OBJECT-ORIENTED SOFTWARE QUALITY METRICS

D.M.RASANJALEE HIMALI, S.R. KODITHUWAKKU
Department of Statistics & Computer Science, Faculty of Science,
University of Peradeniya, Peradeniya, Sri Lanka.
Tel: +94-81-2389134

ABSTRACT

Software quality is a key measure of the software process. It provides a clear record of development progress, a basis for setting objectives, and a framework for current action. There are hundreds of software metrics for traditional and object-oriented software. These already-in-use metrics have certain limitations. Among many other limitations, the most obvious is that many traditional metrics do not capture certain fundamental aspects of the object-oriented paradigm such as inheritance and polymorphism. Some of the proposed object oriented metrics lack theoretical basis, while some others have not yet been validated. Some of these metrics are dependent on the implementation environment. New metrics are being investigated to overcome these limitations.

We investigated the traditional metrics and recently proposed metrics for measuring the overall quality of object-oriented software. A set of metrics for measuring the quality of object-oriented software is proposed by refining the existing metrics. We developed a prototype system to test the applicability of these metrics. Experimental results showed that the proposed metrics could be used to measure the quality of object-oriented software. This paper presents the set of metrics proposed.

1. INTRODUCTION

Object-oriented design becomes more popular than traditional function oriented design for today’s software development. Object oriented development requires not only a different approach to design and implementation, but also a different approach to software metrics. The existing metrics, both traditional and object-oriented, have several limitations in measuring the quality of object oriented software. The most obvious limitation in traditional metrics is that it does not capture certain fundamental aspects of the object-oriented paradigm. For example, they fail to measure inheritance and polymorphism. Many of the proposed object oriented metrics lack theoretical basis, while others have not yet been validated. Some of these metrics are dependent on the implementation environment. As most of the modern software systems are designed using the object-oriented paradigm, there is a need for better quality metrics to assist developers. In this research work we investigated a set of metrics that are not subject to such limitations. We attempted to develop a set of metrics, which are capable of measuring the overall quality of object-oriented software. We propose a set of new metrics by improving the existing traditional and object oriented metrics.
The rest of this paper is organized as follows. Section 2 presents the related work. Our methodology for developing a new set of quality metrics is described in Section 3. Section 4 presents the experimental results and discussion. The conclusion of the research and limitations of the proposed metrics are presented in section 5.

2. RELATED WORK

2.1 Traditional Metrics

Product assessments include traditional metrics such as size, complexity, and readability [11]. The metrics listed below are obtained from using UX-Metric from SET Laboratories [10]. UX-Metric produces McCabe's complexity metrics [9], counts GOTOs and comments, and calculates size metrics (IEEE-Std 1045-1992).

**McCabe Cyclomatic Complexity (CC)**

Cyclomatic complexity is a measure of a module control flow complexity based on graph theory. Cyclomatic complexity of a module uses control structures to create a control flow matrix, which in turn is used to generate a connected graph. The graph represents the control paths through the module. The complexity of the graph is the complexity of the module. Fundamentally, the CC of a module is roughly equivalent to the number of decision points and is a measure of the minimum number of test cases that would be required to cover all execution paths. A high Cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain.

**Source Lines of Code (SLOC)**

The SLOC metric measures the number of physical lines of active code, that is, no blank or commented lines. Counting the SLOC is one of the earliest and easiest approaches to measuring complexity. It is also the most criticized approach. In general the higher the SLOC in a module the less understandable and maintainable the module is.

**Comment Percentage (CP)**

The CP metric is defined as the number of commented lines of code divided by the number of non-blank lines of code. Usually 20% indicates adequate commenting for C or Fortran code. A high CP value facilitates in maintaining a system.

2.2 Object-Oriented Metrics

The recent work in software measurement for object-oriented software development includes:

2.2.1 CK OO Metrics Suite

**Weighted Methods Per Class (WMC)**

WMC measures the complexity of an individual class. Two different approaches are used to calculate the WMC metric. The first uses the sum of the complexity of each method contained in the class. The second approach assigns a complexity of 1 for each method in the class and then sums the result. This is equivalent to using the number of methods per class as a measure for WMC. The number of methods and complexity of methods involved is a direct predictor of how much time and effort is required to develop and maintain the class.

**Depth of Inheritance Tree of a Class (DIT)**

DIT is defined as the length of the longest path of inheritance ending at the current module. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree. The deeper the inheritance tree for a class, the harder it might be to predict its behavior due to the interaction between the inherited features and new features. However, the deeper a particular class is in the hierarchy, the greater the potential for reuse of inherited methods.

**Number of Children (NOC)**

NOC represents the number of immediate subclasses subordinated to a class in the class hierarchy. A moderate value for NOC indicates scope for reuse and high values may indicate an inappropriate abstraction in the design. Classes with a large number of children have to provide more generic service to all the children in various contexts and must be more flexible, a constraint that can introduce more complexity into the parent class.

**Coupling Between Objects (CBO)**

CBO is defined as the count of the number of other classes to which it is coupled. A class is coupled to another class if it uses the member method and/or instance variables of the other class. Excessive coupling indicates weakness of class encapsulation and may inhibit reuse. High coupling also indicates that more faults may be introduced due to inter-class activities.

**Response for a Class (RFC)**

RFC gives the number of methods that can potentially be executed in response to a message received by an object of that class. If a large number of methods can be invoked
in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester.

**Lack of Cohesion in Methods (LOCM)**

LOCM counts the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero. The larger the number of similar methods in a class the more cohesive the class is. Cohesiveness of methods within a class is desirable, since it promotes encapsulation and lack of cohesion implies classes should probably be split into two or more subclasses.

### 2.2.2 Lorenz Metrics

Based on his experience in OO software development, Lorenz (1993) proposed eleven metrics as OO design metrics. He also provided the rules of thumb for some of the metrics. The metrics are listed below:

1. Average Method Size
2. Average Number of Methods per Class
3. Average Number of Instance Variables per Class
4. Class Hierarchy Nesting Level
5. Number of Subsystem/Subsystem relationships
6. Number of Class/Class Relationships in Each Subsystem
7. Instance Variable Usage
8. Average Number of Comment Lines
9. Number of Problem Reports per Class
10. Number of Times a Class is Reused
11. Number of Classes and Methods Thrown Away

### 3. METHODOLOGY

In identifying a set of object oriented metrics, we focus on the primary, critical constructs of object oriented design and select metrics that apply to those areas. The metrics evaluate the object-oriented concepts, such as methods, classes, coupling, inheritance and so forth. In developing such metrics, we put more emphasis on internal object structure that reflects the complexity of each individual entity and on external complexity that measures the interactions among entities. In order to do that, existing traditional and object oriented metrics are refined to address the properties of object oriented software systems. The following sections discuss the metrics proposed for measuring the quality of object-oriented software.

#### 3.1 Object-Oriented Metrics

**3.1.1 System Size:**

*Total function calls (TFC)*
TFC tallies the number of calls to methods and system functions within the system, class, or method. This metric measures size in a way that is more independent of coding style than the LOC metric.

**Number of windows (NOW)**

The NOW metric counts the number of visible windows within the system. This indicates the size of the user interface.

### 3.1.2 Class and Method Size

**LOC and method invocations per class/method (MICM)**

These metrics are similar to the LOC system size metrics but focus on individual classes and methods.

**Public method count per class (PMCC)**

The number of public methods indicates the amount of behavior exposed to the outside world and provides a view of class size and how complex the class might be to use.

**Number of attributes per class and number of instance attributes per class (NAPC)**

The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities. Attributes can either be instance attributes, which are unique to each instance of an object, or class variables, which have the same value for all members of the class.

**Average Method Complexity (AMC)**

This is the sum of individual method complexities of all methods in the application divided by the total number of methods in the application. The individual method complexities are simply the Cyclomatic Complexities of the methods. Higher the average method complexity, more difficult it would be to maintain and understand the system.

### 3.1.3 Coupling and Inheritance

**Reuse Ratio of Inherited Methods Per Class (RRIM)**

\[
RRIM = \sum \frac{\text{Actual number of inherited methods reused within the class}}{\text{Total methods inherited from the base class}} \times \frac{\text{Total number of classes}}{\times 100\%}
\]

Simply defining methods in such a way that they can be reused via inheritance does not guarantee that those methods are actually reused. This metric provides a ratio between 0%
and 100% denoting no use of inheritance and maximum use of inheritance respectively per class in the application.

**Reuse Ratio of Inherited Attributes Per Class (RRIA)**

\[
RRIA = \sum \frac{\text{Actual number of inherited Attributes reused within the class}}{\text{Total attributes inherited from the base class}} \times 100%
\]

This is the ratio between the actual number of attributes reused and the potential number of attributes available for reuse per class in the application. Higher the ratio, higher will be the use of inheritance within the application.

### 3.1.4 Classes and Method Internals

**Percent of commented methods (PCM)**

This metric measures the extent to which you’ve internally documented your code. The metric counts the number of separate comment blocks rather than the number of comment lines.

\[
\text{PCM} = \frac{\sum_{i=1}^{n} \text{number}\_\text{of}\_\text{commented}\_\text{blocks}\_\text{in}\_\text{method}_i}{\text{total}\_\text{no}\_\text{of}\_\text{methods}\_\text{in}\_\text{application}} \times 100%\]

**Number of parameters per method (NPM)**

This metric measures the average number of parameters involved in invoking a method. A high number of parameters indicates a complex interface to calling objects, and should be avoided.

**Object Oriented Design Patterns Usage (OODP)**

Object Oriented Design Patterns usage within application allows design reuse. Thus, higher number of design pattern usage implies higher design reusability of the application. This can be expressed as the fraction of classes involved in the design patterns used in application, multiplied by the number of design patterns.

\[
\text{OODP} = \frac{\text{Number of design patterns used in the application} \times \text{number of classes involved in design patterns}}{\text{Total number of classes in the application}}
\]

### 3.1.5 Polymorphism

**Number of Overridden Methods in the Class Hierarchy (OMCH)**
$OMCH = \frac{\text{Number of overridden methods of the base class in the class hierarchy}}{\text{Total number of methods in the base class}}$

This metric is directly related to the object-oriented concept ‘polymorphism’. Higher the overridden methods in the class hierarchy, greater use of polymorphism may be ensured within a program.

### 3.1.6 Abstraction

**Number of Abstract Methods Per Class (AMPC)**

$$AMPC = \frac{\text{Number of abstract methods of the class}}{\text{Total number of methods in the class}}$$

This metric is associated with the abstraction concept in object orientation.

### 3.1.7 Encapsulation

**Public to Private Method Ratio (PPMR)**

This metric refers to a basic structural mechanism of the object-oriented paradigm as encapsulation.

$$PPMR = \frac{\text{Number of private methods in the project}}{\text{Number of public methods in the project}} \times 100\%$$

The metric is expressed as percentages, ranging from 0% (no use of encapsulation) to 100% (maximum use of encapsulation).

**Public to Private Attribute Ratio (PPAR)**

$$PPAR = \frac{\text{Number of private attributes in the project}}{\text{Number of public attributes in the project}} \times 100\%$$

This metric refers to the object-oriented paradigm concept encapsulation. This metric is also expressed as percentages, ranging from 0% (no use of encapsulation) to 100% (maximum use of encapsulation).

**Number of global/shared references per class (NGRC)**
This metric indicates the number of global variable references found within a class. Global references tend to break encapsulation and inhibit reuse. While global references may be difficult to eliminate entirely, they should be used as sparingly as possible.

### 3.2 Interpretation Guidelines

The interpretation guidelines were derived from the statistics obtained from large software projects. Some of these metrics are guidelines for Object Oriented design and development rather than metrics in the sense of quantitative measurements. While there are many guidelines for interpretation of metrics, there is insufficient statistical data to prove that a value of 8 for one metric is twice as complex or twice as “bad” as a value of 4. We, therefore, propose interpretation guidelines based on a comparison of the values, looking at the outliers to determine why they are different from the other modules of code. This is not an indication of “badness” but an indicator of difference that needs to be investigated. Table 1 gives a summary of the objectives for the values suggested above in the description of these metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Related Aspect</th>
<th>Interpretation Guideline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total function calls (TFC)</td>
<td>System Size</td>
<td>Low</td>
</tr>
<tr>
<td>Number of windows (NOW)</td>
<td>System Size</td>
<td>Low (trade-off)</td>
</tr>
<tr>
<td>LOC and method invocations per class/method (MICM)</td>
<td>Size</td>
<td>Low</td>
</tr>
<tr>
<td>Public method count per class (PMCC)</td>
<td>Class and Method Size</td>
<td>Low</td>
</tr>
<tr>
<td>Number of attributes per class and number of instance attributes per class (NAPC)</td>
<td>Class and Method Size</td>
<td>Low</td>
</tr>
<tr>
<td>Average Method Complexity (AMC)</td>
<td>Class and Method Size</td>
<td>Low</td>
</tr>
<tr>
<td>Reuse Ratio of Inherited Methods Per Class (RRIM)</td>
<td>Coupling and Inheritance</td>
<td>High (~60–70%)</td>
</tr>
<tr>
<td>Reuse Ratio of Inherited Attributes Per Class (RRIA)</td>
<td>Coupling and Inheritance</td>
<td>High</td>
</tr>
<tr>
<td>Percent of commented methods (PCM)</td>
<td>Classes and Method Internals</td>
<td>~20–30%</td>
</tr>
<tr>
<td>Number of parameters per method (NPM)</td>
<td>Classes and Method Internals</td>
<td>Low</td>
</tr>
<tr>
<td>Object Oriented Design Patterns Usage (OODP)</td>
<td>Classes and Method Internals</td>
<td>High</td>
</tr>
<tr>
<td>Number of Overridden Methods in the Class Hierarchy (OMCH)</td>
<td>Polymorphism</td>
<td>High (trade-off)</td>
</tr>
<tr>
<td>Number of Abstract Methods Per Class (AMPC)</td>
<td>Abstraction</td>
<td>High</td>
</tr>
<tr>
<td>Public to Private Method Ratio (PPMR)</td>
<td>Encapsulation</td>
<td>High</td>
</tr>
</tbody>
</table>
Table 1: Interpretation Guidelines

<table>
<thead>
<tr>
<th>Public to Private Attribute Ratio (PPAR)</th>
<th>Encapsulation</th>
<th>High</th>
<th>Encapsulation</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of global/shared references per class (NGRC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, as indicated in the metric Number of Overridden Methods in the Class Hierarchy (OMCH), there is a trade-off with many of the metrics. A higher Number of Overridden Methods in the Class Hierarchy will increase the level of inheritance but also shows improper use of abstraction principle. A high number of windows will increase testing efforts and lower the maintainability but will also accompany increased the extent of user friendliness of the user interface. A developer must be aware of the relationships of the structures and that altering the size of one metric can impact areas such as testing, understandability, maintainability, development effort and reuse [13].

4. RESULTS AND DISCUSSION

In the previous section, we proposed a collection of metrics and guidelines for measuring the quality of object-oriented software. Refining the existing metrics and adding some new metrics derive these metrics. In order to check the applicability of the metrics, a prototype system was developed and was executed over three medium and large size object-oriented systems produced in different application domains. Test result indicated that the proposed metrics could successfully be calculated. A sample output of the system when metrics are applied on an application is depicted in Figure 4.1.

![Figure 4.1: Experimental results](image)

Adhering to good object-oriented principles produces higher quality software. Many traditional metrics do not capture certain fundamental aspects of the object-oriented
paradigm. The metrics proposed in this paper make a better use of object oriented concepts and eliminate most of the drawbacks found in existing metrics.

Since object oriented system implementations use the concept of classes, the proposed metric number of windows can be used to find the size of the user interface, which is very easy to calculate.

Reusability is one of the major objectives of object-oriented paradigm. Higher the use of concepts like inheritance, aggregation and abstraction, the reusability is high. The existing metrics like number of children or number of inherited methods guarantees only that they are defined in such a way that they can be reused; but, do not guarantee that those methods are actually reused. The proposed metrics Reuse Ratio of Inherited Methods and Reuse Ratio of Inherited Attributes denote the level of actual reuse in the inheritance hierarchy.

The complexity and maintainability of a system was measured using existing metrics Lines of Code, Depth of Inheritance Tree and Cyclomatic Complexity and so on. This idea is further improved by the new metric Number of parameters per method. Higher the number of parameters per method, the complexity of the system and degree of difficulty in maintaining the system are also high.

The proposed metrics assist to measure the quality factors such as understandability, maintainability and reusability. The effect of these metrics in measuring the quality is summarized in Table 2.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Objective</th>
<th>Testing Effort</th>
<th>Understandability</th>
<th>Maintainability</th>
<th>Develop Effort</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFC</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOW</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MICM</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMCC</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAPC</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMC</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRIM</td>
<td>↑</td>
<td>↑</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>RRRA</td>
<td>↑</td>
<td>↑</td>
<td>▼</td>
<td>▼</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>PCM</td>
<td>▼</td>
<td>▼</td>
<td>↑</td>
<td>↑</td>
<td>▼</td>
<td></td>
</tr>
<tr>
<td>NPM</td>
<td>▼</td>
<td>▼</td>
<td>↑</td>
<td>↑</td>
<td>▼</td>
<td></td>
</tr>
<tr>
<td>OODP</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>OMCH</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>AMPC</td>
<td>↑</td>
<td>↑</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>PPMR</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>PPAR</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>NGRC</td>
<td>▼</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>▼</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Object Oriented Metrics Effects

6. CONCLUSION AND LIMITATIONS
Object oriented metrics exist and do provide valuable information to object oriented developers and project managers. We have found that a combination of “traditional” metrics and metrics that measure structures unique to object oriented development is most effective. This allows developers to continue to apply metrics that they are familiar with, such as complexity and lines of code to a new development environment. However, now that new concepts and structures are being applied, such as inheritance, coupling, cohesion, methods and classes, metrics are needed to evaluate the effectiveness of their application. Metrics such as Reuse Ratio of Inherited Methods applies in these areas.

Applying the proposed metrics leads to two actions: improve the product and improve the process. Although many software development organizations espouse that a well-defined process results in a good product, there is little evidence that conformance to process standards guarantees good products [14]. Therefore, metric programs should be comprehensive - dynamically evaluate the development process as well as statically evaluate the products.

At this time there are no clear interpretation guidelines for these metrics although there are guidelines based on common sense and experience.

It is difficult, if not impossible, to place a dollar amount on the benefits of a metrics program because as in the case of risk management, you are trying to measure something that did not happen. The benefits derived are also not only applicable to the current project but to future projects. As with any new project, whether it is implementing a new engineering design or a metrics program, start up costs is high. But as management and staff become familiar with the tasks and tools are developed, the costs decrease to a low maintenance level.

Currently, we have started data collection from different industrial strength software systems in different application domains, for software metric analysis to provide better interpretation guidelines based on a statistical model. Significant effort is being spent on methods to use the metrics to forecast the values of the selected goals and attributes forward to project milestones such as delivery of the software and also to propose and test software quality metrics that can evaluate software development process, which is lacking in the current system. Our long-term objective is to establish a numerical metric scale for assessment of all metrics that affect software quality.

REFERENCES


