its architecture are quite obvious. All these parameters contribute toward long life of software systems. Although, architectural metrics proposed by Sarkar *et al.* are reported as important to evaluate complementary areas of software sustainability, however, their usage in practice (e.g., software industry) is not too wide. Combining the results of Table V, VI and VII, we can have substantial assessment of software architecture, but, it would not be easy to make general conjecture for single metric. Furthermore, some of metrics have conflicting traits among the each other; as is the case of *IC* and *APIU*, hence

TABLE VI. CORRELATION COEFFICIENT OF COUPLING MODULARIZATION METRICS BASED ON METHOD INVOCATION

Project	Metric	Baseline Modularization Metrics				Quality Metrics		
		$\mathbf{M}_{newm}$	$\mathbf{M}_{bunch}$	$\mathbf{M}_{g\&g}$	$\mathbf{M}_{rcc}$	**MI**	**QDI**	**TLOC**
JHotDraw(9)	MII	**0.89****	**0.75***	0.44	**0.87****	**-0.87****	**0.90*****	**-0.88***
	NC	-0.45	-0.23	0.01	-0.39	**-0.61***	**-0.60***	**0.52**
	$APIU$	0.47	0.31	0.20	0.43	-0.48	0.52	**0.56**
Apache-ant(14)	MII	0.22	**-0.80*****	**-0.82*****	**-0.68****	**-0.69****	**0.06**	**-0.76****
	NC	0.05	**-0.84*****	**-0.85*****	**-0.78*****	**-0.78****	-0.18	**-0.83*****
	$APIU$	-0.20	**-0.72****	**-0.76****	**-0.90*****	**-0.50***	**-0.46***	**-0.97*****
GWT(11)	MII	**0.89****	**-0.75***	0.44	**0.87****	**-0.85****	**0.90*****	**-0.92*****
	NC	-0.45	-0.23	-0.01	-0.39	**0.61***	**-0.60***	**0.90****
	$APIU$	0.47	0.31	0.20	0.43	-0.48	0.52	**-0.79***

TABLE VII. CORRELATION COEFFICIENT OF COUPLING MODULARIZATION METRICS BASED ON INHERITANCE AND ASSOCIATION

Project	Metric	Baseline Modularization Metrics				Quality Metrics		
		$\mathbf{M}_{newm}$	$\mathbf{M}_{bunch}$	$\mathbf{M}_{g\&g}$	$\mathbf{M}_{rcc}$	**MI**	**QDI**	**TLOC**
JHotDraw(9)	$SAVI$	**0.69***	0.38	0.02	**0.63***	**-0.87****	**0.87****	**-0.88****
	PPI	-0.55	**-0.61***	**-0.92****	**-0.80****	0.31	-0.44	0.34
	CU_m	**0.75***	**0.64***	0.50	**0.87****	**-0.67***	**0.77****	**-0.68***
	CU_l	0.365	0.46	**0.80****	0.64	-0.19	0.27	**-0.72***
Apache-ant(14)	$SAVI$	0.17	**-0.93*****	**-0.94*****	**-0.72****	**-0.89*****	0.25	**-0.76****
	PPI	0.09	**0.60***	**0.63***	**0.77****	0.40	**0.52***	**0.87*****
	CU_m	0.019	**-0.84*****	**-0.84*****	**-0.65***	**-0.81*****	0.39	**-0.59***
	CU_l	-0.34	**-0.50***	**-0.53***	**-0.76***	-0.20	-0.37	**-0.74****
GWT(11)	$SAVI$	**0.65***	**0.68***	0.30	**-0.66***	**-0.67***	**0.23*****	**-0.79****
	PPI	0.02	0.045	-016	-0.009	0.014	-0.022	0.009
	CU_m	0.19	-0.032	-0.001	-0.009	**-0.53***	**-0.60***	0.33
	CU_l	-049	-0.51	-0.36	**0.65***	-0.17	0.23	0.44

optimization of certain architectural features may raise unexpected sustainability vulnerabilities[7]. Despite these precise analytical constraints, architectural metrics can help developers to re-asses their design decision to insure modularization in particular during software evolution process. For instance, result of Table VII suggests that excessive state access violation should be avoided during development of *Apache-ant*and results of Table VI indicate that inter-module connections through method invocations can be minimized to improve modularization of *Apache-ant*. Overall, Our results show that package level modularization metrics bear considerably significant relationship with modularization metrics studied in different engineering domains. Such findings can help the software engineer to develop design plan to ensure optimized source code architecture. Additionally, Sarkar's modularization metrics were also analyzed to reveal significant correlation with different software quality metrics.

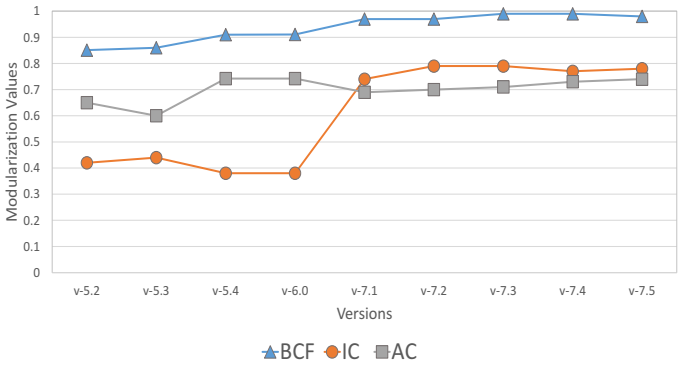
2) Do Sarkar's package modularization metrics provide any significant perspective for improving architectural quality during software evolution? : Software usually undergoes a continuous process of evolution driven by incremental development processes and design improvement. Modularity of software reflects different design considerations. Modularity of software systems has been linked with growth of software development community, code contribution and reduction of design vulnerabilities. As a theoretical construct, modularity also relates with other domain of research, including organization mechanism, industrial economics and product refinement. However, measuring modularity is crucial for maintenance of complex and large software systems.

To analyze the architecture's sustainability, we assessed the evolution scenario of software systems formed with architectural metrics. Each subject system was separately observed. This kind of analysis towards evolution scenario of software systems is helpful to judge the impact on its structural changes and to identify points where architectural mitigation can be incorporated during software development. To elicit the evolutionary changes, each category of Sarkar *et al.*'s metrics is exclusively represented in Fig. 4, 5, and 6.

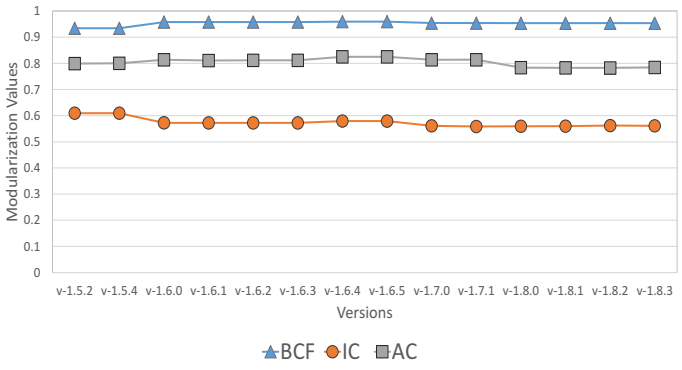
Fig. 4 shows patterns of changes taking place in different software systems against modularization values of Inheritance and Association based coupling. In 14 versions of *Apache-ant*, all the coupling based metrics show the constant pattern. In 9 versions of *JHotDraw*, *AC* and *IC* seem to be improving marginally, whereas, *BCF* has gradually improved to significant extent. In 11 versions of *Google-Web Toolkit*, *BCF* and *IC* has shown slight decline as the software evolves, while *AC* has exhibited an improving trend. Our examination from such graphical representation also tells that *BCF* remains as best modularization among two other metrics, however, evolution process is still followed by fluctuating modularization values of *IC*.

Fig. 5 shows patterns of changes taking place in different software systems against modularization values of method invocation based coupling. It can be clearly seen that through out all the evolutionary period of all software systems, only *MII* has achieved an effective modularization while on the other hand *NC* and *APIU* has shown negative and steady trends. Such graphical representation can help us to understand that inter-module method calls through Application programming interfaces declared for each system should be maximized to modularize the software system.

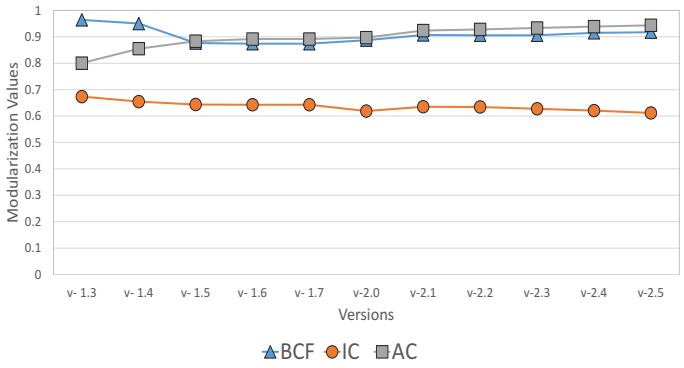
Fig. 6 shows visualization trends for modularization values



(a) *JHotDraw*



(b) *Apache-ant*



(c) *Google-Web-ToolKit*

Fig. 4. Evolutionary changes against Coupling Based on Inheritance and Association

of best practices defined in category 3 of Sarkar's metrics. An important finding is; managing the size of software components in terms of methods and lines of code improves the modularization during its evolution process as shown by CU_m and Cu_l in all cases. Whereas, *SAVI* and *PPI* show low and unstable values raising the vulnerabilities for sustainability. Hence, development process should avoid extensive violation for integrating third party plug-ins or state access to code base.

In summary, architectural decay or improvement is quite dependent on the particular design phenomenon, the software systems are developed for. Although, flexible architecture allows addition of new functionality during evolution process; however, it may come at the cost of certain sustainability concerns and design deterioration as well. Trend of architectural degradation can be result of design time violations, thereby posing sustainability concerns. If these significant mismatches against original design are identified earlier then maintenance and design improvement strategy can be effectively deployed to insure sustainability of software systems. Additionally, we have noticed that certain architectural metrics bear common modularization trends as well in software systems of diverse nature and domain which can be useful to form opinion for architectural-level metrics towards overall sustainability of software systems.

VIII. Discussion

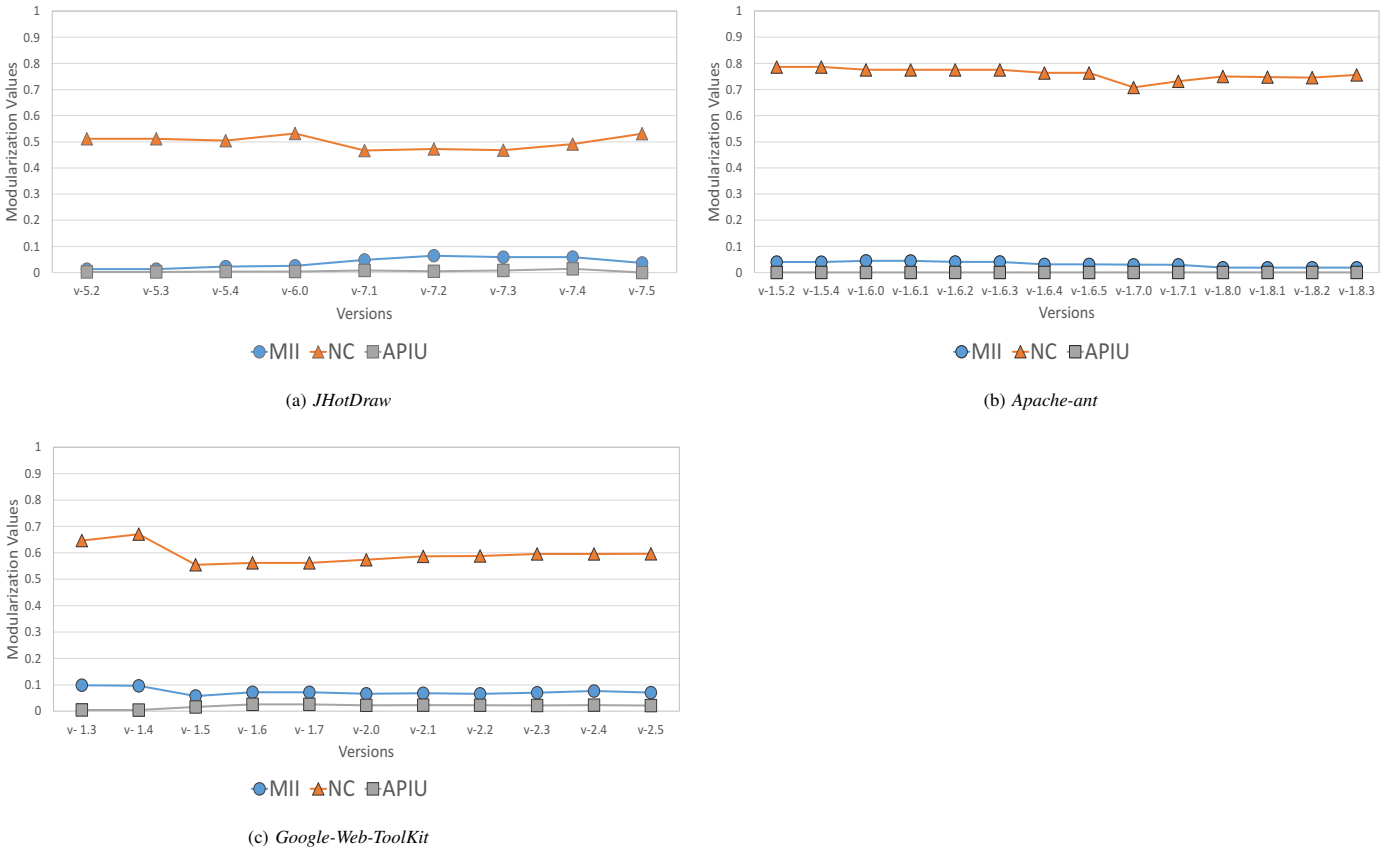
The subject of software sustainability is emerging as benchmark to realize applications of software in social, economic, operational and technical terms. Hence, relevant empirical studies are required to explore the subject further. We presented an experimental analysis over 34 versions of three different open source java systems that includes research objectives, design, data processing and experimental results. In this regard, we reported statistical relationship of Sarkar's modularization metrics with existing validated modularity metrics studied in different domains of engineering [40] and quality metrics. Indeed, from theoretical standpoint, our study has diverse quality assurance focus but with major emphasis of architectural sustainability. We do not rule out other parameters that may have arguably better explanatory power; however, our effort is to explore significance of package based modularization metrics.

IX. Threats to Validity

In this section, we describe the most important threats to construct, internal and external validity.

Construct Validity

Accuracy of metrics calculation and rationale behind setting up *Baseline* modularization and quality metrics are two major factors which account for threats to construct validity. We



(a) *JHotDraw*

(b) *Apache-ant*

(c) *Google-Web-ToolKit*

Fig. 5. Evolutionary changes against Coupling Based on Method invocation

employed our source code analysis procedure with incremental testing to collect the Sarkar's metrics reliably using *Understand* tool which is already utilized in recent empirical studies [37], [47]. Whereas *Baseline* modularization metrics were computed by co-author using our own java based tool and already recognized in research literature [40]. Therefore, the construct validity of metrics computations can be considered quite satisfactory. Although, independent variables (Sarkar's Package modularization metrics, *Baseline* modularization metrics and software quality metrics) are briefly described with research focus and importance, still, it is required to further insure their validity. For example, *MI* and *MQ* have widely been used as software quality assessment metric [48], [49], [50], but other metrics are novel contribution towards setting up determinants of software sustainability.

Internal Validity

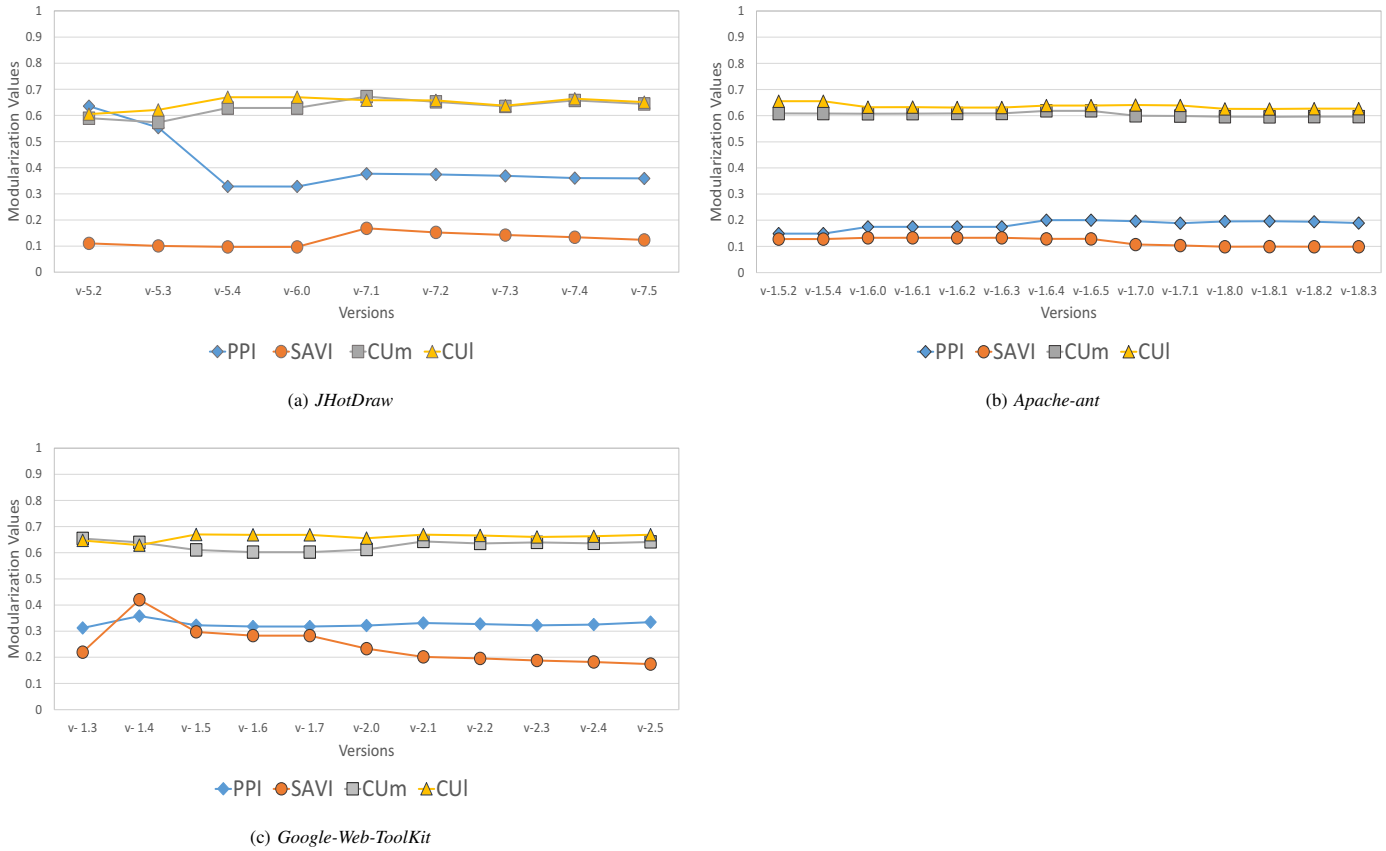
Threats to internal validity can arise from experimental methodology of data analysis applied in our study. Conclusion are drawn on the basis of correlation analysis. We have already mentioned our motivation of using correlation to identify the relationship among the quantities in aforementioned sections. However, these results are not merely based on significance of correlation obtained, but, it expands further through monitoring of evolution scenarios. Moreover, calculating *P-value* for statistical significance in quantitative analysis of software metrics is already utilized methodology [51], [38]. Therefore, sample size and guidelines mentioned by Woh *et al.* [52] and Yin *et al.* [53] counter these threats to internal validity to considerable extent. But, there is need of further study and experimental applications to acquire meaningful conclusion.

External Validity

Our study is based on several versions of open source java systems. This may cause potential threat to external validity due to choice of open source systems developed in particular language. Consequently, it is not possible to generalize the conclusion and results. We experimented our study with open source java system which have been frequently utilized in research literature of empirical software engineering, i.e., *Apache-Ant* and *JhotDraw*. Admitting the fact that this may raise bias over study as software systems of industrial range have not been brought in experimental set up. Indeed, this is an inherent problem in most of empirical software studies. However, we believe that study of 34 versions can substantiate to form at least formidable research opinion.

X. Conclusion

The modularity is an important aspect of software system describing its overall architectural quality and strength. In this paper, we investigated newly proposed modularization metrics based on packages from different perspectives. Our empirical study was mainly based on computation and analysis of modularity metrics through statistical correlation methodology. We presented an assessment for evolution scenarios of



(a) *JHotDraw*

(b) *Apache-ant*

(c) *Google-Web-ToolKit*

Fig. 6. Evolutionary changes against metrics based on Modularization practices

software systems using architectural level metrics. This study can be important to avoid technical risk and develop a formal perspective for managing architectural improvement efforts invested during software development.

Tracking the rules of architectural compliance as the software evolves, there were some unique findings and observation to determine sustainability concerns of software system from different dimensions. Statistical analysis reported on different quality metrics and modularization metrics may help developers to regularly check architectural authenticity and form the code reviews for future quality assurance activities. Due to trade-off among the different architectural metrics, multiple perception are described for software maintenance. In addition to specific empirical view, our study also yielded interesting propositions: First, optimizing of each architectural metrics value is subject to design decisions; however, relative sustainability improvement or decline can be monitored during continuous development process; Second, tracking architectural level metrics and monitoring quality of software design can be possible task when evolution phases of software system are in progress; Third, undesired violations of architectural rules can be controlled with proactive re-factoring or restructuring decisions.

We adopted an integrated approach of evaluating software systems for their sustainable architecture and evolution scenario analysis. Sarkar *et al.* package modularization metrics were statistically tested to check their compliance with existing modularization metrics and their impact over different quality metrics. In addition to this, tracking architectural metrics during software evolution, we could determine ripple effects of different design aspects. Although, there was no as evidence to prioritize these architectural metrics in terms of their application, but, their usage can be helpful to identify critical sensitivities of software sustainability. In broader picture, approach is an effort to explore different dimensions of architectural sustainability.

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] J. Mitchell, C. Laughton, and S. A. Harris, "Atomistic simulations reveal bubbles, kinks and wrinkles in supercoiled dna," *Nucleic acids research*, p. gkq1312, 2011.

[2] S. R. Mounce, R. B. Mounce, and J. B. Boxall, "Novelty detection for time series data analysis in water distribution systems using support vector machines," *Journal of hydroinformatics*, vol. 13, no. 4, pp. 672–686, 2011.

[3] F. C. Maryland, "Measuring software sustainability," 2003.

[4] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent systems*, no. 3, pp. 54–62, 1999.

[5] P. C. Clements, "Software architecture in practice," Ph.D. dissertation, Software Engineering Institute, 2002.

[6] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[7] H. Koziolek, D. Domis, T. Goldschmidt, and P. Vorst, "Measuring architecture sustainability," *Software, IEEE*, vol. 30, no. 6, pp. 54–62, 2013.

[8] W. Albattah and A. Melton, "Package cohesion classification," in *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*. IEEE, 2014, pp. 1–8.

[9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[10] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 193–208, 2006.

[11] H. Abdeen, S. Ducasse, and H. Sahraoui, "Modularization metrics: Assessing package organization in legacy large object-oriented software," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 394–398.

[12] S. Sarkar, G. M. Rama, and A. C. Kak, "Api-based and information-theoretic metrics for measuring the quality of software modularization," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 14–32, 2007.

[13] 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.

[14] H. Koziolek, "Sustainability evaluation of software architectures: a systematic review," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM, 2011, pp. 3–12.

[15] P. Lago, S. A. Koçak, I. Crnkovic, and B. Penzenstadler, "Framing sustainability as a property of software quality," *Communications of the ACM*, vol. 58, no. 10, pp. 70–78, 2015.

[16] C. G. von Wangenheim, A. von Wangenheim, F. McCaffery, J. C. R. Hauck, and L. Buglione, "Tailoring software process capability/maturity models for the health domain," *Health and Technology*, vol. 3, no. 1, pp. 11–28, 2013.

[17] D. Mairiza, D. Zowghi, and N. Nurmuliani, "An investigation into the notion of non-functional requirements," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 311–317.

[18] M. Paulk, "Capability maturity model for software," *Encyclopedia of Software Engineering*, 1993.

[19] P. Clements, R. Kazman, and M. Klein, *Evaluating software architectures*. ACM, 2003.

[20] S. Sehestedt, C.-H. Cheng, and E. Bouwers, "Towards quantitative metrics for architecture models," in *Proceedings of the WICSA 2014 Companion Volume*. ACM, 2014, p. 5.

[21] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 99–108.

[22] R. Martin, "Oo design quality metrics-an analysis of dependencies," *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, 1994.

[23] J. Al Dallal and L. C. Briand, "An object-oriented high-level design-based class cohesion metric," *Information and software technology*, vol. 52, no. 12, pp. 1346–1361, 2010.

[24] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[25] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI. ACM, 1995, pp. 259–262.

[26] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

[27] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 135–144.

[28] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[29] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[30] J. Lakos, *Large-scale C++ software design*. Addison-Wesley Reading, 1996.

[31] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[32] M. A. Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," in *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. IEEE, 2004, pp. 309–318.

[33] K. Sethi, Y. Cai, S. Huynh, A. Garcia, and C. Sant'Anna, "Assessing design modularity and stability using analytical decision models," *Drexel University, Philadelphia, PA, Technical Report DU-CS-08-03*, 2008.

[34] A. Tang and J. Han, "Architecture rationalization: a methodology for architecture verifiability, traceability and completeness," in *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*. IEEE, 2005, pp. 135–144.

[35] E. Bouwers and A. Van Deursen, "A lightweight sanity check for implemented architectures," *Software, IEEE*, vol. 27, no. 4, pp. 44–50, 2010.

[36] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena, "On the modularity assessment of software architectures: Do my architectural concerns count," in *Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD*, vol. 7, 2007.

[37] 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.

[38] E. M. Bouwers, *Metric-based Evaluation of Implemented Software Architectures*. TU Delft, Delft University of Technology, 2013.

[39] L. Yu, A. Mishra, and S. Ramaswamy, "Component co-evolution and component dependency: speculations and verifications," *IET software*, vol. 4, no. 4, pp. 252–267, 2010.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] K. D. Welker, "The software maintainability index revisited," *CrossTalk*, vol. 14, pp. 18–21, 2001.

[46] A. Tahir, S. G. MacDonell, and J. Buchan, "Understanding class-level testability through dynamic analysis," in *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. IEEE, 2014, pp. 1–10.

[47] Y. Zhao, Y. Yang, H. Lu, J. Liu, H. Leung, Y. Wu, Y. Zhou, and B. Xu, "Understanding the value of considering client usage context in package cohesion for fault-proneness prediction," *Automated Software Engineering*, pp. 1–61, 2016.

[48] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[49] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[50] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, p. 4, 2014.

[51] 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.

[52] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[53] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.