1 Introduction

Software complexity is one of the important factors that must be determined by clear procedures or methods that can be used in software production. Specialty there are huge amounts of software systems have been developed over the last three decades. The chapter will survey the methods of software complexity, and develop these methods to be used for measuring the complexity of software systems written by object-oriented languages. Measuring software complexity developed for managing projects is an extremely complex affair. Usually, the evolution of the software is the result of team works or rather several groups of specialists, who individually are experts in their respective disciplines, but probably may not have enough expertise in other disciplines (Nasib, 2005).

Software complexity must be measured to ensure the Quality Assurance (QA) of software. The quality assurance process is primarily concerned with defining or selecting standards that should be applied to the software development process or software product. As the part of QA process you may select and procure tools and methods to support these standards. Sommerville (2006) introduced two types of standards that may be established as part of the quality assurance process these are;

1. *Product standards*: these standards are applied to the software product being developed. These include document standards, such as the structure of documents requirement; documentation standards. Such as a standard comment header for an object class definition, and coding standards that define how a programming language should be used.

2. *Process standards*: these standards define the process that should be followed during software development. It includes definitions of specification design and validation process and a description of the documents that should be written in the path of these processes.

2 Software Complexity

Over the last three decades many measures have been proposed by researchers to analyze software complexity, understandability, and maintenance. Metrics designed to analyze software such as imperative, procedural, and object-oriented programs. Software measurement is concerned with deriving a numeric value for an attribute of a software product, i.e. a measurement is a mapping from the empirical world to the formal world.

Software metrics have been found to be useful in reducing software maintenance costs by assigning a numeric value to reflect the ease or difficulty with which a program module may be understood. There are
hundreds of software complexity measures that have been described and published. For example, the most basic complexity measure is the number of lines of code (LOC), simply counts the lines of executable code, data declarations, comments, and so on. While this measure is extremely simple, it has been shown to be very useful and correlates well with the number of errors in programs (Cardoso, 2006).

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures. There are hundreds of software complexity measures, ranging from the simple, such as source lines of code, to the esoteric, such as the number of variable definition/usage associations (Yanming & Shiyi, 2007).

Ideally, complexity measures should have both descriptive and prescriptive components. Descriptive measures identify software that is error-prone, hard to understand, hard to modify, hard to test, and so on. Prescriptive measures identify operational steps to help control software, for example splitting complex modules into several simpler ones, or indicating the amount of testing that should be performed on given modules (Watson & McCabe, 1996).

An important issue encountered in software complexity analysis is the consideration of software as a human creative artifact and the development of a suitable measure that recognizes this fundamental characteristic. The existing measures for software complexity can be classified into two categories: the macro and the micro measures of software complexity.

Major macro complexity measures of software have been proposed by Basili and by Kearney. The former considered software complexity as “the resources expended” (Basili, 1980). The latter viewed the complexity in terms of the degree of difficulty in programming (Kearney et al., 1986).

The micro measures are based on program code, disregarding comments and stylistic attributes. This type of measure typically depends on program size, program flow graphs, or module interfaces such as Halstead’s software science metrics (Halstead, 1977) and the most widely known cyclomatic complexity measure developed by McCabe (McCabe, 1976).

3 Line of Code Method

Software size is a key measure for many cost and effort estimation models. The predominant definition used today for lines of code (LOC) is a line of program code that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. Traditionally, the number of lines of code (LOC) was often used as an intuitive measure of size. Studies confirm that LOC is an effective measure for maintenance effort prediction. However, LOC cannot be measured early in the software development process. Furthermore, evidence suggests that LOC can be very inaccurate because they depend on language and development tools. LOC is highly language specific; therefore it can be used for comparison of code in the same language. Also, it is not applicable for visual languages because the notion of LOC may not be meaningful. It is not a suitable indicator at the design phase when the code has not been developed because it is not a good indicator of structural complexity. Many programmers use the LOC metric to depict a program’s size, complexity, or programming effort. There are few different ways of calculating the LOC metric. Some involve counting only non-white spaces lines, while others exclude comments (Garcia, 2008).

The most commonly used measure for the length of a code source of a program is the number of lines of code (LOC). The abbreviation NCLOC is used to represent a non commented source line of code. NCLOC is also sometimes referred to as effective lines of code (ELOC). NCLOC is therefore a measure
of the uncommented length. The commented length is also a valid measure, depending on whether or not line documentation is considered to be a part of programming effort. The abbreviation CLOC is used to represent a commented source line of code (Chis, 2008).

The suggested types to the lines of programs, that can be used with structure programming and object oriented programming are;

1. Line of codes (LOC) that do not contain comments.
2. Line of codes that contains several separate instructions, multiple line of code (MLOC=LOC*i); where (i) is the number of separate instructions in the same line.
3. Line of codes that contain comments as in the object oriented languages (LOC+CL), where CL is the comment line.
4. Blank lines BL
5. If the line contains more than one statements of comments, it will be (CL*j), where j is the number of comments statements.
6. Data lines measured by its size in bytes.

The general true program length will be calculated by;

\[
\text{General True Program Length (GTPL)} = \text{LOC} + (\text{LOC} \times i) + (\text{LOC} + \text{CL}) + (\text{CL} \times j) + \text{BL} \quad (1)
\]

The GTPL and data size in bytes both of them can be used in comparison of programs complexity that are written by using the same languages to solve the same problem, but the programmers are different. The GTPL can be used to measure the complexity from both sides of computer view to run a program and from the programmer side view. Equation (1) of the GTPL can be reduced to measure the complexity of the program from the computer side view by using;

\[
\text{Computer Program Length (CPL)} = \text{LOC} + (\text{LOC} \times i) + (\text{LOC} + \text{CL}) \quad (2)
\]

CL will be substituted by zero in this case. The data size will be used to measure the complexity in bytes. The suggested method of measurement can be used as a first indicator to measure the program complexity depends on the program length. But this method alone does not sufficient to measure the complexity; therefore the other methods that will be discussed in the following sections also will be used to measure the complexity.

4 Information Flow

The information flow metric is a structure metric that measures the sources (fan-in) and destinations (fan-out) of all data related to a given software component. These factors are used to compute the communication complexity of the component, which is taken as a measure of the strength of its communication relationship with other components. The fan-in consists of all function parameters and global data structures from which the function retrieves its information, while the fan-out consists of the return values from function calls and the global data structures that the function updates.

Measures of control-flow structure have been the interest of software metrics work since the beginning of the 1970’s. This approach defines a connection between two components to exist if there is a possible flow of information from one to the other. The information flow metrics are based on the structure created by the flow of information within a system. The procedure for calculating the metrics is as follows:

1. Identify the information flows in a system:
• **Local Direct Flow**: if a module invokes a second module and passes information to it or if the invoked module returns a result to the caller.

• **Local Indirect Flow**: if the invoked module returns information, which is subsequently passed to another invoked module.

• **Global Flow**: if there is flow of information from one module to another, via a global data structure.

2. Calculate fan-in and fan-out:
   - **Fan-in**: number of local flows that terminate at a module, plus the number of data structures from which information is retrieved.
   - **Fan-out**: number of local flows that emanate from a module plus the number of data structures that are updated by that module

3. Calculate complexity of each module:
   The most commonly used method is fan-in and fan-out metrics. Henry and Kafura defined the structure complexity as (Henry & Kafura, 1981);

   \[ C = (\text{fan-in} \ast \text{fan-out})^2 \] (3)

   Where C stands for the amount of information flow in a module, fan-in is a module's inputs, and fan-out is the amount of return values and variables changed (Yanming & Shiyi, 2007).

5 **McCabe Method**

McCabe (1976) proposed “A complexity measure” that brought forward the idea of cyclomatic complexity for the first time, and it basically measures decision points or loops of the program. This method can show the intelligibility, testability, and maintainability. Cyclomatic complexity utilizes a graph, which is derived from code. The formula defined as:

\[ MC = V(G) = e - n + 2p \] (4)

Where e is the number of edges, n is the number of nodes, and p is the number of connected components as shown in Figure (1).

![Figure 1: Example for McCabe’s Cyclomatic number](image)
According to McCabe, a 3,000-line program with five IF/THEN statements is less complex than a 200-line program with six IF/THEN statements. Many programmers find this assertion hard to swallow. McCabe complexity calculates total number of possible control paths through a program, using a flow graph. In some programs, it is possible to have an infinite number of loops. From programmer side view, the MacCabe method can be applied, because the method calculates the number of loops. But this is not applicable from the computer side view, because the number of iteration of each loop is not appearing in the MacCabe method, therefore the suggested method of calculation is;

\[ YAS = e^{-n} + 2p \times i \]  \hspace{1cm} (5)

Where i; is the number of iteration of each loop and if there are more than one loop in a program and their iterations are different, the equation will be;

\[ YAS = e^{-n} + 2 \times \sum p_{ij} \]  \hspace{1cm} (6)

Where;

- i: is the number of iteration of each loop, i=0, ..., n
- j: is the number of loops in a program, j=0, ..., m

**Example1**: if a program consist from (15) edges, with (10) nodes, with three loops, where;
- Loop1; iterate 5 times
- Loop2; iterate 10 times
- Loop3; iterate 15 times
The complexity will be;
- \( MC = e^{-n+2p} = 15-10+2*3 = 11 \) (by using MacCabe equation 4)
- \( YAS = e^{-n+2} \times \sum p_{ij} = 15-10+2(5+10+15) = 65 \) (by using our YAS equation 6)
We note from these calculations the complexity from the computer side view is completely different.

**Example2**: if Example1 is repeated with the same number of (15) edges and (10) nodes, but with the following iterations loops.
- Loop1; iterate 10 times
- Loop2; iterate 20 times
- Loop3; iterate 30 times
The complexity will be;
- \( MC = e^{-n+2p} = 15-10+2*3 = 11 \) (the same as in example 1, by using MacCabe equation 4)
- \( YAS = e^{-n+2} \times \sum p_{ij} = 15-10+2(10+20+30) = 125 \) (the complexity will be higher than in example 1, because the number of iterations are higher, by using YAS equation 6)

6 **Halstead’s Method**

In 1976 Maurice Halstead made an attempt to capture notions on size and complexity beyond the counting number of operators and number of operands of the program (Magnus, 2004). Halstead’s metrics are related to the areas that were being advanced in the seventies, mainly psychology literature.

Halstead began defining a program \( P \) as a collection of tokens, classified as either operators or operands. The basic metrics for these tokens where:
- \( \mu_1 \) = number of unique operators
• \( \mu_2 \) = number of unique operands
• \( N_1 \) = total occurrences of operators
• \( N_2 \) = total occurrences of operands

For example the statement: \( f(x) = h(y) + 1 \), has two unique operators ( = and + ) and two unique operands ( \( f(x) \) and \( h(y) \)).

For a program \( P \), Halstead defined the following metrics:

\[
\text{The length } N \text{ of } P: N = N_1 + N_2 \quad (7)
\]
\[
\text{The vocabulary } \mu \text{ of } P: \mu = \mu_1 + \mu_2 \quad (8)
\]
\[
\text{The volume } V \text{ of } P: V = N \times \log_2 \mu \quad (9)
\]
\[
\text{The program difficulty } D \text{ of } P: D = (\mu_1 / 2) \times (N_2 / \mu_2) \quad (10)
\]
\[
\text{The effort } E \text{ to generate } P \text{ is calculated as: } E = D \times V \quad (11)
\]

The volume, vocabulary and length value can be viewed as different size measures. Take the vocabulary metric for instance: It calculates the size of a program very different from the LOC metric and in some cases it may be a much better idea to look at how many unique operands and operators a module has than just looking at lines of code. Halstead’s metrics are most often used during code development in large projects in order to track complexity trends.

Halstead did not distinguish between the different types of mathematical operations such as; addition, subtraction, multiplication and division. From the computer side view, each of these mathematical operations needs specific processing time different from the other operations. Table (1) shows example to the mathematical operations to two microprocessors (\( \mu_p \)), that their processing time are available to us.

<table>
<thead>
<tr>
<th>Operation</th>
<th>( \mu_p )</th>
<th>80486 (33 MHz)</th>
<th>Pentium (100 MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td></td>
<td>0.12</td>
<td>0.04</td>
</tr>
<tr>
<td>Subtraction</td>
<td></td>
<td>0.12</td>
<td>0.04</td>
</tr>
<tr>
<td>Multiplication</td>
<td></td>
<td>0.59</td>
<td>0.13</td>
</tr>
<tr>
<td>Division</td>
<td></td>
<td>9.2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

**Table 1**: Processing time of the mathematical operation to the 80486 and Pentium \( \mu_p \).

To show the effect of mathematical operation (operators) types on processing time, some of the examples with suggested number of operators \( \mu_1 \), are selected, one of them for operator (=) and the others are either addition, multiplication or division (that will be multiplied with the processing time of each microprocessor). The number of operands \( \mu_2 \) is proposed to show the effect of operators’ types to the 80486 and Pentium microprocessors as shown in tables (2 and 3).

<table>
<thead>
<tr>
<th>Example</th>
<th>( \mu_1 )</th>
<th>( \mu_2 )</th>
<th>( N_1 )</th>
<th>( N_2 )</th>
<th>( E+(486) )</th>
<th>( E*(486) )</th>
<th>( E+(486) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>106</td>
<td>129.5</td>
<td>560</td>
</tr>
<tr>
<td>Ex2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td>424</td>
<td>518</td>
<td>2240</td>
</tr>
<tr>
<td>Ex3</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>30</td>
<td>954</td>
<td>1165.5</td>
<td>5040</td>
</tr>
<tr>
<td>Ex4</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>40</td>
<td>1696</td>
<td>2072</td>
<td>8960</td>
</tr>
<tr>
<td>Ex5</td>
<td>2</td>
<td>2</td>
<td>25</td>
<td>50</td>
<td>2650</td>
<td>3237.5</td>
<td>14000</td>
</tr>
<tr>
<td>Ex6</td>
<td>2</td>
<td>2</td>
<td>30</td>
<td>60</td>
<td>3816</td>
<td>4839</td>
<td>20160</td>
</tr>
<tr>
<td>Ex7</td>
<td>2</td>
<td>2</td>
<td>35</td>
<td>70</td>
<td>5194</td>
<td>6345.5</td>
<td>27440</td>
</tr>
</tbody>
</table>

**Table 2**: Effect of addition multiplication, and division operators on the efforts of the 80486 \( \mu_p \).
Note: the addition and subtraction operations are approximately need the same processing time. Figure (2) shows the effort of the division is higher than the addition and multiplication operations.

![Graph showing effort comparisons](image)

Figure 2: Comparison between efforts according to mathematical operations of the 80486 µp.

<table>
<thead>
<tr>
<th>Example</th>
<th>µ1</th>
<th>µ2</th>
<th>N1</th>
<th>N2</th>
<th>E+(p)</th>
<th>E*(p)</th>
<th>E÷(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>102</td>
<td>106.5</td>
<td>175</td>
</tr>
<tr>
<td>Ex2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td>408</td>
<td>426</td>
<td>700</td>
</tr>
<tr>
<td>Ex3</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>30</td>
<td>918</td>
<td>958.5</td>
<td>1575</td>
</tr>
<tr>
<td>Ex4</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>40</td>
<td>1632</td>
<td>1704</td>
<td>2800</td>
</tr>
<tr>
<td>Ex5</td>
<td>2</td>
<td>2</td>
<td>25</td>
<td>50</td>
<td>2550</td>
<td>2662.5</td>
<td>4375</td>
</tr>
<tr>
<td>Ex6</td>
<td>2</td>
<td>2</td>
<td>30</td>
<td>60</td>
<td>3672</td>
<td>3834</td>
<td>6300</td>
</tr>
<tr>
<td>Ex7</td>
<td>2</td>
<td>2</td>
<td>35</td>
<td>70</td>
<td>4998</td>
<td>5218.5</td>
<td>8575</td>
</tr>
</tbody>
</table>

**Table 3**: Effect of addition, multiplication and division operators on the efforts of the Pentium µp.

Figure (3) shows the effort of the division is higher than the addition and multiplication operations to the Pentium µp.

![Graph showing effort comparisons](image)

Figure 3: Comparison between efforts according to mathematical operations to Pentium µp.
7 Cognitive Weights Method

Complexity measure based on weighted information count of the software and cognitive weights has been developed by Kushwaha and Misra (Kushwaha & Misra, 2005). Basic control structures [BCS] such as sequence, branch and iteration are the basic logic building blocks of any software and the cognitive weights (Wc) of a software is the extent of difficulty or relative time and effort for comprehending a given software modeled by a number of BCS’s. These cognitive weights for BCS’s measure the complexity of logical structures of the software. Either all the BCS’s are in a linear layout or some BCS’s are embedded in others. For the former case, we sum the weights of all the BCS’s and for the latter, cognitive weights of inner BCS’s are multiplied with the weight of external BCS’s (Kushwaha & Misra, 2006).

Wang and Shao defined the cognitive weight as a metric to measure the effort required for comprehending a piece of software (Wang & Shao, 2003). Based on empirical studies, they defined cognitive weights for basic control structures. Table (4) shows the basic control structures and its cognitive weights which are a measure for the difficulty to understand a control structure. The cognitive weight of a basic control structure is a measure for the difficulty to understand this control structure.

<table>
<thead>
<tr>
<th>Control Structures</th>
<th>Wi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence (an arbitrary number of statements in a sequence without branching)</td>
<td>1</td>
</tr>
<tr>
<td>Call of a user-defined function</td>
<td>2</td>
</tr>
<tr>
<td>Branching with if-then or if-then-else</td>
<td>2</td>
</tr>
<tr>
<td>Branching with case (with an arbitrary number of selectable cases)</td>
<td>3</td>
</tr>
<tr>
<td>Iteration (for-do, repeat-until, while-do)</td>
<td>3</td>
</tr>
<tr>
<td>Recursive function call</td>
<td>3</td>
</tr>
<tr>
<td>Execution of control flows in parallel</td>
<td>4</td>
</tr>
<tr>
<td>Interrupt</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4: Basic control structures and its cognitive weights

The cognitive weight of a software component without nested control structures is defined as the sum of the cognitive weights of its control structures according to table (4). It seems to be a promising approach to use the ideas to define cognitive weights for Business Process Models (BPMs).

8 Object-Oriented Programming

The software industry is moving toward object-oriented design. Although the move was slow in the beginning, it has received positive acceptance in recent years. However, with the popularity of object-oriented programming languages such as JAVA, C++, C#, .NET, and Visual Basic.NET, object-oriented programming languages prove to be faster to learn and easier to maintain. In addition, object-oriented programming languages provide a way to break large and sometimes difficult-to-manage programming projects into smaller modules that can be managed easily.

Each object oriented language have a number of keywords, sometimes these keywords are different from one language to the other. There are a large number of similarities between Java and C#, table (5) shows the comparison between Java and C# keywords.
Table (5) shows that there are 35 keywords with similar identification in both languages C# and Java. There are 15 keywords having different identification from both languages for example base in C# language is represented with super in Java language, we can predicate that one of the reasons of increasing software complexity’s the absence of some keywords from the languages. From the programmer view it is easier to programming when the keywords less, but from the computer view, it will be more complex and may be need longer access time.

9 Software Complexity Examples

i. Linear search; algorithm is used as an example to compare the complexity of a program if written using C++, Visual Basic, and Java programming languages. The listing of the main program (test program) of the linear search was written in C++ is as shown in figure (4). The LinearSearch method is not included in the comparison for simplicity. From Deitel and Deitel book “C++ How to program” (Deitel & Deitel., 2007): Figure (4) shows the flow graph of the LinearSearch main program.
Table (6) shows the complexity of the main program of the linear search written in C++ language. The table shows that the length in lines of the program is 32, and LOC without comments is 20. The measured LOC+ comments is 7, where LOC+ comments, is that mixed line of code with comments on the same line. The McCabe complexity is measured as 4 while the program difficulty by Halstead method is 9.3. While the file size of this program is 954 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>32</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>20</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>7</td>
</tr>
<tr>
<td>McCabe complexity</td>
<td>4</td>
</tr>
<tr>
<td>The Program difficulty D (As per Halstead method)</td>
<td>9.3</td>
</tr>
<tr>
<td>File Size</td>
<td>954 bytes</td>
</tr>
</tbody>
</table>

Table 6: The complexity of the main program of the linear search written in C++

Figure 4: Main program with the linear search flow graph written in C++ language

```cpp
    // C++
    // Linear search of an array.
    #include <iostream>
    using std::cout;
    using std::cin;
    using std::endl;
    int linearSearch( const int [], int, int ); // prototype

    int main()
    {
      const int arraySize = 100; // size of array a
      int a[ arraySize ]; // create array a
      int searchKey; // value to locate in array a
      int element = linearSearch( a, searchKey, arraySize );
      // display results
      if (element != -1 )
        cout << "Found value in element " << element << endl;
      else
        cout << "Value not found" << endl;
      return 0; // indicates successful termination
    } // end main
```
The listing of the main program (test program) of the linear search written in Visual Basic is shown in figure (5). The method LinearSearch is not used in the comparison for simplicity. Figure (5) shows the flow graph of the linear search algorithm from Deitel and Deitel book “Visual Basic 2008 for Programmers” (Deitel & Deitel, 2008).

Figure 5: Main program with the linear search flow graph written in Visual Basic language

Table (7) shows the complexity of the main program linear search written in Visual Basic language. The table shows that the length in lines of the program is 45, LOC without comments is 27, and LOC +
comments are 6. The McCabe complexity is 6 while the program difficulty by Halstead method is 11.6. The file size of this program is 1789 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>45</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>27</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>6</td>
</tr>
<tr>
<td>McCabe complexity</td>
<td>6</td>
</tr>
<tr>
<td>The Program difficulty D (As per Halstead method)</td>
<td>11.6</td>
</tr>
<tr>
<td>File Size</td>
<td>1,789 bytes</td>
</tr>
</tbody>
</table>

Table 7: The complexity of the main program of the linear search written in Visual Basic

The listing of the main program (test program) of linear search written in Java language is shown in figure (6). The method LinearSearch is not used in the comparison for simplicity. Figure (6) shows the flow graph of the algorithm from book “Java How to program” (Deitel & Deitel, 2009).

```java
// Java
// Sequentially Linear search an array for an item.
import java.util.Scanner;

public class LinearSearchTest {
    public static void main(String args[]) {
        // create Scanner object to input data
        Scanner input = new Scanner(System.in);

        int searchInt; // search
        int position; // location of search key in array

        // create array and output it
        LinearArray searchArray = new LinearArray(10);
        System.out.println(searchArray);

        // get input from user
        System.out.print("Please enter an integer value (-1 to quit): ");
        searchInt = input.nextInt(); // read an int from user

        // repeatedly input an integer; -1 will quit the program
        while (searchInt != -1) {
            // search array linearly
            position = searchArray.linearSearch(searchInt);

            if (position == -1) {
                System.out.println("The integer "+searchInt+" was not found.

                System.out.println("The integer "+searchInt+" was found in position "+position+".

            }

            // get input from user
            System.out.print("Please enter an integer value (-1 to quit): ");
            searchInt = input.nextInt();
        } // end while
    } // end main
} // end class
```
**Figure 6**: Main program with the linear search flow graph written in Java language

Table (8) shows the complexity of the main program linear search written in Java language. The table shows that the length in lines of the program as 44, LOC without comments is 21, and LOC + comments is 6. The McCabe complexity is 7, the program difficulty by Halstead method is 5.8, while the file size of this program is 1524 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>44</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>21</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>6</td>
</tr>
<tr>
<td>McCabe complexity</td>
<td>7</td>
</tr>
<tr>
<td>The Program difficulty D (As per Halstead method)</td>
<td>5.8</td>
</tr>
<tr>
<td>File size</td>
<td>1,524 bytes</td>
</tr>
</tbody>
</table>

**Table 8**: The complexity of the main program linear search written in Java

Figure (7) shows the comparison between the object oriented languages C++, Visual Basic, and Java by using the main program of the linear search algorithm as a case study for comparison.

**ii. binary search** is the second case study taken to measure the complexity written in C++, Visual Basic, and Java languages. The listing of the main program (test program) of the binary search algorithm was written in C++ is shown in Figure (8), while the method of the binarySearch is not used in the comparison for simplicity. Figure (8) shows the flow graph of the algorithm from Deitel and deitel book “How to program C++” (Deitel & Deitel, 2007).
Table (9) shows the complexity of the main program of the binary search written in C++ language. The table shows that the length in lines of the program is 44, LOC without comments as 25, and LOC + comments as 7. The McCabe complexity is 7, the program difficulty by Halstead method is 5.1, while the file size of this program is 1393 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>44</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>25</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 8: Main program with the binary search flow graph written in C++ language
McCabe complexity | 7
---|---
The Program difficulty D (As per Halstead method) | 5.1
File Size | 1,393 bytes

Table 9: The complexity of the main program of the binary search written in C++

The listing of the main program (test program) of the binary search written in Visual Basic is shown below. The method BinarySearch is not used in the comparison for simplicity. Figure (9) shows the flow graph of binary search algorithm written in Visual Basic language from Deitel and Deitel book “Visual Basic 2008 for programmers” (Deitel & Deitel, 2008).

```vbnet
1 ' Visual Basic
2 ' Binary search of an array using Array.BinarySearch.
3 Imports System
4 Public Class FrmBinarySearchTest
5 Dim array1() As Integer = New Integer(19) {}
6 ' create random data
7 Private Sub btnCreate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCreate.Click
8 Dim randomNumber As Random = New Random()
9 Dim output As String = ("Index" & vbTab & "Value" & vbCrLf)
10 ' create random array elements
11 For i As Integer = 0 To array1.GetUpperBound(0)
12 array1(i) = randomNumber.Next(1000)
13 Next
14 ' display sorted array elements
15 For i As Integer = 0 To array1.GetUpperBound(0)
16 output &= (i & vbTab & array1(i) & vbCrLf)
17 Next
18 txtData.Text = output ' displays numbers
19 txtInput.Text = "" ' clear search key text box
20 btnSearch.Enabled = True ' enable search button
21 End Sub ' btnCreate_Click
22 ' search array for search key
23 Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnSearch.Click
24 Dim searchKey As Integer = Convert.ToInt32(txtInput.Text)
25 Dim element As Integer = Array.BinarySearch(array1, searchKey)
26 If element >= 0 Then
27 lblResult.Text = "Found Value in index « & element
28 Else
29 lblResult.Text = "Value Not Found"
30 End If
31 End Sub ' btnSearch_Click
32 End Class ' FrmBinarySearchTest
```

Figure 9: Main program with the binary search flow graph written in Visual Basic language
Table (10) shows the complexity of the main program of the binary search written in Visual Basic language. The table shows that the length in lines of the program with comments is 53, LOC without comments is 31, and LOC+ comments is 5. The McCabe complexity is 9, the program difficulty by Halstead method is 10.1. The file size of this program is 2026 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>53</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>31</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>5</td>
</tr>
<tr>
<td>McCabe complexity</td>
<td>9</td>
</tr>
<tr>
<td>The Program difficulty D (As per Halstead method)</td>
<td>10.1</td>
</tr>
<tr>
<td>File Size</td>
<td>2,026 bytes</td>
</tr>
</tbody>
</table>

Table 10: The complexity of the main program of the binary search written in Visual Basic

The listing of the main program (test program) of the binary search written in Java is shown in figure (10). The method of the BinarySearch is not used in the comparison for simplicity. Figure (10) shows the flow graph of binary search algorithm written in Java language from Deitel and Deitel book “How to program Java” (Deitel & Deitel, 2009).

```
1  // Java
2  // Sequentially Binary search an array for an item.
3  import java.util.Scanner;
4  
5  public class BinarySearchTest
6  {
7      public static void main( String args[] )
8      {
9          // create Scanner object to input data
10         Scanner input = new Scanner( System.in );
11         
12         int searchInt; // search
13         int position; // location of search key in array
14         
15         // create array and output it
16         BinaryArray searchArray = new BinaryArray( 16 );
17         System.out.println( searchArray );
18         
19         // get input from user
20         System.out.print( "Please enter an integer value (-1 to quit): ");
21         searchInt = input.nextInt(); // read an int from user
22         
23         // repeatedly input an integer; -1 will quit the program
24         while ( searchInt != -1 )
25         {
26             // use binary search to try to find integer
27             position = searchArray.binarySearch( searchInt );
28             
29             // return value of -1 indicates integer was not found
30             if ( position == -1 )
31                 System.out.println( "The integer " + searchInt + 
32                                      " was not found.
33                                      ").
34             else
35                 System.out.println( "The integer " + searchInt + 
36                                      " was found in position " + position + 
37                                      ").
38             // get input from user
39             System.out.print( "Please enter an integer value (-1 to quit): ");
40             searchInt = input.nextInt();
41         } // end while
42     } // end main
43 } // end class BinarySearchTest
```

Figure 10: Main program with the binary search flow graph written in Java language
Table (11) shows the complexity of the main program of the linear search written in Java language. The table shows that the length in lines of the program is 44, LOC without comments is 24, and LOC + comments are 6. The McCabe complexity is 7, the program difficulty by Halstead method is 6.4. The file size of this program is 1572 bytes.

<table>
<thead>
<tr>
<th>Complexity Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (in lines)</td>
<td>44</td>
</tr>
<tr>
<td>LOC without comments</td>
<td>24</td>
</tr>
<tr>
<td>LOC + comments</td>
<td>6</td>
</tr>
<tr>
<td>McCabe complexity</td>
<td>7</td>
</tr>
<tr>
<td>The Program difficulty D (As per Halstead method)</td>
<td>6.4</td>
</tr>
<tr>
<td>File Size</td>
<td>1,572 bytes</td>
</tr>
</tbody>
</table>

*Table 11: The complexity of the main program binary search written by Java*

Figure (11) shows the comparison between the object oriented language C++, Visual Basic, and Java, by using the main program of the binary search algorithm as a case study for comparison.

Table (12) shows the comparison between McCabe and Halstead methods for Linear Search of an array taken in consideration three types of languages C++, Visual Basic and Java.

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Visual Basic</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe Method</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Halstead's Method</td>
<td>9.3</td>
<td>11.6</td>
<td>5.8</td>
</tr>
</tbody>
</table>

*Table 12: Comparison of McCabe vs. Halstead methods for linear search of an array*

Table (13) shows the comparison between McCabe and Halstead methods for Binary Search of an array taken in consideration three types of languages C++, Visual Basic and Java.
<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Visual Basic</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe Method</td>
<td>7</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Halstead's Method</td>
<td>5.1</td>
<td>10.1</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Table 13: Comparison of McCabe vs. Halstead methods for Binary search of an array

Figures (12 and 13) are showing the differences between McCabe and Halstead measurement for Linear and Binary Search of an array for the three languages.

10 Conclusion

In this chapter, a brief description about object oriented programming and the development history of C++, Visual Basic, and Java languages was introduced. New way of measuring line of code and command was introduced. The McCabe and Halstead methods were modified to be capable to measure the software complexity by using structural and object oriented programming languages.

Two case studies to measure the complexity of linear search algorithm and binary search algorithm were discussed. These algorithms were written by three most popular object oriented languages C++, Visual Basic, and Java. The complexity to each of these examples were measured the line of code (LOC) without comments, LOC + comments, McCabe method, the program difficulty using Halstead method and file size. It is found that McCabe method has various value of complexity for C++, Visual Basic, and Java languages for linear search and similar measuring value for C++ and Java languages for binary search the value for C++ and Java languages was 7. The measured complexity with McCabe method is higher for Visual Basic languages with value equal to 9 in binary search. Using Halstead method implies various
values of complexity regarding using different programming languages either in linear or binary search cases. On the other words, the measured values of complexity are different for either linear or binary search from one language to another means that if one program is written in C++ language then it is complexity will be different with other type of languages such as Visual Basic and Java.

References


Deitel H..and Deitel P. J., (2009), Java How to Program, Pearson Prentice Hall.


Sommerville I., (2006), Software Engineering, Pearson Education Ltd.
