IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

# Assessing Coding Efficiency Cocomo-Iii Methods And Object Orientated Metrics Are Used. Enhance Software Defects

Neeta Mourya, Shanu K Rakesh Department of computer science and Engineering, Chouksey Engineering College, Bilaspur Chhattisgarh 495004, INDIA

***Abstract:*** Software development has gotten increasingly sophisticated and demanding, necessitating attention to even minor details. Problems in software development include quality degradation, cost, and schedule overruns.

Software organizations rely on measurement programs to control quality, evaluate errors, and manage costs during development. To measure effectively, software metrics must be continuously evaluated and integrated into the development process. Object-oriented design metrics quantify the quality of a class and its attributes.

This paper recommends using a combination of methods to improve coding efficiency and accuracy when evaluating projects using object-oriented approaches, including MOOD Metrics, CK metrics, and COCOMO-III. Using object-oriented ways to evaluate code helps identify certain factors. This directly addresses the software's quality. These findings can help enhance software estimation, quality training, and research, leading to more accurate project milestone estimates and fault-free software systems.

***Index Terms*** **- Object-oriented, mood, CK, COCOMO, Defects, measurement, code, etc.**

## I. INTRODUCTION

Software engineering creates a strategy for software development within a specific scope. Schedule and effort, with the necessary quality. Object-oriented design metrics are vital in the software environment. Analyzing metrics aims to improve software quality.

Software metrics have become crucial in software engineering. Software developers assess software characteristics to ensure consistent and full requirements, high-quality design, and testable code. Effective project managers evaluate process and product attributes to determine when software is ready for delivery and if budget has been exceeded.

Regular feedback from the development process helps determine the progress of tasks and projects. Tracking allows project managers to address unanticipated situations.

## II. PROJECT MANAGEMENT

Project management involves organizing and managing a team to perform work within a set scope, quality, schedule, and cost restrictions. Project management involves identifying necessary activities and allocating resources accordingly.

## III. COCOMO III MODEL

The COCOMO III model is an update on the popular COCOMO II Software Cost Estimation Model.
• A draft version of the model has been formulated and the next step is to calibrate the model to real-world data.
• The updates to the new model include
– functional size inputs
– a new Software Security parameter
– removal of a couple of COCOMO II parameters
– an update to some of the pre-existing COCOMO II parameters
– "Pre-Sets" to cost driver values based on application domain
o Real-Time
o Engineering
o Automated Information Systems

## IV. COMPARISON OF THE COCOMO MODEL

The "Cocomo 3 tools" generally refer to the three levels of the COCOMO model in software engineering: Basic COCOMO, Intermediate COCOMO, and Detailed COCOMO; each increasing in complexity by incorporating more detailed cost drivers, allowing for more accurate estimations based on project attributes like product complexity, hardware constraints, and team experience, making the Detailed COCOMO the most precise but also the most time-consuming to use.

Key Differences between the COCOMO levels:

- Basic COCOMO:
- Simplest model, only considering the size of the software (lines of code) to estimate effort and development time.
- Useful for quick, rough estimations on small projects.
- Intermediate COCOMO:
- Introduces a set of "cost drivers" (e.g., product reliability, developer experience, platform complexity) which are used to adjust the effort estimate based on project characteristics.
- Provides a more accurate estimate for medium-sized projects with moderate complexity.
- Detailed COCOMO:
- Most complex model, further breaking down the project into modules and applying cost drivers to each module individually.
- Offers the most precise estimation but requires extensive project details and is best suited for large, complex projects with diverse components.

Important points to consider when comparing COCOMO tools:

- Accuracy: Detailed COCOMO generally provides the most accurate estimations, followed by Intermediate and then Basic.
- Complexity: Basic COCOMO is the easiest to use, while Detailed COCOMO requires a significant amount of project information to be effective.
- Application: Use Basic COCOMO for quick estimations, Intermediate for most projects, and detailed for highly complex projects with diverse requirements.

## V. INTERNAL QUALITY OF OOD

Internal quality of Object-Oriented Design (OOD) refers to characteristics like maintainability, readability, and testability. Here are some other things to consider when evaluating the quality of OOD:

- Inheritance: While inheritance is useful for reusing code, it can go against the goal of decoupled classes.
- Composition: Composition is another way to achieve code reuse.
- Type versus class: A type is an interface, which is a collection of methods that an object responds to.
- Design for change: Patterns can help address changes that might otherwise require redesign.

- Application framework: An application framework is a set of libraries or classes that can be used to implement the standard structure of an application. This can save time for developers by reducing the amount of code they need to rewrite for each new application.
- Persistent objects: Identify objects that need to last longer than a single application runtime.
- Remote objects: Identify and define remote objects and their variations.

## VI. MOOD METRICS

Application quality is critical to the development of software systems, especially large-scale ones. High quality software would reduce the cost of software maintenance, and it enhances the potential software reuse.

In order to measure the software quality more quantitatively and objectively, software metrics (MOOD) give impression to be a powerful and effective methodology that decide a grade to an object-oriented application. So, in this section, we will discuss the mood factors to assess an object-oriented application.
The MOOD set includes the Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (POF) and Coupling Factor (COF). These metrics are defined at the system or subsystem3 level while in other approaches, such as the well know set proposed in [Chidamber94], the metrics are defined at the class level. Each MOOD metric is associated with such basic structural mechanisms of the object-oriented paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (POF) or message-passing and association (COF). The mathematical definition of each MOOD metric will be introduced after the underlying basic concepts are made clear. Each metric is expressed as a quotient where the numerator represents the actual use of one of those mechanisms for a given design.

The denominator, acting as a normalizer, represents the hypothetical maximum achievable use for the same mechanism within the same universe of discourse that is, considering the same classes and inheritance relations. As a consequence, these metrics are expressed as percentages, ranging from 0% (no use) to 100% (maximum use) and thus are dimensionless. This avoids the misleading, subjective or "artificial" units that are often found in the metrics literature. Being formally defined, the MOOD metrics avoid subjectivity of measurement and thus allow replicability. In other words, different people at different times or places can yield the same values when measuring the same systems.

## VII. CK METRICS

The CK Metrics Suite comprises six metrics, each providing insights into different aspects of software design and implementation. These metrics serve as invaluable indicators for various dimensions of software quality:

7.1 Weighted Methods per Class (WMC):
Measures the complexity of a class by assessing the sum of complexities of its methods.
Identifies potential code smells and helps in evaluating maintainability.

7.2 Depth of Inheritance Tree (DIT):
Indicates the maximum length from the node to the root of the inheritance tree.
Offers insights into the hierarchical structure of the classes and potential complexity.

7.3 Number of Children (NOC):
Represents the number of immediate subclasses a class has.
Provides an understanding of class reuse and potential dependencies.

7.4 Coupling Between Object Classes (CBO):
Measures the number of classes to which a class is coupled.
Helps in evaluating the level of interdependence among classes.

7.5 Response for a Class (RFC):
Counts the number of methods that can potentially be executed in response to a message received by an object of the class.

Highlights the potential interactions and responsibilities of a class.

7.6 Lack of Cohesion in Methods (LCOM):
Measures the lack of cohesion among methods in a class.
Indicates how closely related or unrelated the methods within a class are.

## VIII. FACTOR CALCULATION

### 8.1 Method Hiding Factor (MHF)

MHF can be calculates by using the following mathematical formula:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Where,
$M_h(C_i)$ = hidden Methods in class $C_i$
$M_d(C_i) = M_v(C_i) + M_h(C_i)$: Methods defined in $C_i$
$M_v(Ci)$: visible Methods in class $C_i$
TC: Total number of Classes

### 8.2 Attribute Hiding Factors (AHF)

AHF can be calculates by using the following mathematical formula:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Where,
$A_h(C_i)$ = hidden attributes in class $C_i$
$A_d(C_i) = A_v(C_i) + A_h(C_i)$: attributes defined in $C_i$
$A_v(Ci)$: visible attributes in class $C_i$
TC: Total number of classes

### 8.3 Inheritance Factor (MIF)

MIF can be calculated by using the following mathematical formula:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Where,
**$M_i$:** inherited methods
**$M_a(C_i)$** = $M_d(C_i) + M_i(C_i)$: attributes defined in $C_i$
**$M_d(Ci)$:** defined methods
**TC:** Total number of classes

At first sight, we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability fade away quickly [1,2,3,4].

### 8.4 Attribute Inheritance Factor (AIF)

AIF can be calculated by using the following mathematical formula:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Where,
**$A_h(C_i)$** = hidden attributes in class $C_i$
**$A_d(C_i)$** = $A_v(C_i) + A_h(C_i)$: attributes defined in $C_i$

**$A_v$(Ci):** visible attributes in class $C_i$
**TC:** Total number of classes

### 8.5 Coupling Factor (COF)

It measures the coupling between classes.
COF can be calculated by using the following mathematical formula:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

Where,
Is client ($C_c$,$C_s$) =| 1 if ($C_c \Rightarrow C_s$)^($C_c \neq C_s$) , 0 otherwise
**TC:** It denotes the total number of classes.

### 8.6 Polymorphism Factor (POF)

Polymorphism means having the ability to take several forms. In OO systems, polymorphism allows the implementation of a given operation to be dependent on the object that "contains" the operation 2, 3, 4, and 6.

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}\left[M_n(C_i) \times DC(C_i)\right]}$$

Where,
**$M_o$($C_i$):** overriding Methods in class $C_i$
**$M_n$(Ci):** new Methods in class $C_i$
**$D_C$($C_i$):** number of Descendants of Class $C_i$ (derived classes)
**TC:** Total number of Classes

## IX. Programmer Coding Evaluation And Performance Measurement

Our strategy prioritized programmer efficiency and skill. This tool evaluates and analyses software metrics using Chidamber & Kemerer and MOOD metrics for typical Java libraries and applications. It also adds Java bindings for these metrics. Examining the results provides insights into how different technologies execute object-oriented approaches. Combining this knowledge with validation studies from other academics helps optimize software design and save costly maintenance duties, resulting in higher-quality software. The tool was created for the project management team, who are responsible for completing the project.

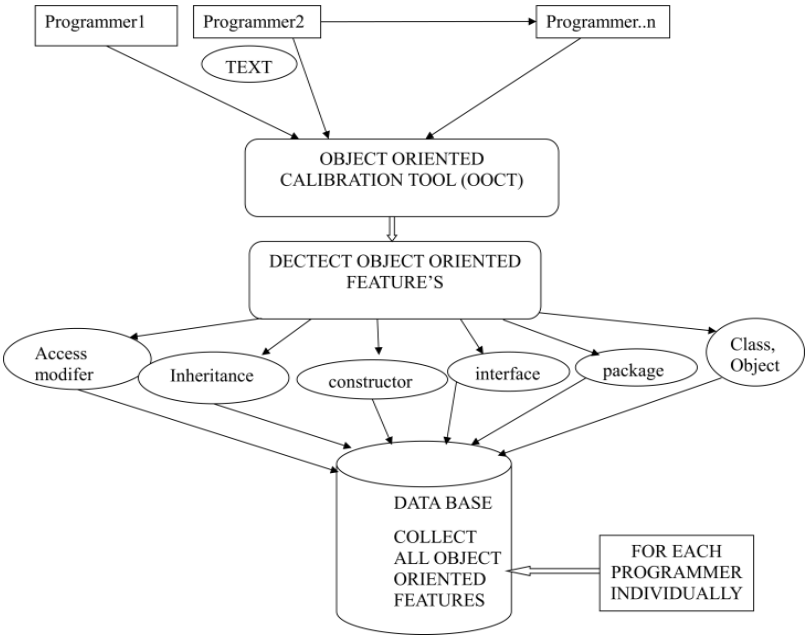


Fig 9.1 programmer coding key evaluation

Metric-based examination of programming language libraries can reveal structural and design similarities between them. Thus, we can get a more generalized picture of

Software design heuristics. Analysis data can expose the intrinsic complexity of standard libraries, which may be inherited by software applications that use them. Additional future research directions.

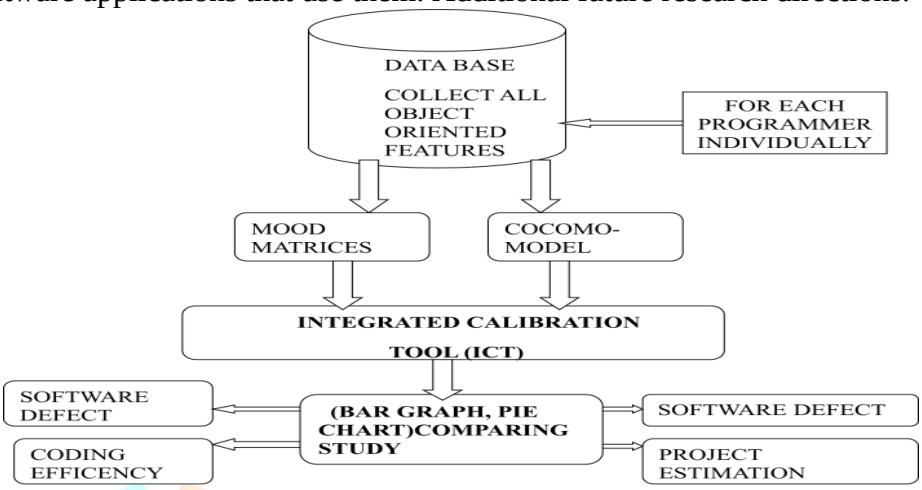


Fig 9.2 coding evaluation & performance measurement.

## X. DEFECT EVALUATION

Defect evaluation is the process of assessing the quality of a product or material to identify and characterize any defects. It can involve collecting and analysing data, creating defect evaluation lists, and developing defect pattern libraries.

Applications:
1. Pharmaceutical manufacturing
Defect evaluation lists and defect pattern libraries are used to assess the quality of batches of pharmaceutical products.
2. Packaging
Defect evaluation lists help manufacturers and suppliers deal with customers and ensure quality assurance.
3. Composite materials
Defect evaluation can involve using ultrasonic reflections to detect defects in complex microstructures.
4. Pipelines
Defect evaluation can involve assessing the geometry of corrosion defects to determine residual strength and failure pressure.

Steps
1. Identify and list potential defects
2. Categorize potential defects
3. Create a defect pattern library
4. Create a defect evaluation list
5. Analise data to determine defect frequency and behaviour
6. Compare defect evaluation results to other methods, such as X-ray CT scans.

Related terms Defect detection, Defect assessment, Defect evaluation lists, and Defect pattern libraries.

## XI. FUTURE WORK

Analysis a specific set of object-oriented metrics for various Java technology libraries. A similar analysis can be performed for other competing technologies such as .NET C++ etc. CK and MOOD belong to the class of structural and complexity metrics. We can also evaluate the efficiency with the help of AI technology.

- Programmer coding evaluation using multimedia data.
- Training and Research area.
- Reduce execution time and space complexity.
- Better report generation of the project which can be a blueprint for forwarding engineering process.
- Better HR management.
- Better result of the program efficiency.
- Better to explain the project execution time.

## XII. REFERENCES

[1] Jaechang Nam , Wei Fu, Student Member, IEEE, Sunghun Kim, Member, IEEE, Tim Menzies , Member, IEEE, and Lin Tan, Member, IEEE "Heterogeneous Defect Prediction" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 9, SEPTEMBER 2018.
[2] Ping Cao a , Ke Yang b, Ke Liu c "Optimal selection and release problem in software testing process: A continuous time stochastic control approach" European Journal of Operational Research March 2, 2019.
[3] Magne Jørgensen and Martin Shepperd, "A Systematic Review of Software development Cost Estimation Studies," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 1, JANUARY 2007.
[4] Tirimula Rao Benalaa, Rajib Mallb "DABE: Differential evolution in analogy-based software development effort estimation "Swarm and Evolutionary Computation 38 (2018) 158–172.
[5] Manish Agrawal and Kaushal Chari "Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 3, MARCH 2007.
[6] Barbara A. Kitchenham, Robert T. Hughes, and Stephen G. Linkman, "Modelling Software Measurement Data," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 27, NO. 9, SEPTEMBER 2001.
[7] Alexander Egyed, Member, IEEE "Automatically Detecting and Tracking Inconsistencies in Software Design Models" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 37, NO. 2, MARCH/APRIL 2011.
[8] Ning Nan and Donald E. Harter, Member, IEEE "Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 35, NO. 5, SEPTEMBER/OCTOBER 2009.