# Measuring Software Architecture Stability Evolution in Object-Oriented Open Source Systems

Hassan Almousa and Mamdouh Alenezi

College of Computer and Information Sciences, Prince Sultan University,
11586 Riyadh, Saudi Arabia

**Abstract:** Stability is the capability of a software artefact to stay intact while adapting to new changes and requirements. Software architecture demonstrates the structure of a software system which can be presented as software components and their interconnections. Measuring the stability of software architecture is an important endeavor that can help developers or project managers to make them aware about the situation of the software being developed. Many software metrics have been introduced to measure the stability of the software architecture. Some of these metrics measure software architecture at package-level while other ones measure it at class-level. The principal goal of this study is to come up with a new easy mechanism to measure the stability evolution in open source Java systems. Five different systems with ten versions are analyzed with respect to the new suggested mechanism of measuring stability evolution.

**Keywords:** Software architecture, maintainabiliy, stability, software metric, measurement

## INTRODUCTION

Measurement in software engineering is a central feature to assess software quality characteristics such as maintainability, reliability and portability. Software measurement (Ruchika, 2015) is not only about assessing software products but it can be used to assess the software development process. These measurements can contribute heavily in understanding software and processes in many ways for example by using software metrics, project managers will be able to know about the software status and evaluate the quality of various artifacts produced throughout software development. Software metrics are needed to measure various software attributes at different phases of the software development. Software metrics can be helpful in effectively measure different stages of the software development life cycle. Continuously measure software characteristic and development process usually supply timely management useful information about improving both process product. In this study, a new mechanism will be designed to evaluate a sub quality characteristic which is the software stability and then a case study will be discussed to see how the new mechanism will evaluate a set of systems.

**Software architecture:** Software structure is usually designed right after collecting and studying the software requirements. Throughout this stage the software is built with regards of components and interconnections that tie these components with each other (Ahmed *et al.*, 2003). This structure will demonstrate the architecture for specific software. Moreover, the software system architecture is the initial design piece that reports quality objective such as maintainability, stability, modifiability and performance (Tonu *et al.*, 2006). Software architecture divides the system structure with regards of components and their connections from the detailed design of these individual components. Software architecture can be defined at two levels, high level and lower level. At high level, software is decomposed into subsystem where at lower level, a software is decomposed into modules and components. Interfaces are used to connect subsystems or components with each other (Gomaa, 2011). Software architecture signifies the critical design decisions that are toughest to change and decide the overall system proprieties (Olague *et al.*, 2006). These decisions should be before working on the detailed aspects of the system. These architecture decisions not only made at the component level but they usually include the overall system components and constrains. After deciding about all architecture decisions, the team can work in designing and delivering individual components (Jazayeri, 2002).

Software architecture is not just how software is constructed but it guides the software evolution (Jazayeri,

**Corresponding Author:** Hassan Almousa, College of Computer and Information Sciences, Prince Sultan University,
11586 Riyadh, Saudi Arabia

2002). Now a days, it is impossible to have a software product remains as it was since it is being developed. The software will evolve continuously, requiring continuous development and maintenance (Hassaine *et al.*, 2012). Changes come throughout software's life in term of adding new features, modifying existing features, removing some features or fixing software defects. All these actions that may accrue on the software architecture can be referred as signs of aging. Signs of aging can negatively affect the software architecture and make it deviated from the original architecture. This deviation could harm the software architecture and make it complex and hard to maintain. By observing a software evolution, information from the software architecture will be obtained by evaluating this architecture. Therefore, software metrics are needed to evaluate and assess to avoid the negative impacts of signs of aging. Quantitative information that obtained from the software metrics can help developers to build systems with better quality.

**Stability:** The main goal of evaluating software architecture is to validate the software architecture using systematic procedures (Tonu *et al.*, 2006). This evaluation goal is to make sure that the architecture under examination fulfil one or more of software quality characteristics. According to ISO/IC square quality standards, quality characteristics consist of four characteristics: functionality, usability, reliability, efficiency, portability and maintainability (Chidamber and Kemerer, 1994). Maintainability is defined as the capability of a software product to be modified. Maintainability is ether categorized into a set of sub characteristics: stability, analyzability, changeability and testability. At the architecture level, these characteristics will be refined into coupling and modularity to be able to be evaluated and assessed (Yu and Ramaswamy, 2009). As mentioned before stability is a sub characteristic under maintainability and it is defined as the capacity of the software product to stay unaffected when facing new requirements and/or changing in the environment. Stability evolution is measured by computing the differences between the stability of two versions while the software is evolving. On other hand, it is impossible to keep a software product unchanged. Then, what the software is supposed to do? The answer is these new changes have to be accommodated in the software architecture and if not these changes will lead to the degradation of the usefulness of the software product (Ebad and Ahmed, 2015). From what have been said before, it noticeable that stability is a primary criterion for evaluating an architecture which can be seen as an indicator for software maturity (Hassaine *et al.*, 2012;

Roden *et al.*, 2007). If a software product has some components and these components are unstable, the software architecture will require high maintenance cost and effort (Alenezi and Khellah, 2015). On other hand, once an architecture for a software component is recognized as a stabile component, this component can be considered as a reusable component.

**Litreruare review:** This study browses different attempts in the literature to measure the stability of specific software. Aversano evaluated the software architecture for a set of open source software projects. 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 were proposed to measure the stability of each release, Core Design Instability (CDI) and Core Call Instability (CCI). Both metrics provided 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.* (2011) mentioned that none of the existing measures have included all class aspects such as class relationships, attributes and methods. First, they identified all properties that affect the class stability; 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, they proposed 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 the summation of all these properties is divided by the number of the properties which is eight. The result their empirical study indicated that their metric is highly negatively correlated with maintenance effort. Li *et al.* (2000) proposed new metrics to measure the stability of software designs. Researchers in Li *et al.* (2000) highlighted that metrics that are discovered by Chidambe and Kemerer (1994) can't measure change in the class name, class number and class inheritance relations. Tackling this deficiency of C and K metrics, they proposed three new metrics: System Design Instability (SDI), Class Implementation Instability (CII) and System Implementation Instability (SII). The main purpose was to justify how the information that is gathered from theses metrics can help project managers to modify software

project plan. Their experimental analysis showed that SDI and CII can measure object oriented aspects that are different from the aspects that are measured by C and K metrics.

Abdeen *et al.* (2011) proposed a number of coupling and cohesion metrics that evaluate packages modularization in legacy object-oriented systems. These metrics are 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). They defined these metrics 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 respect to inter-class dependencies: method call and inheritance relationships. They validated these against the mathematical properties that have to be existed in any cohesion or coupling metric.

Jazayeri (2002) did retrospective analysis to assess the architecture stability for twenty releases of telecommunication software. The evaluation was intended to help project managers to predict how the future of the architecture will look like. They used simple metrics, 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 to observe the effect of evolution on pair of releases.

Abreu and Melo (1996) evaluated the impact of object oriented design on software quality characteristics such as defect density, failure density and normalized rework. They used a set of object oriented design metrics called MOOD. These metrics are 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, they developed a predictive model. The results show that the design alternatives may have a strong influence on resulting quality.

Olague *et al.* (2006) utilized entropy to reduce spikes in the original SDI metric that is produced by Li *et al.* (2000) and proposed the new SDIe metric. They discussed the dynamic nature of the agile development process that could obscure an analysis of software stability. Furthermore, this study noted that the SDIe metric is easier than the original SDI metric in computing the stability. The justification is the fact that 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 versions. SDIe metric is theoretically investigated and validated using the Kitchenham criteria and the Zuse (1997) requirements (Chidamber and Kemerer, 1994) 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 (1994).

Tonu *et al.* (2006) proposed an approach that evaluates stability for a particular software architecture. The approach is based on analyzing the changes in the software's aspects form one release to another. Software aspects can be structural, behavior or economical, their focus was on the structural aspects only. Growth rate, changes rate, coupling, cohesion are the measures that are applied in their retrospective analysis. Then, evolution sensitivity and evolution critical parts are identified by observing how the subsystems are interconnected. This approach is empirically evaluated on two spreadsheet applications by selecting nine releases for each application.

Ratiu started by defining two threshold measurements that are used to identify which structure is considered a god class or data class. 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 the other one 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. 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 discussed whither classes are god classes or data classes.

Bansiya (2000) introduced a framework to evaluate architecture stability that is based on quantitative assessment on the changes in versions using object oriented metrics. The framework consists of four steps 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.

## MATERIALS AND METHODS

**Designing stability metric:** Object oriented design concepts are very important factors that contribute in the software development. The reason behind that is these concepts address fundamental concerns about software adaption and evolution. Information hiding is one of the object oriented concepts and it is used to decide which information should be visible and which information should be hidden. Inheritance is another concept of object oriented that is used for sharing and reusing properties between classes. Concurrent processing is another concept of object oriented and it describes how software's objects will interact with each other by invoking classes methods (Gomma, 2011). These concepts can be evaluated for particular software in order to recognize the software architecture. In this study, the focus will be on the concepts that affect the software architecture in term of concurrent processing. It has been recognize from the literature review that have been reviewed in the previous section that most infusing attributes which are used in order to determine the stability for the software architecture are size, coupling and cohesion. Some researches use coupling and cohesion at the class level while other use coupling and cohesion at architecture level by considering the software packages. Furthermore, most of object oriented metrics that measure the software architecture for particular system are class-level metrics (Alshayeb *et al.*, 2011; Chidamber and Kemerer, 1994). In this study we are aiming to introduce a metric that is used to measure the stability of the software architecture based on these three attributes; size, coupling and cohesion.

**Identifying software architecture attributes:** Software architecture attributes that will be used to measure the stability for the software architecture will be discussed in this section. Each one of these attributes (Size, Cohesion and Coupling) nominated to contribute in designing the new stability metric. Size provides high view about the amount of functionalities that have been developed for a particular system during its life cycle. These functionalities will be represented at the end for each class in the system either by adding new line of codes or methods or imported packages or adding new interfaces. Therefore, size will be used as a good indicator to know if new functionalities have been added in a system. This indicator can be used as a measure to recognize the stability for a particular system and many researches used this measure to study the stability in their studies like, (Olague *et al.*, 2006; Bansiya, 2000). Coupling is the degree that can be pointed out in order to know how a class has dependences among other classes within the software architecture. In a brief way, coupling illustrates the relationships of a class with other classes. There is a contradiction between coupling and software engineering factors like maintainability, testability and fault-proneness.

Let's consider the case of maintainability to present how coupling will be accrued. If a developer edits the software architecture by making a modification in a class, this change may require a modification on one of classes that connect with this class. It is clear from the above that coupling can be used as good indicator to recognize the change in the software architecture by observing coupling between the software versions. This reorganization will tack place by measuring the coupling for a class during the software life cycle. There are many metrics have been introduced to measure the coupling at class-level. These metrics are coupling between objects, message passing coupling, efferent coupling, afferent coupling, response for class and local methods calls. Many researches used coupling as a measure in their study like (Li *et al.*, 2000; Abdeen *et al.*, 2011; Tonu *et al.*, 2006; Roden *et al.*, 2007; Alenezi and Khellah, 2015).

Cohesion is used to identify the degree of how class's elements are related to each other. In another way class cohesion describes the relationships between the class attributes and methods. A class will be pointed out as strong cohesion class, if there is a strong overlap between its attributes and its methods. Moreover, designing strong class cohesion is seen as a good quality attribute when evaluating the software architecture. On the other hand, there is a contradiction between designing high cohesive and complex class. Because designing very high cohesive class will lead this class to some degree of

Table 1: Class-level metrics

| Group | Metric | Detention |
|---|---|---|
| Size | NLOC | Counts number of lines of code in a class |
| | PACK | Counts number of packages that are imported in a class |
| | INTER | Counts number of Interfaces that are implemented in a class |
| | No.methods | Counts number of methods that are defined in a class |
| Cohesion | LCOM 1 | Lack of Cohesion of Methods and it shows how the methods of a class are not related to each other to achieve the aims of the class |
| | LCOM 2 | Lack of Cohesion of Methods and it shows how the methods of a class are not related to each other to achieve the aims of the class |
| | COH | Cohesion and it shows how well the methods of a class co-operate to achieve the aims of the class |
| Coupling | CBO | Coupling between objects and it counts the number of classes that are coupled to a particular class |
| | FAN-IN | Afferent coupling and it counts the number of other classes that reference a class |
| | FAN-OUT | Efferent coupling and it counts the number of other classes that referenced by a class |
| | RFC | Response for class and it is defined as a set of methods that can be executed in response to a message received by another class |
| | MPC | Message passing coupling and it counts the numbers of messages passing among objects of a class |
| | LMC | Number of local method called in a class |

complexity. The optimal way to avoid this kind of concern is having threshold for cohesion while designing the software architecture. The previous sentences were giving overview about the class cohesion and it benefits and drawback and it is clear that cohesion can be used as an attribute to recognize the stability for the software architecture when a version of system compare with other system versions. Many researches used cohesion as a measure in their study like (Li *et al.*, 2000; Olague *et al.*, 2006; Tonu *et al.*, 2006; Roden *et al.*, 2007).

**Selecting the software metrics for size, cohesion and coupling attribute:**. Software quality characteristics can't be evaluated directly. They have to be refined into sub-characteristics. These sub-characteristics will be refined into attributes. The refinement attributes will be measured by using some metrics. In our case, the refinement attributes are size, coupling and cohesion. In this section, the software metrics for each one of refinement attributes that have been discussed previously will be identified and defined. In order to identify the metrics that will be nominated to be helped in designing the new metric, we used a software metrics tool to identify all metrics that give measurement at the class-level. Only metrics that measure class-level with concerning size, coupling and cohesion are selected. Table 1 groups the metrics by the attributes and each metric in these groups is defined. However, not all metrics that are identified in Table 1 will be selected to design the new metrics. The reason behind that is the attention will be given only to metrics that their results are not correlated when they are calculated for software. If there are two metrics their results are strongly correlated, only one of theme will be selected. Spearman correlation analysis will be used to identify the metrics that are strongly correlated in each group (size, cohesion and coupling) (Quah and Thwin, 2003).

**Designing the stability metric:** Stability is the ability of a class to remain unchanged when new requirements are introduced in a system. On the other hand, preventing a system from adding new features something is hardly to

achieve. The system has to accommodate any new requirement to get the end user's satisfaction. If this not happened, the end user's will look to the systems competitors. The stability of a system can be measured by observing the system evolution. In our case the attributes that have been selected will be observed. This observation will take place by recognizing the difference between the system's versions (Yu and Ramaswama, 2009). One of the measurements proposed by Ratiu is the stability of a class (Stab). They mentioned a class A is considered stable with respect to measurement M version i and number of versions N if there is no change in the measurement M as shows in Eq. 1 and 2:

$$\text{stab}_i(C,M) = 1, M_i(C) - M_{i-1}(C) = 0 \qquad (1)$$

$$\text{stab}_i(C,M) = 1, M_i(C) - M_{i-1}(C) \neq 0 \qquad (2)$$

Once the stability of a class is identified in version i, the stability of this class in version N is calculated as the number of versions in which a class was changed over the total number of versions. Equation 3 describes how the stability will be calculated:

$$\text{stab}_{i\ldots 1} = \frac{\sum_{i=2}^{n} \text{stab}_i(C,M)}{n-1} \qquad (3)$$

Equation 3 measure the stability at class-level but in this study the concern is paid to measure the stability at architecture-level. Moreover, Eq. 3 identify the stability based on one method but in this study there is three attributes (size, cohesion and coupling) and each one of these attributes has a set of metrics. In this study, Eq. 1 and 3 will be modified to measure the stability for the software architecture for a system. Instead of calculating the stability for a system by considering each class in the system and using only a single metric, the stability for the whole system will be measured by considering more than one metrics. The following steps show how the stability for the software architecture will be measured.

**Step 1:** Extracting class-level metric using software metrics tool. The value of the class-level metrics for the selected attributes will be calculated by extracted their data using a software metrics tool for each class in the system.

**Step 2:** Calculating the average for the class-level metrics. The average will be calculated for each metric in the system after the metrics values being extracted.

**Step 3:** Calculating the aggregated metric. The average of all metrics will be aggregated in one metric to be used to compare its value with the system's versions. Equation 4 describes how all metrics M in version i will be combined:

$$\text{aggregated metric}_i = \sum_{j=0}^{j=n} AVG(M)_i \qquad (4)$$

**Step 4:** Do comparison between the system versions. Step 1-3 will be repeated for each version i for the selected system to obtained the aggregated metric. Each version i will be considered stable with respect to the aggregated metric version i and number of versions N if there is no change in the aggregated metric. Metric (Eq. 5) and metric (Eq. 6) illustrate how the aggregated metric will be compared between the system versions:

$$\text{stab}_i = 1, \text{Aggregated Metric}_i - \qquad (5)$$
$$\text{Aggregated Metric}_{i-1} = 0$$

$$\text{stab}_i = 0, \text{Aggregated Metric}_i - \qquad (6)$$
$$\text{Aggregated Metric}_{i-1} \neq 0$$

**Step 5:** Measuring the stability for the selected system. The stability for the selected system will be measured by evaluating the $\text{stab}_i$ which is described in step 4, between a pair of versions for N versions. After the stab evaluated for each version i, stab i will be aggregated for all Versions N then it will be divided by total number of versions N-1. Equation 7 shows how the system stability will be measured:

$$\text{System stab}_{i...1} = \frac{\sum_{i=2}^{n} \text{stab}_i}{n-1} \qquad (7)$$

## RESULTS AND DISCUSSION

In this research, a case study will be presented in which five object-oriented java systems are selected. All these systems are open source systems and they downloaded from Sourceforge library. Since our stability metric at its early stage requires at least a pair of versions, it is mandatory to have a large number of versions to be trained. From this concern we agreed to download ten versions for each one of the selected systems. One of the crucial requirements to start applying this study is to choose a software metric tool to extract the results of class-level metrics. At the first of this section we will talk about the benefits of open source systems. Then, the selected software metric tool will be discussed. After that, a summary about the selected systems will be given. Finally, the results of applying our stability metric will be shown.

**Open source systems:** Open source systems have received a lot of attention to be used by practitioners in these days. The reason behind that is the availability of a large number of systems in different domains. A researcher downloads an open source system for applying the required analysis form his or her study. A programmer modifies an open source system to develop a particular need. It is obvious from the above that the availability of open source system will reduce cost, effort and time to obtain a system meets a desired need (Ding *et al.*, 2014). There are many open source libraries are available on the internet. These libraries are sourceforge, Gethub, Google code and Tigers. In this study sourceforge will be used to download five systems with ten versions. In order to evaluate our stability metric, we realized that it is necessary to put some criteria to help us in selecting an open source system. Following are the criteria that we stick on to select a set of open source systems:

- All selected systems have to be written in Java
- All selected systems have to be object-oriented
- The selected systems have to be from different domains
- There are multiple versions for the selected systems.
- There is a variation in the number of classes for the selected systems

Table 2 gives a brief summary for the selected open source systems.

**Software metric tool:** One of the fundamental requirements for out stability metric is to have a tool provides many metrics at class level for the attributes that are selected to design our stability metric. Another requirement is the tool should be able to analysis systems that are written in Java. During our search for the most appropriate tools that accommodate our needs, we found many software metric tools. Some of these tools provide

Table 2: Selected open source system

| System | Domain | Description |
|---|---|---|
| SQSim | Scientific/Engineering | This system used to model simple probabilistic queue, dynamic, factories, business, services and other |
| BarCode generator | Development/Dynamic content | This system used to generate barcode of different types like EANB, UPCA, etc. |
| Ezbilling | Finance | This system used to manage the invoices that will be issued |
| Catan | Game | This system allows anyone to play board game against the computer |
| EuroBudget | Accounting | This system is a free personal accounting tool |

Table 3: Class-level metrics grouped by size, cohesion and coupling

| Class name | Size | | | | Cohesion | | | Coupling | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NLOC | No methods | INTR | PACK | LCOM | LCOM2 | COH | RFC | CBO | MPC | F-IN | FOUT | LMC |
| BarSet | 50 | 6 | 0 | 0 | 0.2 | 0 | 0.33 | 6 | 24 | 0 | 24 | 0 | 1 |
| BarcodeEncoder | 4 | 2 | 0 | 0 | 0 | 2 | 0 | 2 | 16 | 0 | 14 | 2 | 0 |
| BarcodePainter | 3 | 1 | 0 | 2 | 0 | 1 | 0 | 1 | 8 | 0 | 7 | 1 | 0 |
| BaselineTextPainter | 24 | 3 | 1 | 5 | 1 | 4 | 0.33 | 3 | 5 | 0 | 4 | 1 | 0 |
| CircularPainter | 55 | 3 | 1 | 8 | 1 | 4 | 0.33 | 3 | 3 | 0 | 1 | 2 | 0 |
| CodeabarEncoder | 84 | 10 | 1 | 0 | 0.01 | 30 | 0.08 | 11 | 7 | 1 | 4 | 3 | 1 |
| Code11Encoder | 59 | 6 | 1 | 0 | 0.1 | 14 | 0.17 | 7 | 6 | 1 | 3 | 3 | 1 |
| Code12Encoder | 186 | 7 | 1 | 1 | 0.02 | 17 | 0.11 | 8 | 7 | 1 | 4 | 3 | 2 |
| Code39Encoder | 100 | 7 | 1 | 0 | 0.12 | 17 | 0.14 | 8 | 8 | 1 | 5 | 3 | 2 |
| Code39ExtEncoder | 45 | 6 | 0 | 0 | 0.4 | 14 | 0.17 | 7 | 7 | 1 | 3 | 4 | 1 |
| Code93Encoder | 127 | 7 | 1 | 0 | 0.12 | 17 | 0.14 | 8 | 8 | 1 | 5 | 3 | 2 |
| Code93ExtEncoder | 51 | 6 | 0 | 0 | 0.4 | 14 | 0.17 | 7 | 7 | 1 | 3 | 4 | 1 |
| DirectGif89Frame | 29 | 3 | 0 | 3 | 0 | 0 | 1 | 3 | 3 | 0 | 2 | 1 | 0 |
| EAN13Encoder | 46 | 4 | 0 | 0 | 0.33 | 8 | 0.25 | 5 | 8 | 1 | 5 | 3 | 0 |
| EAN13TextPainter | 35 | 3 | 1 | 5 | 1 | 4 | 0.33 | 3 | 5 | 0 | 4 | 1 | 0 |
| EAN8Encoder | 36 | 4 | 0 | 0 | 1 | 8 | 0.25 | 4 | 7 | 0 | 4 | 3 | 0 |
| EAN8TextPainter | 34 | 3 | 1 | 5 | 1 | 4 | 0.33 | 3 | 5 | 0 | 4 | 1 | 0 |
| EANEncoder | 35 | 2 | 1 | 0 | 4 | 2 | 0 | 3 | 7 | 1 | 4 | 3 | 1 |
| Gif89Encoder | 163 | 22 | 0 | 11 | 0.02 | 189 | 0.18 | 24 | 7 | 2 | 2 | 5 | 7 |
| Gif89Frame | 67 | 12 | 0 | 3 | 0.04 | 68 | 0.13 | 13 | 6 | 1 | 4 | 2 | 2 |
| GifColourTable | 94 | 11 | 0 | 0 | 0.06 | 40 | 0.25 | 11 | 5 | 0 | 1 | 4 | 3 |
| GifPixelsEncoder | 240 | 12 | 0 | 0 | 0.14 | 66 | 0.17 | 12 | 1 | 0 | 1 | 0 | 10 |
| HeightCodedPainter | 34 | 3 | 1 | 4 | 1 | 4 | 0.33 | 3 | 6 | 0 | 4 | 2 | 0 |
| ImageUtil | 33 | 4 | 0 | 7 | 0 | 8 | 0.25 | 4 | 3 | 0 | 1 | 2 | 2 |
| IndexGif89Frame | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 3 | 0 | 2 | 1 | 0 |
| Interleaved2of5Encoder | 38 | 4 | 0 | 0 | 0.11 | 8 | 0.25 | 5 | 7 | 1 | 4 | 3 | 0 |
| InvalidAttributeException | 15 | 4 | 0 | 0 | 1 | 4 | 0 | 4 | 23 | 0 | 23 | 0 | 0 |
| JBarcode | 121 | 23 | 0 | 9 | 0.03 | 244 | 0.12 | 23 | 9 | 0 | 4 | 5 | 4 |
| JBarcodeComponent | 170 | 35 | 1 | 10 | 0 | 0 | 0.2 | 36 | 5 | 0 | 1 | 5 | 1 |

many metrics for the selected attributes while other provides few metrics. Examples of these tools are JHWAK, Source Code Metric Plugin and Simple Code Metric Plugin. In this study we used JHAWK tool to extract the results of class-level metrics. JHAWK is a standalone application provides plenty of metrics at different levels like package level, class-level and method-level. One of the most important features that attract us to select JHAWK is the ability to choose only the metrics that we want to be extracted when analyzing a system. These metrics can be managed through preferences in tool setting. JHAWK has the ability to extract the results of the metrics in different formats like CSV, HTML and XML

**Appling the stability metric:** Our stability metric needs to be validated by selecting a set of open source systems. The main objective from this validation is to see how our stability metric will evaluate the selected systems. As mentioned previously not all metrics that contribute to

measure the size, cohesion and coupling will be nominated to design our stability metric. Therefore, spearman correlation analysis is applied to discard the metrics that are strongly correlated. All size, cohesion and coupling metric's results for all system's versions are extracted. Then, the related metrics are grouped with each other. For examples, coupling metrics will be combined in one group. Table 3 shows how metrics and their results are grouped by size, cohesion and coupling attributes. A spearman correlation analysis will be applied in each one of these groups. Table 4-6 show the spearman correlation analysis results for size, cohesion and coupling attributes, respectively.

When we see the results for spearman correlation analysis results for size group, we realized that there is a strong correlation between NLOC and number of methods. Therefore, only one of them will be chosen and in our case study we select NLOC. When we see the results for spearman correlation analysis results for cohesion group, we realized all metrics are not correlated

Table 4: Spearman correlation analysis results for size metrics

| Variables | Size | | | |
| | NLOC | No. Meth | INTR | PACK |
| --- | --- | --- | --- | --- |
| NLOC | 1 | | | |
| No. Meth | 0.684145 | 1 | | |
| INTR | 0.144081 | 0.309831 | 1 | |
| PACK | -0.0677 | -0.12603 | -0.06122 | 1 |

Table 5: Spearman correlation analysis results for cohesion metrics

| Variables | Cohesion | | |
| | LCOM | LCOM2 | COH |
| --- | --- | --- | --- |
| LCOM | 1 | | |
| LCOM2 | -0.03591 | 1 | |
| COH | 0.263737 | -0.03994 | 1 |

Table 6: Spearman correlation analysis results for coupling metrics

| Variables | Coupling | | | | | |
| | RFC | CBO | MPC | F-IN | FOUT | LMC |
| --- | --- | --- | --- | --- | --- | --- |
| RFC | 1 | | | | | |
| CBO | 0.712879 | 1 | | | | |
| MPC | 0.46271 | 0.567566 | 1 | | | |
| F-IN | 0.589835 | 0.869623 | 0.5455 | 1 | | |
| FOUT | 0.577303 | 0.726984 | 0.333192 | 0.301873 | 1 | |
| LMC | 0.867593 | 0.655508 | 0.354002 | 0.530947 | 0.558953 | 1 |

Table 7: Stability result for SQSim

| Variables | SQSim |
| --- | --- |
| 1 | 66.77478 |
| 2 | 66.75304 (0) |
| 3 | 66.75304 (1) |
| 4 | 66.75304 (1) |
| 5 | 66.75304 (1) |
| 6 | 66.84 (0) |
| 7 | 66.75304 (0) |
| 8 | 66.75304 (1) |
| 9 | 66.75304 (1) |
| 10 | 66.75304 (1) |

Stability: 67%

Table 8: Stability result for barcode generator

| Variables | BarCodeGenerator |
| --- | --- |
| 1 | 112.8988 |
| 2 | 112.8988 (0) |
| 3 | 112.8988 (1) |
| 4 | 112.8988 (1) |
| 5 | 112.8988 (1) |
| 6 | 112.8988 (1) |
| 7 | 112.8988 (1) |
| 8 | 112.8988 (1) |
| 9 | 112.8988 (1) |
| 10 | 112.8988 (1) |

Stability: 89%

with each other. Therefore, all of them are nominated to contribute in designing our stability metric. Finally, spearman correlation analysis results for coupling metric shows all coupling metrics are strongly correlated. Therefore, selection one of them is enough and we select Coupling Between Objects (CBO). The reason for selecting this metric is CBO includes all dependences ingoing and outgoing (Bakar, 2016).

Table 9: Stability result for Catan

| Variables | Caton |
| --- | --- |
| 1 | 92.3574 |
| 2 | 90.57128 (0) |
| 3 | 90.57128 (1) |
| 4 | 90.56011 (0) |
| 5 | 90.57128 (0) |
| 6 | 90.57128 (1) |
| 7 | 90.57128 (1) |
| 8 | 90.57128 (1) |
| 9 | 90.57128 (1) |
| 10 | 90.57128 (1) |

Now size, cohesion and coupling metrics that will be nominated to design our stability metrics are identified. These metrics are NLOC, LCOM, LCOM 2, COH and CBO. We are ready now to apply the steps in following for each one of the selected systems to measure its architecture.

**Step 1:** Extracting class-level metric.

**Step 2:** Calculating the average for the class-level metrics.

**Step 3:** Calculating the aggregated metric.

**Step 4:** Do comparison between the system versions.

**Step 5:** Measuring the stability for the selected system.

Table 7 shows the result for our stability metric after applying the previous steps on SQSim. Our stability metric evaluates the architecture for this system and it gives 76%. We compare a pair of subsequence versions and we realized the aggregated metric results for V2, 6 and 7 are different when they compare with V1, 5 and 6. Table 8 shows the result for our stability metric after applying the previous steps on Barcode Generator. Our stability metric evaluates the architecture for this system and it gives 89%. We compare a pair of subsequence versions and we realized the aggregated metric results for V2 is different when it compared with V1.

Table 9 shows the result for our stability metric after applying the previous steps on Catan. Our stability metric evaluates the architecture for this system and it gives 67%. We compare a pair of subsequence versions and we realized the aggregated metric results for V3-V5 are different when they compare with V2-V4.

Table 10 shows the result for our stability metric after applying the previous steps on EuroBudget. Our stability metric evaluates the architecture for this system and it gives 22%. We compare a pair of subsequence versions and we realized the aggregated metric results for V2, 3, 4, 5, 6, 7 and 10 are different when they compare with V1, 2, 3, 4, 5, 6 and 9. Table 11 and 12 shows the result for our stability metric after applying the previous steps on

Table 10: Stability result for EuroBudget

| Variables | EuroBudget |
|---|---|
| 1 | 92.0466 |
| 2 | 91.39917 (0) |
| 3 | 91.29543 (0) |
| 4 | 93.62827 (0) |
| 5 | 88.73376 (0) |
| 6 | 88.72273 (0) |
| 7 | 88.70638 (0) |
| 8 | 88.70638 (1) |
| 9 | 88.70638 (1) |
| 10 | 88.72273 (0) |

Stability: 22%

Table 11: Stability result for Ezbilling

| Variables | Ezbilling |
|---|---|
| 1 | 58.16569 |
| 2 | 58.16569 (1) |
| 3 | 58.16569 (1) |
| 4 | 58.16569 (1) |
| 5 | 58.16569 (1) |
| 6 | 58.16569 (1) |
| 7 | 58.16569 (1) |
| 8 | 58.16569 (1) |
| 9 | 58.19569 (1) |
| 10 | 58.19569 (1) |

Stability: 100%

Table 12: Stability result for all systems

| System | Stability (%) |
|---|---|
| EuroBudget | 22 |
| BarCodeGenerator | 89 |
| Catan | 67 |
| Ezbilling | 100 |
| SQSim | 67 |

EzBilling. Our stability metric evaluates the architecture for this system and it gives 100%. We compare a pair of subsequence versions and we realized the aggregated metric results for all versions are the same.

## CONCLUSION

In conclusion, the stability metric that will be used to measure the architecture for a system is designed. There are five steps have to followed in order to calculate the stability for a selected system. These steps are extracting class-level metric using software metrics tool, calculating the average for the class-level metric, calculating the aggregated metric, do comparison between the system versions and measuring the stability for the selected system. We have demonstrated our stability metric by applying it on five systems with ten versions. In our case study, we started by talking about the benefits of open source systems and we ended by showing the results our stability metric. The stability results for each one of the selected system. The stability metrics evaluate the selected systems and based on it is results we realized that the most stabile system is Ezbilling while Eurobudget system is instable. The stability for Catan and SQSim are

the same. Without having a large set of versions for a particular system, our stability metric will not be able to work in order to give a useful stability result.

## REFERENCES

Abdeen, H., S. Ducasse and H. Sahraoui, 2011. Modularization metrics: Assessing package organization in legacy large object-oriented software. Proceedings of the 2011 18th Working Conference on Reverse Engineering, October 17-20, 2011, IEEE, Lille, France, ISBN:978-1-4577-1948-6, pp: 394-398.

Abreu, F.B. and W. Melo, 1996. Evaluating the impact of object-oriented design on software quality. Proceedings of the 3rd International Symposium on Software Metrics: From Measurement To Empirical Results, March 25-26, 1996, IEEE, Berlin, Germany, pp: 90-99.

Ahmed, M., R. Rufai, J. AlGhamdi and S. Khan, 2003. Measuring architectural stability in object oriented software. Proceedings of the UML Conference on Stable Analysis Patterns: A True Problem Understanding with UML, October 20, 2003, UML, San Francisco, California, USA., pp: 21-28.

Alenezi, M. and F. Khellah, 2015. Evolution impact on architecture stability in open-source projects. Int. J. Cloud Appl. Comput., 5: 24-35.

Alshayeb, M., M. Naji, M.O. Elish and A.J. Ghamdi, 2011. Towards measuring object-oriented class stability. IET. Software, 5: 415-424.

Bakar, N.S.A.A., 2016. The analysis of object-oriented metrics in C++ programs. Lect. Notes Software Eng., 4: 48-48.

Bansiya, J., 2000. Evaluating framework architecture structural stability. ACM. Comput. Surv., Vol. 32, 10.1145/351936.351954

Chidamber, S.R. and C.K. Kemerer, 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng., 20: 476-493.

Ding, W., P. Liang, A. Tang, V.H. Vliet and M. Shahin, 2014. How do open source communities document software architecture: An exploratory survey. Proceedings of the 2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS), August 4-7, 2014, IEEE, Wuhan, China, ISBN:978-1-4799-5482-7, pp: 136-145.

Ebad, S.A. and M.A. Ahmed, 2015. Measuring stability of object-oriented software architectures. IET. Software, 9: 76-82.

Gomaa, H., 2011. Software Modeling and Design: UML, Use Cases, Patterns and Software Architectures. 1st Edn., Cambridge University Press, Cambridge, ISBN-13: 978-0521764148, Pages: 576.

Hassaine, S., Y.G. Gueheneuc, S. Hamel and G. Antoniol, 2012. Advise: Architectural decay in software evolution. Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), March 27-30, 2012, IEEE, Quebec, Canada, ISBN:978-1-4673-0984-4, pp: 267-276.

Jazayeri, M., 2002. On Architectural Stability and Evolution. Springer, Berlin, Germany.

Li, W., L. Etzkorn, C. Davis and J. Talburt, 2000. An empirical study of object-oriented system evolution. Inf. Software Technol., 42: 373-381.

Olague, H.M., L.H. Etzkorn, W. Li and G. Cox, 2006. Assessing design instability in iterative (agile) object-oriented projects. J. Software Maint. Evol. Res. Pract., 18: 237-266.

Quah, T.S. and M.M.T. Thwin, 2003. Application of neural networks for software quality prediction using object-oriented metrics. Proceedings of the International Conference on Software Maintenance (ICSM03), September 22-26, 2003, IEEE, Singapore, ISBN:0-7695-1905-9, pp: 116-125.

Roden, P.L., S. Virani, L.H. Etzkorn and S. Messimer, 2007. An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes. Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM07), September 30-October 1, 2007, IEEE, Florence, Alabama, ISBN: 978-0-7695-2880-9, pp: 171-179.

Ruchika, M., 2015. Empirical Research in Software Engineering: Concepts, Analysis and Applications. CRC Press, New York, USA.

Tonu, S.A., A. Ashkan and L. Tahvildari, 2006. Evaluating architectural stability using a metric-based approach. Proceedings of the Conference on Software Maintenance and Reengineering (CSMR06), March 22-24, 2006, IEEE, Canada, ISBN:0-7695-2536-9, pp: 10-10.

Yu, L. and S. Ramaswamy, 2009. Measuring the evolutionary stability of software systems: Case studies of Linux and Free BSD. IET. Software, 3: 26-36.

Zuse, H., 1997. A Framework of Software Measurement. Walter de Gruyter Company, New York, USA., Pages: 402.