Volume 2, Issue 11, November 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

The Software Quality Assurance Framework for CBSD

Aman Kumar Sharma*
*Computer Science Department,
Himachal Pradesh University,
Shimla, India.*

Arvind Kalia
*Computer Science Department,
Himachal Pradesh University,
Shimla, India.*

Hardeep Singh
*Computer Sc. & Engg. Department,
Guru Nanak Dev University,
Amritsar India.*

Abstract— A number of quality models for software processes have been published, each of which is intended to encompass the totality of quality factors and issues relevant to a specific notion of process quality. Quality models may be used to develop a new process, measure the quality of existing processes, or guide improvement of existing processes. It is desirable to have mechanisms to select the component of highest intrinsic quality and greatest relevance. In this study a quality assurance framework is proposed, on the basis of relationship between the metrics and the quality sub-factors for the assessment of high-level design quality factors in object-oriented designs.

Keywords— Software Quality, Metrics, Factors, Component, McCall model, Boehm model, ISO 9126 model

I. Introduction

The use of Component Based Software Development (CBSD) products to implement significant portions of software system has grown in both government and industry over the past decade. The use of CBSD indeed have a beneficial effect in terms of less costly system construction, faster development of software, easy maintenance, technological advancements happening in the competitive market and greater reuse [1].

Quality is a functional and artistic measurement used to specify user satisfaction with a product or how well the product performs compared to similar products. A model is an abstract form of reality, enabling details to be eliminated and an entity or concept to be viewed from a particular perspective [2]. There are various kinds of models: cost estimation models, quality models, maturity models etc. Models can be presented in different manner, such as in the form of equation, function and / or diagram [3]. Relation between components are examined to form opinions, thus quality assurance framework have become essential for ensuring that the product caters to the need of customers. Models have been proposed by numerous researchers over a period of time to name a few prominent studies McCall quality factors proposed in 1976 [4], Barry Boehm quality model presented in 1978 [5], FURPS in 1987 [6], Gillies Relational model [7], ISO 1926 in 1991 [8] and Dromey model in 1996 [6]. Such models are intended to evaluate the quality of software in general and are neither oriented towards CBSD and nor are they able to quantify quality.

Certification is important to assess quality. Evaluation at different stages of development gives an edge to the developer to improve quality of the finished product. Software product certification is based on models. The models lack internal properties into the product to demonstrate the external quality attributes desired [9].

The objective of this study is to propose software quality assurance framework to obtain systemic quality. The paper is structured as follows: second section presents the literature survey, the third section gives the framework of software quality assurance. The fourth section concludes the paper along with mention of the future scope.

II. LITERATURE SURVEY

The literature on the subject contains study of quality models that have gradually evolved. The well known software models are McCall model, Boehm model, FURP model, Gillies Relational model, ISO 9126 model and Dromey model [10] [11].

James A. McCall grouped software qualities into three sets of quality factors: Operation, Revision and Transition. These three sets constitute eleven quality factors which further contain twenty three sub-factors of quality [12] [13]. However, the model lack in quantifying the quality [14].

The Boehm model has additions of characteristics to McCall model with emphasis on the maintainability of software product. The model proposes to break the general utility to sub-factors, has wider range of characteristics and incorporates criteria [15].

Robert Grady and Caswell proposed the FURP model which decomposes characteristics into two categories of requirements: functional and non-functional. FURP model sets priorities and defines quality attributes but fails to take account all the software characteristics [16].

Gillies' Relational model focuses on the relation between the attributes. However, valuation of quality attributes is not assigned to judge the importance of the attribute [11].

ISO 9126 model divides factors into two levels: Characteristics and Sub-Characteristics without attempting to measure quality [10].

The Dromey model introduced by R. G. Dromey intends to increase the understanding of the relationships between the attributes and sub-attributes. But, the model lacks in measurement of relationship and attributes [6].

The quality characteristics present in all the models studied are efficiency, reliability, maintainability, portability and usability. The remaining factors are indirectly integrated with other factors [17]. Further, Sharma et. al. have identified quality sub-factors of quality factors in their study [17].

To quantify the quality attributes software metrics suite exists in the software quality evaluation domain namely CK metrics suite [18] and MOOD metrics suite [19]. Studies related to evaluation of metrics suite prevail to notify the better metrics in terms of evaluation of software quality such studies are [20] [21] [22] [23]. These studies have highlighted that the CK metrics namely Number of Children (NOC), Weighted Methods per Class (WMC), Depth Inheritance Tree (DIT), Response For a Class (RFC) and Coupling Between Objects (CBO) are capable to evaluate quality of software component. Similarly, the metrics of MOOD: Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF) and POF (Polymorphism Factor) are quality evaluator of software component [23].

A software quality assurance framework was designed after identification of principal elements of the quality models.

III. **SOFTWARE QUALITY ASSURANCE FRAMEWORK**

Framework for software quality assurance for CBSD is presented. A common approach for formulating the framework for software quality is to first identify a set of high level quality factors. The selected high level quality factors pertaining to this study are efficiency, maintainability, portability, reliability and usability. These factors were selected on the basis that these quality factors were proposed by all the software quality models and as such are significant. Then in a top down fashion these factors are decomposed into a set of sub-factors e.g. efficiency is decomposed into sub-factors namely time behaviour, resource behaviour, reply time, processing speed, execution efficiency, robust, hardware independence and compliance. Similarly, sub-factors for maintainability are analyzability, changeability, stability, testability, adaptiveness, understandabilty, error debugging, reusability and extensibility. Sub-factors of portability are adaptiveness, replaceability, reusability and transferability. Reliability sub-factors are fault tolerance, recoverability, survivability, error tolerance and simplicity. Identified usability sub-factors are attractiveness, understandability, learnability, operability, ease of use and documentation. As a next step for the formulation of software quality framework various component quality metrics were identified. A quality metric provides a way to measure & quantify a quality sub-factor. The metrics suites used are CK metrics suite and MOOD metrics suite. The selection of the metric suite for the usage is based upon the studies [20] [21] [23]. The framework relates design properties of object oriented programming such as encapsulation, inheritance, coupling, polymorphism and modularity to high level quality factors such as efficiency, maintainability, portability, reliability and usability using anecdotal information. The relationship or links from design properties to quality factors and further to quality sub-factors are in accordance with their influence and are analyzed to form the framework.

The efficiency quality factor and its sub-factors are affected by the metrics in the following manner:

Time Behavior with RFC: RFC represents set of methods that can be executed in response to a message received by an object of that class. Time behavior worsens with higher RFC because more communication among classes will require more and higher execution complexity.

Time Behavior with WMC: WMC is the sum of the complexities of the methods of a class. Classes with larger NOM will have increased number of instructions. Thus increase in WMC worsens the time behavior.

Time Behavior with DIT: Deeper class hierarchy depicts that more methods and variables can be inherited. Time Behavior gets worse with increasing DIT as increasing DIT means more complexity.

Time Behavior with CBO: Increase in CBO will worse time behavior. Excessive coupling between the object classes implies that the execution is not localised but might involve many other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system. Time behavior becomes worse with increasing CBO.

Time behavior with MHF: Time behavior improves with increasing MHF. Private methods restrict access resulting in faster processing.

Time behavior with AIF: Due to the fact that higher inheritance involves indirect calls more complexity as it is likely to inherit more variables. Time behavior has inverse relation with AIF.

Resource Behavior with CBO: Coupling involves other parts, devices and classes. Thus execution during operation does involve huge parts of the system. Resource Behavior gets bad with increasing CBO.

Resource Behavior with DIT: Deeper class hierarchy would need more resources to keep a track of lengthier available attributes, methods and classes. Resource Behavior behaves poorly with increasing DIT.

Resource Behavior with WMC: More classes mean higher WMC and more size. Size indicates higher memory utilization. Resource Behavior gets worse if the WMC increases.

Resource Behavior with AIF: Higher Inheritance implies more reference to memory and other resources denoting that with increase in AIF the resource behavior deteriorates. More resources are used in case of higher inheritance.

Resource Behavior with MHF: Increase in hiding factor improves resource Behavior. Increased encapsulation reduces the load of resources being utilized. Burden on resource Behavior gets reduced with increase in MHF.

Reply time with WMC: Additional methods per class surely require extra processing. Reply time increases with increasing WMC value.

Reply time with RFC: With the increase in RFC, reply time also increases. Added number of responses for a class increases the time required to reply.

Reply time with DIT: Reply time increases with increasing DIT. More depth in the tree of classes results in more time to reply.

Reply time with NOC: Reply time worsens with increasing value of NOC. More number of classes' breadth wise implies more time requirements to process it.

Processing speed with WMC: It declines with increasing WMC. More classes require more time to process.

Processing speed with RFC: It declines with increasing RFC. Processing speed deteriorates with increasing response set size.

Processing speed with DIT: Deeper access to classes consumes more time. Processing speed drops with increasing DIT.

Processing speed with NOC: Processing more number of immediate child classes derived from a base class declines the speed. Processing speed slips with increasing NOC.

Processing speed with MHF: Processing speed increases with higher degree of encapsulation. Rise in MHF increases processing speed.

Processing speed with AIF: Processing speed decreases with an increase in Inheritance levels. Processing speed declines with increase in AIF.

Execution Efficiency with WMC: Higher values of WMC require more execution efforts. Execution efficiency declines with increase in WMC.

Execution Efficiency with RFC: Execution efficiency declines with increase in RFC.

Execution Efficiency with DIT: More penetration in the depth of classes demands extra effort. Execution efficiency weakens with increase in DIT.

Execution Efficiency with NOC: A class with higher number of children requires higher effort for executing the software. All depending classes need to be taken care of as well. Execution efficiency decreases with increasing NOC.

Execution Efficiency with AIF: Deeper access of attributes and methods require more hard work. Execution Efficiency declines with increase in AIF.

Execution Efficiency with MHF: Increasing values of MHF means more encapsulation and less access to attributes and methods. Efficiency would increase with more encapsulation.

Robust with DIT: Robustness declines with increasing DIT. Deeper trees constitute greater design complexity since more methods and classes are involved.

Robust with NOC: Robustness deteriorates with increasing NOC.

Robust with CBO: Enriched coupling involves more complexity resulting into decrease in robustness with increasing CBO.

Robust with AIF: Inheritance implies penetration into the class hierarchy having implications in a number of areas; hence AIF increase reduces robustness.

Robust with MHF: Encapsulation implies restricted access, more of specialization and loss of flexibility. Such systems are sturdier. MHF increase makes the system more robust.

For maintainability factor and its sub-factors, the impact of metrics is as follows:

Analyzability with WMC: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of parts to be modified is directly related to the size and complexity of the class. Depending on the weight metric, WMC allows an assessment of size or complexity. Analyzability declines with increasing WMC.

Analyzability with RFC: The effort and time for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified is directly related to the number of executed methods in response to a message. Analyzability declines with increasing RFC.

Analyzability with DIT: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of parts to be modified is related to the number of methods of the class. Classes that are deep down in the class hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it. Analyzability declines with increasing DIT.

Analyzability with CBO: Analyzability decreases with increasing CBO since the parts of the system are also using other parts of the system which need to be analyzed as well.

Analyzability with NOC: To analyze a class having a higher number of children requires higher effort. Diagnosis of deficiencies or causes of failures involves the children of a parent class. They have access to the functionality and data provided by a class and have to be involved in the analysis. Identifying parts in a parent class which need to be modified requires the analysis of all child classes, which are effected by the modification. To analyze a class completely it is also necessary to look at its children to be able to have the complete picture. Analyzability decreases with increasing NOC.

Analyzability with AIF: Attributes and methods available deep down in the classes' hierarchy are difficult to be analyzed. Analyzability decreases with increase in AIF.

Analyzability with MHF: Having limited or less access to methods and / or attributes helps in analysis as it restricts the domain area. Analyzability increases with increase in MHF.

Changeability with WMC: Changing a class requires prior understanding, which, in turn, is more complicated for large and complex systems. Depending on the weight metric, WMC allows an assessment of size or complexity. Changeability declines with increasing WMC.

Changeability with RFC: Each modification must be correct for all execution paths. The size of the response set for a class (RFC) gives an idea about how many methods are potentially contributing to the size of the execution paths. Changeability declines with increasing RFC.

Changeability with DIT: Changing a class requires prior understanding, which, in turn, is more complicated for classes with many methods. Classes that are deep down in the class hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to understand it. Changeability declines with increasing DIT.

Changeability with NOC: A class having a higher number of children has a lower changeability. The effort spent on modification, fault removal or environmental change is increased, since many child classes are extending its functionality, depending on the parent class. The side effects of modifications are harder to predict. Fault removal effect child classes. The need for environmental change has to consider the child classes. Changeability decreases with increasing NOC.

Changeability with CBO: Changeability decreases with increasing CBO as the change in one part of system might involve the other parts which also need to be changed.

Changeability with AIF: Even a minor change in an attribute may have wider impact in case of higher inheritance values. Increase in values of AIF results in the decline of changeability.

Changeability with MHF: Higher values of encapsulation favour changeability. Increasing MHF values increase changeability.

Changeability with POF: The impact of polymorphism on changeability is on the lesser side. Overriding of methods at times make it difficult to change a component. Changeability reduces with increase in POF.

Stability with CBO: Stability correlates with the metrics which measure attributes of the software that indicate about the risk of unexpected effects as a result of modification. Stability decreases with increasing CBO since the effects of the modification may effect other parts as well. Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them. Stability decreases with increasing CBO.

Stability with WMC: Due to reduced analyzability stability is influenced negatively by size. Stability declines with increasing WMC.

Stability with RFC: Due to reduced analyzability stability too may be influenced negatively by response of a class. Stability declines with increasing RFC.

Stability with DIT: Due to reduced analyzability and testability, also stability may be influenced negatively by size. Stability might decline with increasing DIT.

Stability with NOC: A class having a higher number of children bears a higher risk of unexpected effect of modification. Child classes are re-using parent classes in various ways. They are directly affected by modifications making it difficult to predict how stable a class or software product would be after modification. Stability decreases with increasing NOC.

Stability with AIF: Attributes of a class available deep down in the hierarchy cause stability problems. Stability declines with an increase in AIF. Increase in AIF indicates more and more attributes are inherited with increasing AIF and stability declines as the risk of unexpected effects as a result of modifications rise.

Stability with MHF: Private methods indicate loss of flexibility and hence lesser modification ability. In the midst of increasing MHF stability too increases.

Testability with WMC: Testability correlates with the metrics which measure the efforts needed for test coverage of all execution paths. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. WMC allows an assessment of the number of methods and their complexity. Testability declines with increasing WMC.

Testability with RFC: Complete testing requires coverage of all execution paths needed for test. Response For a Class computes the number of methods (directly) involved in handling a particular message. Testability declines with increasing RFC.

Testability with DIT: The number of possible execution paths of a class increases deep down in the classes hierarchy. Due to late binding, super-class methods need to be tested again in the sub-class context. This makes it potentially harder to test classes deep down in the class hierarchy. Testability declines with increasing DIT.

Testability with CBO: Testability decreases with increasing CBO since one part of the system is using other part of the system which results in increased number of possible test paths.

Testability with NOC: Testability decreases with increasing NOC as a class having a higher number of children requires a higher effort for validating the modified software. All depending child classes need to be included in the tests as well, since modifications in the parent class have direct impact on the extending child classes.

Testability with AIF: Increasing values of inheritance decreases testability. Growing AIF diminishes testability.

Testability with MHF: Higher encapsulation assists testability in a way that rising values of MHF make the class less inheritable and thus reducing the test paths. MHF help in raising testability.

Testability with POF: Increase in polymorphism lowers understanding and lowers testing abilities.

Adaptiveness with WMC: Adaptiveness correlates with the metrics related to the amount of changes needed for adaptation of component to new environment. Adaptiveness declines with increasing WMC since each modification requires understanding of the system and which is more complicated for large systems.

Adaptiveness with RFC: Adaptiveness declines with increasing RFC. The increase in the number of methods executed in response to message will also increase the number of methods to be changed for adaption.

Adaptiveness with DIT: Increase in DIT will increase complexity. Adaptiveness declines with increasing DIT because the higher complexity will make the adaption more difficult.

Adaptiveness with CBO: Adaptiveness decreases with increasing CBO. Increase in CBO results in higher number of other classes that are coupled to a class hence making the changes difficult.

Adaptiveness with AIF: More attribute inheritance factor means higher reuse. However, with the higher reuse the complexity increases which in turn lowers adaptability.

Adaptiveness with MHF: Adaptiveness grows with higher values of MHF since higher MHF reduces the complexity and makes the component more adaptable.

Adaptiveness with NOC: A class having a higher number of children is difficult to adapt to different specified environments. Adapting the parent class to a new environment, can make it unsuitable for the children, requiring adaptation in them as well. Adaptiveness decreases with increasing NOC.

Adaptiveness with POF: Increase in polymorphism leads to more of overriding of methods resulting in reduced adaptability.

Understandability with WMC: Understandability declines with increasing WMC. Increase in the WMC indicates increase in the sum of complexities of the methods of the class thus, reducing understandability.

Understandability with RFC: Understandability declines with increasing RFC as understandability of a class depends upon its complexity and response size of the method.

Understandability with DIT: Classes that are deep down in the class hierarchy potentially inherit many methods from super-class. The increased DIT signifies greater design complexity thus reduced understandability.

Understandability with NOC: Understanding a class is supported if it has a high number of children. Numerous children show directly a high usability of the class under concern. Understandability increases with increasing NOC.

Understandability with CBO: Parts, which have a high coupling, may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well. Understandability decreases with increasing CBO.

Understandability with AIF: Increasing values of AIF decrease understandabilty because increased reusability makes the component more complex and less understanding.

Understandability with MHF: MHF increased values also enable understandability to increase.

Error Debugging with DIT: More DIT implies more difficulty in error debugging.

Error Debugging with WMC: Increased number of methods in a class makes it cumbersome to debug errors. Higher value of WMC lowers error debugging.

Error Debugging with CBO: Increase in CBO lessens error debugging.

Error Debugging with AIF: Highly inherited attributes are prone to errors. Further, identifying such attributes is cumbersome. Increased AIF reduce error debugging.

Error Debugging with MHF: Increased MHF easies the process of error debugging as the highly encapsulated classes are less complex and easy to debug.

Re-Usability with DIT: It is positively influenced by DIT. A large DIT value indicates that many methods might be reused.

Re-Usability with NOC: It is positively influenced by attributes assessed with NOC as higher NOC denotes higher use of the base class.

Re-Usability with AIF: Inheritance promotes the reuse of attributes and methods. Increase in AIF results in increase in reusability.

Re-Usability with MHF: MHF negatively influence re-usability since high MHF indicate very little functionality and the methods are not available for reuse.

Extensibility with RFC: If communication among classes exists on a larger scale then it requires more effort for adding new components. Increase in RFC declines extensibility.

Extensibility with CBO: To provide an extendable program it is a good practice to reduce the dependencies between implementing classes. With an increase in CBO, extensibility decreases.

Extensibility with DIT: Adding a component in the inheritance tree needs extra effort to check for any bugs. It implies that in the absence of inheritance tree extensibility flourishes.

Extensibility with AIF: While enhancing software by adding new features, it is needed to closely monitor the adverse effect of the new feature on the existing components. Component having minimum inheritance levels will be less affected by the addition of a new feature. More AIF denote less extensibility.

Extensibility with MHF: MHF is directly related to extensibility. Inaccessible objects from one another would lead to confinement in their scope. Addition of a new component, in a limited scope zone produces fewer hassles.

The mapping effects of metrics onto the portability and its sub-factors are as follows:

Replaceablity with WMC: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by the WMC. Replaceablity declines with increasing WMC.

Replaceablity with DIT: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size increases for classes that are deep down in the class hierarchy. Replaceablity declines with increasing DIT.

Replaceability with NOC: A class having a higher number of children is difficult to replace. The children are dependent on it, by extending specific functionality and can depend on certain functionality the parent class provides. It is difficult to find another class satisfying these specific needs, allowing replacing the parent class. Replaceability decreases with increasing NOC.

Replaceablity with RFC: Replaceablity declines with increasing RFC as components with complex control structures and response sets might have a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component.

Replaceability with CBO: Coupling between objects raises issues pertaining to complexity. Uncoupled objects are easier to augment than those with high degree dependencies due to lower interactions and interconnections. Higher CBO reduces replaceability. Replaceability with AIF: Replacing a method or an attribute with a substitute deep down in the class hierarchy involves more effort. Replaceability declines with increasing AIF.

Replaceability with MHF: Substituting in a limited domain area is manageable. It implies that higher values of MHF restrict the methods accessibility making it easier for replaceability to increase.

Replaceability with POF: Polymorphism has an effect on interaction, interconnection and linkages making it tougher for replacing a component with another. Increasing POF values reduce replaceability.

Transferability with WMC: Handling more methods require more efforts which declines transferability with increasing WMC.

Transferability with RFC: A lot of communication makes the component more complex thereby declining transferability with increasing response set size (RFC).

Transferability with DIT: Deeper class hierarchy based systems are more organized and are better placed for transferability. Transferability increases with increasing DIT.

Transferability with CBO: More coupling decreases the ability to transfer.

Transferability with NOC: Breadth wise spread classes within a component require more efforts to coordinate among them. Transferability is negatively influenced by attributes assessed with NOC.

Transferability with AIF: Inherited classes within a component are more structured. Transferability increases, with increasing value of AIF.

Transferability with MHF: Encapsulation leads to data abstraction. Hidden methods can easily be ported thus the ability to transfer increases with increase in MHF.

Transferability with POF: With an increase in POF transferability also increases.

The software metrics have relationship with reliability and sub-factors of reliability as follows:

Maturity with WMC: Maturity correlates with metrics which measure the frequency of failure due to faults in the component. Maturity declines with increasing WMC. It is due to reduced testability that bugs might be left in the component.

Maturity with RFC: Maturity might decline with increasing RFC since large response set size reduces testability thus making the system more bug prone.

Maturity with DIT: Increased DIT suggests reduced testability implying bugs might be left in the software. Therefore, Maturity might decline with increasing DIT.

Maturity with AIF: Maturity diminishes with increase in AIF as it is difficult to keep a track of increasing inheritance, which makes ability to test more difficult.

Maturity with MHF: Maturity increases with increase in MHF since the higher MHF shows less complexity and reduced bug density.

Maturity with POF: Use of polymorphism implies that one bug may lead to number of problems as the methods are linked, interacting and interconnected. Enhancing POF worsens maturity.

Fault-tolerance with WMC: Relating WMC to fault-tolerance requires being able to locate the parts of a system responsible for fault tolerance. The size of these parts might indicate a better ability to interact. Fault-tolerance might increase if the WMC increases.

Fault-tolerance with DIT: Relating DIT to fault-tolerance requires being able to locate tree hierarchy of a system responsible for interoperability. A high DIT in these hierarchy structures might indicate a better ability to interact. Fault-Tolerance might increase with increasing DIT.

Fault-tolerance with CBO: Increase in CBO indicate high coupling between the different parts of the system and can be affected by faults in other parts of the system. Fault-Tolerance might decrease with increasing CBO.

Fault-tolerance with MHF: Increase in MHF results in decrease in defect density and effort to fix defects. Increasing MHF value results in more fault tolerance.

Fault-tolerance with AIF: Deeper inheritance levels imply more complexity it means the component is less fault tolerant. Higher value of AIF means less fault tolerance.

Recoverability with WMC: Recoverability correlates with metrics which measure the ability of the component to recover the data in case of a failure. Recoverability requires being able to locate the parts of a system responsible for recoverability. Higher values of WMC might indicate a higher recoverability. Recoverability might increase if the WMC increases.

Recoverability with DIT: Relating DIT to recoverability requires being able to locate hierarchy structures of a system responsible for recovery. A high DIT in these hierarchy structures might indicate a higher recoverability. Recoverability might increase with increasing DIT.

Recoverability with CBO: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult. Recoverability might decrease with increasing CBO.

Recoverability with MHF: Encapsulation leads to data hiding and MHF hide methods. Higher values of MHF denote that the concerned method and/or attribute are created for a specific cause only. Its working or malfunctioning will block only a limited number of methods / attributes. The remaining ones shall work normally. Thereby, increase in MHF increases recoverability.

Recoverability with AIF: Inheritance enables access to its methods and attributes. Inherited objects will have a vast implication and dependence increases, which may lead to break down of many objects. Increase in values of AIF decrease the chances of recoverability.

Recoverability with POF: With the increase in POF, recoverability declines. More overloading of objects reduces restorability.

Survivability with WMC: Survivability correlates with the metrics which measure the degree to which essential services continues to be provided in spite of accidental harm. More methods per class increase specialty. Survivability increases with increase in WMC.

Survivability with DIT: Increase in DIT decreases survivability.

Survivability with CBO: CBO reduces survivability. It can be due to the fact that if one part of the system is affected by malicious harm then it can affect other parts of the system due to high coupling.

Survivability with AIF: Increase in inheritance levels makes the system more vulnerable to problems. Increase in AIF decrease survivability.

Survivability with MHF: Increase in encapsulation results in a secure system. A secure system will be less prone to malicious harm. Hence increase in MHF increases survivability.

Survivability with POF: Increase in the polymorphism factor leads to the decrease in survivability. It is due to the reason that interdependence of the methods of child class on the parent class is against the endurance. It means that if the method in the child class is harmed by malicious software the parent class shall also be affected.

Error-tolerance with WMC: Relating WMC to error-tolerance requires being able to locate the parts of a system responsible for error tolerance. The size of these parts might indicate a better ability to interact. Error-tolerance might increase if the WMC increases.

Error-tolerance with DIT: Relating DIT to error-tolerance requires being able to locate tree hierarchy of a system responsible for interoperability. A high DIT in these hierarchy structures might indicate a better ability to interact. Error-Tolerance might increase with increasing DIT.

Error-tolerance with CBO: Increase in CBO indicate high coupling between the different parts of the system and can be affected by errors in other parts of the system. Error-Tolerance might decrease with increasing CBO.

Error-tolerance with MHF: Increase in MHF results in decrease in defect density and effort to fix defects. Increasing MHF value results in more error tolerance.

Error-tolerance with AIF: Deeper inheritance levels imply more complexity it means the component is less error tolerant. Higher value of AIF means less error tolerance.

Simplicity with RFC: More communication within classes and between classes lead to complexity. Increase in RFC is inversely related to simplicity.

Simplicity with NOC: Increase in NOC shows the way for small manageable units. Simplicity increases with increase in NOC.

Simplicity with CBO: Coupling leads to more dependence on other objects. Simplicity decreases with an increase in CBO.

Simplicity with WMC: Managing bigger size is more complex than smaller ones. Size is weighed by the weighted number of classes. Simplicity decreases with an increase in WMC.

Simplicity with DIT: Deeper a class within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour, depth in the class hierarchy implies methods are being inherited from super classes. Simplicity decreases with increasing DIT.

Simplicity with AIF: Inherited methods and attributes have the ability to make an impact on a wider front; consequently more care is needed. Simplicity decreases with increasing values of AIF.

The relationship between the metrics and sub-factors of usability is as follows:

Attractiveness with WMC: Attractiveness of a class depends on the size and complexity of the potentially reused code. Depending on the weight metric, WMC allows an assessment of size or complexity. Attractiveness increases with increasing WMC.

Attractiveness with RFC: Attractiveness of a class depends on the complexity of the potentially reused code. Response For a Class allows an assessment of complexity. Attractiveness increases with increasing RFC.

Attractiveness with DIT: Attractiveness of a class depends on the size of the potentially reused code. Classes that are deep down in the class hierarchy potentially inherit many methods from super-classes. Attractiveness increases with increasing DIT.

Attractiveness with NOC: A class having a higher number of children appears more attractive to the software engineer/ developer. It is an eye-catcher in a class diagram or other representations since it stands out by having many children. Associated assumptions could be, that the class is stable, since it is tested each time a child is tested, that is well documented and understood, since it has been extended so often, that it is easier to understand, since there are many examples of usage, that it helps to understand the children, if the parent class has been understood, that is plays a central role in the design, since many classes extend its functionality. Attractiveness increases with increasing NOC.

Attractiveness with CBO: Parts that have a high (outgoing) coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies. Attractiveness decreases with increasing CBO.

Attractiveness with AIF: Inheritance promotes reusability as well as usability. Attractiveness increases with increasing values of AIF.

Attractiveness with MHF: Values of MHF restrict reuse as accessibility is confined. Increasing values of MHF decline attractiveness.

Attractiveness with POF: Since other parts of the system are being used which need to be understood and represent dependencies. Increase in POF decreases attractiveness.

Learnability with WMC: Learnability is correlated to the measure of the user's effort to learn an application. Learnability of a class depends upon the size of its interfaces. Learnability declines with increasing WMC.

Learnability with RFC: Learnability declines with increasing RFC as learnability depends on the complexity of its interface and the number of methods in other classes called in response to a received message.

Learnability with DIT: As DIT becomes high it makes the classes harder to analyze and in turn to learn. Learnability declines with increasing DIT.

Learnability with NOC: Learnability increases with increasing NOC. High NOC denotes high number of children which require versatile knowledge on how to reuse a particular class in different situations.

Learnability with CBO: Learnability decrease with increasing CBO because due to high CBO some parts of the system may have high coupling with other parts which also needs to be understood.

Learnability with AIF: Learnability declines with increasing AIF. Lengthier class hierarchy leads to keeping the knowledge of various inherited attributes hence making the learnability difficult.

Learnability with MHF: Restricting the scope of a method in its definition makes understanding easy. Learnability increases with increase in MHF.

Learnability with POF: POF is inversely related to learnability. In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden, resulting in more efforts to learn and understand.

TABLE I
SOFTWARE QUALITY ASSURANCE FRAMEWORK, SHOWING QUALITY FACTORS, SUB-FACTORS, METRICS AND THEIR RELATIONSHIP

MHF	**AIF**	**POF**	**WMC**	**DIT**	**RFC**	**CBO**	**NOC**	**Metrics** / **Sub Factors**	**Factor**
↑	↓	⌂	↓	↓	↓	↓	⌂	Time Behavior	**Efficiency**
↑	↓	⌂	↓	↓	⌂	↓	⌂	Resource Behavior	
⌂	⌂	⌂	↓	↓	↓	⌂	↓	Reply Time	
↑	↓	⌂	↓	↓	↓	⌂	↓	Processing Speed	
↑	↓	⌂	↓	↓	↓	⌂	↓	Execution Efficiency	
↑	↓	⌂	⌂	↓	⌂	↓	↓	Robust	
↑	↓	⌂	↓	↓	↓	↓	↓	Analyzability	**Maintainability**
↑	↓	↓	↓	↓	↓	↓	↓	Changeability	
↑	↓	⌂	↓	↓	↓	↓	↓	Stability	
↑	↓	↓	↓	↓	↓	↓	↓	Testability	
↑	↓	↓	↓	↓	↓	↓	↓	Adaptiveness	
↑	↓	⌂	↓	↓	↓	↓	↑	Understandability	
↑	↓	⌂	↓	↓	⌂	↓	⌂	Error Debugging	
↓	↑	↑	⌂	↑	⌂	⌂	↑	Reusability	

↑	↓	⌂	⌂	↓	↓	↓	⌂	Extensibility	
↑	↓	↓	↓	↓	↓	↓	↓	Adaptiveness	**Portability**
↑	↓	↓	↓	↓	↓	↓	↓	Replaceability	
↑	↑	↑	↓	⌂	⌂	↓	↑	Reusability	
↑	↑	↑	↓	↑	↓	↓	↓	Transferability	
↑	↓	↓	↓	↓	↓	⌂	⌂	Maturity	**Reliability**
↑	↓	⌂	↑	↑	⌂	↓	⌂	Fault Tolerance	
↑	↓	↓	↑	↑	⌂	↓	⌂	Recoverability	
↑	↓	↓	↑	↓	⌂	↓	⌂	Survivability	
↑	↓	⌂	↑	↑	⌂	↓	⌂	Error Tolerance	
⌂	↓	⌂	↓	↓	↓	↓	↑	Simplicity	
↓	↑	↓	↑	↑	↑	↓	↑	Attractiveness	**Usability**
↑	↓	↓	↓	↓	↓	↓	↑	Understandability	
↑	↓	↓	↓	↓	↓	↓	↑	Learnability	
⌂	⌂	⌂	↓	↓	↓	↓	↑	Operability	
⌂	↑	⌂	↑	↑	↑	⌂	⌂	Documentation	

Where the symbols signify the following:

No relation	is denoted by ⌂
+ve related	is represented by ↑
-ve related	is represented by ↓

Relation not evaluated is represented by ☼

Operability with WMC: Operability declines with increasing WMC. It is due to the fact that increased WMC shows higher complexity and thus reduced operability.

Operability with RFC: Operability decreases with increased RFC. Operability depends upon the complexity of its interfaces and dependency on other classes.

Operability with DIT: Increase in the DIT reduces operability. The definitions of inherited methods are not local to the class which makes a class harder to analyze.

Operability with NOC: A high number of children indicate that a particular class is well integrated into an existing software system. This means, that it is suitable for several different tasks, self-descriptive since there exist many examples on its usage, having a higher error tolerance, since it is involved each time one of its children is tested. Operability increases with increasing NOC.

Operability with CBO: Operability might decrease with increasing CBO as higher CBO represents higher dependency between the different parts of the system.

The relationship of documentation quality sub factor is already discussed in the maintainability quality factor.

The proposed software quality assurance framework is presented in Table 1. The framework has been validated in other related studies [21] [24] [25] and expert opinion to compare with the framework results on several large object-oriented components.

In the Table 1, the metrics are tabulated in the first row, the factors being put on the extreme last column, with sub-factors placed preceding to the factor's column. In the framework few symbols are used to designate the relationship between the metric and sub-factors. An upside arrow symbol (↑) depicts a direct positive relation between the metrics and the software quality sub-factor. A direct positive relation denotes that if the metric increases so does the quality sub-factor. Similarly, a down side arrow symbol (↓) signifies an inverse relation between the metrics and the software quality sub-factor. The inverse relation means that if the metric increases then the quality sub-factor decreases and vice versa. The symbol (⌂) is used to signify no relation between the metrics and the quality sub-factor. The results obtained and further analysis has concluded that for certain relationships between sub-factors and metrics no relation exists. Incline or decline of the metrics value has insignificant effect on the software metric. The study was confined to evaluation of software quality keeping in view the developer's perspective. The sub-factor's which do not associate with development stage are not evaluated. The symbol (☼) signifies that the specific quality sub-factor was not evaluated. Summarized in the Table 1 are the relations between 33 software quality sub-factors with 8 software quality metrics. Among them 63 times sub-factors were directly positive related with metrics and 136 times quality sub-factors were inversely related with the metrics. These relationships were derived based on the rational analysis of the factors and the sub-factors. Depending upon the relationships it can be derived that efficiency can be measured from the metrics associated with the sub-factors listed for efficiency quality factor in the allusion table. On examining the Table 1 from the viewpoint of the sub-factor it can be concluded that the sub-factor time behavior of efficiency quality factor is negatively associated with AIF, WMC, DIT, RFC and CBO. If the developer intends to increase the time behavior of a component based software then the value of the above mentioned metrics need to be reduced. However, MHF has a direct relation with time behavior. Thereby, to enhance time behavior the developer may attempt to increase the value of MHF. Further, the same analogy may be applied for the remaining quality factors, sub-factors, metrics and on their relations.

IV. CONCLUSION

Mapping of metrics with quality factors and with quality sub-factors was accomplished. The study framed a software quality assurance framework comprising of 8 metrics, 5 quality factors and 33 quality sub-factors. Subsequently, the quality assurance framework featuring the type of relationship between the metrics and quality sub-factors was proposed. In future a capability maturity model for components may be created.

REFERENCES

[1] Brereton B. and Budgen D., "Component Based Systems: A Classification of Issues", *IEEE Computer 2000*, pp. 54-62, 2000.

[2] Fenton N. and Pfleeger S.L., "*Software Metrics: A Rigorous and Practical Approach*", 2nd ed., PWS Publishing, 1997.

[3] Adnan Rawashdeh and Bassem Matalkah, "A New Software Quality Model For Evaluating COTS Components", *Journal of Computer Science*, vol. 2, no. 4, pp. 373-381, 2006.

[4] Côté and Marc-Alexis, Witold Suryn, and Elli Georgiadou, "In Search For A Widely Applicable And Accepted Software Quality Model For Software Quality Engineering", *Software Quality Journal,* vol. 15, no. 4, pp. 401-416, 2007.

[5] Dorfman and Merlin, "System And Software Requirements Engineering", *IEEE Computer Society Press Tutorial,* 1990.

[6] G. Dromey, "A Model for Software Product Quality", *IEEE Transactions on Software Engineering*, vol. 146, pp. 20-22, 1995.

[7] Slhoub and Khaled Ali M., "Managing Software Quality in Small-scale Projects", University of New Brunswick (Canada), 2008.

[8] Dzida, and Wolfgang, "Standards for user-interfaces", *Computer Standards & Interfaces,* vol. 17, no. 1, pp. 89-97, 1995.

[9] Dromey G., "Cornering the Chimera", *IEEE Software*, pp. 33-43, 1996.

[10] Aggarwal K.K., and Yogesh Singh, "*Software Engineering*", 2nd ed., New Age International Publishers, Delhi, 2005.

[11] Gillies A. "*Software Quality: Theory and Management*", 2nd ed., International Thomson Computer Press, 1997.

[12] Agarwal Bharat Bhushan, and Tayal Sumit Prakash, "*Software Engineering*", 1st ed., University Science Press, Delhi, 2007.

[13] Cavano J. P., and McCall J. A., "A Framework for the measurement of Software Quality", *In Proc. Of ACM Software Quality Assurance Workshop,* 1978.

[14] J. A. McCall, P. K. Richards, and G. F. Walters, "*Factors in Software Quality*", Griffiths Air Force Base, NY Rome Air Development Center Air Force Systems Command, 1977.

[15] Khan R.A., Mustafa K., and Ahson S.I., "*Software Quality: Concepts and Practices*", 1st ed., Narosa, Delhi, 2006.

[16] Grady R. B., and Caswell Deborah L., "*Software Metrics: Establishing a Company Wide Program*", Prentice Hall, USA, 1987.

[17] Sharma Aman Kumar, Arvind Kalia, and Hardeep Singh, "An Analysis of Optimum Software Quality Factors", *IOSR Journal of Engineering,* vol. 2, no. 4, pp. 663-669, 2012.

[18] Chidamber S R and Kemerer C F, "A Metrics Suite for Object Oriented Design", *IEEE Transaction Software Engineering,* vol. 20, no. 6, pp. 476–493, 1994.

[19] Abreu F B, Coulao Miguel and Esteves Rita, "Toward the Design Quality Evaluation of Object-Oriented Software Systems", *Proceedings of the 5th International Conference on Software Quality,* Austin, Texas, USA, 1995.

[20] Harrison R., Counsell Steve J., and Nithi Reuben V., "An Evaluation of the Mood Set of Object-Oriented Software Metrics", *IEEE Transaction Software Engineering,* vol. 24, no. 6, pp. 491–496, 1998.

[21] Sharma Aman Kumar, Arvind Kalia, and Hardeep Singh, "Empirical Analysis of Object Oriented Quality Suites", *International Journal of Engineering and Advanced Technology (IJEAT),* vol. 1, no. 4, pp. 163-167, 2012.

[22] Harrison R., Counsell S., and Nithi R., "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems", *Journal of Systems and Software,* vol. 52 no. 2-3, pp. 173–179, 2000.

[23] Sharma Aman Kumar, Arvind Kalia, and Hardeep Singh, "Metrics Identification for Measuring Object Oriented Software Quality", *International Journal of Soft Computing and Engineering (IJSCE),* vol. 2, no. 5, pp. 255-258, 2012.

[24] Sharma Aman Kumar, Arvind Kalia, and Hardeep Singh, "Taxonomy of Metrics for Assessing Software Quality", *International Journal of Engineering Research and Technology (IJERT),* vol. 1, no. 02, 2012.

[25] Sharma Aman Kumar, Arvind Kalia and Hardeep Singh, "Software Quality Evaluation Using MOOD Metrics: A Case Study of OSS", *In proceeding of Application Tools and Techniques of Ubiquitous Computing in Building Intelligent Space World,* 17-18 September 2012, Kurukshetra University, 2012.