A comparative study on different object oriented metrics

D.J.SAMATHA NAIDU1, P.CHITTI BABU2
1MCA Department, Annamacharya PG College of Computer Studies, Rajampet, Kadapa,India
2MCA Department, Annamacharya PG College of Computer Studies, Rajampet, Kadapa,India
1samramana44@gmail.com , 2peb_mca@yahoo.com

ABSTRACT

At present a huge research is going on software metric quality issues. Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts. Various object oriented metrics with other metrics like traditional metrics, orthogonal metrics and CK metrics were discussed in this paper. Orthogonal object oriented metrics describe what it meant for two or more object-oriented metrics to be orthogonal. The main focus of this study is to produce a minimal set of Orthogonal Object-Oriented metrics capable of analyzing code quality with the same degree of accuracy as afforded by a metrics set of a significantly larger cardinality. This comparative study on different object oriented metrics shows a direct positive correlation between the degree of object-oriented constructs and the level of quality for each software application.

Keywords: Object oriented metrics, Traditional metrics, Orthogonal metrics.

1. INTRODUCTION

Object-oriented modelling and design is a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behaviour in a single entity [RUM]. There are three fundamental characteristics required for an object-oriented approach: encapsulation, polymorphism, and inheritance. Encapsulation is not unique to the object-oriented paradigm; however polymorphism and inheritance are two aspects unique to the object-oriented approach [TEG]. These three aspects of the object-oriented paradigm are described below.

Encapsulation: encapsulation is a method to wrapping the external aspects of an object, which are accessible to other objects from the internal implementation details of the object that are hidden from other objects. Encapsulation prevents a program from becoming so interdependent that a small change has massive effects. For example, the implementation of an object can be changed without affecting the application that uses it. One may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or for porting.

Polymorphism: Polymorphism means having the ability to take several forms. For object-oriented systems, polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation. For example, In an organization a single employee may play many fertilise roles in an organization, so it shows the ability to take several forms.

Inheritance: Acquiring the properties of other objects, it allows programmers to define objects incrementally by reusing previously defined objects as the basis for new objects. For example, sharing of properties from parents to kids. In simple words we say that, Inheritance is a reuse mechanism.

2. COMPARISON BETWEEN DIFFERENT SOFTWARE METRICS

2.1 Traditional Metrics

Traditional metrics have been applied to the measurement of software complexity of structured systems since 1976 [MCC76]. This subsection presents the McCabe Cyclomatic Complexity metric along with two other popular traditional software design metrics, Source Lines of Code and Comment Percentage.

McCabe Cyclomatic Complexity (CC): Cyclomatic complexity is a measure of a module control flow complexity based on graph theory [MCC99]. 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 [MCC76], [MCC99]. 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 code [LOR94]. Counting the SLOC is one of the earliest and easiest approaches to measuring complexity. It is also the most criticized approach [TEG]. 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 [ROS95]. A high CP value facilitates in maintaining a system.
2.2 Object-Oriented Metrics

In 1994 Chidamber and Kemerer [CHI] proposed a now widely accepted suite of metrics for an object-oriented system. Basili validated the metrics suite in 1996 [BAS] and Tang in 1999 [TAN]. The six object-oriented metrics are listed below.

**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 [CHI]. 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 [CHI]. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree [CHI]. The deeper the inheritance tree for a class, the harder it might be to predict its behaviour 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 [CHI]. 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 [CHI]. 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 [CHI]. 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 required 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 [CHI]. 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.

2.3 Orthogonal Object-Oriented Metrics

**Orthogonal:** Orthogonality is a measure of intrinsically different characteristics of the code, therefore any correlation among the measured values is due to relationships among the target modules and not due to any relationships among the actual metrics themselves. For example, lines of code and number of comments are said to be non-orthogonal since adding comments simultaneously increases lines of code. However, source lines of code and number of comments are said to be orthogonal since source lines of code can be increased without any changes in the comment count.

**Orthogonal Object-Oriented (OOO):** Two object-oriented metrics are said to be Orthogonal Object-Oriented metrics if they are orthogonal.
3. RELATED WORK

The evolution categories proposed by Lanza were first described in his study on using evolution matrices to visualize the evolution of object-oriented software systems (Lanza 2001). Rows in evolution matrices represent classes, whereas columns denote versions of the target system. Each cell encodes two metrics at a time: its height scales to the number of instance variables (NIV) of the class denoted by the containing row, and its width is proportional to the number of methods (NOM) in the same class. Lanza evaluates evolution Software Qual J matrices using two Smalltalk systems, in which supernovas, white dwarfs, pulsars, stagnant, and dayfly classes are visually identified in terms of NIV and NOM. The evaluated systems, however, are not representative of industrial-strength object-oriented systems. Lanza and Ducasse (2003) propose the use of polymeric views, which contrary to Lanza’s evolution matrices, can encode up to five metrics measurements. A polymeric view is a graph representing a given relationship between source code entities (e.g.,classes), where nodes encode metric measurements by means of colours, position, and size.Colors are in gray scale, with white standing as the least value and black the maximum metric measurement. Positions are (x and y) coordinates, and node size encodes two measurements by its size and width. Polymeric views are designed to assist developers in understanding the structure and the design quality of software systems, along with information on how these two properties evolve over time. The authors argue that such an approach help detecting evolution patterns in terms of class, method, and attribute metrics, but excluded CK metrics from their analysis. Godfrey and Tu (2000) study the Linux Kernel and its evolution over 96 versions, showing that it follows a super-linear rate, contradicting Lehman and Turki’s hypothesis of an inverse square growth rate (Lehman et al.1998). Israeli and Feitelson perform a larger study, with the analysis of 810 Linux kernel releases over 14 years (Israeli and Feitelson 2010). Their analysis suggests that the kernel agrees with Lehman’s Law of Software Evolution. In particular, the authors report strong evidence toward continuing growth and change laws, but anecdotal evidence of the self-regulation and feedback system laws. Gonzalez-Barahona et al. (2009) study the evolution of the Debian Linux distribution, opposed to studies focusing on the kernel alone, and point out that the package mean size in Debian is often constant across stable releases. However, the number of packages and the LOC size of the distribution (the sum of all LOC in each source file) double at each release. They also find that 7 % of packages from version 2.0 are still present in version 4.0, and that around 18 % of such packages remained unchanged (something we would characterize as a stagnant behaviour). Herráiz and Hassan (2010) investigate the correlation between LOC and other software metrics. In particular, the authors analyze how LOC/SLOC measurement of C files in Arch Linux packages correlates with McCabe’s control flow complexity and Halstead’s metrics. Overall, they find a high correlation (C 84 %) between size and Halstead’s metrics, but an average correlation (60 %) between SLOC and McCabe’s cyclomatic complexity for non-header files. Finally, a low correlation between McCabe’s complexity and SLOC/LOC is present. As discussed by the authors, this is expected, as header files do not contain control flow information. Similar results are reported in Herráiz et al. (2007), but with FreeBSD as a subject of analysis. Altogether, these works measure correlations between metric values taken from a single version of the target systems, which are restricted to the domain of operating systems. Opposed to that, we use different versions of the systems in our experiments, thus taking into account the temporal variations over the metric values. Furthermore, our target systems comprise a rich set of applications from different domains. El Emam et al. (2001) points out the potential confounding effect of class size on CK metrics when predicting fault-proneness (El Emam et al.2001). Subramanyan and Krishnan further investigate such effect (Subramanyan and Krishnan 2003), but opposed to Eman, show a strong association of defects to a subset of CK metrics, even after controlling for size. Their results, however, are dependent on the programming language. Future work, intends to investigate correlations between evolution categories and bugs.

3.1 Previous work: The development of a large software system is a time and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. There is also a great interest in software metrics due to their potential for use as a cost saving device. The research intended to find a way to produce cheaper and higher quality software. The orthogonal object-oriented set of metrics will be selected from the core set of metrics that the SATC uses for code analysis. The set of orthogonal object-oriented metrics obtained in the study was applied to three real world industrial strength object-oriented systems to predict their overall quality. The level of quality found in these three systems is classified into three types: low, high, medium.

3.1.1 Existing work
1. The first study shows that stagnant, supernovas, white dwarfs, and dayflies are probable events in the lifetime of classes.
2. The second study shows that by monitoring only the occurrence of the evolution categories, reliable predictions of the values of metrics designed to measure coupling (CBO), both coupling and size (RFC), size (WMC), and to a less extent cohesion (LCO), can be made. On the other hand, there is no connection between the evolution categories considered in the paper and properties derived from inheritance relations (as measured by NOC and DIT metrics).
(a) External software quality metrics, such as number of warnings raised by static analysis tools and number of bugs can be considered.
(b) The use of evolution categories as an independent variable in bug prediction models is considerable.
(c) How software evolution categories in general (and their particular relations to software metrics as investigated in this paper) can be incorporated in software quality monitoring tools and models may be investigated.

A Simple and easy way to use minimal set of Orthogonal Object-Oriented metrics in the form of an equation and one standalone metric, which can be used to evaluate software quality, using the CK suite of object-oriented metrics as a superset were constructed. This was accomplished by observing that the correlation of the metrics contained in the CK metrics suite increased as the quality of code decreased. This motivated a relationship between some of the metrics in the CK suite of object-oriented metrics.

The reduced metrics suite was validated using three industrial strength software systems. By comparing the results obtained from the reduced metrics set approach with the results obtained from a full-scale code analysis conducted using the entire CK object-oriented metrics suite together with traditional metrics. The reduced metrics set approach was able to classify the software systems with respect to the level of code quality. Both the reduced metrics set approach and the full metrics set (CK metrics suite and traditional) approach resulted in the same software quality system classification. System A was high quality software, System B was medium, and System C low.

Table 1 summarizes this descriptive information along with other information for each System. The last two rows in the table were obtained from the SATC full-scale code analysis of these systems. The table shows a direct positive correlation between the degree of object-oriented constructs and the level of quality for each software application.

**Table 1: comparison between applications used to validate reduced orthogonal and ck metrics set.**

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code used</td>
<td>150k</td>
<td>300k</td>
<td>670k</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>56k</td>
<td>2000</td>
<td>2617</td>
</tr>
<tr>
<td>Language</td>
<td>Java</td>
<td>Java</td>
<td>C++</td>
</tr>
<tr>
<td>Type of Application</td>
<td>Commercial Software</td>
<td>Commercial Software</td>
<td>Commercial Software</td>
</tr>
<tr>
<td>Code Construct</td>
<td>Orthogonal and Ck metrics</td>
<td>Excellent Object oriented metrics</td>
<td>Traditional object oriented metrics</td>
</tr>
<tr>
<td>Quality</td>
<td>High</td>
<td>Medium</td>
<td>low</td>
</tr>
</tbody>
</table>

**CONCLUSIONS**

The reduced CK metrics set approach is very promising. A word of caution is prudent at this point: the reduced metrics set approach is not a silver bullet. The approach is more applicable to identify low quality code than high, due to specific theoretical concepts employed to develop the approach. However, this is a mammoth step in the right direction in reducing the turnaround time it takes to perform a code analysis on industrial strength software. Future research should be conducted with the aim of developing more appropriate reduced metrics set models for identifying high quality code and how this reduced object-oriented metrics set approach can be integrated into the software development lifecycle.

**ACKNOWLEDGEMENTS**

My sincere heartful thanks to my co-author Dr.P.Chitti Babu garu who gave a morale support to do this research work.

**REFERENCES**


AUTHOR BIOGRAPHY

D.J.Samatha Naidu, In 2005 received the MCA degree in Computer applications from university of SVU. In 2008 received MPhil degree in Computer Science from the University of MKU, India. In 2010 received the MTech Degree in Computer Science and Engineering. At present PhD scholar of Computer Science in Vikrama Simhapuri University, respectively. She is currently working as Assistant Professor at APGCCS College. Her research interests includes Computer networks, wireless and sensor networks, software testing. She has published more than 40 papers in related national and international conference proceedings and National & International journals. She is a life member of IACSIT, IAENG, CSTA, ACM and ISTE etc.

Dr. P. Chitti Babu has obtained MCA and Ph.D (Computer Science) form S.V. University, Tirupati, M.Phil ( Computer Science ) from Alagappa University, Karaikudi, M.Tech (Computer Science & Engineering ) from Acharya Nagarjuna University, Guntur. He is working as a Professor & Principal, Annamacharya PG College of Computer Studies, Rajampet, Andhra Pradesh with the experience of 13 years. He has published 15 papers in International and National Journals. He is a life member of CSI, IACSIT, IAENG and ISTE.