Classification of Technical and Management Metrics in Object Oriented Software Engineering

Devesh Kumar Srivastava and Ayush Singh

Abstract From the dawn of technical age, software projects have always been significantly difficult to estimate and manage well. Over costing, delays in schedule, and out front cancellations of these projects have been a common issue we are facing for over 50 years now. Poor quality of the software, when delivered, remains to be a highlighted issue. Although, several tools for management of software projects are available, when even used by experienced managers to estimate the complexity of software project raises the odds of unsuccessful completions. Project management tools have many subcategories. However, they can be classified into two major groups of tools: [1] For estimation and planning of software projects; [2] For reporting and tracking of costs and status of project while they are underway.

Keywords Object oriented programming · Software metric · Java

1 Introduction

Over nearly past 50 years, this industry has grown into one of the leading industries of this century. On global basis, software and its applications are the main tools of various corporations, govt. agencies and even allied forces. This industry employs thousands of professionals every year [3]. Because of the high costs and importance of development and maintenance of software combined with lower optimal quality,

D.K. Srivastava (✉) · A. Singh
SCIT, Manipal University Jaipur, Jaipur, India
e-mail: devesh988@yahoo.com

A. Singh
e-mail: mail.ayushsingh@gmail.com

N. Modi et al. (eds.), *Proceedings of International Conference on Communication and Networks*, Advances in Intelligent Systems and Computing 508,
DOI 10.1007/978-981-10-2750-5_29

it becomes mandatory to measure both productivity and quality of the software with a high precision. This research paper is divided broadly into 2 sections, Technical and Management Software Metrics and they are analyzed as follows.

1. For Technical Software Metrics, Object Oriented programs were studied and selected software metrics are applied to estimate their complexity to explore and compute whether each of these proposed metrics are independent of each other and effective in calculating complexity of any proposed program
2. For Software Management Metrics, Requirement Engineering is performed on several aspects of projects development. Work is broken down for providing an overview of development and 11 composite software management metrics are derived for every stage of development to support. Both sets of these metrics aim to describe a quantitative method for the prediction of difficulty it for designing, implementing, and maintaining the system. Their secondary goal is to create a mutual understanding for to initiate some important cost changes to decrease unnecessary costing over lifespan of given software

1.1 Technical Metrics

As we know, metrics are the key source of knowledge used for making decisions, a vast majority of Object Oriented metrics were proposed over a period of 10–15 years to exhibit the functioning and architecture of an Object Oriented program and also are directly related with the other extrinsic factors to measure quality [4]. As the total count of metrics which are available today is large, the sequential method to perform the calculation of the required metrics and obtain result from the resulting values becomes tedious. Also, as the count of these metrics which are proposed are bigger as compared to features like cohesion, coupling, size, polymorphism and inheritance exhibited by these metrics, our objective is to explore and compute whether each of these proposed metrics are independent of each other or is it possible to select a portion of the metrics which have equal measures and use like the preselected set.

To achieve that target, a set of 11 metrics is first selected and is defined with examples. The corresponding metric values are calculated for a standard project study and their interrelationships from the values are interpreted and recorded [5]. The faulty classes were defined based on previous investigations which were derived empirically.

2 Research Methodology

The OO metrics that were selected for the analysis of this project can be further grouped into 4 different categories which are coupling, class, reuse and inheritance metrics. Metrics taken in consideration are defined below [3, 6–10].

1. **Response For Classes (RFC)**. The class with a mathematical set of response (RFC) can be technically termed as a mathematical group of all methods which could be interpreted as a reply to any message that is received by any object made for the class.
2. **Weighted Methods for every Class (WMC)**. It is calculated by counting the total of the complexities of individual methods of that class.
3. **Data Abstraction Coupling (DAC)**. This metric formally represents the total number of all the instances of classes other than given class and within it. It is the number of all the external classes that the given classes may use.

$$\text{DAC} = \text{count of ADTs defined per class}$$

4. **Message Passing Coupling (MPC)**. This coupling metric helps us to measure the total count of all those calls by a method that are defined inside the methods of that sample class to the methods in others.
5. **Inheritance Tree's Height/Depth (DIT)**. The metric measures the degree of effect of ancestor/parent classes on the given class. The class's depth according to the tree made by estimating inheritance is directly proportional to the behavior inherited from its super class(s).
6. **Count of Subunits (NUS)**. The total count of subunits represents the count of all procedures and functions that are defined for a given class.
7. **Number Of Children (NOC)**. This metric measures the total count of immediate children in the model of hierarchy.
8. **Inheritance Dependencies**. The metric aims to exhibit characteristics of tree of inheritance. Inheritance Tree's Height/Depth = max (length of the path of the inheritance tree)
9. **Factoring Effectiveness**. Hierarchies of Inheritance can be controlled by the process named factoring. This process aims at minimizing the count of places inside the hierarchy tree of inheritance within which a selected method is executed. It can be estimated by:

$$\text{Factoring Effectiveness} = \text{Count of specific methods}/\text{Total count of every method}$$

10. **Reusability Ratio (RR)**. This metric is informally represented as

$$U = \text{Total Count of super class}/\text{Total count of all the classes}$$

11. **Specialization Ratio (S)**. This metric is mathematically represented as

$$S = \text{Count all the subclasses}/\text{Count of superclass}$$

To further understand the application of these metrics, we are going to calculate these metrics on a sample java source code which exhibits the characteristic of OOPs like polymorphism, inheritance and data abstraction to know if these metrics fulfill our demand of an accurate estimation of complexity or not.

2.1 Source Code in Java

Page 1

```
import java.io.InputStream;
import java.util.Scanner.;
class sportsman
{ static Scanner in = new
Scanner( System.in );
     public int no; private
string fullname; public void
readinput()  {
System.out.println("Please
Input Full Name:"); try {
String fullname=
in.nextLine();
System.out.println ("Please
Input Number :");
    int no=in.nextInt();}
catch(Exception ex){} }
public void showoutput()  {
```

Page 2

```
System.out.println("Your
Fullname is:" +fullname);
System.out.println ("The ID
is="
+number); }
public void display()
{
System.out.println("This
completes Sportsman Class");
} }

class golfplayer extends
sportsman
{ protected float penalties ;
protected String trophyname ;
public void readinput()
{ try{
```

Page 3

```
super. readinput();
System.out.println("Please
Input Trophy Name:");  String
trophyname= in.nextLine();
System.out.println("Please
Input Golf Club Penalties:");
float penalties =
in.nextFloat();}}
catch(Exception ex){}
} public void showoutput()
{
super. showoutput();
System.out.println ("Trophy
Name:" +trophyname);
System.out.println("Penalties
:"+penalties);
} public void display1()
{
System.out.println("This
completes golfplayer class");
} }
class badmintonplayer extends
sportsman { protected int
shuttle ; public void
readinput()
{
 try{
super. readinput();
System.out.println ("Please
Input the count of shuttles
used:") ; int shuttle=
in.nextInt();}
;} catch(Exception ex){}
} public void showoutput()
{super. showoutput();
System.out.println ("Total
Count shuttles used:"
+shuttle);
} }
class coach inherits
sportsman
{ protected int workhrs;  int
sum=0;
void readinput()
{ try{
super. readinput();
System.out.println ("Please
Input no. of Work Duration:")
;
```

Page 4

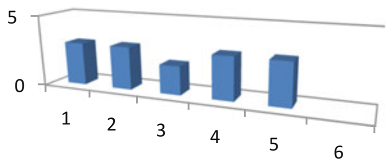
```
measure();
} catch(Exception ex){}
}
public void measure( )
{
total = workhrs*50;
}void showoutput();
{
super. showoutput();
System.out.println ("Work
Duration:" +workhrs) ;
System.out.println ("Sum:"
+sum);
} }
class hourwisecoach inherits
coach
{ protected int workhrs;
protected double inc; void
readinput()
{ try{
 super.readinput(); income();
}
catch(Exception ex){}
}public void income()
{
inc=super.workhrs*200;
}
void showoutput()
{
super. showoutput();
System.out.println ("The
Total Income is " +inc);

public static void
main(String args[])
{ golfplayer g1 = new
golfplayer();
golfplayer g2 = new
golfplayer();
badmintonplayer b1 = new
badmintonplayer();
coach c1 = new coach();
hourwisecoach s1 = new
hourwisecoach();
System.out.println ("Please
Input credentials of first
golf player"); g1.
readinput();
System.out.println ("Please
```

Page 5	Page 6
`int workhrs= in.nextInt(); golf player"); g2. readinput(); System.out.println Please Input credentials of first Badminton player"); b1. readinput(); System.out.println ("Please Input credentials of firstSystem.out.println ("Credentials of first coach"); c1. showoutput(); System.out.println ("Credentials of first hourwise coach "); s1. showoutput(); coach”); c1. readinput();`	`Input credentials of second System.out.println ("Please Input credentials of first hourwise coach"); h1. readinput(); System.out.println ("Credentials of first golf player "); g1. showoutput(); System.out.println ("Credentials of second golf player "); g2.showoutput(); System.out.println ("Credentials of first badminton player"); b1. showoutput(); } }`

2.2 Calculation of Selected Object Oriented Metrics

1. **WMC (Weighted Methods for every Class)**: This metric is estimated by analyzing and counting all methods present in every class.

a	b	c	d	e	f
WMC	3	3	2	3	3



2. **RFC (Response For a Class)**: This metric is calculated by the count of every procedures that may be interpreted in unique class [11].

a	b	c	d	e	f
RFC	3	5	4	5	7

3. **NOC (Count of Immediate Children)**: The count of immediate children is calculated by counting all direct subclasses of a class.

a	b	c	d	e	f
NOC	3	0	0	1	0

4. **DIT (Inheritance tree's Depth)**: The height of the phenomena of inheritance is the basic level of a specific class within the scheme of hierarchical inheritance, and the base class being on level Zero.

a	b	c	d	e	f
DIT	0	1	1	1	2

5. **NUS (Count of all Subunits)**: The total count of subunits is basically the quantity of all the procedures and functions termed for any given class.

a	b	c	d	e	f
NUS	3	3	2	3	3

6. **DAC (Data Abstraction Coupling)**: This metric mathematically specifies the total number of instances of any specifically selected classes present in a selected class [12].

a	b	c	d	e	f
DAC	0	1	0	1	0

7. **MPC (Massage Passing Coupling)**: This metric exhibits the total number of a function/method calls or calls of procedure that were directed to any extrinsic units.

a	b	c	d	e	f
MPC	0	2	2	2	4

8. **Factoring Effectiveness (FE)**: Effectiveness of Factoring = Total count of specific methods/Count of methods = 4/14 i.e. 0.29
9. **Inheritance Dependencies (ID)**: Depth of the Inheritance tree = max (length of path of inheritance tree), according to the class specific diagram, Depth of Inheritance tree = 3
10. **Specialization Index (SI)**: The index of specialization (S) = Total count of subclasses/Total count of all the superclass. Index of Specialization = 5/2 = 2.5
11. **Reusability Ratio (RR)**: Reusability ratio = U = Total count of superclass/Actual count of classes. Reuse ratio = 2/5 = 0.4
    After studying and analyzing the above, following traits were derived with the relation of the complexity of selected programs written in java.

1. Weighted Methods for every Class (WMC). A large value of WMC leads towards larger quantities of errors. Classes having a greater count of methods are actually more software specific, and limits reusability. Study suggests that increasing the average of WMC also elevates complexity but lowers quality.
2. Response for a Class. Higher count of methods from any class which may be called through messages, the larger the complexity of that class. Programs written in java are somewhat not complex because the average value for a specific metric is less for such codes.
3. Depth of Inheritance tree. If any specific class is deep in that hierarchy, more methods will be inherited, increasing its complexity. Deep trees have greater design complexity, as more number of classes and methods are involved, but reusability also increases due to inheritance.
4. Number of Children. A high count of immediate children indicate a larger chance of malpractice of abstraction, which might be a case of misusing of sub classing ability. However, higher NOC exhibits higher reuse, as inheritance is another form of reuse but also increases complexity [4]. A class with more children requires more testing but fewer errors due to higher reuse.
5. Message passing and coupling. A higher number of passing of messages indicates higher coupling between given classes in a code. It makes them highly dependent and spikes the overall complexity of that java code and also makes the scalability and modeling difficult.
6. Data Abstraction Coupling. Complexity of the software increases as DAC increases. For Java, data is more important than methods and procedures. The data is usually not shown to the customer or user. DAC is generally not high for all programs of java.
7. Count of Subunits. As frequently as the count of methods and functions spike, classes grows more prone to error. So, complexity somehow also increases with a growth in quantity of such metric. So, the final value of NUS metric is discovered as low generally for programs in java.
8. Inheritance Dependencies. As a tree with a greater depth is somewhat more difficult in testing, a greater value of ID indicates greater complexity of programs in java. Comprehensibility might be decreased for a larger count of layers of inheritance.
9. Factoring effectiveness. A smaller count of places of implementation for an average method means that fewer mistakes were made while designing. An inheritance hierarchy with a high factor is the largest degree until which a function can be reused. A highly factored application indicates a smaller count of places of implementation for an average function with lower complexity.
10. Specialization Index. It misses the empirical and theoretical validation. If Specialization Index is increased, class maintainability becomes more difficult. The value of SI is usually high for java programs increasing usability and hence complexity.

11. Reuse ratio. If the final value of RR is coming to be zero, nothing is inherited. As this value approaches one, the tree of inheritance deepens as a chain with single root and single leaf exactly. When RR was estimated for several other programs in java, we discovered that the results were intermediate.

3 Concluding Results and Further Research

The Canonical aim of the aforementioned research study was basically to validate and verify the utility of proposed Management Metrics of Software and the applicability of carefully selected OOP Metrics to calculate the total complexity of an Object Oriented code or software. Complexities of such software and application can be measured with several types of metrics. But in this study we evaluated and classified a defined set of 11 well known OOP metrics which are measured to serialize software codes with the complexities to estimate maintainability for those programs. By this work, we may deduce that we must compromise the intrinsic attributes of software to continue maintaining a high scalability while also maintaining complexity and coupling as low as possible. Although, further in depth study may be focused on empirical validation of metrics for an environment of multi languages, we can still expect from our study and analysis that it can be further used by software project and application developers for developing a reliable, error free, maintainable Java software product.

References

1. Patrick Naughton & Herbert Schildt "Java: The Complete Reference", McGrawHill Professional, UK, 2008.
2. S. Chidamber, and C. Kemerer, "Towards a Metrics Suite for Object Oriented Design," Object Oriented Programming Systems, Languages and Applications (OOPSLA), Vol 10, 1991, pp 197–211
3. Brij Mohan Goel, Pradeep Kumar Bhatia, "Analysis of reusability of object oriented systems using object-oriented metrics ACM SIGSOFT Software Engineering Notes" Volume 38 Issue 4, Pages 1–5 July 2013.
4. Briand, W. Daly and J. Wust, Exploring the relationships between design measures and software quality. Journal of Systems and Software, 5 245–273, 2000.
5. K. Morris, "Metrics for Object-oriented Software Development Environments," Master Thesis, MIT, 1989.
6. Churcher, N.I. and M.J. Shepherd, "Towards a Conceptual Framework for Object Oriented Metrics," ACM Software Engineering Notes, vol. 20, no. 2, April 1995, pp. 69–76.
7. Roger S. Pressman: Software Engineering, A practioner's Approach, Fifth Edition, 2001.
8. S.R Schach, Object-Oriented and Classical Software Engineering. Tata McGraw-Hill, 2002 http://www.mhhe.com/engcs/compsci/schach5/student/airgourmet.java.java
9. Mahfuzul Huda, Dr. Y.D.S. Arya, and Dr. M. H. Khan. "Testability Quantification Framework of Object Oriented Software: A New Perspective." International Journal of Advanced Research in Computer and Communication Engineering, Vol. 4, Issue 1, Jan, 2015.

10. Abdullah, Dr, M. H. Khan, and Reena Srivastava. “Testability Measurement Model for Object Oriented Design (TMMOOD).” International Journal of Computer Science & Information Technology (IJCSIT) Vol. 7, No 1, February 2015.
11. Arti Chhikara1and R.S. Chhillar, “Analyzing the Complexity of Java Programs using Object - Oriented Software Metrics” IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 3, January 2012.
12. Er. V.K. Jain. “The Complete Guide to Java Programming”, First Edition, 2001.