Software Architecture Quality Measurement Stability and Understandability

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

Abstract—**Over the past years software architecture has become an important sub-field of software engineering. There has been substantial advancement in developing new technical approaches to start handling architectural design as an engineering discipline. Measurement is an essential part of any engineering discipline. Quantifying the quality attributes of the software architecture will reveal good insights about the architecture. It will also help architects and practioners to choose the best fit of alternative architectures that meets their needs. This work paves the way for researchers to start investigating ways to measure software architecture quality attributes. Measurement of these qualities is essential for this sub-field of software engineering. This work explores Stability and Understandability of software architecture, several metrics that affect them, and literature review of these qualities.**

Keywords*—*Software Engineering; Software Architecture; Quality Attributes; Stability; Understandability

I. Introduction

Software systems are becoming complex, larger, more integrated, and are implemented by the use of several varieties of technologies. These various technologies need to be managed and organized to deliver a quality product. Quality attributes usually assessed and analyzed at the architecture level not at the code level. It is usually the case that when we decide on a appropriate architectural choice (i.e. the system will exhibit its required quality attributes) without the need to wait until the system is developed and deployed, since software architecture enables to predict system qualities.

The software architecture field has been inspired by other engineering domains. This inspiration led the movement to these well-known concepts such as stakeholders and concerns, analysis and validation, styles and views, standardization and reuse, best practices and certification. However, software is inherently different from all other engineering disciplines. Rather than delivering a final product, delivery of software means delivering blueprints for products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them.

In this work, we pave the way for researchers to start investigating ways to measure software architecture quality. The remainder of this paper is organized as follows. Section II introduces and defines software architecture and discusses its importance. Software metrics are discussed in Section III. Software architecture measurement is presented in Section IV. Tow samples of software architecture quality attributes are discusses is Section V. Section VI presents the software architecture measurement validation techniques. Conclusions are presented in Section VII.

II. Software Architecture

Over the past years software architecture has became an important sub-field of software engineering. There has been substantial advancement in developing new technical approaches to start handling architectural design as an engineering discipline. However, much research is yet to be carried to achieve that. Moreover, the changing nature of technology raises a number of challenges for software architecture.

Designing a software structure is the phase that comes immediately after gathering and analyzing the software requirements. During this phase the software is constructed in terms of components and relationships that link these components with each other [1]. These components and their relationships will illustrate the architecture for particular software. The software architecture of a system has many definitions in the field of software engineering. Software Architecture is defined in the IEEE standards [2] as “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”. The authors of the book ’Software Architecture in Practice’ [1] defined the software architecture as “the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.” Software architecture represents the design decisions that are hardest to change and determine the overall system proprieties [3]. Those decisions have to be made before concurrent work on a system can be started. Architecture decisions will not be at a component level, but they span the overall system components and determine their interconnections and constrains. Once all architecture decisions are made, work in individual components can proceed independently [4].

Software architecture is the name of a particular form of abstraction, or model, of software systems. It is considered as abroad abstraction of the system, which contains information about both functional and non-functional requirements. The architecture lays the foundation of the shared communication platform for the various stakeholders. Software architecture typically bridges between requirements and implementation. Software architecture embodies the earliest decisions that shaped the impact on the success/failure of the software system. Software architecture serves as a reasoning, important communication, analysis, and growth tool for software systems [1].

Several authors came to an agreement that software architecture is a skeleton produced in the early phase of design. Documenting the software architecture is useful both as a means of communication between stakeholders and in providing an overall picture of the system that is to be developed. Architecting the software system is a crucial task since it lays the groundwork for later activities in the software development process. The architecture plays an important factor in the software success or failure. Understanding the architecture is very important for both architects and developers to relate it to requirements, product, and process.

The software architecture influences greatly the system's quality as it can inhibit or enable product's quality attributes. The quality of a software system is largely attributed to its software architecture [5]. Thus, evaluation of this software architecture should be done on a regular basis. Such repeated evaluations ensure that the system remains sustainable and evolvable over a longer period of time [6].

The software architecture can be decomposed into more granular levels, namely packages, components, and modules. Package is used to represent a set of classes that might be hierarchically structured and to perform a series of related tasks [7]. A package is a group of classes that are related to each others or perform one higher purpose. Classes in the same package have special access privilege with respect to one another and may be designed to work together closely. Component in the context of object-oriented design is for organization purposes. Component contains a group of classes and other components as well. A component provides one or two similar system functionalities [8]. Module consists of a large number of classes and sometimes a module is referred to as a package. It provides information hiding for the module allowing a software engineer to see it as a black box [9].

The field of software architecture remains reasonably immature. Although it has an engineering foundation for software architecture, it is not clear yet, there are still several challenges. As a result, we anticipate major new advancements and developments in the software architecture field in the future.

A. *Importance*

The software architecture is very important in the software development life-cycle. It is considered as the blueprint of the system where important decisions are documented. It is a reference for the whole system in design, development, and maintenance. A poor software architecture may lead to a deficient software product that does not satisfy its customers and can not be adaptive to new changes. David Garlan [10] summarized the importance of software architecture in six aspects of software development:

1) Understanding: Software architecture can be seen as mechanism to simplify our ability to understand complex-large systems by presenting them at a higher level of abstraction [1]. Furthermore, the architecture exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices [11].
2) Reuse: Software architecture supports reuse of components and frameworks. Platforms, frameworks, components, architectural patterns, libraries of plug-ins, add-ins, apps, and domain-specific software architectures are different promoters of reuse.
3) Construction: Software architecture provides a blueprint for development and implementation by showing the major components and dependencies between them. For instance, a layered architecture documents abstraction boundaries between parts of a system's implementation [11].
4) Evolution: Software architecture exposes the dimensions along which a system is expected to evolve. Software maintainers can easily understand the ramifications of changes, and accurately estimate costs of modifications [12].
5) Analysis: Software architecture can be seen as a way to analyze the whole system. These analyses can include satisfaction of quality attributes [1], system consistency checking [1], conformance to constraints forced by an architectural style, and domain-specific analyses for architectures built in specific styles.
6) Management: Software architecture can be seen as a viable milestone in any industrial software development process. Critical evaluation of an architecture leads to clear understanding of requirements, implementation plans, and possible risks, which will reduce the amount of rework required to address problems later in a systems life-time [1].

B. *Quality Attributes*

This section summarizes several important quality attributes across the software architecture domain. A quality attribute (QA) is a measurable feature of a system, which is utilized to stipulate how well the system satisfies stakeholders. You can consider a quality attribute as measuring the goodness of that property. ISO/IEC 9126 [2] classifies quality attributes of software as functionality, maintainability, usability, efficiency, reliability, and portability. These characteristics are attributes that can describe a software system. These quality attributes are further derives the sub-characteristics with more attributes. The quality characteristics are refined to sub-characteristics and these sub-characteristics are refined to attributes or measurable properties using several metrics. A metric is a defined measurement method that assigns a value to that attribute.

Quality attributes are strongly related to non-functional requirements of a system. One of the responsibilities of the software analyst to come up with a complete list of quality attributes before architecting and designing the system. Quality attributes commonly include efficiency (time, efficiency, resource economy), functionality (completeness, security, interoperability), maintainability (expandability, modifiability, testability), portability (hardware independence, software independence, installability, reusability), reliability (error tolerance, availability), and usability (understandability, user interface, learnability). Figure 1 shows the quality attributes in ISO/IEC 9126.

When software architects are able to measure and quantify these quality attributes, they will be able to enumerate feasible

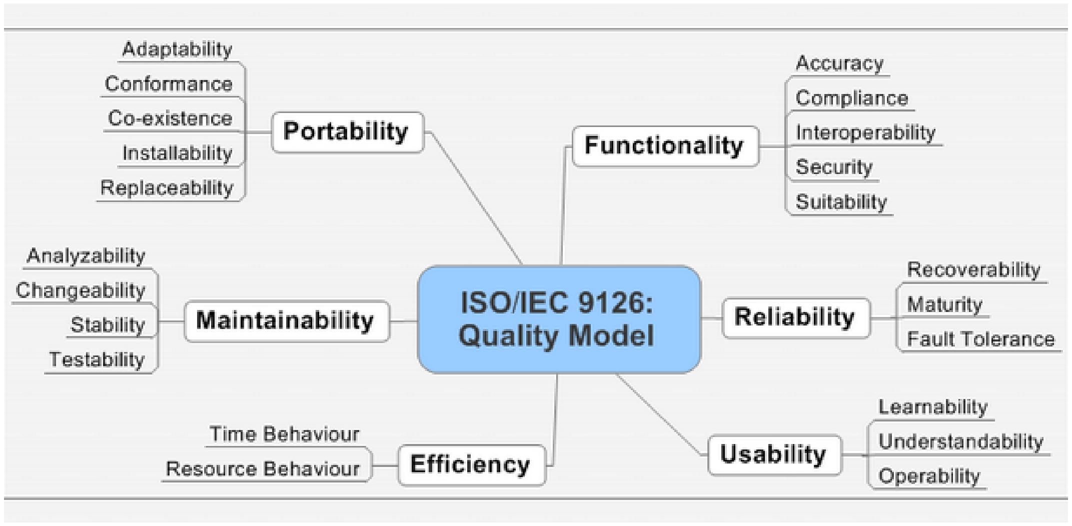

Fig. 1. Quality Attributes in SQuaRe

architecture design and evaluate all quality attributes. As each quality attribute is assigned a measure, a total score can be calculated in order to help the architect which design alternative to use. When the evaluation process is complete, the design with high score will be chosen.

III. Software Metrics

Measurement is crucial for any science or engineering field. Organizations strive to come up with meaningful measures that indicate progress or performance. Measurement in software engineering is considered a crucial factor to evaluate the software quality characteristics such as functionality, usability, reliability, efficiency, maintainability, and portability. In software engineering, there is still a lack in that discipline. We still need to work in consolidating terminology, principles and methods of software measurement [13]. Software measurement activities consist of direct and indirect assessments, as well as predictions [13], [14]. Measurement allows us to understand the current situation and to come up with clear benchmarks that are useful to set goals for the future behavior. Software measurement [15] is not limited only to evaluate a software product but it will be used to evaluate the software development process. Measurement is a crucial activity in all empirical studies.

Software metrics field is an interesting field in the software engineering community since more than 30 years. The interest in metrics by both academician and practitioners is growing rapidly. Software metrics are defined as [16] "standard of measurement, used to judge the attributes of something being measured, such as quality or complexity, in an objective manner". Software metrics are measures utilized to evaluate the process or product quality. These metrics helps project managers to know what is the progress of software and evaluate the quality of the various artifacts produced during development. The software requirements engineers can validate and verify requirements. Software metrics are required to capture various software attributes at different phases of the software development [17]. Software metrics are required to adequately measure various points in the software development process.

Software metrics constitute the main approach to software measurement [13], [18], [19]. Software metrics and quality are major players in measurement of software quality. Measuring software artifacts should focus on selecting the right metrics for each software and on how to apply them [13].

IV. Software Architecture Measurement

Software architecture measurement suffers from what people calls the tyranny of the dominant architectural principle. The assessments of certain principles are overstressed, other equally important design principles have been omitted in architecture measurement processes.

Each quality attribute can be measured using different characteristics of the software architecture. The characteristics can be of size, complexity, coupling, cohesion, or others. Furthermore, each quality attribute can be measured by combining several existing measures. Several quality attributes are very similar and can complement each other. Several software metrics can be combined together to measure a certain property using either composition or aggregation [20]. In composition, software metrics used to assess a property can be composed by (1) simple or weighted average of the metrics. This can be used only when the different metrics have similar range and semantic; (2) thresholding; (3) interpolating; or (4) a combination of these methods. In aggregation, several steps are required. (1) a weighting function is applied to each metric then (2) average the weighted values of the metrics then (3) we compute the inverse function of the average.

V. Architecture Quality Attributes

In this section we review several attempts to measure two quality attributes of the software architecture, namely stability and understandability.

A. *Stability*

The primary goal from the architecture evaluation is to assess and validate the software architecture using systematic methods and procedures [21]. This evaluation is accomplished to ensure the examined software architecture satisfy one or more of quality characteristics. One desired quality of the software architecture is stability. Stability is one of the maintainability characteristics of the ISO/IEC SQuaRe quality standard [2]. According to this standard, stability is defined as the degree to which the software product can avoid unexpected effects from modifications of the software [2]. Architectural stability reduces unnecessary architecture rework as the software system's functionality is expanding over multiple versions, thus reducing implementation costs.

As architectures have a profound effect on the operational life-time of the software and the quality of the service provision, architectural stability could be considered a primary criterion towards achieving the long-livety of the software. Architectural stability is envisioned as the next step in quality attributes, combining many inter-related qualities.

Several researchers proposed several metrics to measure the stability of software architecture. Stability is the ability of software to remain unchanged while facing new requirements or changing the environment. The software has to accommodate some of these changes and they should not affect the software stability, while other may harm the software stability. This section presents an overview of several attempts in the literature to measure the stability of particular software.

Ahmed et al. [22] proposed a new way to measure the architectural stability of an object oriented system by using similarity metrics. These metrics compare pair versions of a system. First metric is Shallow Semantic Similarity Metric (SSSM), and the purpose of this metric is to the measure the semantic similarity between components in a pair of systems. Second metric is Relationship-Based Similarity Metric (RBSM) and it is aiming to measure similarity between the relationships that exist in a pair of systems. A regression line is generated for the architecture changes with releases from these similarity values. A higher value indicates a stable architecture.

Ebad et al. [3] continued the work that is proposed in [22] by developing a new architecture stability metric (ASM) that measure cross-architecture components communications in term of inter-package connections (IPC). The idea behind IPC is when a pair of releases of a software system is compared; there are three types of changes that may happen (addition, deletion, and modification). ASM value will be between 0 and 1, where 1 means lowest possible amount of changes between two releases which means stable software architecture. ASM is validated by a set of mathematical properties which are: non-negativity, normalization, null value, maximum value, transitivity, package cohesion impact and change impact. Moreover, this metric is experimentally validated by using two open source projects: JHotDraw and abstract windowtoolkit. Measurements of the ASM are illustrated by lines of code for original IPCs and deleted IPCs, and added IPCs across releases in the two previously mentioned projects.

Aversano et al. [23] evaluated the software architecture for a set of open source software projects. Most of these projects are selected from sourceforge. Stability is the characteristic that is examined in order to evaluate the software core architecture. The evolution of certain software is considered when the software components are changed during the software releases. Two metrics are proposed to measure the stability of each release. These two metrics are Core Design Instability (CDI) and Core Call Instability (CCI). Both metrics provide a measure of how much the architecture of a software system changed passing from a release to another one. CDI metric finds the change in terms of number of packages and CCI finds the change in terms of number of the interactions among packages. Smaller values mean less change which means greater stability. All these metrics are based on calculating fan-in and self-call for software packages.

Alshayeb et al. [24] mentioned that none of the existing measures have included all class aspects such as class relationships, attributes, and methods. At the first of this study all properties that affect the class stability are identified; these properties are class access level, class interface level, inherited class name, class variable, class variable access- level, method signature, method access level , method body. Then from these properties the proposed metric is recognized. The name of the discovered metric is Class Stability Metric (CSM). Stability is calculated by counting the number of unchanged properties between two classes in version i+1 and version i divided by the maximum possible change value, then summation of all these properties is divided by the number of the properties which is eight. This metric is theoretically validated by some properties. Moreover, this metric is empirically validated through two Java systems. The result of this empirical study indicates that this metric is highly negatively correlated with maintenance effort.

Li et al [25] proposed new metrics to measure the stability for the software design. They highlighted that metrics that are discovered by Chidambe & Kemerer [26] cant measure all aspects of Object Oriented. Examples of these aspects are the change in the class name, class number, and class inheritance relations. From this imperfection of C&K metrics, authors proposed these three metrics: System Design Instability (SDI), Class Implementation Instability (CII), and System implementation Instability (SII). The main goal that pointed out in this study is to justify how the information that is gathered from theses metrics can help project manager to adjust the project plan. These metrics are experimentally examined against C&K metrics. They found out that SDI and CII measure Object Oriented aspects that are different from the aspects that are measured by C&K metrics.

Abdeen et al. [27] introduced a complementary set of coupling and cohesion metrics that assess packages organization in large legacy object-oriented software. These metrics are aiming to measure the modularization for an object-oriented system. Here are the metrics that are discovered by Abdeen: Index of Inter-Package Usage (IIPU), Index of Inter-Package Extending (IIPE), Package Focus (PF), Index of Package Service Cohesion (IPSC), and Index of Package Changing Impact (IPCI). These metrics are defined with respect to some modularity principles that are related to packages. Examples of these principles are information hiding, changeability and communality of goal. These metrics are defined with regard to two different types of object-oriented inter-class dependencies: method call and inheritance relationships. All metrics that are discovered in this work are validated against the mathematical properties that have to be existed in any cohesion or coupling metric.

Sethi et al. [28] mentioned that none of existing metrics, that are used to assess the modularity and stability of architecture decomposition, has considered environmental factors that drive software changes. From this point a suite of metrics is proposed: decision volatility, design volatility, impact scope, concern scope, concern overlap and independent level. Theses metrics consider environmental factors that drive software changes. Moreover they tried to measure how well a particular architecture produces independently substitutable modules. Furthermore, these metrics did not require having knowledge about the implementation details. In these metrics design dimensions and environmental conditions are modeled as variables and their relations are modeled as logical constraints. These metrics are evaluated using eight aspect-oriented and object-oriented releases of software product-line architecture.

It has been realized the work that is produced by Chidamber and Kemerer [26] has been cited in most of other studies. They developed a suite of metrics that is used for measuring a particular object- oriented design. The primary goal is to develop and validate theoretically and empirically a set of object-oriented metrics. These metrics are Weighted Method per Class (WMC), Depth of Inheritance of Tree (DIT), Number of Children (NOC), Coupling Between Object Class (CBO), Response for Class (RFC) and Lack of Cohesion of Metric (LOCM). Each one of this matric is evaluated theoretically against six properties: noun-coarseness, non-uniqueness, design details are important, Monotonicity, noun-equivalence of intersection and intersection increases complexity. Then, the implementation of these metrics is demonstrated through data collection from both C++and SMALLTALK implementations. From the obtained data, it has been shown how each one of this metric can help project managers and senior designers to obtain useful information about the entire evolution of a particular application.

Hassaine et al. [29] proposed a novel approach to investigate some metrics (code decay indicators) on software, that serve as symptoms, risk factors, and predictors of decay, in the context of an evolving architecture. The name of their approach is ADvISE and it aims for analyzing the evolution of certain software architecture at various levels (classes, triplets, and micro-architectures). The first step in observing architectural decay is to use a diagram matching technique to identify structural changes among versions of architectures. The second step is detecting the class renaming by using structure-based and text-based techniques. The third step is architecture diagram matching by using a bit vector algorithm to perform diagram matching between two programs versions in order to find the common triplets. The fourth step is architecture diagram clustering by applying the incremental clustering algorithm to find the sets of connected triplets. These sets will form the stable micro-architecture between two program versions. The fifth step is architecture evolution by performing a pairwise matching for programs architectures in order to identify sets of stable triplets and micro-architecture. The authors applied their approach on three open-source systems: JFreeChart, Rhino and Xerces-J to answering the following research questions: RQ1: What are signs of architectural decay and how can they be tracked down? The authors studied the graph of architectures evolution for each system, and then they showed these indicators to provide useful insights regarding the signs of software aging. RQ2: Do stable and unstable micro-architectures have the same risk to be fault prone? The authors showed stable micro-architectures, which are belonging to the original design, are significantly less bug-prone than unstable micro-architectures.

Jazayeri et al. [30] did retrospective analysis to evaluate and assess the architecture for telecommunication software. Twenty releases are selected to observe the evolution of this software. This kind of evaluation helps project managers to predict about how the future of the architecture will be look like. Metrics that are used in this work are likely to be the observation of the some simple measures between a pair of releases while the software is being evolved. Examples of these metrics are module size, number of modules changed, number of modules added, number of modules changed in the same sequence of release, number of programs in the same version of release.

Abreu and Melo. [31] evaluated the impact of object oriented design on software quality characteristics such as defect density, failure density, and normalized rework. There is a set of metrics for object oriented design MOOD. These metrics are empirically evaluated against the software quality characteristics by calculating the correlation coefficients where the quality characteristics are the dependents variables and the design metrics are the independent variables. Examples of these design metrics are 1) method hiding factor (MHF), attribute hiding factor (AHF), method inheritance factor (MIF), attribute inheritance factor (AIF), polymorphism factor (POC) and coupling factor (COF).To quantify the impact of OO design on software quality, a predictive model is developed. The results show that the design alternatives may have a strong influence on resulting quality. For the study validity multiple R, R square and adjusted R square are calculated for the software quality characteristics.

Olague et al. [32] utilized entropy to reduce spikes in the original SDI metric that is produced by Li.et al [25] and proposed the new SDIe metric. This study highlighted that the dynamic nature of the agile development process could obscure an analysis of software stability. Also, this study notified that the SDIe metric is easier than the SDI metric to compute the stability for a particular software system. The reason behind that is SDIe is able to be automated instead of requiring the close investigation of code by human judgment. The SDIe metric is calculated using the number classes added, deleted, changed and unchanged from the previous iteration. SDIe metric is theoretically investigated and validated using the Kitchenham criteria [33] and the Zuse requirements [34] for software measures. Moreover SDIe is empirically tested over two software projects by comparing SDIe metric with the original SDI, using SDIe to assess the software evolution, and comparing SDIe metric to the Chidamber and Kemerer [26].

Tonu et al. [21] proposed an approach that can helps developers in evaluating stability for a particular software architecture. Evaluating the software architecture is based on analyzing the changes in the softwares aspects form one release to another. Software aspects can be structural, behavior, or economical, in this research work the focus is only on the structural aspects. Growth rate, changes rate, coupling, cohesion are the measures that are applied in this approach to do retrospective analysis. Then, evolutionsensitive and evolution critical parts are identified by observing how the subsystems are interconnected between each other. This approach is empirically evaluated on two spreadsheet applications by selecting nine releases for each application, and then the results of the architecture stability are discussed.

Roden et al. [35] performed empirical study on six different highly iterative projects with multiple iterations by observing the system packages. These projects are evaluated using Total Quality Index (TQI), System Design Instability (SDI) and System Design Instability using Entropy (SDIe) metrics. TQI is calculated by summation of quality factors. Each quality factor is calculated by a weighted formula of quality properties. This statistical analysis gives a string relation between TQI and SDI. Because of the similarity between TQI and stability metrics and since the stability metrics require human participation, the authors in this work suggest to use TQI instead of SDI.

Yu and Ramawamy [36] reintroduced an approach to represent and normalize the evolution stability of software modules. This approach is based on version differences of evolving software models by measuring the normalized distance of two versions for a module structure and module source code. For example by giving two versions V_i and V_j of a software component in a given release period let say m months, the component is said to be more stable in this period, if the measurement structure distance Di, j(source_code) or Di, j (structure) is considered to be small. A case study is applied on this model by evaluation the evolution of Linux and FreeBSD applications by selecting two versions.

Raemaekers et al. [37] highlighted backward compatibility as a very important concern to build an Application Programming Interface (API). API developers have to ensure the public interfaces are stabile because other systems are depending on theme. From this point, a way to measure interface and implementation stability of a library is introduced. Four metrics are proposed in this study to provide different insights in both implementation and interface stability. These four metrics are weighted number of removed methods, the change in metric values in existing units, the ratio between change in new and old methods and the percentage of new methods. Smaller value indicates greater stability. Moreover, they illustrate who these metrics can be used to help project managers or developers to make a decision regarding interface libraries by applying three scenarios. These metrics are theoretically evaluated by applying theme on the most frequently used Apache common libraries by selection a set of industrial systems which making use of Apache libraries. From the architectural perspective, the drawback of these metrics is the granularity level of the metric; they are not at package level (coarse-grain) but method level (fine-grain)

Ratiu et al. [38] started by defining two measurements that are used to identify which structure is considered a god class or data class. These measurements are based on object oriented design metrics and threshold for each metric. First measurement is used to identify god classes and it is based on these metrics: Access to Foreign Data (ATFD) and Weighted Method Count (WMC), Tight Class Cohesion (TCC) ,Number of Attributes (NOA). While the another measurement is used to identify data classes and it is based on these metrics: Weight of a Class (WOC), Number of Methods (NOM) ,Weighted Method Count (WMC) ,Number of Public Attributes (NOPA) and Number of Accessory Methods (NOAM). Then, they proposed two measurements that are applied on the history of a design structure. One of these measurements is used to measure the stability of a class (Stab) and another is used to measure the persistence of a design flaw (Pers). A class is considered stable with respect to measurement M version i and number of versions if there is no change in the measurement M. while a flaw is considered persistence in a class with respect to measurement M version i and number of versions if this flaw is exist in all versions of this class. Their approach is applied on three case studies: two in house projects, and one on a large open source framework. By observing the data while applying their approach, they discuss which classes, ether these classes are god classes or data classes, are considered to be harmless or harmful classes.

Bansiya et al. [39] introduced a methodology to evaluate framework architecture characteristics and stability that based on quantitative assessment on the change in framework versions using object oriented metrics. This approach consists of four steps that need to follow in order to calculate the extent-of-change measure. First step is identifying structure characteristics that evaluate the architecture of framework. There are two types of structure characteristics: static and dynamic. Example of static structure characteristics are number of classes, number of class hierarchies, number of single and multiple inheritances, and average depth and width of class inheritance hierarchies. Examples of dynamic structure characteristics are number of services a class provides, class coupling, and number of inheritance related classes. Second step is defining metrics for each one of these structure characteristics. Third step is collecting the data from the defined metrics by applying theme on a case study. Finally, for each release the extent-of-change is calculated by normalizing the values of these metrics. Once all values are normalized, the aggregate-change is calculated by summation of these values. Then the extent-of-change is calculated by taking the difference of the aggregate change value of a version i with the aggregate change value of the first version. The extent of change measure can be used as an indicator to identify the stability for a particular system structure, low number indicates high stability.

Alenezi and Khellah. [6] highlighted in their work that most of previous studies have not considered measuring the system instability at the system architecture level and most of them are focusing at the package level. From this point, they introduce a new approach to compute the instability changes for particular software architecture at a certain release while observing its evolution. Their approach is based on the instability I metric that is introduced by Martin [40] which tells how a flexible a package is able to change, the ration of the efferent coupling to the total the coupling for a package. The proposed approach is to reflect the instability change to evolution by expressing the aggregate system instability change for certain release as being composed of the average of two elements: ΔI: the amount of change in the system stability for all common packages and ASI: Aggregative System Instability for the current added packages. They illustrate how an improvement by just calculating ASI is incorrect and how the incorrectness would be solved by the proposed approach. This approach is empirically validated on two open source systems implemented in Java JEdit and PDFBox by selecting twelve releases.

Table I summarizes the discussed papers with their metrics and granularity level. The literature reveals that the available architectural stability measures have some limitations. For instance, the metrics proposed by [39] and the methodology presented by [41] consider the method/class level which is fine-grain level. Bansiya work is suitable only when having cost and economic as his backdrop [39].

B. Understandability

One desired quality of the software architecture is understandability. Understandability in the context of software architecture simply means whether system architecture is understandable to average architects. Understandability is one of the usability characteristics of the ISO/IEC SQuaRe quality standard [2]. Understandability refers to the capability that to what extent users with different backgrounds can understand the architecture. Understandability is an essential characteristic of software quality since the difficulty of understanding the software architecture system inhibits its reuse and maintenance. Understandability is the capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Several researchers have explored the relationships between several metrics and understandability. Table II summarizes their efforts, goals, and research methodology. Gupta and Chhabra [7] proposed a package coupling metric and empirically validated it against package understandability. Their study used one metric and they performed correlation analysis. They validated the package coupling metric with regard to the understandability of packages measured by assessing the effort required to fully understand the packages' functionalities. They concluded that there is a strong correlation between package coupling and the effort required to understand a package.

Elish [42] used several metrics (Size (NC), Coupling (Ca, Ce), and Stability (I, D)) and conducted a case study to correlate these metrics with package understandability of two open source software systems. The results of the study indicated statistically significant correlation between most of the metrics and understandability of a package.

Hwa et al. [9] have proposed hierarchical quality model (consist of 4 levels and 3 links to connect these levels) to assess the understandability of the modular design of an Object-Oriented software system. At the level 2 of their proposed model 6 design properties were identified that affect understandability of the modular design of a system. One of these properties is the coupling and they have found that the coupling property has a negative influence on understandability and that means the higher number of coupling the harder is to understand the system. in their proposed hierarchical quality model have found that there is a positive influence of the design size on understandability. The larger the size the harder is to understand.

Stevanetic and Zdun [8] carried a study to examine the relationships between the effort required to understand a component and component level metrics that describe component's size, complexity and coupling. Correlation, collinearity and multivariate regression analysis were performed. The results of the analysis show a statistically significant correlation between the metrics and the effort required to understand a component.

Stevanetic and Zdun [43] found that the architecture at the abstraction level that is sufficient to adequately map the systems relevant functionalities to the corresponding architectural components (i.e. , each component in the architecture corresponds to one systems relevant functionality) significantly improves the architecturelevel understanding of the software system, as compared to two other architectures that have a low and a high number o f elements . This means highly abstracted system; low in number of elements by merging systems relevant functionalities into one component would decrease the understandability of such systems. As well as, highly detailed systems; high number of elements by scattering the functionalities into several components would decrease the understandability of such systems. However, when each component in the software architecture corresponds to one of the systems functionalities would significantly improves the understandability at the architectural level.

Stevanetic et. al [44] have done a controlled experiment on 75 students of the Software Architecture lecture. The students were divided into 3 groups and each group had been given a different architectural representation of the same large system. The first architectural representation was hierarchical representation where all components at every abstraction level in the hierarchy are present. Second architectural representation concentrates on the lowest level no hierarchy used. Third architectural representation at concentrate on the highest level component in th hierarchy by does not use hierarchal abstraction. The conclusion of the experiment was that by using the hierarchical architecture would result on a better understandability at the architecture-level.

VI. SOFTWARE ARCHITECTURE MEASUREMENT VALIDATION

It is commonly accepted that a software metric should be validated following two different validations: theoretical and empirical validations. This will ensure that the metric measures the attribute that it is supposed to measure and provide evidence on the usefulness of the metric. Many existing software metrics are criticized from two standpoints, theoretically and empirically. Several researchers pointed that most software metrics were developed with no or little theoretical basis [46], [47]. Furthermore, even though some metrics are theoretically valid, they lack empirical evaluation [48].

A. Theoretical Validation

Theoretical validation makes sure that a metric is measuring what is supposed to measure. The first requirement for theoretical validation is that either the analyst has an intuitive understanding of the concept that is being measured and/or that the software engineering community has a consensual intuitive understanding of the concept. There are several frameworks for the theoretical validation of metrics. Some of them are mainly subjective, while others rely on either axiomatic or measurement theory foundations. Briand et al. [49] have discussed the application of measurement theory in software engineering. The theoretical validation is generally carried out using measurement frameworks based on property-based approaches. Property-based approaches [46] allow one to prove that a measure satisfies properties characterizing a concept (e.g., size, complexity, coupling). This approach is

TABLE I. SUMMARY OF THE DISCUSSED PAPERS

Reference	Metrics	Granularity level
Abdeen et al. [27]	Index of Inter-Package Usage (IIPU), Index of Inter-Package Extending (IIPE), Package Focus (PF), Index of Package Service Cohesion (IPSC), and Index of Package Changing Impact (IPCI)	Package, architecture
Olague et al. [32]	Software Design Instability using Entropy (SDIe) metric	Architecture
Abreu and Melo. [31]	Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (POC) and Coupling Factor (COF)	Architecture , classes
Ebad et al. [3]	Architecture Stability Metric (ASM)	Package, architecture
Ahmed et al. [22]	Shallow Semantic Similarity Metric (SSSM),Second metric is Relationship-Based Similarity Metric (RBSM)	Component, architecture
Aversano et al. [23]	Core Design Instability (CDI) and Core Call Instability (CCI)	Package, architecture
Sethi et al. [28]	Decision volatility, design volatility, impact scope, concern scope, concern overlap and independent level	Architecture
Hassaine et al. [29]	ADvISE: Architectural Decay In Software Evolution	Architecture
Li et al [25]	System Design Instability (SDI), Class Implementation Instability (CII), and System implementation Instability (SII)	Architecture , classes
Alshayeb et al. [24]	Class Stability Metric (CSM)	Class
Chidamber and Kemerer [26]	Weighted Method per Class (WMC), Depth of Inheritance of Tree (DIT), Number of Children (NOC), Coupling Between Object Class (CBO), Response for Class (RFC) and Lack of Cohesion of Metric (LOCM)	Class
Jazayeri et al. [30]	Module size, number of modules changed, number of modules added, number of modules changed in the same sequence of release, number of programs in the same version of release	Architecture
Tonu et al. [21]	Growth rate, changes rate, coupling, cohesion	Architecture, classes
Roden et al. [35]	Total Quality Index (TQI)	Package, architecture
Raemaekers et al. [37]	Weighted number of Removed Methods (WRM), the amount of Change in Existing Methods (CEM), the Ratio of Change in New to Old methods (RCNO), and the Percentage of New Methods (PNM)	Library, method
Yu and Ramawamy [36]	Distance (source code) or Distance (structure)	Component
Ratiu et al. [38]	Stability of a class (Stab) and persistence of a design flaw (Pers)	Class
Bansiya et al. [39]	The extent-of-change measure	Architecture
Alenezi and Khellah. [6]	Aggregative System Instability	Architecture

TABLE II. SUMMARY OF THE DISCUSSED PAPERS

Reference	Goal	Methodology
Gupta and Chhabra [7]	To Propose new metrics for measurement of package level coupling.	Theoretical and Empirical
Elish [42]	To explore the relationships between five package-level metric (Size, Afferent, Efferent, Instability and Distance) and the average effort required to understand a package in O.O. design.	Empirical
Stevanetic and Zdun [45]	Systematic mapping study on software metrics related to the understandability concept of such higher-level software structures with regard to their relations to the system implementation	Systematic Mapping Study
Hwa et al. [9]	To propose a hierarchical model to assess understandability of modularization in large-scale O.O. software.	Empirical
Stevanetic and Zdun [8]	To examine the relationships between the efforts required to understand a component, measured through the time that participant spent on studying a component and component level metrics that describe components size, complexity and coupling.	Experimental
Stevanetic and Zdun [43]	To examine the effect of the level of abstraction of the software architecture representation (3 levels) on the architecture-level understandability of a software system.	Experimental
Stevanetic et. al [44]	To examine the impact of hierarchies on architectural-level software understandability.	Empirical

comprehensive framework which defines the structural properties of software system mathematically which matches with the methodology of the proposed metrics in this thesis. The theory provides an empirical interpretation of the numbers (of software measures) by the hypothetical empirical relational system.

Arvanitou el al. [50] used a property-based approach to theoretically validate their new metric, which measures the coupling and class proneness to the ripple effect. Khoshkbarforoushha et al. [51] theoretically validated their new metric using a property-based framework. Tripathi and Kushwaha [52] theoretically validated their package level coupling metric using a property-based framework. Lenhard et al. [53] theoretically validated their new metric that measures installability of service orchestrations using a property-based approach. Gupta and Chhabra [7] introduced a coupling metric at the package level and theoretically validated it through property-based approach.

B. *Empirical Validation*

The purpose of empirical validation is to show the usefulness of the metric in real application using real data from software projects. The goal is to show that this new metric has a concrete value in a real settings. The empirical validation of a software metric can be done using different empirical techniques. These techniques include controlled experiments, surveys, or case studies. A controlled experiment is a rigorous and controlled study. A Survey is research performed in retrospect, when the method has been in use for a certain period of time. A Case Study is an observational study, and data are collected for a specific purpose throughout the study. Experiments provide a high level of control and are useful for validating software metrics.

Arvanitou el al. [50] compared their new measure, which measures the coupling and class proneness to the ripple effect and several coupling metrics empirically to evaluate the usefulness of their metric. Khoshkbarforoushha et al. [51] empirically validated their new metric using an experiment in how the new metric can predict design-level estimation of the potential reusability of the BPEL processes. Tripathi and Kushwaha [52] empirically validated their package level coupling metric by comparing it to other package level coupling metrics. Lenhard et al. [53] empirically validated their new metric using BPEL engines to evaluate their installability and the deployability of a set of functionally different processes. Gupta and Chhabra [7] introduced a new coupling metric at the package level and empirically validated it using package understandability.

VII. Conclusion

In this work, we have laid the foundation for researchers and practitioners to come up with better ways of measuring the software architecture quality attributes. The definition and importance of software architecture were discussed. How to evaluate these measurements were also comprehensively presented in this work. Stability and Understandability were given more focus for their importance and effect on software. Future directions include devising new metrics to measure both stability and understandability of software architecture.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[2] ISO/IEC/IEEE, "Systems and software engineering – architecture description," *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1 –46, 1 2011.

[3] S. A. Ebad and M. A. Ahmed, "Measuring stability of object-oriented software architectures," *IET Software*, vol. 9, no. 3, pp. 76–82, 2015.

[4] M. Alenezi and F. Khellah, "Architectural stability evolution in open-source systems," in *Proceedings of the The International Conference on Engineering & MIS 2015*. ACM, 2015, p. 17.

[5] 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.

[6] M. Alenezi and F. Khellah, "Evolution impact on architecture stability in open-source projects," *International Journal of Cloud Applications and Computing (IJCAC)*, vol. 5, no. 4, pp. 24–35, 2015.

[7] 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.

[8] S. Stevanetic and U. Zdun, "Exploring the relationships between the understandability of components in architectural component models and component level metrics," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, p. 32.

[9] J. Hwa, S. Lee, and Y. R. Kwon, "Hierarchical understandability assessment model for large-scale oo system," in *Asia-Pacific Software Engineering Conference, 2009. APSEC'09*. IEEE, 2009, pp. 11–18.

[10] D. Garlan, "Software architecture: a travelogue," in *Proceedings of the Future of Software Engineering*. ACM, 2014, pp. 29–39.

[11] F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2011.

[12] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, "Evolution styles: Foundations and tool support for software architecture evolution," in *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture.* IEEE, 2009, pp. 131–140.

[13] C. G. P. Bellini, R. D. C. D. F. Pereira, and J. L. Becker, "Measurement in software engineering: From the roadmap to the crossroads," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 01, pp. 37–64, 2008.

[14] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2016.

[15] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. CRC Press, 2015.

[16] K. Khosravi and Y.-G. Guéhéneuc, "On issues with software quality models," in *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 172–181.

[17] M. Alenezi and I. Abunadi, "Quality of open source systems from product metrics perspective," *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 5, p. 143, 2015.

[18] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson, "Measurement programs in software development: determinants of success," *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 863–875, 2002.

[19] I. Abunadi and M. Alenezi, "An empirical investigation of security vulnerabilities within web applications," *Journal of Universal Computer Science*, vol. 22, no. 4, pp. 537–551, 2016.

[20] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1117–1135, 2013.

[21] S. A. Tonu, A. Ashkan, and L. Tahvildari, "Evaluating architectural stability using a metric-based approach," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*. IEEE, 2006, pp. 10–pp.

[22] M. Ahmed, R. Rufai, J. AlGhamdi, and S. Khan, "Measuring architectural stability in object oriented software," *Stable Analysis Patterns: A True Problem Understanding with UML*, p. 21, 2004.

[23] L. Aversano, M. Molfetta, and M. Tortorella, "Evaluating architecture stability of software projects," in *20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 417–424.

[24] M. Alshayeb, M. Naji, M. O. Elish, and J. Al-Ghamdi, "Towards measuring object-oriented class stability," *IET Software*, vol. 5, no. 4, pp. 415–424, 2011.

[25] W. Li, L. Etzkorn, C. Davis, and J. Talburt, "An empirical study of object-oriented system evolution," *Information and Software Technology*, vol. 42, no. 6, pp. 373–381, 2000.

[26] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

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

[28] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. WICSA/ECSA 2009*. IEEE, 2009, pp. 269–272.

[29] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "Advise: Architectural decay in software evolution," in *16th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 267–276.

[30] M. Jazayeri, "On architectural stability and evolution," in *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*. Springer-Verlag, 2002, pp. 13–23.

[31] F. B. E Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Software Metrics Symposium, 1996., Proceedings of the 3rd International*. IEEE, 1996, pp. 90–99.

[32] H. M. Olague, L. H. Etzkorn, W. Li, and G. Cox, "Assessing design instability in iterative (agile) object-oriented projects," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 4, pp. 237–266, 2006.

[33] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929–944, 1995.

[34] H. Zuse, *A framework of software measurement*. Walter de Gruyter, 1998.

[35] P. L. Roden, S. Virani, L. H. Etzkorn, and S. Messimer, "An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2007*. IEEE, 2007, pp. 171–179.

[36] L. Yu and S. Ramaswamy, "Measuring the evolutionary stability of software systems: case studies of linux and freebsd," *IET Software*, vol. 3, no. 1, pp. 26–36, 2009.

[37] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *28th IEEE International Conference on Software Maintenance (ICSM), 2012*. IEEE, 2012, pp. 378–387.

[38] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings. Eighth European Conference on Software Maintenance and Reengineering, 2004.* IEEE, 2004, pp. 223–232.

[39] J. Bansiya, "Evaluating framework architecture structural stability," *ACM Computing Surveys (CSUR)*, vol. 32, no. 1es, p. 18, 2000.

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

[41] M. Alshayeb, "The impact of refactoring on class and architecture stability," *Journal of Research and Practice in Information Technology*, vol. 43, no. 4, p. 269, 2011.

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

[43] S. Stevanetic and U. Zdun, "Empirical study on the effect of a software architecture representation's abstraction level on the architecture-level software understanding," in *14th International Conference on Quality Software (QSIC), 2014*. IEEE, 2014, pp. 359–364.

[44] S. Stevanetic, M. A. Javed, and U. Zdun, "The impact of hierarchies on the architecture-level software understandability-a controlled experiment," in *24th Australasian Software Engineering Conference (ASWEC), 2015*. IEEE, 2015, pp. 98–107.

[45] S. Stevanetic and U. Zdun, "Software metrics for measuring the understandability of architectural structures: a systematic mapping study," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2015, p. 21.

[46] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[47] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Transactions on Software Engineering*, vol. 20, no. 3, pp. 199–206, 1994.

[48] M. Alshayeb and W. Li, "An empirical validation of object-oriented metrics in two different iterative software processes," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1043–1049, 2003.

[49] L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," *Empirical Software Engineering*, vol. 1, no. 1, pp. 61–88, 1996.

[50] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Introducing a ripple effect measure: A theoretical and empirical validation," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[51] A. Khoshkbarforoushha, P. Jamshidi, M. F. Gholami, L. Wang, and R. Ranjan, "Metrics for bpel process reusability analysis in a workflow system," *IEEE Systems Journal*, vol. 10, no. 1, pp. 36–45, 2016.

[52] A. Tripathi and D. Kushwaha, "A metric for package level coupling," *CSI Transactions on ICT*, vol. 2, no. 4, pp. 217–233, 2015.

[53] J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the installability of service orchestrations using the square method," in *IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2013, pp. 118–125.