OBJECT ORIENTED METRICS IN SOFTWARE DEVELOPMENT

Ramesh Ponnala and Santhosh Kumar N

1Asst. Professor, Dept. of CSE, Sree Chaitanya Institute of Technological Sciences, Karimnagar, Telangana

ABSTRACT:
Object-oriented design and development has become a popular way of software development. Object oriented development requires not only a different approach to design and development which needs a different approach to software metrics. As an object-oriented programming language uses objects as its fundamental building blocks, the conventional oriented programs must be different from the standard metrics. Hence, a new set of metrics have been developed metrics may not be possible to apply directly in this area. So that software metrics for object oriented to fulfill the needs of developers, practitioners and quality controllers.

Keywords: Object-Oriented, Software Development, Quality, Size, Coupling, Cohesion Metrics

[1] INTRODUCTION

Object-oriented design and development are popular concepts in today's software development. Object oriented development has proved its value for systems that must be maintained and modified. Object oriented software implementation require a different approach from more conventional functional decomposition and data flow development methods. The behavioral and data flow approaches will be done by considering the systems functionality and/or attributes/data separately, object oriented analysis approaches the problem by looking for system entities that group them. Object oriented analysis and design focuses on objects as the primary agents involved in a computation; each class of data and related operations are collected into a single system entity. Software metrics are generally required features in the software management functions of project planning and project evaluation methods, they are of special importance with a new technology such as the object-oriented approach. This is due to the significant need to train current and new software engineers in generally accepted object-oriented principles.

In this paper, I present obtainable and new software metrics useful in the different phase of the Object-Oriented Software Development Life Cycle (SDLC). Metrics are used by the software industry to itemize the implementation and maintenance of the software.

Development and validation of software metrics is expected to provide a number of practical benefits. In general, techniques that provide measures of the size and of the complexity of a software system can be used to aid management in:

- Estimating the cost, schedule and size of future projects,
- Evaluating the productivity impacts of new tools and techniques,
- Establishing productivity with new technologies over time,
OBJECT ORIENTED METRICS IN SOFTWARE DEVELOPMENT

- Improving software product quality,
- Estimating future staffing needs, and
- Anticipating, reducing future maintenance requirements

More specifically, given the relative newness of the OO approach, metrics oriented towards OO can aid in evaluating the degree of object orientation of an implementation as a learning tool for staff members who are new to the approach. They may also in due course useful objective criteria in setting design standards for a vendor.

[2] RESEARCH WORK

There are two types of analysis that can be applied to current software metrics. The first category of analysis is that are leveled at conventional software metrics as they are applied to traditional software i.e., non object oriented programming software design and implementation. These metrics are generally analyzed as being without solid theoretical bases and failing to display what might be termed normal predictable behavior [Weyuker, 1988].

The second category is more specific to OO design and development. The OO approach centers on modeling the real world in terms of its objects, which is in contrast to older, more conventional that concentrate on procedure or algorithm implementation view that separated behaviors and attributes. The fundamentally different notions inherent in these two views, that software metrics are developed with traditional methods in mind do not readily lend themselves to notions such as data encapsulation, data abstraction, classes, objects, inheritance, polymorphism, and message passing. Therefore, given that current software metrics are subject to some general criticism and are easily seen as not supporting key object oriented synopsis, it seems appropriate to develop a set of new metrics especially designed to measure uniqueness of the object oriented approach. Some early work has recognized the shortcomings of existing metrics and the need for new metrics especially designed for object oriented. Some proposals are given by Morris, although they are empirically suggested rather than theoretically driven [1988]. Pfleeger also suggests the need for new measures and uses counts of objects and methods to develop and test a cost estimation model for object oriented development [Pfleeger, 1989; Pfleeger and Palmer, 1990]. Moreau and Dominick suggest other 3 metrics for object oriented systems, but they do not provide formal definitions [1989]. Lieberherr and his colleagues present a well-articulated, formal approach in documenting the Law of Demeter™ [1988] The Demeter system represents a formal attempt at defining the rules of correct object oriented programming style, building on concepts of coupling and cohesion that are used in traditional programming.

[3] OBJECT ORIENTED METRICS

Booch [2] defines object oriented design to be the process of identifying objects and their attributes, identifying operations required on each object and establishing interfaces between objects. Design of classes involves three steps: 1) definition of objects 2) attributes of objects and 3) communication between objects. Methods design involves defining procedures which implement the attributes and operations suffered by objects. Class design is therefore at a higher level of abstraction than the traditional data/procedures approach (which is closer to methods design). It is the task of class design that makes OOD different than data/procedure design [Taylor & Hechi, 1990]. An introduction to the fundamental concepts and terminology of object-oriented design is as shown below

Some of the terminologies commonly used in object oriented metrics are as follows

1. **Object**: Object is an entity able to save a state and offers a number of operations/behavior to either examine or affect this state.
2. **Message**: A request that an object makes of another object to perform an operation.

Ramesh Ponnala and Santhosh Kumar N
3. **Class:** A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which object can be created.

4. **Method:** An operation upon an object, available to all instances of class, need not be unique.

5. **Instantiation:** The process of creating an instance of the object and binding or adding the specific data.

6. **Inheritance:** A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes.

7. **Cohesion:** The degree to which the methods within a class are related to one another.

8. **Coupling:** Object A is coupled to Object B, if and only if A sends a message to B.

---

### Figure 1. Elements of Object Oriented Design

Pictorial view of object oriented terms is as shown below

- **Object Oriented**
  - **Object Design**
  - **Methods Design**
  - **Object Definition**
  - **Attributes of**
  - **Communication**

---

- **Class Super**
  - Attribute(s)
  - Method(s)

- **Class Sub1**
  - Attribute(s)
  - Method(s)

- **Class Sub2**
  - Attribute(s)
  - Method(s)

- **Class Sub3**
  - Attribute(s)
  - Method(s)

- **Object obj1**
  - Data Information

- **Object obj2**
  - Data Information

**Message 1**
Figure: 2. Pictorial View of Some Object Oriented Terms

In the above diagram the class Super is the super class of sub1, sub2, and sub3 classes. Here these three classes are acquiring the attributes and methods from its base class Super in addition to having its own attributes and operations. A sub class can also become a super class for other classes forming hierarchical structure. When an object is created to contain data or information, it is an instantiation of the class. Two objects obj1 and obj2 are the objects of sub1 and sub2 classes respectively. Classes communicate or interact by passing messages. When a message is passed between two objects, the classes are coupled. Message 1 is the way of coupling two objects.

The selected object oriented metrics are primarily applied to the classes, coupling, and inheritance. For some of the object-oriented metrics discussed here, multiple definitions are given; developers, researchers and practitioners have not reached a common definition. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics. Object oriented metrics are categorized like size, coupling, cohesion, inheritance etc.

A class is a blueprint/template from which objects can be instantiated. This set of objects shares a common structure and a common behavior defined by the set of methods. A method is a set of instructions that performs an operation upon an object and is defined in the class definition. A message is the way of communicating an object with another object to perform required operation. The operation executed as a result of receiving a message is called a method. Cohesion is defined as the degree to which the elements of a module belong together. It is a measure of how tightly related each piece of functionality expressed by the source code of a software module. Effective object oriented designs maximize cohesion because cohesion promotes encapsulation. In software engineering, coupling is the manner and degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. Classes (objects) are coupled when a message is passed between objects; when methods declared in one class use methods or attributes of another class.

Inheritance is the process of acquiring the attributes and methods of one class into another class in hierarchical relationship among classes that enables programmers to reuse previously defined objects including variables and operations.

Figure: 3. Object Oriented Example Application with 3 Classes as Sub Classes (Class Diagram)
3.1 Size Metrics

The size metrics given here measure the size of the system in terms of attributes and methods included in the class and capture the complexity of the class.

a. **Number of attributes per class (NOA)**

The NOA counts the number of attributes defined in a class. The Figure 3 shows the class diagram of Person information system. In this system, Number of Attributes (NOA) for person class is 4. So NOA=4 for Person class.

b. **Number of methods per class (NOM)**

The NOM counts the number of methods defined in the class. In Figure 3, class Person has two methods setData() and getData(). Hence NOM=2 for class Person.

c. **Weighted methods per class (WMC)**

The WMC is the count of sum of complexities of all methods in a class. The method complexity is measured using cyclomatic complexity. This should be normalized so that nominal complexity for a method is taken as unity. Consider a class K1, with methods M1, M2, M3…… Mn that are given in the class. Let C1, C2, C3……Cn be the complexities of the methods. WMC is defined as

\[
\text{WMC} = \sum_{i=1}^{n} C_i
\]

If method’s complexities are normal (value=1), then WMC=n, which is equal to number of methods. In the Figure 3, WMC for Person is 2, Student is 3, Employee is 3 and Worker is 2 (considering each method complexity to be unity).

d. **Response for a class (RFC)**

The RFC is the count of the set of all methods that can be called in response to a message to an object of the class or by some methods in the class. Member method in the class and member methods of other classes are both counted equally. It is “considered a measure of attributes of an object. Since it specifically includes methods called from outside the object, it is also a measure of communication between objects.”

This includes all methods accessible within the class hierarchy. 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 larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater 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.

The RFC for Student class in Figure 3 is the number methods that can be invoked in response to messages by itself, and by Person class. Hence The RFC for Student = 3 (self) + 2 (Person) = 5

Ramesh Ponnala and Santhosh Kumar N
3.2 Coupling Metrics

Coupling relations increase complexity, reduce encapsulation, potential reuse, and limit understanding and maintainability. An improvement of modularity is achieved when inter object class couples is minimized. Some of the coupling metrics are as follows

a. **Coupling between objects (CBO)**

The CBO for a class is the count of the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables by the other class. The more independent a class is, the easier it is to reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a class is harder to understand, change or correct by itself if it is interrelated with other classes. Complexity can be reduced by designing systems with the weakest possible coupling between classes. This improves modularity and promotes encapsulation.

In **Figure 4**, the Book class contains instances of classes Publications and Sales. The Book class delegates its publication and sales issue to instances of the publication and sales classes. The value of metric CBO for class book is 2 and for class Publication and Sales is zero (0).

b. **Data abstraction coupling (DAC)**

The DAC is a technique of creating new data types suited for an application to be developed. It provides the ability to create user defined data called Abstract Data Type (ADT) Li and Henry defined Data Abstraction Coupling (DAC) as:

\[ \text{DAC}=\text{number of ADTs defined in a class} \]

In **Figure 4**, there are two ADTs in class Book, pub and market. So DAC for book class is 2.

c. **Message passing coupling (MPC)**

The MPC is defined as the number of different messages sent out from a class to other classes excluding the messages sent to the objects created as local objects in the local methods of the class. Two classes can be coupled because one class sends a message to an object of another class, without involving the two classes through inheritance or abstract data type. The metric
an indication of how many methods of other classes are needed to fulfill the class’ own functionality.

In Figure 4, MPC value of class Book is 4 as methods in class book calls pub.getData (), pub.display (), market.setData (), market.display () methods.

d. Coupling Factor (CF)

Coupling can be due to message passing (dynamic coupling) or due to semantic association links (static coupling) among class instances.

It is desirable to reduce communication amongst classes and even if they communicate, very little information should be exchanged. It defined as:

$$ CF = \frac{\sum_{i=0}^{TC} \sum_{j=0}^{TC}[IS\_Client(C_i, C_j)]}{TC^2 - TC} $$

Where, TC is the total number of classes.

$$ IS\_Client(C_i, C_j) = \begin{cases} 1 & \text{if } C_i \text{ and } C_j \text{ are coupled} \\ 0 & \text{otherwise} \end{cases} $$

Coupling due to the use of the inheritance is not included in CF, because a class is heavily coupled to its ancestors via inheritance. If no classes are coupled, CF=0%. If all classes are coupled with all other classes, CF=100%.

3.3 Cohesion Metrics

Cohesion is the measure of the degree to which the elements of a module are functionally related. A strongly cohesive model implements functionality that is related to one feature of the software and requires little or no interaction with other modules. We have to maximize the interactions within a module.

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book_id, pub_id, book_name, auth_name, price</td>
</tr>
<tr>
<td>Add_book(), delete(), search_name(), search_author()</td>
</tr>
</tbody>
</table>

Figure: 5. Class Diagram of class Book in Library Information System

a. Lack of cohesion of methods (LCOM)

This metric is a count of the number of disjoint method pairs minus the number of similar method pairs. The disjoint methods have no common instance variables, while the similar methods have at least one common instance variable.

Let I_i be the set of all instance variables used by methods M_i and if there n such sets
Let \( P = \{ (I_i, I_j) | I_i \cap I_j = \emptyset \} \) and 
\[ Q = \{ (I_i, I_j) | I_i \cap I_j \neq \emptyset \} \]
If all \( n \) sets \( \{ I_1, I_2, \ldots, I_n \} \) are \( \emptyset \) then \( P = \emptyset \)

\[
LCOM = |P| - |Q|, \text{ if } |P| > |Q| \\
= 0 \text{ otherwise}
\]

A positive high value of LCOM implies that classes are less cohesive. Hence, low value of LCOM is desirable.

b. **Tight class cohesion (TCC)**

It is defined as percentage of pairs of public methods of the class with common attribute usage. Here in **Figure 5**, the class book has 6 pairs of public methods. Among them 3 methods are with common attributes usage

Hence, \( TCC = \frac{3}{6} \times 100 = 50\% \)

c. **Loose class cohesion (LCC)**

In addition to direct attributes, this measure considers attributes indirectly used by a method. Methods \( m \) directly or indirectly invokes a method \( m' \), which uses attribute \( a \). LCC is same as TCC except that this metric also considers indirectly connected methods. The LCC is defined as the percentage of pairs of public methods of the class, which are directly or indirectly connected. In **Figure 5**, LCC is same as TCC (50\%) since no method is invoked indirectly by the methods of the class book.

d. **Information flow base cohesion (ICH)**

ICH for class is defined as the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method. In **Figure 5** the method delete () will invoke the method search_name () which is the method of same class. Hence ICH is 1.

### 3.4 Inheritance Metrics

Inheritance Metrics are based on the property of Inheritance of object oriented software.

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

The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number 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. In **Figure 3**, DIT for the classes Student, Employee, and Worker is 1 since these 3 classes ancestor is only one class Person.

b. **Number of Children (NOC)**

The NOC is the number of immediate sub classes of a class in a hierarchy. In **Figure 3** NOC of the class Person is 3.

c. **Method inheritance factor (MIF)**
The MIF is Ratio of the sum of inherited methods in all classes of the system to the total number of methods for all classes.

\[ MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \]

Where \( M_d(C_i) = M_i(C_i) + M_d(C_i) \)
TC= total number of classes
\( M_d(C_i) = \) the number of methods declared in a class
\( M_i(C_i) = \) the number of methods inherited in a class

In Figure 3,
TC=4 (Total Number of Classes)
Let C1 = Person class
C2 = Student class
C3 = Employee class
C4 = Worker class
Therefore
\( M_i(C_1) = 0, M_i(C_2) = 2, M_i(C_3) = 2, M_i(C_4) = 2 \)
\( M_d(C_1) = 2, M_d(C_2) = 2, M_d(C_3) = 5, M_d(C_4) = 4 \)

Thus, \( MIF = \frac{0+2+2+2}{2+5+5+4} = \frac{6}{16} \)

d. Attribute inheritance factor (AIF)
AIF is defined as the ratio of the sum of inherited attributes in all classes of the system to the total number of attributes for all classes.

\[ AIF = \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} \]

Where, \( A_d(C_i) = A_i(C_i) + A_d(C_i) \)
TC= Total number of classes
\( A_d(C_i) = \) Number of attributes declared in a class
\( A_i(C_i) = \) Number of attributes inherited in a class

In Figure 3,
TC=4
Let C1 = Person class
C2 = Student class
C3 = Employee class
C4 = Worker class
Therefore
\( A_i(C_1) = 0, A_i(C_2) = 2, A_i(C_3) = 2, A_i(C_4) = 2 \)
\( A_d(C_1) = 4, A_d(C_2) = 5, A_d(C_3) = 8, A_d(C_4) = 3 \)

Thus, \( AIF = \frac{0+2+2+2}{4+9+13+7} = \frac{6}{33} \)

[4] CONCLUSION

Object oriented metrics exist and do provide valuable information to object oriented developers and project managers. The primary objective of this research was to investigate the applicability of Object–Oriented software metrics to measure the complexity of a object oriented software application. Complexity of object oriented applications can be evaluated at several dimensions like Size, method, class, inheritance, cohesion etc., using a variety of available
software metrics from Software Engineering Domain. From this, we conclude that there should be a compromise among internal software attributes in order to maintain a high degree of reusability while keeping the degree of complexity and coupling as low as possible. However, it needs further in depth future work to perform in depth object oriented software evaluation for developing high performance, efficient software applications using object oriented programming paradigm.

REFERENCES


