2020

Introducing a new object-oriented complexity metric

Group Members

Registration No	Name	Responsible Complexity Factor
IT18015836	**R.M.M.P.K. Rajapakse**	**Weight due to the number of functions**
IT18042634	**L.A.D.Y.S. Perera**	**Weight due to using access modifiers with attributes**
IT18166170	**J.M.A.I. Mudalige**	**Weight due to type of control structure**
IT18098310	**R.N.A.M.S.C.B. Warakagoda**	**Weight due to using abstraction in class hierarchy**

Abstract.

Quality in software is a key measure of the software process. It offers a consistent record of progress towards growth, a basis for setting goals and a structure for current action. There are hundreds of software metrics exist for both conventional and object-oriented applications [3]. In OOP, core principles of programming such as Encapsulation, Polymorphisms, Abstraction and Inheritance influence the programming, structure and style of coding. The challenge is how to reduce the OOP Complexity and comply with the main OOP design concepts. In this report we focus on introducing a new object-oriented complexity metric. The new complexity metric has added the following characteristics:

Weight due to using abstraction in class hierarchy, Weight due to using access modifiers with attributes, Weight due to the number of functions, Weight due to type of control structure.

Table of Content.

1.Introduction.

Software metrics take a significant role of the software development process.so it’s considered as a key quality driver for each software developer. Software metrics provide maintainability, understandability, readability, performance and availability of programming code. The complexity in measuring code should shape how developers build their code and how to make it less complicated. Therefore, reducing the complexity of the code would reduce the risk of errors in development and production environments and will have significant advantages in terms of testability and robustness. We proposed new metric by improving the existing traditional and object-oriented metrics. We used four factors to build our new complexity. Two of them are object oriented related and other two is non object oriented. We used Encapsulation, Abstraction, no of functions and control structures.

2. Factors that used to new metric

2.1 Weight due to using abstraction in class hierarchy (W_A)

In our new complexity metric we choose abstraction as one of the object oriented factors. Abstraction is one of three main concepts of object-oriented programming.

Abstraction is the method of adding or eliminating properties from something in order to reduce it to a collection of basic features. Its main objective is to deal with complexity by hiding unnecessary user information. This allows the user to apply more complex logic on top of the given abstraction without knowing the hidden meaning or even thinking about it.

Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. Similarly as abstraction often works in art, the remaining object is a representation of the original, omitting unnecessary detail.

To consider the complexity added due to using abstraction, in the proposed matric,

- A weight of zero is assign for statements without any kind of abstraction
- A weight of one is allocated for statements which are having a abstract class or abstract method.

2.2 Weight due to using access modifiers with attributes (W_{AM})

• In our complexity metric, we get one factor as an encapsulation. Encapsulation is one of the key principles in object-oriented programming. It is frequently used to enforce a framework for hiding information. This method restricts attribute accessibility to the current class and uses public getter and setter methods to monitor and limit external access to these attributes. The major objective of software development is reducing the complexity of a program. Encapsulation reduces the complexity and increases the reliability, maintainability, and understandability of the system.[1][2]

• To consider the complexity encapsulation, we get its access modifiers and assign values to those access modifiers. Object data will always be restricted to private or protected for better encapsulation. So out of these two access modifiers, we get private access modifier as the most suitable one for information hiding. because Only class methods have access to such data and from the instance of the class, private elements are inaccessible. so, we assigned value 2 to the private access modifier. Then we get protected access modifier and assigned value 1 . because in protected access modifier only accessible for its derived class. and we assigned value 0 for public access modifier. Because If a class member is declared with the public access modifier, then this class member is accessible from any code in the given package and outside the package for all methods of all classes.

- Private access attribute weight two
- Protected access attribute weight one
- Public access attribute weight zero

2.3 Weight due to the number of functions (W_{NF})

In our new complexity metric we choose number of functions as one of the non-object oriented factors. Functions are among the most fundamental building blocks of programming.

Simply put, a programming function is a collection of code called for performing a particular task. If you define a function in anywhere in the program, you can transfer information to the function and return some results by giving the function a name and specifying parameters it should accept. Functions are reusable, so you can call them anywhere else in your program once you write one.

For the same code that is factored into functions and not factored into functions, the version with functions will have a higher time complexity due to a constant factor based on a function call overhead, as well as a higher space complexity due to the call stack. Optimizing compilers will inline the appropriate functions, taking care of the overhead function, so use of the appropriate functions in your code will reduce the time complexity of the reader's comprehension.

There are two types of functions as parameterize functions and non-parameterize functions.

To consider the complexity added due to number of functions, in the proposed matric,

- A weight of one is assign for function without any parameters (non-parameterize functions)
- A weight of two is assign for functions with parameters (parameterize functions)

2.4 Weight due to type of control structure (W_{CS})

- Control Structures are essentially a means of defining data flow in programs. Any algorithm or program, if they use self-contained modules called as logic or control structures, can be clearer and more understandable. It essentially analyzes and selects the path in which a system flows based on specific parameters or conditions. With the type of control structures Program gets more complex and harder to understand the different type of control structures. Control Structures increases and consequently they add more complexity to a software program.

- To consider the complexity added due to the control structures, in the proposed metric,

  - a weight of one is assigned for sequence
  - weight 2 is assigned for If-Then-Else
  - weight 3 is assigned for For-do, Repeat-until and While-do
  - weight n is assigned for ‘n’ number of Switch cases

3. Newly constructed equation

According to object oriented and non-object-oriented factors that we discussed above we came up with an equation with the combination of all 4 factors. In this equation we decide to add all the factors together because, all of these factors are in same unit (weight).

Complexity = $W_A + W_{AM} + W_{NM} + W_{CS}$

W_A = Weight due to using abstraction in class hierarchy

W_{AM} = Weight due to using access modifiers with attributes

W_{NM} = Weight due to the number of functions

W_{CS} = Weight due to type of control structure

4.Sample complexity calculation using new metric

Example 01

Line No	Program Statement	W_A	W_{AM}	W_{NF}	W_{CS}	Final Complexity
1	public abstract class Undergraduate {	1			1	2
2	private String name;	0	2		1	3
3	private String degree;	0	2		1	3
4	private String specialize;	0	2		1	3
5	public Undergraduate (String name, String degree, String specialize) {	0		2	1	3
6	System.out.println("Registering an Undergraduate");	0			1	1
7	this.name = name;	0			1	1
8	this.degree = degree;	0			1	1
9	this.specialize = specialize;	0			1	1
10	}	0			0	0
11	public String display() {	0		1	1	2
12	return name + " " + degree+ " " + specialize;	0			1	1
13	}	0			0	0
14	public String getName() {	0		1	1	2
15	return name;	0			1	1
16	}	0			0	0
17	public String getDegree() {	0		1	1	2
18	return degree;	0			1	1
19	}	0			0	0
20	public String getSpecialize() {	0		1	1	1

21	return specialize;	0			1	1
22	}	0			0	0
23	public void setSpecialize(String newSpec) {	0		2	1	3
24	if(newSpec.equals("SE"||"DS"))	0			2	2
25	specialize = newSpec;	0			1	2
26	else	0			0	2
27	specialize = "IT";	0			1	2
28	}	0			0	0
29	}	0			0	0
Total		1	6	8	26	41

Weight due to abstraction (W_A) = 1

Weight due to access modifiers (W_{AM}) = 6

Weight due to number of functions (W_{NM}) = 8

Weight due to control structures (W_{CS}) = 26

By using new equation, $\text{Complexity} = W_A + W_{AM} + W_{NM} + W_{CS}$

$$= 1 + 6 + 8 + 26$$

$$= 41$$

Example 02

Line No	Program Statement	W_A	W_{AM}	W_{NM}	W_{CS}	Final Complexity
1	public abstract class Worker {	1			1	2
2	private String name;	0	2		1	3
3	private String position;	0	2		1	3
4	public Worker(String name, String position) {	0		2	1	3
5	System.out.println("Assigning Workers");	0			1	1
6	this.name = name;	0			1	1
7	this.position = position;	0			1	1
8	}	0			0	0
9	public String display() {	0		1	1	2
10	return name + " " + position;	0			1	1
11	}	0			0	0
12	public String getName() {	0		1	1	2
13	return name;	0			1	1
14	}	0			0	0
15	public String getPosition() {	0		1	1	2
16	return address;	0			1	1
17	}	0			0	0
18	public void setPosition(String newPosition) {	0		2	1	3
19	Switch(newPosition) {	0			3	3
20	Case “Engineer”:	0			0	0
21	Position = newPosition;	0			1	1
22	Break;	0			0	0
23	Case “Assistant”:	0			0	0
24	Position = newPosition;	0			1	1

25	Break;	0			0	0
26	Default:	0			0	0
27	position = “labours”;	0			1	1
28	}	0			0	0
29	}	0			0	0
30	}	0			0	0
Total		1	4	7	20	32

Weight due to abstraction (W_A) = 1

Weight due to access modifiers (W_{AM}) = 4

Weight due to number of functions (W_{NM}) = 7

Weight due to control structures (W_{CS}) = 20

By using new equation, Complexity = $W_A + W_{AM} + W_{NM} + W_{CS}$

$= 1 + 4 + 7 + 20$

$= 32$

Example 03

Line No	Program Statement	W_A	W_{AM}	W_{NM}	W_{CS}	Final Complexity
1	public abstract class Pattern {	1	0	0	1	1
2	private int row;	0	2	0	1	3
3	public Pattern(int row){	0	0	2	1	3
4	this.row = row;	0	0	0	1	1
5	}					
6	public void printTraingle(){	0	0	1	1	2
7	for(int i = 1; i <= rows; i++){	0	0	0	3	3
8	for(int j = 1; i <= j; j++){	0	0	0	3	3
9	System.out.print(“* ”);	0	0	0	1	1
10	}					
11	System.out.print(“”);	0	0	0	1	1
12	}					
13	public void printPyramid(){	0	0	1	1	2
14	for(int i = 1; i <= rows; i++){	0	0	0	3	3
15	for(int j = 1; i <= rows-1; j++){	0	0	0	3	3
16	while(k != 2 * i - 1){	0	0	0	3	3
17	System.out.print(“* ”);	0	0	0	1	1
18	k++;	0			1	1
19	}					
20	System.out.println();	0	0	0	1	1
21	}					
22	}					
23	}					
Total		1	2	3	26	32

Weight due to abstraction (W_A) = 1

Weight due to access modifiers (W_{AM}) = 2

Weight due to number of functions (W_{NM}) = 3

Weight due to control structures (W_{CS}) = 26

By using new equation, **Complexity = $W_A + W_{AM} + W_{NM} + W_{CS}$**

= 1 + 2 + 3 + 26

= 32

Example 04

Line No	Program Statement	W_A	W_{AM}	W_{NM}	W_{CS}	Final Complexity
1	public abstract class PascalTraingle {	1	0	0	1	1
2	private int row;	0	2	0	1	3
3	public PascalTraingle (int row){	0	0	2	1	3
4	this.row = row;	0	0	0	1	1
5	}					
6	public void printPascalTraingle(){	0	0	1	1	2
7	for(int i = 0; i <= rows; i++){	0	0	0	3	3
8	for(int x = 1; i <= rows-i; ++x){	0	0	0	3	3
9	System.out.print(“ ”);	0	0	0	1	1
10	}					
11	for(int j = 0; j <= i; j++){	0	0	0	3	3
12	if(j == 0 || i ==0)	0	0	0	2	2
13	coef = 1;	0	0	0	1	1
14	else	0	0	0		
15	coef = coef * (i – j + 1) / j;	0	0	0	1	1
16	System.out.print(“%4d”,coef);	0	0	0	1	1
17	}					
18	}					
19	}					
Total		1	2	3	20	26

Weight due to abstraction (W_A) = 1

Weight due to access modifiers (W_{AM}) = 2

Weight due to number of functions (W_{NM}) = 3

Weight due to control structures (W_{CS}) = 20

By using new equation, **Complexity = $W_A + W_{AM} + W_{NM} + W_{CS}$**

= 1 + 2 + 3 + 20

= 26

Example 05

Line No	Program Statement	Factor 1 values for a program statement (W_A)	Factor 2 values for a program statement (W_{AM})	Factor 3 values for a program statement (W_{NM})	Factor 4 values for a program statement (W_{CS})	Final complexity value for a program statement based on all the four factors
1	public class Result{	**0**	**0**	**0**	**1**	**1**
2	public void res(int marks) {	**0**	**0**	**2**	**1**	**3**
3	if (marks > 0 && marks < 50)	**0**	**0**	**0**	**2**	**2**
4	System.out.println("Fail");	**0**	**0**	**0**	**1**	**1**
5	else	**0**	**0**	**0**	**1**	**1**
6	System.out.println("Pass");	**0**	**0**	**0**	**1**	**1**
7	}	**0**	**0**	**0**	**1**	**1**
8	public static void main(String args[ ]){	**0**	**0**	**2**	**1**	**3**
9	Result r = new Result( );	**0**	**0**	**0**	**1**	**1**
10	r.res(50);	**0**	**0**	**0**	**1**	**1**
11	}	**0**	**0**	**0**	**1**	**1**
12	}	**0**	**0**	**0**	**1**	**1**

Weight due to abstraction (W_A) = 0

Weight due to access modifiers (W_{AM}) = 0

Weight due to number of functions (W_{NM}) = 4

Weight due to control structures (W_{CS}) = 13

By using new equation, **Complexity = $W_A + W_{AM} + W_{NM} + W_{CS}$**

= 0 + 0 + 4 + 13

= 17

5.Conclusion

In this report a new approach for complexity measure has been presented. The proposed metric has computed by considering four factors. Two factors related on object-oriented concepts and next two based on non-object oriented factors. The presented metric is simple and can measure easily code complexity due to Encapsulation, Abstraction, Control structures and Number of functions. To understand metric some java programs complexity has been measured using the new approach.

6.References

1] Yadav, A. & Khan, Prof. Raees. (2012). Development of Encapsulated Class Complexity Metric. Procedia Technology. 4. 754-760. 10.1016/j.protcy.2012.05.123.

2] Available on : http://www.scs.pdn.ac.lk/personal/salukak/papers/HimaliFINAL.pdf

3] D.M. Rasanjalee Himali, S.R. Kodithuwakku. “Object-oriented software metrics”. Department of Statistics & Computer Science, Faculty of Science, University of Peradeniya, Peradeniya, Sri Lanka.

4] Daniel Rodriguez, Rachel Harriso. (March 2001). “An Overview of Object-Oriented Design Metrics”. RUCS/2001/TR/A.

5] Hussam Hourani, Hiba Wasmi, Thamer Alrawashdeh. (2019). “A Code Complexity Model of Object-Oriented Programming”. 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT).