CALCULATING MCCABE'S CYCLOMATIC COMPLEXITY METRIC AND ITS EFFECT ON THE QUALITY ASPECTS OF SOFTWARE

Marwa Najm Abd Jader
Software Engineering Department
College of Computer Sciences and Mathematics
Mosul, Iraq

Dr. Riyadh Zaghlool Mahmood
Software Engineering Department
College of Computer Sciences and Mathematics
Mosul, Iraq

***Abstract*—Most software developers aims at having a high-quality software that is easily maintainable, easily understandable, well structured, reliable, etc. Measuring software complexity is quite important as high complexity has been identified to be the reason behind difficult understanding and reading and problematic changes in the future. Added to that, it has been the source of defects and poor software quality in the present time. Accordingly, high complexity entails the presence of serious faults in the software and duly higher costs to maintain and fix. Software complexity metrics determine the improvement of software quality and project controllability to a large extent. In this work, we have calculated the software complexity metric (McCabe's Cyclomatic Complexity (CC)). Simplifying testing by guaranteeing the execution of at least one statement during testing. That is to say, the number of test cases will be equal to the Cyclomatic Complexity of the program. Such information is important for testing. Identifying reliability risk level, testability level; and cost/effort so as to maintain (maintenance risk level) by depending on Cyclomatic Complexity CC. Also, identifying Bad fix probability of an error that is unintentionally set into a program at the time of fixing a previous error, or while maintaining a code. As the complexity of the software increases, the probability to introduce new errors also increases. Finally, we have explained the results which are under discussion by specific tables and figures to illustrate the value and information of the metric in detail.**

Keywords*— Software Complexity Metrics, CC Metric, Cyclomatic Complexity, McCabe's Cyclomatic Complexity, Quality Attributes, CC number, Software Quality

I. Introduction

The most jaded software developers unanimously agree upon the importance of the high-quality software. Broadly speaking, software quality is defined as an effective software process that applied in a way that brings about a useful product of a measurable value to both producers and users [1]. A useful product provides the final users with desirable contents, functions, and features in the form of assets that are delivered in a reliable and error-free way [1]. Complexity appears everywhere in the life cycle of the software represented by the requirements, analysis, design, and implementation. It stands for an unfavourable property that renders the software quite hard to read and understand, and accordingly harder to change. It is also believed to be the main source of defects in the software. This has resulted in the belief that software complexity is just the opposite of its good quality represented by its easy maintenance and understanding, well-designed structure, reliability, etc. [2]. The measurement of software complexity and high complexity software is an important issue since understanding and reading it are quite difficult and its due change in the future become quite problematic. A main reason behind the defects are complex software. Accordingly, poor software quality is said to be due to software complexity [3]. A software metric is defined as a degree standard measurement of which a software system or process has some features [4]. On measuring, software complexity is a main part of the software metrics that focus on direct measurement of the qualities just contrary to the indirect software measures such as project milestone status and reported system failures. Many software complexity measures are available. They, range from the simple measures such as Source Lines of Code, to the esoteric represented by a number of variable definition/usage associations [5]. Well-known software metrics form the basis for Software complexity. They may minimize the time spent and cost estimation during the testing phase of the software development life cycle (SDLC). This can only be used after program coding is done. Improving software quality forms a quantitative measure of the source code quality [6]. The effort needed to analyze and describe the requirements, design, code, test and debugging of the system comes under the strong effect of complexity during the development phases of software. Also, in the maintenance phase, the difficulty in error correction and the required effort to change different software module are specified by complexity [7]. Complexity metrics form a good source of help in the intricate judgments about the strategy and planning of projects. Added to that, complexity information can help in [8]: (1) adjusting programming estimates, and duly schedules and costs, (2) deciding where more thorough analysis is necessary, (3) deciding which resources are most appropriate for a task, (4) developing more appropriate and detailed testing plans, (5) advising the business of additional project risks, and (6) deciding on alternative design plans to minimize changes to the highly complex code. Similarly, preparing both project and testing plans are derived from understanding the complexity metrics of a program to be tested [8]. LOC, Halstead’s measure of complexity and cyclomatic complexity are three software complexity metrics of main use. Yet there are problems concerning the use of LOC and Halstead’s measure complexity metric, and they can be overcome by the strongest metric cyclomatic complexity metric was introduced, three methods are used to measure cyclomatic complexity [3].

II. Literature Survey

This study analyzes the effectiveness of complexity in security, maintainability and errors prediction. It puts forward some various software metrics to measure the complexity of a system during the design phase, and highlight the cyclomatic complexity metric introduced by McCabe and studied extensively by most researchers. It also proposes improving cyclomatic complexity metric to measure both the intra-modular and the inter-modular complexity of a system [2].

This research attends to some more efficient software complexity metrics, namely Cyclomatic complexity, Line of code and Hallstead complexity metric and sheds light on their impacts on software quality. It also discusses and analyzes the correlation between these metrics. Finally, it demonstrates their relation with the number of errors through the use of a real dataset as a case study [9].

In this study, a framework or an algorithm is suggested to measure interaction between object classes based on all the unique external class references. Also, an approach is proposed to improve the concept of cyclomatic complexity [3].

This study investigates acquisition of knowledge on the basis of path testing by considering a sample of code and the implementation of path testing stated with its merits and demerits [10].

III. Software Complexity Matrics

Many different metrics for complexity have been proposed. The most important and popular ones have been chosen due to their noticeable impact on the project design and code quality, and as follows: Line of Code (LOC), Halstead Complexity (HC) and Cyclomatic Complexity (CC) [9].

A. Line of Code (LOC)

Generally speaking, LOC is computed when the lines of a program codes are counted. It is a common means for the evaluation of the software size, yet for measuring software complexity, it has been an inadequate indication [9].

For measuring a program size through counting the number of the lines of the source code, the oldest and the simplest metric Source line of code (SLOC) is most widely used to measure the size of a program by counting the number of lines of the source code. It was basically developed for the estimation of a project's man-hours. Every line including comments and blank lines is counted by LOC. Kilo Lines of Code (KLOC) is LOC divided by 1000. The effective line of code, except parenthesis, blanks and comments, is estimated by Effective Lines of Code (ELOC) [2].

For programming language, LOC is counted speedily and is easy for understanding [9]. Immediately after the calculation of the LOC, the following attributes can be measured [2].

- Productivity = LOC/Person months
- Quality = Defects/LOC
- Cost = $/LOC

The advantages of the LOC metric are as follows.

LOC is mostly used and measured after project completion. Its language independent that has been proved as a useful predicator of program effort [3].

The disadvantages of the LOC metric are as follows.

1) No counting of LOC is possible unless the application is done, and its counting is always performed at the completion of the life cycle. In other words, it is difficult and impossible to have a counting of LOC at a stage of early life cycle [3].

2) A main feature of this metric is that it makes no distinction between LOC complexities. For instance, the code "i=1" does not differ from the code "i= (++x + max (a,b)) / power (c,d)". As a matter of fact, the second code is more complex than the first one, yet LOC metric counts the number of lines at the time nothing else is taken into consideration [9].

3) Both complexity caused by the decisional statements or conditional statements, if present in the program, and data complexity are ignored by LOC. According, in 1977, a new metric Maurice Halstead, also known as Halstead software science or as Halstead metric, was introduced to overcome such difficulty [3].

B. Halstead Complexity (HC)

Maurice Halstead introduced a set of metrics, known as Halstead software science or Halstead metrics, in 1977.He was the pioneer in writing a scientific formulation of software metrics. His objective was to provide an alternative measurement for counting of LOC as a measure of both size and complexity. His measurement has been used as a predictive measure of the error of a program [2]. It computes the difficulty of a program by counting both the "operators" and the "operands" [11].

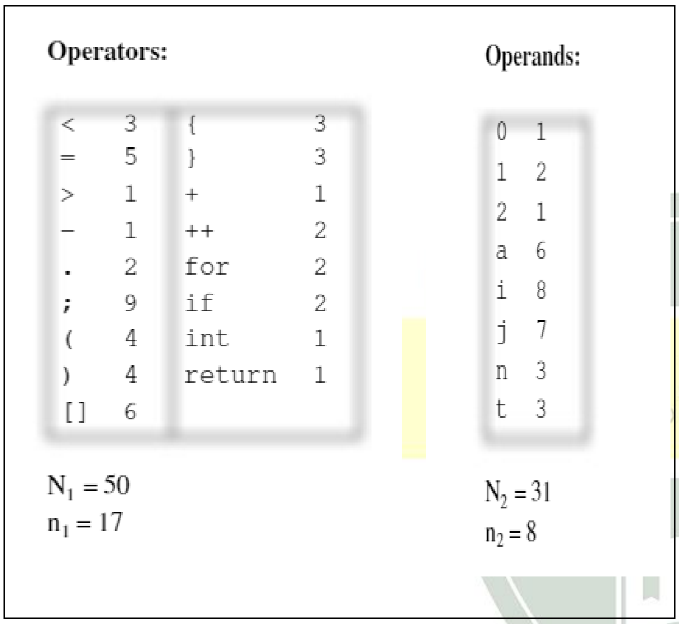
Halstead views a computer program as the application of an algorithm in the form of a collection of tokens that are liable to classification into either operators or operands. That is to say, a program is viewed as a chain of operators with their associated operands. All Halstead's metrics work as functions for the counting of these tokens [12].

These tokens are basically defined as either operators or operands. The following indices for the bases for Halstead's metrics [2].

- n1 - distinct number of operators in a program
- n2 - distinct number of operands in a program
- N1 - total number of operators in a program
- N2 - total number of operands in a program

```
void sort(int *a, int n) {
int i, j, t;
if (n < 2)return;
for (i=0 ; i<n-1; i++) {
for (j=i+1 ; j<n ; j++) {
if (a[i] > a[j]) {
t = a[i];
a[i] = a[j];
a[j] = t;
}
}
}
}
```

Now, the operators and operands can be identified from this C program (HALSTEAD VOLUME EXAMPLE) [13].



Operators:

<	3	{	3
=	5	}	3
>	1	+	1
–	1	++	2
.	2	for	2
;	9	if	2
(	4	int	1
)	4	return	1
[]	6		

$N_1 = 50$
$n_1 = 17$

Operands:

0	1
1	2
2	1
a	6
i	8
j	7
n	3
t	3

$N_2 = 31$
$n_2 = 8$

Fig. 1. Calculation of operators and operands of the program [13].

Then: HV = 81 * log2 (25) = 81 * 4.64385618977≈376 [13].

Halstead puts forward the definition of some software attributes based on the notions of operators and operands [2].

*1) Program vocabulary:R*represents counting the number of both unique operators and operands as: n =n1+n2.

*2) Program length: S*tands for counting of total number of both operators and operands: N=N1+N2.

3) Program Volume: Represents a further measure of program size. It is defined by V=N*log2(n). The Program Volume (HV) of the preceding C program is.

*HV = 81 * log2 (25) = 81 * 4.64385618977≈376* [13]

4) Difficulty Level: The relation between this metric and the number of distinct operators (n1) in the program is proportional. It is based on the total number of operands (N2) and the number of distinct operands (n2). If the same operands are used several times in the program, there will be more chances for errors to occur.

D= (n1/2)(N2/n2)* [2].

The Difficulty of the previous C program is.

*Example C program: D = (17/2) * (31/8) ≈ 36* [14].

5) Program Level: Is the inverse of the level of difficulty. A low-level program is more subject to errors compared to a high-level program. L=1/D.

6) Programming Effort: This is limited to the mental activity that is needed to change an existing algorithm into an actual implementation in a programming language.

*E =V*D*

7) Programming Time: Can be taken from the formula: T=E/18 [2].

8) Bugs delivered: (E ^ (2/3)) / 3000 [14]..

The advantages of the Halstead metric are as follows.

- The overall quality of a program is measured by Halstead's metric.
- Its programming does not require deep knowledge and its calculation is quite simple.
- The rate of errors and effort can also be measured by its means.
- It is used to calculate the complexity from the data flow of software [3].

The disadvantages of the Halstead metric are as follows

- No distinction is there in the computation of some operators and operands of the codes and some branches and jumps. Definitely, there is more complexity in the computations of the branches and jumps [9].
- Halstead's metric method cannot measure inheritance and Interaction between modules [3].
- Halstead's metric method also ignores the complexity from the decision statements like if, loops etc. [3].

Yet these metrics (LOC, Halstead's Measures of Complexity, etc..) have some problems as illustrated through previously stated disadvantages; a point that has made the concept of cyclomatic complexity become more common [3].

C. Cyclomatic Complexity (CC metric)

Before the introduction of the Cyclomatic complexity method, physical size, i.e. LOC metric was not considered adequate due to the existence of 40-50 lines of code comprising different consecutive conditional statements such as "while", "if". There might be a program code that has million distinct control paths, yet only a small proportion of them would probably be subject to testing [3].

In an attempt to estimate software program maintenance cost in the future, reference [15] made use of three software complexity measures, namely McCabe's Cyclomatic Complexity, Halstead's E and "the length as measured by the number of statements". They point out that the maintenance cost of the program is three times the cost of the initial development. As such, software managers view this measurement as, reference [15].

McCabe's Cyclomatic Complexity is a mathematical technique that both provides a quantitative basis for modularisation and allows the identification of software modules difficult to test or maintain. The LOC is rejected due to the fact that no obvious relationship exists between length and module complexity, and hence McCabe is viewed as a complexity measurement that examines the number of control paths through a module, reference [15].

Cyclomatic complexity is a software metric (measurement), developed by Thomas J. McCabe, Sr. in 1976, by means of which the complexity of a program is identified.It quantitatively measures the number of linearly independent paths through a program's source code [16].

Complexity measurement is defined by McCabe (1976) based on a graph theory by which a number of paths through a program are measured and controlled. Through the presentation of examples in FORTRAN programs, McCabe stated that complexity has nothing to do with physical size; it only depends on a program's decision structure", reference [15].

Cyclomatic complexity metric has always been correlated with certain qualities of the software represented by reusability, maintainability, security and prediction of faults. It works as an efficient guide for the development of test cases [2].

D. *Calculating Cyclomatic Complexity (CC)*

McCabe's Cyclomatic number which is used to study program complexity may be applied. Three methods are recommended for the calculation of the Cyclomatic Complexity number from a flow Diagram [3].

- Number of edges – number of nodes + 2
- Number of binary decisions + 1
- Number of closed regions + 1.

In the following pages, light will be shed on the three methods to calculate the number of Cyclomatic Complexity.

E. *First Method*

The control flow graph of the program plays a crucial role in the definition of the Cyclomatic Complexity of a structured program. It is a directed graph that contains the basic blocks of the program and has an edge between two basic blocks in case the control may pass from the first to the second. Accordingly, the complexity M is defined as [16]:

$M = E - N + 2P$,

Where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

In the case of a single program (or subroutine or method), P always equals 1. The following is a simpler formula for a single subroutine:

$M = E - N + 2$.

Cyclomatic complexity may simultaneously be applied to a number of programs or subprograms (e.g. to all the methods in a class). In such a case, Since each subprogram will appear as a disconnected subset of the graph, P will be equal to the number of programs [16].

This program is mainly applied to calculate any number's power [3].

```
1. Begin
2. int A, B, power;
3. float C;
4. input(A,B);
5. if (B < 0)
6. power = -B;
7. else
8. power = B;
9. C = 1;
10.while ( power!=0) {
11.C=C*A;
12.power = power -1;
13.}
14.If ( B<0)
15.C = 1/C;
16.Output( C );
17.End
```

This is the program for a "raised to power B".

The flow graph stands for the flow of source code and 9 nodes and 13 edges exist in the control flow graph (Fig. 2).

From the graph, the number of edges and nodes is calculated and set in the formula and as follow:

Cyclomatic complexity number = #edges - #nodes +2P.

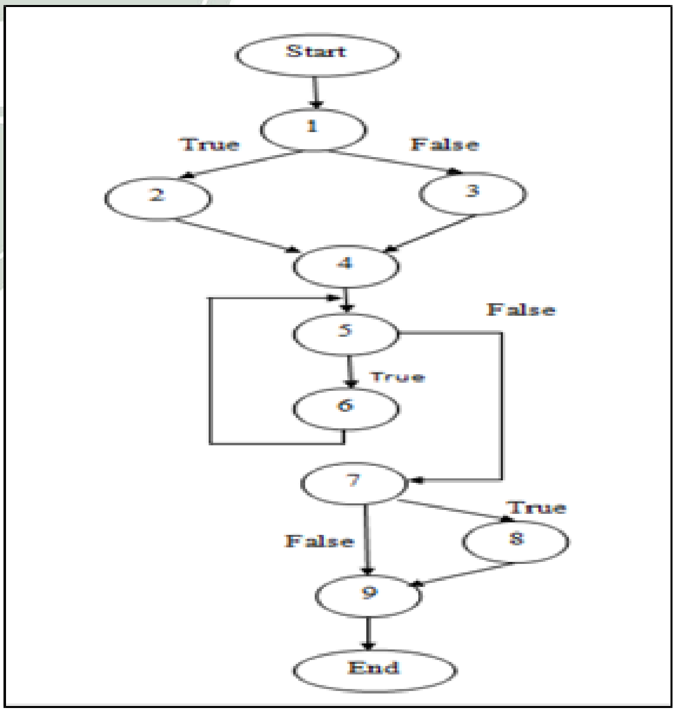


Fig. 2. The Control Flow Graph.

From the above program #edges =13, #nodes = 9. Hence, the Cyclomatic complexity number =13-11+2 = 4.

Four independent paths exist in this graph. The Cyclomatic complexity number equals the number of independent paths in the graph [3].

The number of test cases will equal that of the Cyclomatic complexity of the program [16].

F. *The Second Method*

Here, McCabe Cyclomatic Complexity is calculated by determining the number of decision statements caused by the conditional statements in a program + one.

Cyclomatic Complexity = number of decision statements + 1 [3].

This method will be adopted in our proposed system.

Five basic rules can be used for the calculation of CC.

- Calculating the number of “if/then”, “else if”, yet no counting of the “else statements” in the program is made.
- Finding any select (or switch) statements, and counting the number of the cases within them. Then, finding the total of the cases in all the select statements combined together. No counting of the default or "else" case is made.
- Calculating all the loops in the program.
- Counting all the try/catch statements [17].
- Counting the conditional operator &&, || operator and ternary operators like? from the expression [3]..

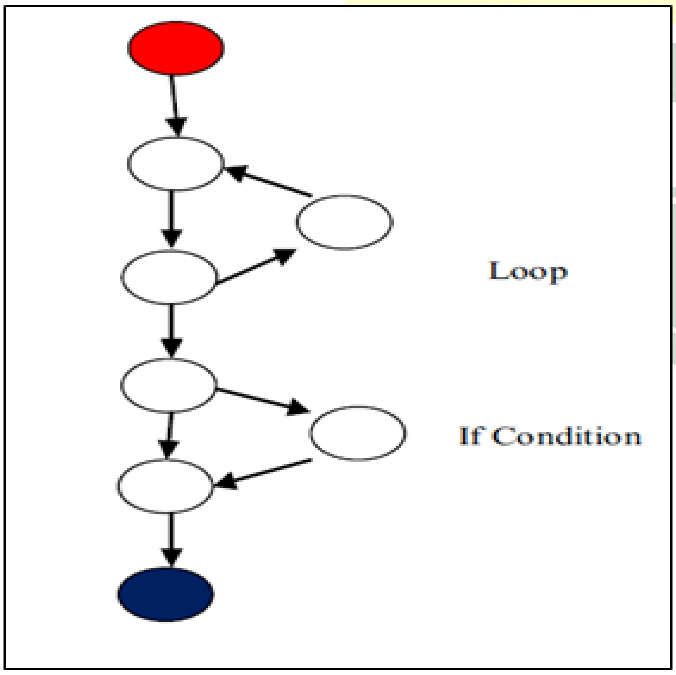


Fig. 3. Decision Control Graph [18].

Now, one is added to the numbers from the previous step numbers. That will be the number of final cyclomatic complexity of the program. Accordingly, in the program stated above for calculating the power of any number), the number of decisional statements is three: two “if” statements and one “while” statement. Then, one is added to the final cyclomatic complexity number and the final cyclomatic complexity number for the above program according to the second method is = 3+1=4 [3].

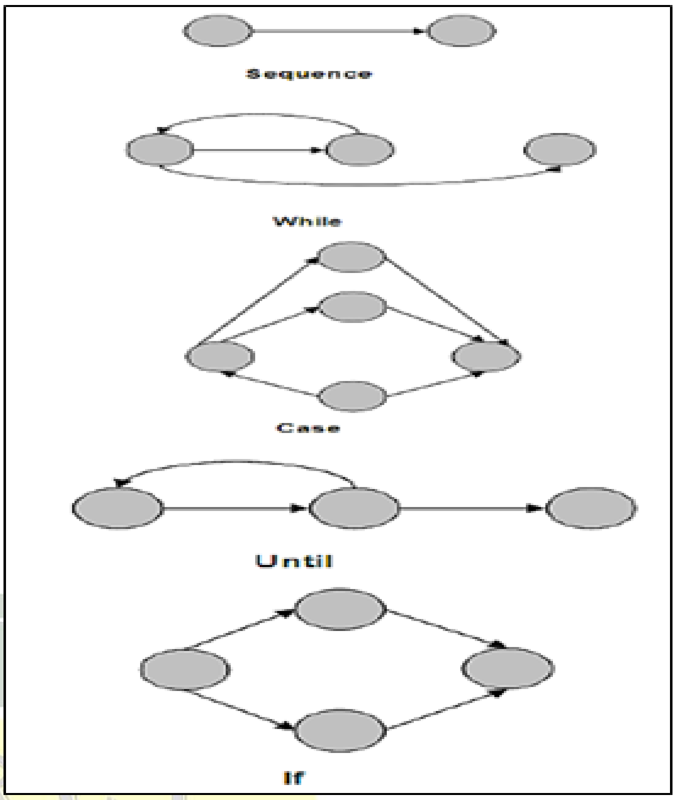


Fig. 4. The Different Types of Flow Graphs [19].

G. *The Third Method*

Cyclomatic complexity = Number of enclosed areas + 1.

Accordingly, the control flow diagram is first drawn followed by the calculation of the number of closed regions in the control flow diagram. The number of closed regions is 4 here.

Then, cyclomatic complexity is= 3+1=4 (Fig. 5).

It is worthy to note that from the three preceding methods, the same Cyclomatic complexity number is obtained [3]

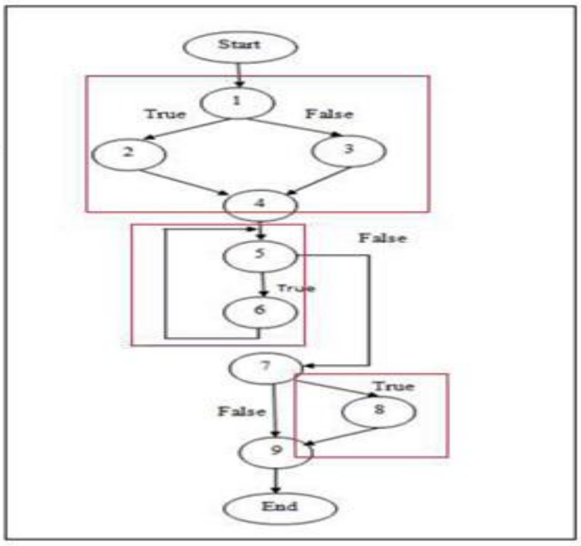


Fig. 5. Regions calculation from the flow graph [3].

IV. The Significance of the McCabe's Cyclomatic Complexity Number

As stated in the following sections, the number of McCabe (CC) noticeably influences the quality of the product (software).

A. *The Correlation between CC number and Reliability Risk*

TABLE I. The Relationship Between Cyclomatic Complexity & Reliability Risk [20]

CC number	Reliability risk
1 – 10	Simple procedure, little risk
11 - 20	More Complex, moderate risk
21 – 50	Complex , high risk
>50	Untestable, VERY HIGH RISK

Software Reliability is defined as the probability that a software operation will not fail for a specified duration of time within a specified environment [21].

The Cyclomatic complexity number should range from 1 to 10. Only by that time, software is considered as risk free. If the range is 10-20, it would be considered as a target of moderate risk. A 30-40 range makes the module highly risky, and a range that exceeds 40 exempts it from the candidacy for testing. Based upon some data, it has been proved that the higher the cyclomatic complexity number will be, the lower the quality of software will be, table(I) [3].

The Software Engineering Institute at Carnegie Mellon University provides threshold values concerning the CC metric, table(I). Based on the resulted complexity, a risk associated with the specific procedure's complexity comes into existence. As such, appropriate measures should be taken to reduce the complexity and duly to avoid future maintenance [22].

Based on certain facts, if programs with McCabe Cyclomatic complexity are greater than 10, there will be a high probability of having errors and defects and it will be very difficult to understand. As a result, more numbers of test cases will be required to execute the paths in the program. In other words, the higher the Cyclomatic number is, the higher will be the error rate, and the more required will be the code maintenance or refactoring [3].Also, the greater the complexity (by some measures), the more fault the software will have and the higher the cost will be [23].

B. *The Correlation between High CC and Failure*

In spite of the intuitive appeal concerning the correlation between high aggregated CC and higher-than-expected maintenance problems, several main factors are expected to have a role to play (figure 6). For instance, it is shown that the CC metric has been strongly correlates with the method size. So, in case a large system has many methods with high CC, the methods are probably the methods that are longer. This may duly indicate an inability by the programmers to form coherent abstractions and build robust, reusable units of code [24].

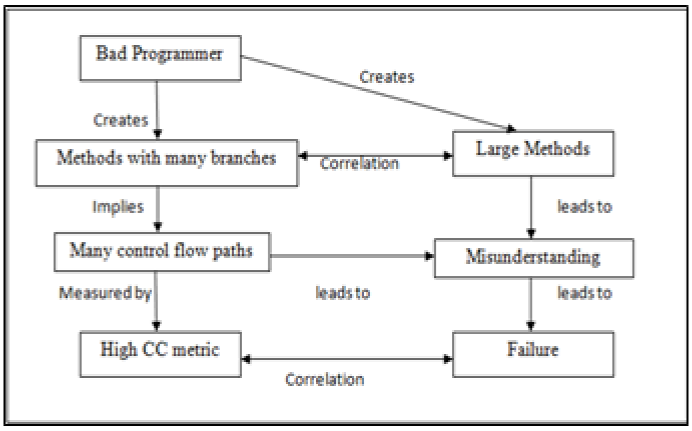


Fig. 6. Two Comparable Explanations for Correlation between high CC and Failure [24].

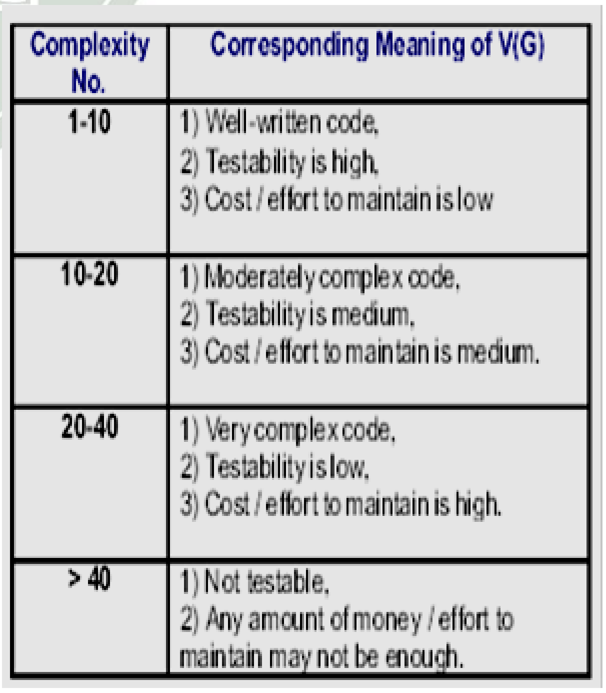
Higher software complexity results in the software failure. In this respect, two possible cases are:

- An incompetent programmer may write a program with large methods and that results in failure.
- A programmer may write some methods including many jump and branch statements and many control paths, all of which result in failure [3].

C. *The CC number and Its Corresponding Meaning*

High complexity, more bugs and security flaws [25]. Table (II) presents an overview of the complexity number and the corresponding meaning of v (G).

TABLE II. The Complexity Number And Its Corresponding Meaning [25]

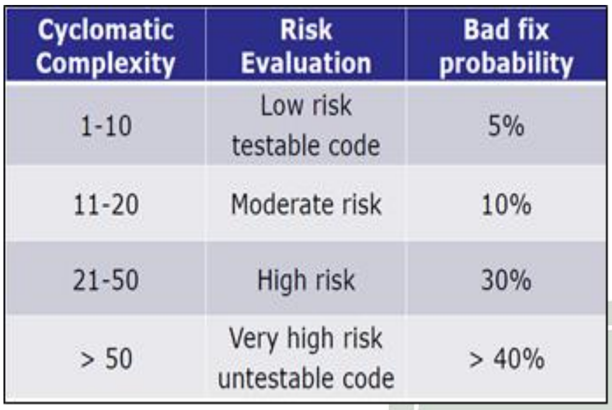


Complexity No.	Corresponding Meaning of V(G)
1-10	1) Well-written code, 2) Testability is high, 3) Cost / effort to maintain is low
10-20	1) Moderately complex code, 2) Testability is medium, 3) Cost / effort to maintain is medium.
20-40	1) Very complex code, 2) Testability is low, 3) Cost / effort to maintain is high.
> 40	1) Not testable, 2) Any amount of money / effort to maintain may not be enough.

D. The Correlation between CC number and Bad Fix Probability

"Bad Fix Probability" refers to the probability of an error that is accidentally inserted into a program at the time of fixing an error. With the complexity reaching high values, new errors are quite likely due to the changes in the program [26].

TABLE III. "EMPIRICAL VALUES SEI" (THE CORRELATION BETWEEN CC NUMBER AND BAD FIX PROBABILITY) [27].



Cyclomatic Complexity	Risk Evaluation	Bad fix probability
1-10	Low risk testable code	5%
11-20	Moderate risk	10%
21-50	High risk	30%
> 50	Very high risk untestable code	> 40%

E. The Correlation between Software Complexity and Software Maintainability and Cost

Software complexity attends to the difficulty of comprehending and working with a program [6]. According to IEEE, software complexity is defined as "the degree to which a system or component has a design or implementation that is difficult to understand and verify". While "Maintainability" refers to the ease of maintaining a product so as to [28].

- Isolate defects or their causes,
- Correct defects or their causes,
- Repair or replace faulty or worn-out components without replacing the parts that are still working,
- Prevent unexpected breakdowns,
- Maximize a product's useful life,
- Maximize efficiency, reliability, and safety,
- Meet new requirements,
- Make future maintenance easier, and
- Cope with a changing environment

Fig. 7 shows the dependencies.

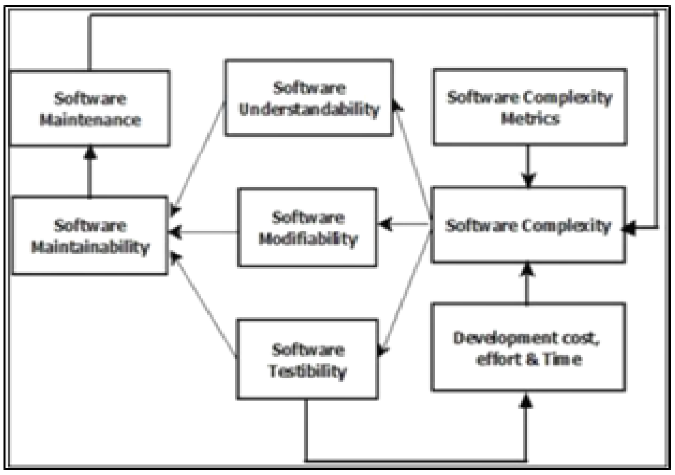


Fig. 7. The Relationship between software complexity metrics and software systems [6].

Complexity is a measure of understandability and the lacking of understandability results in errors. A more complex system may be harder to specify, to design, to implement, to verify, to operate, to predict its behavior and risky to change [29].

A large number of employees is required to maintain large software systems and the high estimated costs of software maintenance are good reasons behind the efforts exerted by software managers to monitor and control complexity [29].

A direct relationship exists between software complexity and maintenance costs. As a code lines increase, software becomes more complex and the probability of more bugs increases. In such a case, the cost of maintaining the software also increases [29].

Algorithm of Calculation of the Complexity Metric

(McCabe Cyclomatic Complexity "*CC*")

The steps of this algorithm are as follows:

1) Loading Java file that belongs to the user by choosing its specific track on the condition that the file to be loaded is of Java extension only.

2) After loading Java file, a medial textual file of the extension "txt" will be formed to be written on later, by using (File and FileWriter) as follows.

```
File file= new File ("file.txt");
FileWriter  fileWriter= new FileWriter(file);
```

3) Reading the contents of Java file to be loaded and treated by using a special function, we have removed all the comments from the file and not counting them programmingly. There are three types of comments in Java.

a) Single-Line Comments that start with // and are used for one programming line.

b) Multi-Line Comments that start with / and ends with */ to limit the number of the programming lines inside it.*

*c) Javadoc Comments that start with the symbol /** and a number of * is vertically written until the symbol of the end */ is arrived at. Within these two symbols there is a set of* *information that may be the name of the writer of code (Author)…etc., in addition to illustrating information about the code, the functions and the variables in the light of the need for writing it so as to understand the code in a better way in terms of clarification and management especially in the complex codes, and are saved in the form of HTML document.*

4) On reading and treating Java file, the contents of the file that had been previously treated will be written on the medial textual file (file.txt) that had been previously formed. In other words, we would read one programing line, treat as far as the comments are concerned and write them on the medial file. The process goes on until we finish with all the contents (lines) of Java file, and this step will be done as follows.

```
FileReader fr = new FileReader((Java file extention));
BufferedReader br = new BufferedReader(fr);
line = br.readLine();
while (line != null)
```

// here, the contents of Java file will be treated by using a special function, as we will remove all the comments from the file and not counting them programmingly. Then, the contents of the file that had been previously treated will be written on the medial file ("file.txt") that had been previously formed.//

```
//////////////write on textual file line by line//////////
fileWriter.write(line);
fileWriter.write("\n");
fileWriter.flush();
line = br.readLine();
end while
```

5) Saving the medial textual file ("file.txt"), with its Java file contents that had been previously treated in the same package where Java file had been previously loaded.

6) After loading and opening Java file that belongs to the user as we have previously explained, we will calculate the metric CC as in the following steps.

7) At the beginning, an array of string will be formed. It is named Keywords []. Inside it are put the Decisional statements and operators and as follows.

```
String [] keywords = {"if", "while", "case", "for", "&&", "||",
"?", "catch"};
```

8) Reading from "file.txt" which is the medial text file that has been formed and written on previously when opening and loading Java file which includes all the contents of Java file without comments. This step will be done as follows.

```
BufferedReader br = new BufferedReader(new
FileReader(file.toString()));
line = br.readLine();
while (line != null)
    line = line.trim();
```

9) Making parse or analysis of each line in the medial textual file ("file.txt") so as to find out the Decisional statements and required to calculate the metric "CC" by using the SringTokenizer Class. This will be done by dividing the string into tokens, i.e. the lines in the medial textual file will be divided into tokens by depending on the delimiters or certain identifiers to divide the lines according to the treatment requirements.

```
StringTokenizer stTokenizer =new StringTokenizer(line, "(
{ ) } ; , .");
```

10) To look for the Decisional statements and operators, _they are as follows.

a) ***If/then*** *(don't count "else"),* ***all cases of switch statement*** *(don't count default).*

b) ***All loops (for, while, do-while, try/catch).****.*

c) Conditional operators ***(&& , || , ?: ternary operator)****,_ in the textual file ("file.txt"), and as we have previously mentioned by depending on the tokens we have obtained by diving the line in the medial file and then putting them in a string that is called words. Then compare the tokens or words that have been obtained from the textual file with the matrix of keywords [] that have been identified at the beginning of the algorithm that includes Decisional statements and operators. If the comparison is correct, the counter is increased by 1 and as follows.*

```
while (stTokenizer.hasMoreTokens())
words = stTokenizer.nextToken();
for (i = 0; i < keywords.length; i++)
if (keywords[i].equals (words))
        counter++;
        end if.
        end for.
        end while.
line = br.readLine();
end the first while in the step(8)
```

11) Calculating the "CC" by means of summing the counter of the Decision statements that have been found out and calculated previously and adding 1 to them as the number of decision statements represents the previous counter that has been formed and increased according to the comparison. The special rule for counting the CC is.

CC=number of decision statements +1 , and this step will be done as follows: CC = counter + 1.

12) Demonstrating the value of the metric CC of the user and the information concerning this metric such as.

Quality Attributes like:

a) Reliability risk level: Reliability is the probability of failure-free software operation for a specified period of time in a specified environment.

b) Testability level: The ease with which a computer program can be tested.

c) Cost/effort to maintain: Maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes.

Bad fix probability: *the probability of an error accidentally inserted into program while trying fix previous error.*

V. Results And Discussion

The metric CC was tested on 2 examples or programmes written in Java language. The results were then discussed. They are illustrated in details in the form of figures and tables.

1) Sample1: "Example of String": In this example, the number of the entered words, letters, digits, uppercase letters and lowercase letters of the string are entered is calculated with a reversed calculation of this string. Figure (8) outlines the source code specific to this example. We arrived at a set of results. Table (IV) illustrates the results of this example in detail.

```
package exampletest;
import java.util.Scanner;
public class ExampleOfstring {
public static void main(String[] args) {
int i, countcharacters = 0, countwords = 0, countdigits = 0,
upperCase = 0, lowerCase = 0, len = 0;
Scanner scanstr = new Scanner(System.in);
System.out.print("Enter your string: " + "\n");
String str = scanstr.nextLine();
char ch[] = new char[str.length()];
for (i = 0; i < str.length(); i++) {
    ch[i] = str.charAt(i);
if (((i > 0) && (ch[i] != ' ') && (ch[i - 1] == ' '))
        || ((ch[0] != ' ') && (i == 0))) {
        countwords++;//count words
        }
  if (((ch[i] != ' '))) {
     countcharacters++;//count characters
        }
      }
  for (i = 0, len = str.length(); i < len; i++) {
    if (Character.isDigit(str.charAt(i))) {
          countdigits++;//count digits
          }
      }
  for (i = 0; i < str.length(); i++) {
          // Check for uppercase letters.
     if (Character.isUpperCase(str.charAt(i))) {
             upperCase++;
          }
     // Check for lowercase letters.
      if (Character.isLowerCase(str.charAt(i))) {
             lowerCase++;
             }
           }
       String stt = ("Your string has ");
   System.out.println(stt + (countwords) + " words." + "\n");
   System.out.println(stt + (countcharacters) + " characters."
+ "\n");
   System.out.println(stt + (countdigits) + " digits." + "\n");
   System.out.println(stt + (upperCase) + " uppercase letters"
+ " and " + lowerCase + " lowercase letters" + "\n");
   String reverse = new StringBuffer(str).reverse().toString();
   System.out.println("Your string before reverse:" + str +
"\n");
   System.out.println("Your string after reverse:" + reverse +
"\n");
   System.out.println("--------------------------------------------
----------------------------" + "\n");
       }
   }
```

Fig. 8. Source code of the Sample1. *(Example of String)*

With regard to the results in Table (IV), the metric CC is calculated from the rule that is previously mentioned:

Cyclomatic Complexity=number of decision statements + 1

We have also illustrated its effect on the quality attributes such as: Reliability risk, Testability, Cost/effort to maintain (maintenance) as they ratios are based on the value of this metric (CC). We have also illustrated the Bad Fix Probability. It also depends on the value of this metric. This ratio is adopted as we have previously stated from certain resources and according to certain tables and by depending the value of the metric) CC) .

As for this example, CC value = 13, depending on the decision statements and adding 1 to it. In this example, we have (for ,if , && ,&& ,|| ,&& ,if ,for ,if ,for , if, if). They are 12 in number and we have added 1 so that CC becomes 13. The value of Bad Fix Probability equals 10% based on the value of the metric (CC).

TABLE IV. The Results Of The Sample1 (*Example Of String*)

Sample1 (*Example of String (count letters, words,…..etc)*)					
Cyclomatic Complexity "CC" metric	**"CC" number**	**"Meaning of "CC" number (Quality Attributes)**			**Bad Fix Probability**
		"Reliability risk"	***"Testability"***	***Cost/Effort to maintain***	
	13	More complex, moderate risk	Medium	Medium	10%

2) Sample2: "Simple Program of Cyclomatic Complexity Examples"

In this example, we have illustrated CC and focused on the value of this metric.

We have tried in this example to put all the cases or rules of the metric occurrence CC as we have dealt with it in detail in the Algorithm for calculating this metric. We have put in this example a set of functions where each function implies a certain case or a certain rule such as:

(If statement, switch statements, while, for, try/catch statements, conditional operator (&& and || operator and also ternary operators like? :)).

The source code for this example is in figure (9). We arrived at a set of results. Table (V) illustrates the results of this example in detail.

```
package SimpleprogramCyclomaticexample;
import java.util.Scanner;
public class SimpleProgramCyclomaticExample {
///////////////example of While function/////////////////
public static void WhileFunction() {
   System.out.println("******While function to print numbers
from (1 to 10)******* ");
      int num = 1;
      System.out.print("The numbers from (1 to 10) are : ");
      while (num <= 10) {
         System.out.print(num + " ");
         num++;
      }
      System.out.println("\n" + "-------------------------------------
-----------------------------------");
   }
///////////////example of for loop  function/////////////////
public static void ForloopFunction() {
   System.out.println("******For loop function to print the
names of animals that starting with letter 'B'******* ");
   String[] values = {"Cat", "Bear", "Dog", "Bird", "Bee",
"Butterfly"};
 System.out.print("The names of animals that starting with
letter 'B' : ");
      // Loop over all Strings.
      for (String value : values) {
         // Skip Strings starting with letter b.
         if (value.startsWith("B")) {
            System.out.print(value + ", ");
         }
      }
      System.out.println("\n" + "-------------------------------------
----------------------------------");
   }
////////example of If statement and contains &&  And  ||
//////////////
   public static void IfStatementAndOrFunction() {
      int score = 90;
      System.out.println("******If statement function to print
your score according to some conditions******* ");
      System.out.println("Your score = " + score);
      if (score <= 70) {
         System.out.println("You did not receive a passing
score!");
      } else if (score > 70 && score < 90) {
         System.out.println("You received a passing score!");
      } else if (score >= 90 && score < 99) {
         System.out.println("You received a perfect score!");
      } else if (score == 99 || score == 100) {
         System.out.println("You are genius");
      } else {
         System.out.println("I don't know your score!");
      }
      System.out.println("\n" + "-------------------------------------
-----------------------------------");
   }
/////////////example of try-Catch block
function/////////////////////////////
public static void tryCatchfunction() {
      int num1, num2;
      System.out.println("******Try-catch block function to
handle code that may cause exception (divide-by-zero
error)**** ");
      try {
         // Try block to handle code that may cause exception
         num1 = 0;
         num2 = 62 / num1;
         System.out.println("Try block message");
      } catch (ArithmeticException e) {
         // This block is to catch divide-by-zero error
         System.out.println("Error: Don't divide a number by
zero");
      }
      System.out.println("I'm out of try-catch block in Java. so
continue excute your program");
      System.out.println("\n" + "-------------------------------------
----------------------------");
   }
////////example of short if statement Ternary Operator ( ? :
)/////////
  public static void ShortIfStatementTernaryOperator() {
      System.out.println("******Short if statement function
(Ternary operator ? : )*****");
      String str = "London";
      System.out.println("Your string is " + str);
      String data = str.contains("L") ? "Your String contains
'L'" : " Your String doesn't contains 'L'";
System.out.println(data);

System.out.println("\n" + "-------------------------------------------
----------------------");
   }
```

```
        switch (monthNumber) {
            case 1:
                System.out.println("(January)");
                break;
            case 2:
                System.out.println("(February)");
                break;
            case 3:
                System.out.println("(March)");
                break;
            case 4:
                System.out.println("(April)");
                break;
            case 5:
                System.out.println("(May)");
                break;
            case 6:
                System.out.println("(June)");
                break;
            case 7:
                System.out.println("(July)");
                break;
            case 8:
                System.out.println("(August)");
                break;
            case 9:
                System.out.println("(September)");
                break;
            case 10:
                System.out.println("(October)");
                break;
            case 11:
                System.out.println("(November)");
                break;
            case 12:
                System.out.println("(December)");
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("\n" + "--------------------------------------
----------------------------"); }
////////////////////////////////////////////////////////////////////////////////
    public static void main(String[] args) {
        WhileFunction();
        ForloopFunction();
        IfStatementAndOrFunction();
        tryCatchfunction();
        ShortIfStatementTernaryOperator();
        SwitchCases();
    }
//////////////////example of switch cases function///////////////////
public static void SwitchCases() {

        System.out.println("******Switch case function to print
the month's name according to month of your birth*****");
    int monthNumber = 12;
        System.out.print(" Month of your birth is : " +
monthNumber + " ");
}
```

Fig. 9. Source code of Sample2. *(Simple Program of Cyclomatic Complexity Examples)*

The results will be demonstrated in the form of a table. And as we have also illustrated, we will calculate the metric CC. It is noticed in this example that the value of the metric CC= 25 and according to the rule:

CC= number of decision statements+ 1.

Since the number of decision statements is 24 and as follows:

(while , for , if , if , else if , && , else if , && , else if , || , try-catch , ?: , (switch statements (case 1 , case2 , case3 , case4 , case5 , case6 , case7 , case8 , case9 , case 10 , case 11 , case 12))

We then add 1 to them according to the rule, the value of the metric CC equals 25.

As we have explained in the previous example, there will be quality attributes such as Reliability risk, Testability, Cost/effort to maintain (maintenance) that are affected by the metric CC , as they ratios are based on the value of this metric (CC). We also have the Bad Fix Probability. This ratio is adopted, as we have previously stated, from certain resources and according to certain tables and by depending on the value of the metric CC. In this example, the value of Bad Fix Probability equals 30% based on the value of the metric (CC). Table(V) illustrates in detail the results of this example.

TABLE V. THE RESULTS OF THE SAMPLE2 (*SIMPLE PROGRAM OF CYCLOMATIC COMPLEXITY EXAMPLES*)

Sample2 (*Simple Program of Cyclomatic Complexity Examples*)					
Cyclomatic Complexity "CC" metric	**"CC" number**	**"Meaning of "CC" number (Quality Attributes)**			**Bad Fix Probability**
		"Reliability risk"	***"Testability"***	***Cost/Effort to maintain***	
	25	Complex, high risk	Low	High	30%

And now, after the implementation of all the 2 examples, we have written one Table (VI) which contains the results of these examples. The value of CC has been written in addition to the value of the Bad Fix Probability of each one of the previous examples. Also a graph has been drawn to illustrate the results of all the examples; i.e. represented the results of table(VI) by what is in figure (10) where focus has been on the value of the metric CC and the value of the Bad Fix Probability.

TABLE VI. THE RESULTS OF THE 2 SAMPLES (*PROGRAMS*) OF JAVA

Samples	**CC number**	**Bad Fix Probability**
1 Example of String (count letters, words,…..etc)	13	10%
2 Simple Program of Cyclomatic Complexity Examples	25	30%

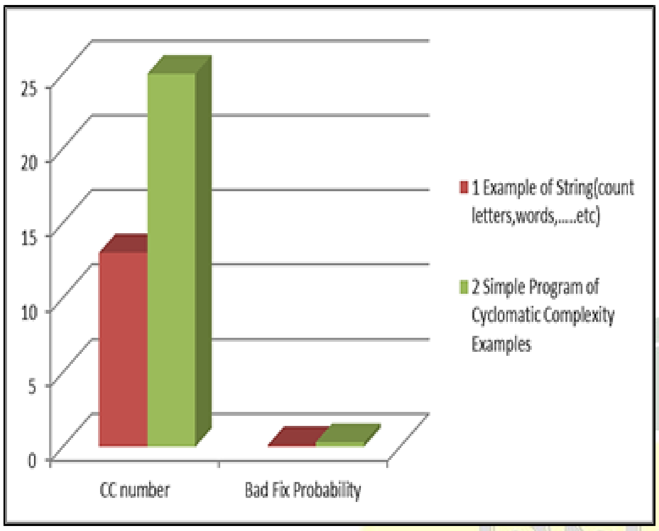


Fig. 10. The Graph to represent the results of the 2 Samples (*programs*) of Java.

Conclusion

After discussing a number of software complexity metrics, we have found out that Cyclomatic Complexity, namely CC can be regarded as the strongest metric compared to other complexity metrics that have been discussed (Halstead, LOC).

There is noticeable advantage from calculation of the Cyclomatic Complexity. Through it, we could calculate the value of the programme complexity. There should be measuring of the programme complexity (software) since this complexity is closely and directly related to a number of quality factors such as:

Maintenance cost and effort, understandability, reusability, testability, reliability, testing effort, time prediction and many others also.

By calculating the value of the CC, the values of the quality factors may be affected by depending on the value of CC. By doing so, we can contribute to the improvement of the concepts for the quality of the software to a large extent and also help in preparing strategies or certain technologies to control the complexity of the software by depending on these concepts.

Also, simplifying testing by guaranteeing the execution of at least one statement during testing. It checks each linearly independent path through the program. That is to say, the number of test cases will be equal to the cyclomatic complexity of the program**.** Such information is important for testing.

In our work system, we calculated the metric CC by depending on one of the methods of calculating CC, that is by depending on the decisional statements. We also Identified reliability risk level, testability level; and cost/effort so as to maintain (maintenance risk level) by depending on cyclomatic complexity CC. The study was implemented on 2 examples or programmes written in Java language (OOP) as we wanted to include all probabilities that can affect the value of the metric(CC). We have added what is called Bad Fix Probability which is also an important ratio whose value depends on the value of the metric (CC). This ratio or probability is that of the occurrence or emergence of new errors during the process of our correction of the current programme errors. That is to say, after doing modifications to the programme or what is called (maintenance).

We also clarified the results by certain tables and figures to illustrate the value of metric and its information in details.

Future Research

1) The inclusion or addition of the concept of cohesion i.e. (Intra –Module concept)and also the concept of coupling (Inter--Module concept), i.e. Coupling Between Object classes(CBO) so, the improvement of the concept of Cyclomatic Complexity of the system or the software will depend on these new concepts(cohesion and coupling).

2) The possibility of transferring the source codes that are written in the Object-oriented programming languages to other samples written in the Unified Modelling Language such as the Activity diagram or Class diagram or any other model by using a certain tool or technology. Finally, we calculate the complexity of these models, (i.e. loading Java programmes in the form of diagrams also and not only in the form of source code).

References

[1] Software Engineering, 7th edition, Roger S. Pressman, vol. 7. 2010.

[2] International Journal of Software Engineering and Its Applications Vol. 7, No. 2, March, 2013, Ayman Madi, Oussama Kassem Zein and Seifedine Kadry on the Improvement of Cyclomatic Complexity Metric.

[3] Ankita (2014) [A Framework for Improving the Concept of Cyclomatic Complexity in Object-Oriented Programming, 2014.

[4] Software metric [online] https://en.wikipedia.org/wiki/Software_metric

[5] NIST Special Publication 500-235, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Arthur H. Watson Thomas J. McCabe September 1996.

[6] A Review and Analysis of Software Complexity Metrics in Structural Testing. Mrinal Kanti Debbarma, Swapan Debbarma, Nikhil Debbarma, Kunal Chakma, and Anupam Jamatia. International Journal of Computer and Communication Engineering, Vol. 2, No. 2, March 2013.

[7] Metric for Early Measurement of Software Complexity. Ghazal Keshavarz et al., International Journal on Computer Science and Engineering (IJCSE) Vol. 3 No. 6 June 2011

[8] Complexity Metrics and Difference Analysis for better Application Management. Steve Kilner.

[9] Yahya Tashtoush, Al-Maolegi, and Bassam Arkok (2014): International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970) Volume-4 Number-2 Issue-15 June-2014 414, The Correlation among Software Complexity Metrics with Case Study.

[10] Dr. Madhavi (2016): A White Box Testing Technique in Software Testing: Basis Path Testing, Journal for Research, Volume 02, Issue 04, June 2016 ISSN: 2395-7549.

[11] Measurement of Quality, Mastering Software Quality Assurance Best Practices, Tools and Techniques for Software Developers, J. Ross 2010.

[12] An Analysis of the Design and Definitions of Halstead's Metrics Rafa E. AL QUTAISH, Alain ABRAN, Al-Qutaish, R. E. and Abran, A., "An Analysis of the Designs and the Definitions of the Halstead's Metrics",

In Proceedings of the 15th International Workshop on Software Measurement (IWSM'2005), September 12-14, 2005.

[13] [Online] https://www.coursehero.com/file/p1qe7un/Spinellis-2006-Code-Quality-The-Open-Source-Perspective-By-D-Spinellis-Addison

[14] Software metrics (2), Alexander Serebrenik, 2IS55 Software Evolution, 2011

[15] Repurposing Code Metrics for Use within Modern Day Programming, Luke Rickard, 2016.

[16] [Online] (Wikipedia, Cyclomatic_complexity https://en.wikipedia.org/wiki/Cyclomatic_complexity

[17] Abhilasha , Harshitha , Bhavya , Pavithra , and Dr.Saroja (2015): International Journal of Advanced Research in Computer Science and Software Engineering Research Paper Available online at: www.ijarcsse.com , Approaches to Improve Code Complexity Analysis, Volume 5, Issue 7, July 2015 ISSN: 2277 128X.

[18] Cyclomatic Complexity for WCF: A Service Oriented Architecture, Mir Muhammd Suleman Sarwar, Ibrar Ahmad, Sara Shahzad

[19] International Journal of Software Engineering and Its Applications Vol. 5 No. 3, July, 2011 Different Approaches to White Box Testing Technique for Finding Errors Mohd. Ehmer Khan.

[20] Software Reliability and Testing: Know When to Say When SSTC June 2007 Dale Brenneman McCabe Software.

[21] IJISET- International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 3, May 2014. www.ijiset.com ISSN 2348 - 7968 Software Reliability, Metrics, Reliability Improvement Using Agile Process Gurpreet Kaur1, Kailash Bahl.

[22] A Coupling-Complexity Metric Suite for Predicting Software Quality. Christopher L. Gray June 2008/ the Faculty of California Polytechnic State University San Luis Obispo]

[23] J. Software Engineering & Applications, 2009, 2: 137-143, doi:10.4236/jsea. 2009.23020 published Online October 2009, http://www.SciRP.org/journal/jsea Copyright © 2009 SciRes JSEA137 Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship Graylin Jay1, Joanne E. Hale2, Randy K. Smith1, David Hale2, Nicholas A. Kraft1, Charles Ward1.

[24] J. J. Vinju and M. W. Godfrey, “What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric”, IEEE 12th International Working Conference on Source Code Analysis and Manipulation, pp.154-163,2012.

[25] COSC 310, Software Engineering, Dr. Bowen Hui, 2017.

[26] Metrics Tool for Software Development Life Cycle. Thilagavathi Manoharan1, IJRIT International Journal of Research in Information Technology, Volume 2, Issue 1, January 2014.

[27] Management by Numbers© SE-CURE AG1SE-CURE AG (www.se-cure.ch) Dr. Hans Sassenburg, SPIN –Oct. 2nd, 2009.

[28] Wikipedia, Maintainability https://en.wikipedia.org/wiki/Maintainability

[29] On the Relationship between Software Complexity and Maintenance Costs Edward E. Ogheneovo Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria, Journal of Computer and Communications, 2014, 2, 1-16..