An Experiment to Measure the Cognitive Weights of Code Control Structures

David Adamo Jr.

2014-12-06

Abstract

Various measures of software complexity have been proposed over the years. More recently, cognitive complexity measures of software have been proposed as a complement to other more widely adopted measures that are primarily based on physical size and/or a systematic analysis of code. Many of these cognitive measures consider the notion of cognitive weight to be an integral part of code comprehension and complexity. Previous work has proposed experiments to derive cognitive weight values for code control structures. Unfortunately, there are a number of flaws in these experiments. These flaws in previous works’ experiments need to be addressed by a new experiment if reasonable cognitive weight values for code control structures are to be derived. The experimental design and tool presented in this paper attempt to address some of the problems in previous experiments. Cognitive weight values for code control structures are derived using data gathered from our experiments and are then compared with cognitive weight values from previous works.

1 Introduction

A common way of measuring software complexity is Lines of Code (LOC). However, it can be argued that LOC does not give a sufficient representation of overall code complexity since it fails to take code semantics, functionality and ease of comprehension into account. Therefore, it can be concluded that measuring the complexity of a program takes much more than just counting lines of code [1].

It is quite common for organizations to have coding standards and ‘complexity metrics’ that guide the manner in which developers write code and also help identify potentially error-prone code that is difficult to comprehend [2]. Typically, these coding standards affect a programmer’s choice of data structures, control structures, variable names and overall code design with the aim of achieving a high standard of code quality and expressiveness. Maintaining such standards of code quality can positively influence the ability of a programmer to comprehend an already existing codebase. The manner in which code is written (choice of control structures, data structures, variable names, etc.) significantly affects its overall cognitive complexity and maintainability. Software development is driven
by software comprehension. Managing a software development process is dependent on controlling software comprehension [3]. Therefore, there is a need for suitable methods of reliably and empirically measuring factors that influence software comprehension. Such a measurement can help organizations further refine their coding standards to take code comprehension and functional complexity into consideration. This paper attempts to provide a reliable and reasonable way to measure the cognitive weights of basic control structures in high-level imperative programming languages. Such cognitive weight measures can subsequently serve as a basis for determining the overall cognitive complexity of programs written in a particular programming language.

2 Existing Software Complexity Measures

This section reviews some existing measures of software complexity.

2.1 Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly-independent paths through a program or software component [4]. The Cyclomatic Complexity (CC) of a program is calculated using a connected graph that represents the control flow of the program, i.e.

\[
CC = e - n + p
\]  

where \( e \) is the number of edges of the flow graph, \( n \) is the number of nodes, and \( p \) is the number of connected components in the graph. In the connected graph, the nodes are blocks of code delimited by statements that affect that control flow e.g. if, while. Two nodes have edges between them if there is an execution path from one node to the other.

The cyclomatic approach considers only internal structures (loops and branches) while I/Os that may affect the functional complexity of a program are not considered.

2.2 Halstead’s Software Metrics

Halstead’s complexity metrics place emphasis on computational complexity derived directly from source code [5]. This complexity measure focuses on operators and operands in a software component or program. Complexity measures of a program are derived based on four scalar properties: number of distinct operators, number of distinct operands, total number of operators and total number of operands.

Halstead’s software metrics are best suited for programs written in assembly language since the approach focuses on small details of measurement [6].

2.3 Complexity Measures Based on Physical Size

Complexity measures such as LOC that are based on the physical size of programs are very well known. LOC provides a means for estimating the size and complexity
of a program. Admittedly, complexity measures based on physical size provide pretty good estimates of complexity. However, one downside of such complexity measures is the fact that the complexity estimates can vary quite significantly for different high level languages. Furthermore, the accuracy of such complexity measures is heavily dependent on a programmer’s skill level and programming style. Also, the accuracy of LOC complexity estimates may be affected by the features of programs, such as difficulty of algorithms, high reliability requirements and real-time functional requirements [6].

3 Cognitive Complexity Measures

Cognitive informatics plays an important role in understanding the fundamental characteristics of software [7]. Attempts at deriving a suitable cognitive complexity measure for computer programs have been mostly inspired by a need to take the internal structure of software and the I/Os it processes into account. Even further, cognitive complexity measures attempt to directly factor program comprehension effort into the measurement framework. Some attempts at deriving suitable cognitive complexity measures for computer programs will be discussed in this section.

One of the failings of commonly used code complexity metrics is that they do not take a programmer’s experience into account. A veteran programmer and a novice are very likely to face different challenges when attempting to comprehend the same piece of code. Many of these challenges have been shown to have very little to do with cyclomatic complexity or LOC measures by Hansen et al. [8]. Hansen et al. present an experiment in which participants with a wide variety of Python and overall programming experience predict the output of ten small Python programs. The aim of their experiments was to find and quantify the relationship between program text, programmer experience, speed of response and correctness of response. They also attempted to identify errors that are peculiar to experienced programmers and novices. The results of the experiments conducted by Hansen et al. indicate that the relationship between program text, experience and response is quite complex even for simple programs. The experimental design employed by Hansen et al. forms a strong basis for the experimental design used in this paper. However, this paper does not focus on finding a relationship between program text, experience and response, but rather attempts to come up with experiments that help determine reasonable cognitive weight values for basic control structures in a programming language.

Klemola and Rilling [3] have proposed a cognitive complexity measure based on category learning. Their approach is based on the premise that code comprehension consists of several processes including recognition, learning, grouping concepts or chunking, searching for occurrences of a term and familiarity of the individual with the artifacts in question. They introduce the concept of Identifier Density (ID). ID can be calculated as the number of identifiers divided by LOC. Klemola and Rilling also define Kind of Line of Code Identifier Density (KLCID). KLCID represents the number of unique lines of code. Lines that have the same
operands with the same arrangement of operators are considered equal. The metrics proposed by Klemola and Rilling are complex to calculate since it involves comparing each line of code with every other line of code in a software component or program. Furthermore, the proposed metric does not include human subjects in the derivation process.

Shao and Wang have made one of the most influential attempts to derive a cognitive complexity measure for computer programs in [6]. Their approach is based on the premise that comprehending a given program usually begins with a focus on the architecture and basic control structures of the code. Arriving at a measure of overall cognitive complexity starts with assigning cognitive weights to basic control structures. Common basic control structures typically fall into either of three categories: sequential, branching and iteration. According to Shao and Wang [6], the cognitive weight is the degree of difficulty or relative time and effort required for comprehending a given piece of software modeled by a number of basic control structures. Shao and Wang assign cognitive weights to the three categories of control structures mentioned previously and also go ahead to extend these categories by considering recursion, parallel, function call and interrupt [6].

In Shao and Wang’s work, cognitive weights for basic control structures derived from psychological experiments form the basis for an overall complexity measure. In most programs, it is quite common to have control structures that are embedded in other control structures. When this is not the case, the weights of all the control structures in a program can simply be summed. However, when there are control structures embedded in others, a simple summation would not suffice. Shao and Wang [6] provide a definition for total cognitive weight. The total cognitive weight of a software component \( W_c \), is defined as the sum of the cognitive weights of its \( q \) linear blocks composed of individual basic control structures. Each block may consist of \( m \) layers of nesting control structures and each layer of \( n \) linear basic control structures. Therefore, the total cognitive weight, \( W_c \) can be calculated by

\[
W_c = \sum_{j=1}^{q} \left[ \prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j, k, i) \right] \tag{2}
\]

If there are no embedded control structures in any of the \( q \) blocks, that is, when \( m=1 \), then (2) can be simplified as:

\[
W_c = \sum_{j=1}^{q} \sum_{i=1}^{n} W_c(j, i) \tag{3}
\]

Shao and Wang also define unit of cognitive weight and cognitive functional size. The unit of cognitive weight (CWU) of a program is the cognitive weight of the simplest software component with only a single I/O and a linear structured basic control structure. The cognitive functional size of a basic software component is defined as a product of the sum of inputs and outputs and the total cognitive weight [6].

A number of modified cognitive complexity measures have been proposed based on the work of Shao and Wang. Kushwaha and Misra [9] have proposed Cognitive
**Information Complexity Measure (CICM).** CICM is defined as the product of **Weighted Information Count of Software (WICS)** and the cognitive weight ($W_c$) of the basic control structures in the program or software component. Kushwaha and Misra also provide definitions for information contained in one line of code ($I_k$), **Information Contained in Software (ICS)**, **Weighted Information Count of a Line of Code (WICL_k)** and **Weighted Information Count of Software (WICS)**.

$I_k$ is given by:

$$I_k = (\text{Identifiers} + \text{Operands})_k IU = (ID_k + OP_k)IU$$

where $ID = \text{total number of identifiers in the } k\text{th line}$, $OP = \text{total number of operators in the } k\text{th line}$ and IU is the *Information Unit* representing that any identifier or operator has at least one unit of information in them.

The ICS is the sum of information contained in each line of code, that is:

$$ICS = \sum_{k=1}^{LOC} I_k$$

where $I_k = \text{information contained in the } k\text{th line of code}$ and $LOC = \text{total lines of code in the program or software component}$.

The WICL for the $k\text{th}$ line of code is given by:

$$WICL_k = ICS_k / [LOC - k]$$

where $WICL_k = \text{weighted information count for the } k\text{th line}$ and $ICS_k = \text{information contained in a software for the } k\text{th line}$.

The WICS is defined as follows:

$$WICS = \sum_{k=1}^{LOC} WICL_k$$

Having provided the aforementioned definitions, Kushwaha and Misra note that a cognitive complexity measure should also consider the basic control structures of software in order to be complete and robust. Therefore, a definition of CICM is provided as follows:

$$CICM = WICS \times W_c$$

Similar to the approach proposed by Klemola and Rilling [3], CICM is quite complex to calculate since it requires calculating the weighted information count of each line. Also, Kushwaha and Misra claim that CICM is based on cognitive informatics. However, in cognitive informatics, the functional complexity of software only depends on input, output and internal architecture and not on operators [7, 10]. In an attempt to deal with these issues and provide a complexity measure more in line with cognitive informatics, Misra proposed a Cognitive Program Complexity Measure (CPCM) in [10]. CPCM considers the total occurrences of inputs and output variables as well as the cognitive weights of basic control structures. The complexity measure due to inputs and outputs is calculated as:

$$S_{IO} = N_{i1} + N_{i2}$$
where \( N_{i1} \) is the total occurrences of input variables and \( N_{i2} \) is the total occurrences of output variables.

In combination with the total cognitive weight of a software component \( W_c \) defined in (2), (3) and proposed in [6], Misra suggests a simple cognitive complexity formula for CPCM as follows:

\[
CPCM = S_{IO} + W_c
\]

### 4 Cognitive Weight of Control Structures

The previous section gave an overview of some of cognitive complexity measures of software that have been proposed in previous work. The cognitive weight of control structures forms an integral part of some of these cognitive complexity measures. In order to assign realistic cognitive weights to the basic control structures of a program or software component, psychological experiments must be conducted. The idea to assign cognitive weights to control structures is not new. In [11], McQuaid defined "inherent complexities" for Ada95 statements. However, McQuaid emphasizes that the complexity weights used in the study are entirely subjective and that no experiments were carried out.

The cognitive complexity measures proposed by [9, 10] are based on the basic control structure cognitive weight values provided by Shao and Wang in [6, 7]. Unfortunately, there are a number of issues with Shao and Wang’s experiments for determining the cognitive weights of basic control structures. Many of these issues are raised by Gruhn and Laue in [12]. Most disturbing among the issues raised by Gruhn and Laue is the fact that the layout of the experiments used by Shao and Wang in [6] have never been published.

Wang describes an experiment for calibrating cognitive weights for control structures in [13]. Undergraduate and graduate software engineering students were given ten Java code fragments, each representing one control structure. The students were required to read the code and answer questions about the value of a variable resulting from executing the code. The time each student took to read the program and provide the answer was then recorded.

The most obvious issue with Wang’s experiments in [13] is the fact that the code fragments used were not syntactically correct Java programs. This threatens the validity of the experiments since comprehension problems could result from the invalid syntax rather than from difficulty understanding the execution of the program. Also, the experiments used in [13] do not do anything to alleviate inconsistencies raised by the differing lengths of the code fragments. Furthermore, no information is given about the correctness of the answers provided by the experimental subjects. This suggests that no difference was made between the time needed to give a wrong answer and the time needed to give a right one. Psychological experiments to determine the cognitive weights of basic control structures in a programming language have to deal with these issues in order to have any hope of deriving realistic values. In [12], Gruhn and Laue specify some requirements for experimental settings to be used in determining the cognitive weights of basic
control structures. Some of these requirements form the basis for the experimental design presented in this paper.

Basic control structures are fundamental aspects of any program written in an imperative programming language. Previous work has provided a strong foundation for this project especially in the aspect of creating a mathematical model that can be used to derive an overall cognitive complexity measure of programs or software components. However, a survey of the aforementioned key publications seems to suggest that it is quite challenging to design an effective experimental process for coming up with some of the variables that drive the mathematical models.

This paper identifies and addresses the need for a sound experimental tool and framework that can be used to measure cognitive weights of control structures in currently existing imperative programming languages. In an attempt to meet this need, this paper addresses some of the shortcomings in the experimental layout of previous work. The cognitive weights derived using the experimental setup proposed in this paper can then form a suitable basis for an overall cognitive complexity measure of programs written in the imperative programming languages that have been experimented with. The overall cognitive complexity measure of such programs can then be determined using the definitions briefly discussed earlier and given in more detail in [6, 9].

5 General Experimental Approach

The data required to derive cognitive weights for basic control structures is acquired from experiments involving participants that have some programming experience. A post-activity survey was conducted to gather information about the subject’s education and experience. Fourteen (14) graduate students participated in the experiments. Five (5) participants reported having greater than five (5) years Java experience, one (1) participant reported having less than one year of experience, while eight (8) participants reported having between one (1) and five (5) years Java programming experience. Before the actual experiments begin, participants were given access to a short Java ‘refresher’ that presents some code snippets and their respective outputs.

The primary metric for determining code comprehension is the time taken for a participant to determine the output a snippet of code if it were to be executed. Response times for both correct and incorrect attempts were recorded. The amount of time it took for participants to provide predicted output for each control structure represented by a code snippet was used to determine the cognitive weight of that control structure. For this paper, the code snippets were written in Java.

A simple interactive web app has been designed and built as part of this project. This web app serves as the primary tool for administering experiments and collecting data as accurately as possible. The web app presents code snippets to participants and tracks how long it takes participants to visually trace the code execution and provide the correct value of some variable via input into a text field. A snapshot of the web app environment is given in Figure 1.
Since the time it takes to read through a piece of code is often directly proportional to the number of code lines, the recorded times are normalized to account for the varying length of code snippets. This normalization involves presenting participants with code snippets that represent the most basic control structure; a sequence. Each of the code snippets used for normalization has lengths that are equal to the lengths of code snippets used for the other more advanced control structures. The times recorded for sequences are used to normalize the times recorded for the other more advanced control structures.

At the end of the experiments, the recorded times for each of the control structures are normalized and averaged over the number of participants in the experiment. This average value is used to calculate the cognitive weights of the control structures.

The following basic code control structures were used in experiments to determine each of their cognitive weights:

<table>
<thead>
<tr>
<th>Category</th>
<th>Basic Control Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Sequence</td>
</tr>
<tr>
<td>Branch</td>
<td>if-else if-else</td>
</tr>
<tr>
<td>Iteration</td>
<td>Switch-Case</td>
</tr>
<tr>
<td></td>
<td>For loops</td>
</tr>
<tr>
<td></td>
<td>While loops</td>
</tr>
<tr>
<td></td>
<td>Do-While loops</td>
</tr>
<tr>
<td>Embedded Component</td>
<td>Recursion</td>
</tr>
</tbody>
</table>

Table 1: Basic code control structures to be used in experiments

6 Code Fragments

Participants are presented with code snippets that focus on a single control structure. The following are some code snippet selections that were used in the experiments. Subjects were required to predict the output of the code snippets if they were to be executed.
6.1 Sequence

```java
public class Activity1{
    public static void main(String[] args){
        ArrayList<Integer> scores = new ArrayList<Integer>();
        int firstScore;
        int secondScore;
        int result;
        scores.add(10);
        scores.add(20);
        scores.add(30);
        scores.add(40);
        scores.add(50);
        firstScore = scores[4];
        secondScore = scores[0];
        result = firstScore + secondScore;
        System.out.println(Integer.toString(result));
    }
}
```

Listing 1: Source code listing in Java for the most basic control structure; a sequence

Sequences like these are used to normalize for the varying lengths of code snippets for other more complex control structures.

6.2 Branching

During the experiments, code snippets for branching control structures are each presented multiple times with the value of $i$ varied in such a manner as to explore all branches.

6.2.1 If-else if-else

```java
public class Activity2{
    public static void main(String[] args){
        int i = 32;
        if(i % 15 == 0){
            String result = "FizzBuzz";
        } else if(i % 3 == 0){
            String result = "Fizz";
        } else if(i % 5 == 0){
            String result = "Buzz";
        } else{
            String result = Integer.toString(i);
        }
        System.out.println(result);
    }
}
```

Listing 2: Source code listing in Java for the if-else if-else control structure

6.2.2 Switch-Case

```java
public class Activity3{
    public static void main(String[] args){
```
int i = 2;
switch(i){
    case 0:
        System.out.println("Buster");
        break;
    case 1:
        System.out.println("Dragon");
        break;
    case 2:
        System.out.println("Hodor");
        break;
}

Listing 3: Source code listing in Java for the Switch-Case control structure

6.3 Iterations

6.3.1 For Loops

public class Activity8{
    public static void main(String[] args){
        int i, j, result;
        for (i=0, j=0; i+j < 20; ++i, j+=i--){
            result = i + j;
        }
        System.out.println(Integer.toString(result));
    }
}

Listing 4: Source code listing in Java for For Loop control structure

6.3.2 While Loops

public class Activity4{
    public static void main(String[] args){
        int n = 10;
        int result = 5;
        while (n > 0){
            n--;
            result++;
        }
        System.out.println(Integer.toString(result));
    }
}

Listing 5: Source code listing in Java for While Loop control structure

6.3.3 Do-While Loops

public class Activity5{
    public static void main(String[] args){
        int n = 10;
        int result = 5;
        do {
            n++;
result --;
} while (n <= 20);
System.out.println(Integer.toString(result));
}

Listing 6: Source code listing in Java for Do-While Loop control structure

6.4 Embedded Components

6.4.1 Recursion

public class Activity6{
    public static int myfunc(int p, int q){
        if (q == 0) return p;
        else return myfunc(q, p % q);
    }

    public static void main(String[] args){
        int p = 20;
        int q = 10;
        int result;
        result = myfunc(p, q);
        System.out.println(Integer.toString(result));
    }
}

Listing 7: Source code listing in Java for Do-While Loop control structure

7 Results

The most basic control structure is a sequence. The length of each code sequence used in the experiments corresponds to the length of one or more code snippets representing other more complex control structures. In other words, each code snippet representing a control structure has a corresponding sequence of equal length. Table 2 shows median response times for code sequences of varying lengths. The lines of code that need to be read in order for a user to provide a response to a code snippet affects response times. In order to adjust for this dependency of response time on length, recorded response times for a given snippet are normalized by the median response time derived from a sequence of corresponding length.

<table>
<thead>
<tr>
<th>Sequence Length</th>
<th>Median Response Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 lines</td>
<td>10</td>
</tr>
<tr>
<td>8 lines</td>
<td>13</td>
</tr>
<tr>
<td>9 lines</td>
<td>15</td>
</tr>
<tr>
<td>10 lines</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 2: Sequences and Median Response Times

A participant’s response time, $t(C_i)$ for a particular code snippet $C_i$ is given
by:
\[ t(C_i) = \sum_{j=1}^{n_{\text{attempts}}} t(\text{att}_j) + t(\text{att}_{\text{correct}}) \]  
(11)

where \( n_{\text{attempts}} \) is the number of incorrect attempts a participant makes, \( t(\text{att}_j) \) is the time taken to provide the \( j^{th} \) incorrect attempt and \( t(\text{att}_{\text{correct}}) \) is the time taken to make a correct attempt. Note that each attempt has a response time ranging from 0-∞.

For each code snippet \( C_i \), the normalized response time across all users is calculated as:
\[ t_{\text{norm}}(C_i) = \frac{t_{\text{median}}(C_i)}{t_{\text{seq}}} \]  
(12)

where \( t_{\text{median}}(C_i) \) is the median response time across all participant responses for a particular code snippet and \( t_{\text{seq}} \) is the median response time for a sequence of equal length.

Control structures are represented by multiple code snippets. In our experiments, we used three code snippets for each control structure. The cognitive weight of a control structure is calculated as the average of all the normalized response times recorded for each code snippet representing the control structure. Therefore, the cognitive weight, \( CW_{\text{struct}} \) of a control structure is given as:
\[ CW_{\text{struct}} = \frac{N_{\text{snippets}}(\text{struct})}{\sum_{i=1}^{N_{\text{snippets}}(\text{struct})} t_{\text{norm}}(C_i)} \]  
(13)

where \( t_{\text{norm}}(C_i) \) is the normalized response time for each code snippet \( i \), representing the control structure as defined in (12) and \( N_{\text{snippets}}(\text{struct}) \) is the number of code snippets for the control structure.

Table 3 shows the values derived from the aforementioned definitions.

Table 4 shows the derived cognitive weight values for each of the control structures used in our experiments.

8 Threats to Validity

Participants’ experience level in the particular programming language used in the experiments may affect their response times. Also, the difficulty of code snippet selections for each control structure may also have an effect. An attempt has been made to mitigate the effect of code snippet selections by using multiple code snippets of varying difficulty for each of the different control structures. The response times are averaged over the different snippets used for a particular control structure.

There is a risk of priming participants with similar lines of code and variable names across code snippets. This may affect participant’s response times as they get familiar with the code snippets used in the experiment. To mitigate this, code snippets have been made as different as possible from one another.
<table>
<thead>
<tr>
<th>Category</th>
<th>Control Structure</th>
<th>Snippet #</th>
<th>Median Response Time (Seconds)</th>
<th>t_{norm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branching</td>
<td>if-else if[else]</td>
<td>1 (9 lines)</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (9 lines)</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (9 lines)</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>16</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td>switch-case</td>
<td></td>
<td>1 (9 lines)</td>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (9 lines)</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (9 lines)</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>21</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td>Iteration</td>
<td>for loops</td>
<td>1 (5 lines)</td>
<td>58</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (5 lines)</td>
<td>57</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (5 lines)</td>
<td>47</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>54</strong></td>
<td><strong>11</strong></td>
</tr>
<tr>
<td></td>
<td>do-while loops</td>
<td>1 (8 lines)</td>
<td>92</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (8 lines)</td>
<td>69</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (8 lines)</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>81</strong></td>
<td><strong>10</strong></td>
</tr>
<tr>
<td></td>
<td>while loops</td>
<td>1 (8 lines)</td>
<td>57</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (8 lines)</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (8 lines)</td>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>46</strong></td>
<td><strong>6</strong></td>
</tr>
<tr>
<td>Embedded Component</td>
<td>Recursion</td>
<td>1 (9 lines)</td>
<td>69</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (10 lines)</td>
<td>83</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (10 lines)</td>
<td>48</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td><strong>Averages</strong></td>
<td></td>
<td><strong>67</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>

Table 3: Analysis of Collected Data

9 Evaluation

Table 5 shows a comparison between cognitive weights for basic control structures derived in previous works and those derived in this paper.

Previous work has claimed that cognitive complexity based on cognitive weights is a more robust measure of code complexity than lines of code (LOC) [6]. This claim is corroborated by showing how algorithm implementations in different languages of the same type and abstraction level can vary significantly in length while the cognitive complexity according to complexity measures that integrate the cognitive weights of control structures essentially remains the same. Even further, it is possible that different programmers using the same language to implement a particular algorithm will write programs that vary noticeably in length while the cognitive complexity based on cognitive weights of control structures may not
Control Structure | Cognitive Weight
--- | ---
if-else if-else | 2
switch-case | 2
for loops | 11
do-while loops | 10
while loops | 6
Recursion | 7

Table 4: Cognitive Weight Values for Control Structures

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Branching</td>
<td>if-else if-else</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>switch-case</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Iteration</td>
<td>for loops</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>do-while loops</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>while loops</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Embedded Component</td>
<td>Recursion</td>
<td>3</td>
<td>11</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Comparison of Cognitive Weight Values

vary as widely.

Software cognitive complexity measures that utilize cognitive weight values of control structures can be evaluated using Weyuker properties as shown in [14].

10 Conclusions and Future Work

This paper proposes an experimental design and tool for determining the cognitive weights of code control structures. Most importantly, it attempts to address some of the problems that have been identified in previous works’ experiments. Participants were required to predict the output of 22 code snippets representing various code control structures and their response times were recorded. The data collected from the experiments was analyzed and used to derive cognitive weight values for some common code control structures.

It may be interesting to gather eyetracking data in addition to response times. Participants’ eye movements may provide insight into the complexity of the code snippets they are working on. Also, these experiments need to be extended to involve many more participants, more control structures such as exceptions and other programming paradigms such as functional and logic programming.

The tool presented in this paper may be adapted for various purposes. It could be used in introductory programming classes to determine areas of basic programming in which students are currently experiencing difficulty. Also, the tool may be extended to support user-centric programming language design by providing a means of collecting data from potential users of a new programming
language with regard to syntax and semantic design issues.

11 Acknowledgments

This paper was written with valuable contributions from Natalie Parde, Curtis Chambers and Sam Cook.

References


