PREDICTION OF THE QUALITY OF SOFTWARE PRODUCT USING OBJECT ORIENTED METRICS

Arti Chhikara*, R. S. Chhillar** and Sujata***

Metrics and Measurement are key components of any engineering discipline and object oriented software engineering is no exception. Object-oriented software development requires a different approach from traditional development methods, including the metrics used to evaluate the software. This paper focuses on several object oriented software metrics and effect of these metrics on quality of software product.

Keywords: Object Oriented, Design, Product Quality, Metric, Measure.

1. INTRODUCTION

Object Oriented designs and development are popular concepts in today’s software development environment—some even herald them as the “silver bullet” for solving software problems. Although there is no silver bullet, object-oriented (OO) development has proved its value for systems that must be maintained and modified. OO software development requires a different approach from more traditional functional decomposition and data flow development methods, including the metrics used to evaluate OO software. Before we start concentrating on object oriented metrics and their impact on software quality it is important to discuss some important points about software quality itself. Software Quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software. [1, 5]

Product quality describes the attributes of the products of the software process. It would then include, for example, the completeness of the design documents, the traceability of the design, the reliability and maintainability of the code and coverage of the tests. Software quality thus compromises of measuring following factors:

1. Efficiency
2. Complexity

* Department of Computer Science, Maharaja Agrasen College, University of Delhi, E-mail: phd.arti@gmail.com
** Department of Computer Science and Applications, M.D.U. Rohtak
*** Department of Computer Science, Deen Dyal Upadhyaya College, University of Delhi
Whether a metric is traditional or object oriented, it must effectively measure one or more of these attributes of software quality \[4, 5, 6, 7\].

2. OBJECT ORIENTED METRICS

Many different metrics have been proposed for object oriented systems. The object oriented metrics measure principle structures and, if they are improperly designed, negatively affect the design and code quality attributes. The selected object oriented metrics are primarily applied to the concepts of classes, coupling and inheritance \[3\].

2.1. Class

A class is a way to bind data and its associated functions together. A class is a template from which objects can be created. This set of objects shares a common structure and common behavior manifested by the set of methods. The following class metrics described here measure the complexity of a class using the classes:

(i) Methods
(ii) Messages
(iii) Cohesion
(iv) Coupling

2.1.1. Methods

The method is an operation upon an object.

2.1.1 (a) Weighted Methods Per Class (WMC): It is defined as the sum of the complexities of all methods of a class. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class. The WMC is calculated as the sum of McCabe’s cyclomatic complexity of each local method:

WMC = summation of McCabe’s cyclomatic complexity of all methods ranging from 0 to N; where N is positive integer \[2, 3\].

2.1.1 (b) Relation with Quality: The larger the number of methods in a class, greater is the potential impact on children because children will inherit all the methods defined in the class. Classes with large numbers of methods are likely to
be more application specific, limiting the possibility of reuse. Also more the control flows class methods have, harder is to understand them and thus difficult is to maintain them. \[1, 4, 5, 6, 7\]

This metric measures reusability, understandability and maintainability attributes of software quality.

2.1.2. Messages
A class can have any number of objects. These objects can communicate with each other by passing messages much the same way as people pass messages to one another. A message is a request that an object makes to another object to perform an operation. The operation executed as a result of receiving a message is called a method. The next metric looks at methods and messages within a class.

2.1.2(a) Response for A Class (RFC): The response for a class (RFC) metric measures the cardinality of the response set of a class. The response set of a class consists of all local methods and all the methods called by local methods. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes.

The calculation of RFC is given by,

$$RFC = number\ of\ local\ methods + number\ of\ methods\ called\ by\ local\ methods;$$

Ranging from 0 to \(N\); where \(N\) is a positive integer \[3, 4\].

2.1.2(b) Relation with Quality: The larger the number of methods that can be invoked from a class through messages, greater is the complexity of a class. If a large number of methods can be invoked in response to a message then testing and debugging of the class becomes complicated because it requires a high level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. Also larger the RFC metric, harder is to maintain the class because calling a large number of methods in response to a message makes tracing an error difficult. This metric evaluates understandability, maintainability and testability attribute of software quality. \[1, 5, 6, 7\].

2.1.3. Cohesion
Cohesion is the measure of the relative functional strength of a module. Effective object-oriented designs maximize cohesion since it promotes encapsulation. Low cohesion is likely to produce a higher degree of errors in the development. Low cohesion adds complexity which can translate into reduction in application reliability. \[2\] The third class metric investigates cohesion.

2.1.3(a) Lack of Cohesion of Methods (LCOM): The LCOM metric measures the lack of cohesion of a class. The cohesion of a class is characterized by how closely
the local methods are related to the local instance variables in the class. The calculation of LCOM is the number of disjoint sets of local methods. Disjoint sets are the collection of sets that do not intersect with each other. Any two methods in one disjoint set access at least one common local instance variable.

\[
\text{LCOM} = \text{number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer.} \quad [2, 3, 4]
\]

2.1.3(b) Relation with Quality: High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during development. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. The more cohesive a class, the easier it is to maintain the class. The larger the metric, harder is to maintain the class. This metric evaluates efficiency, reusability, maintainability and testability. [1, 2, 6, 7].

2.1.4. Coupling

Coupling is a measure of interconnection among modules in a program structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module and what data pass across the interface. In software design lowest possible coupling is targeted. Since excessive coupling is detrimental to modular design and prevents reuse. Classes (objects) are coupled in following ways:

(i) When a message is passed between the objects then objects are said to be coupled.

(ii) Classes are coupled when methods declared in one class and use methods or attributes from the other classes.

(iii) Inheritance introduces significant tight coupling between superclasses and their subclasses.

2.1.4(a) Coupling Between Objects (CBO): CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct noninheritance-related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. [3, 4]

2.1.4(b) Relation with Quality: The more the independent a class, the easier it is to reuse in another application. The larger the number of couples, higher is the sensitivity to change in other parts of the design; maintenance is therefore more difficult. Strong coupling complicates a system because a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible
coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates efficiency and reusability [1, 6, 7].

2.2. Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects including variables and operators. Inheritance supports the process of hierarchical classification. The concept of inheritance provides the idea of reusability. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

2.2.1. Depth of Inheritance Tree (DIT)

It is defined as the maximum length from the node to the root of the tree. The DIT metric is a measure of how many ancestor classes can potentially affect the class [3].

2.1.1(a) Relation with Quality: The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited. This metric primarily evaluates efficiency and reuse but also relates to understandability, maintainability and testability [1, 4, 5, 6, 7].

2.2.2. Number of Children (NOC)

The number of children (NOC) metric measures the number of direct children which a class has. It is an indicator of the potential influence that a class can have on the design and on the system [3].

2.2.1(b) Relation with Quality: This metric addresses the inheritance concept mentioned earlier in a better perspective than depth of inheritance tree (DIT). It seems logical that the more direct children which a class has, the more classes it may potentially affect due to inheritance and it may be an indication of subclassing misuse.

But more the number of children, greater are the reusability, because inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. Also as the value of NOC metric increase maintaining the class will become harder. NOC, therefore, primarily evaluates efficiency, reusability, maintainability and testability [1, 2, 6, 7].
3. CONCLUSION

Object oriented metrics have a great impact on the quality of the software product because they supply hard data that an organization can use to obtain valuable information about project progress, organizational productivity, and enterprise profitability. All practitioners employ some measurement of their code (even if they do not realize it) whenever they use a compiler or source library configuration management tools. Currently, many companies and other institutions are trying to adopt more complex and automated measurement tools into their development processes as they strive to improve their software products. The information obtained, can be used to intelligently plan and focus dynamic testing efforts and apply valuable resources where they will deliver the greatest impact.

References


