ON MEASURING SOFTWARE COMPLEXITY

Adrian COSTEA[1]
PhD, University Lecturer
Academy of Economic Studies, Bucharest, Romania

E-mail: acostea74@yahoo.com **Web page:** http://www.abo.fi/~acostea

Abstract: *In this paper we measure one internal measure of software products, namely software complexity. We present one method to determine the software complexity proposed in literature and we try to validate the method empirically using 10 small programs (the first five are written in Pascal and the last five in C++). We have obtained results which are intuitively correct, that is we have obtained higher values for average structural complexity and total complexity for the programs which "look" more complex than the others, not only in terms of length of program but also in terms of the contained structures.*

Key words: *software; complexity; measurement*

1. Introduction

Software quality is the degree to which software possesses a desired combination of attributes such as maintainability, testability, reusability, complexity, reliability, interoperability, etc. (IEEE, 1992). In other words, quality of software products can be seen as an indirect measure and is a weighted combination of different software attributes which can be directly measured. Moreover, many practitioners believe that there is a direct relationship between internal and external software product attributes. For example, a lower software complexity (seen here as a structural complexity) could lead to a greater software reliability (Fenton & Pfleeger, 1997).

Measuring complexity of software products was, and still is, a widely distributed research subject. The scope of studying it was to control the levels of the external attributes of software via internal attributes, like complexity is. The most well-known internal attribute is software length. Another is complexity. While in the case of length is a quite well defined consensus about the ways the length should be measured, in the case of complexity is still a lot of confusion.

It is not wrong to say that there is a relationship between complexity and the length of the program. But, all authors agree that when measuring complexity one should take into account something different from length and length at the same time. This approach was followed in Törn *et al*. (1999) where a new measure of software complexity called *structural complexity* is derived.

In the literature there are several measures of complexity, the most used ones being:

- *length* defined as the number of lines of codes and

- *McCabe's cyclomatic complexity* which measures something else than just length:

$$v = e - n + 2 = 1 + d$$

where v is the cyclomatic complexity of the flowgraph *f*, *e* is the number of edges, *n* the number of nodes, and *d* is the number of predicate nodes of *f*.

The longer the program is, the more predicate nodes it has. This leads to normal conclusion that McCabe's cyclomatic number is strongly correlated with length. The problem with McCabe's cyclomatic number is that is not a "good" measure of complexity, since smaller programs (in terms of lines of code) are much more complex in terms of their intrinsic functions. In order to eliminate the correlation, some authors proposed other measures such as *complexity density* defined as the ratio of cyclomatic complexity to thousand of lines of code (Gil & Kemerer, 1991).

McCabe (1976) proposed a derived complexity measure called *essential complexity measure* (ev): $ev = v - m$ where v is the cyclomatic number and *m* represents the number of sub-flowgraphs of *f* that are D-structured primes (Fenton & Pfleeger, 1997, p. 288).

Bache (1990) proposed a series of axioms and a number of measures that satisfy the axioms, the *VINAP measures*, to characterize the software complexity. Woodward *et al*. (1979) proposed the so-called *knot measure* that is defined as the total number of points at which control-flow lines cross.

When measuring software complexity we have to be very cautious on which metrics we use. The authors in Törn *et al*. (1999) state that one can obtain wrong result if he/she compares two different programs using one complexity measure and other two using another one. Another problem raised by the authors is that of establishing acceptable axioms for complexity measures. The failure to realize the existence of different views about complexity leads to conflicting axioms.

Next we present an overview of the software complexity model proposed in Törn *et al*. (1999), and then, we apply this methodology on some example programs in order to empirically validate it.

Methodology

The model to calculate the total complexity of a piece of software (*p*) is as follows:

$$e(p) = l(p)c(p)$$

where *e(p)* is the total complexity, *l(p)* is the length of the software, and *c(p)* is the average structural complexity. For a collection of software units $P = \{p_1, p_2, ..., p_n\}$ we calculate *c(P)* as the average of the individual units complexity:

$$c(P) = c(p_1, p_2, ..., p_n) = \frac{\sum_{i=1}^{n} l(p_i)c(p_i)}{\sum_{i=1}^{n} l(p_i)}$$

The individual unit lengths and total complexities are additive. So, if we add them we obtain the length of the collection (*l(P)*) and, respectively, the total complexity of the collection (*e(P)*).

In *Törn et al*. (1999) the authors use the above equations for the software collections and define new formulas that use some constants. The constants are different from one control structure to the other. Next we give the three formulas (sequence, choice, iteration) for ***average structural complexity*** using these constants:

$c(p_1; p_2;...; p_n) = c_s c(p_1, p_2,..., p_n)$ - sequence

$c(if) = c_{if} c(b, p, q)$ - choice: "if *b* then *p* else *q*"

$c(while) = c_{do} c(b, p)$ - iteration: "while *b* do *p*"

In general, when applying the model we consider $c_s = 1.1 < c_{if} = 1.3 < c_{do} = 1.5$ which is intuitive since we assign to the more complex structure a greater importance when calculating the complexity.

We can have the same reasoning (adding some constants) when we calculate the lengths of different control structures. For example: $l(if) = l_{if} + l(b, p, q)$. For simplification (when applying the methodology for our example programs) we will consider all these constants zero ($l_{if} = l_{do} = l_{go} = l_d = 0$).

Using these formulas the complexity of any program can be computed given the lengths and average complexities of the smallest parts (atoms): assignment statement, expressions, procedure calls and *goto*'s. *In our experiment we consider all these to have the value of 1 (as it is suggested in Törn et al. (1999))*.

The unique feature of the model resides in the fact that no other complexity model found in literature has such a two dimensional structure in representing the complexity. Also, the theoretical properties of the model that cover unstructuredness, sequencing, nesting, modularization and layout are intuitively correct. In order to be able to apply the methodology we have to write the programs in "node representation". Using this representation, decision nodes, assignment statements and *goto* nodes are given as: (*b* l_a c_a), (*n* l_a c_a), and (*go* l_a c_a) respectively, where l_a and c_a are the length and complexity of the atoms.

In the next section we apply this methodology on some example programs and try to validate it empirically.

Results

We start the empirical evaluation by testing the complexity of some basic structures (p) and changes of the basic structures (p'):

Sequential structure:

Let p = {a;b}, where *a, b* are simple assignment statements. Then this can be written using the "node notation" as: (S(n 1 1)(n 1 1)) = (n 2 1.1) =>l = 2; c = 1.1 and e = 2.2.

Now let p' = {a;b;c}, where *a, b, c* are simple statements (assignment statements or defining statements). Then, in node notation the structure will be (s(n11)(n11)(n11)) = (n 3 1.1*(1+1+1)/3) = (n 3 1.1) => l = 3, c =1.1 and e = 3.3.

Conclusion: p' is more complex than p. It is obvious that the average complexity or the complexity density is equal for p and p', since both structures consists only of program nodes. But the total complexity of p' is greater than the total complexity of p.

Choice structure (IF a then p else q):

Let p = (If a then b else c), where *a* is a decision node and *b, c* are simple statements (program nodes). In node notation this will be written as:

(if (b 1 1) (n 1 1) (n 1 1)) = (n 3 1.3*(1+1+1)/3) = (n 3 1.3) => l = 3, c = 1.3 and e = 3.9.

Let p'=(If (a and b) then c else d), where *a, b* are decision nodes (Boolean expressions) and *c, d* are program nodes. Using node notation:

(if (b 2 1) (n 1 1) (n 1 1)) = (n 4 1.3) =>l = 4, c = 1.3 and e = 5.2.

Now let p'' = (If a then (b and c) else d), where *a* is a decision node and *b, c, d* are program nodes. Using node notation:

(if (b 1 1) (s(n 1 1) (n 1 1))(n 1 1)) = (if (b 1 1) (n 2 1.1) (n 1 1)) = (n 4 1.3*(2+2.2)/4) =

(n 4 1.365) => l = 4, e = 1.365 and e = 5.46.

Conclusion: p'' is more complex than p', which is more complex than p. Here the average complexity or complexity density is the same for p and p'. It is right because the both structures p and p' are basic, with the only difference that the decision node in p' is of length 2. In the p'' structure is included a sequential structure, which has the average complexity 1.1. This will increase the average complexity of *if* structure, and implicitly the total complexity.

Iteration structure (Do-while)

Let p = (While a do b), where *a* is a decision node and *b* is a program node. In node notation this is written as: (do (b 1 1) (n 1 1)) = (n 2 1.5). That is, the complexity density c=1.5 and the overall complexity e=3, and the length is l = 2.

Now let p' = (While a do (b and c)). In node notation, the structure will be:

(do (b 1 1) (s(n 1 1) (n 1 1))). This will be written further on as:

(do (b 1 1) (n 2 1.1)) = (n 3 1.5*(1+2.2)/3) = (n 3 1.6). This means that the average complexity (complexity density) is c=1.6, the length of the structure is l=3, and the overall complexity is e=4.8.

Conclusion: p' is more complex than p. The complexity density of p' is greater than that of p, and also the overall complexity of p'(e'=4.8) is greater that that of p (e=3).

Then we collected 10 programs for which we computed the values (l, c, e). The programs and the calculations are:

1st Program

program ex1;
var x: integer;
procedure p(y:integer);
begin
writeln(Y:3);
if y>3 then
begin
write('123');
writeln;
end
end;
begin
x:=3;
while x<=5 do
begin
p(x);
x:=x+1
end;
end.
	(s(n 1 1)
	(n 1 1)
	(s(n 1 1)
	(do(b 1 1)
	(s(n 1 1)
	(if(b 1 1)
	(s(n 1 1)
	(n 1 1)
	)
	)
	(n 1 1)
	)
	)
	)
	)

The average structural complexity ***c = 1.947***.
The program length ***l = 9***
Overall complexity ***e = 17.52***.

2nd Program

program ex2;
var counter: integer;
begin
counter:=1;
while counter<20 do
begin
write('We are inthe loop, wainting');
write('for the counter to reach 20. It is', counter:4);
writeln;
counter:=counter+2;
end;
end.
	(s(n 1 1)
	(n 1 1)
	(s(n 1 1)
	(do(b 1 1)
	(s(n 1 1)
	(n 1 1)
	(n 1 1)
	(n 1 1)
	)
	)
	)
	)

The average structural complexity ***c = 1.65***.
The program length ***l = 8***
Overall complexity ***e = 13.211***

3rd Program

program ex3;
const string_size=30;
type low_set=set of 'a'..'z';
var data_set: low_set ;
storage: string[string_size];
index: 1..string_size;
print_group:string[26];
begin
data_set:=[];
print_group:='';
	(s(n 1 1)
	(n 1 1)
	(n 1 1)
	(n 1 1)
	(n 1 1)
	(n 1 1)
	(n 1 1)
	(s(n 1 1)
	(n 1 1)
	(n 1 1)

```
storage:='This is a set for test';
index:=1;
while index<=length(storage) do begin
 if storage[index] in ['a'..'z'] then
  if storage[index] in data_set then
    writeln(index:4, ' ',storage[index],' is already in the set')
    else begin
    data_set:=data_set+[storage[index]];
    print_group:=print_group+storage[index];
    writeln(index:4,",storage[index],' added to group, complete
group= ',print_group);
    end;
  else
  writeln(index:4,'   ',storage[index],' is not a lower case letter');
index:=index+1;
end;
end.
```

```
(n 1 1)
(do(b 1 1)
  (s(if(b 1 1)
      (if(b 1 1)
        (n 1 1)
        (s(n 1 1)
          (n 1 1)
          (n 1 1)
         )
        )
        (n 1 1)
      )
      (n 1 1)
    )
  )
 )
)
```

The average structural complexity ***c = 1.97***
The program length ***l = 20***
Overall complexity ***e = 39.425***

4th Program

```
program ex4;
var index, count: integer;
checkerboard: array[1..8]of array[1..8] of integer;
value: array[1..8,1..8] of integer;
begin
index:=1;
while index<=8 do begin
  count:=1;
  while count<=8 do begin
    checkerboard[index,count]:=index+3*count;
    value[index,count]:=index+2*checkerboard[index,count];
    count:=count+1;
    end;
  index:=index+1;
end;
writeln('Output of checkerboard');
writeln;
index:=1;
while index<=8 do begin
    count:=1;
    while count<=8 do begin
       write(checkerboard[index,count]:7);
       count:=count+1;
    end;
    writeln;
    index:=index+1;
    end;
value[3,5]:=-1;
value[3,6]:=3;
value[3,7]:=2;
count:=1;
while count<=3 do begin
writeln;
```

```
(s(n 1 1)
  (n 1 1)
  (n 1 1)
  (n 1 1)
  (s(n 1 1)
    (do(b 1 1)
      (s(n 1 1)
        (do(b 1 1)
          (s(n 1 1)
            (n 1 1)
            (n 1 1)
           )
        )
        (n 1 1)
       )
      )
      (n 1 1)
    )
  )
  (n 1 1)
  (n 1 1)
  (n 1 1)
  (do(b 1 1)
    (s(n 1 1)
      (do(b 1 1)
        (s(n 1 1)
          (n 1 1)
         )
       )
       (n 1 1)
       (n 1 1)
     )
   )
   (n 1 1)
   (n 1 1)
   (n 1 1)
```

```
count:=count+1;
end;
writeln('output of value');
writeln;
count:=1;
while count<=8 do begin
   index:=1;
   while index<=8 do begin
      write(value[count,index]:7);
      index:=index+1;
   end;
   writeln;
   count:=count+1;
   end;
end.
```

```
      (n 1 1)
      (do(b 1 1)
        (s(n 1 1)
          (n 1 1)
        )
      )
      (n 1 1)
      (n 1 1)
      (n 1 1)
      (do(b 1 1)
        (s(n 1 1)
          (do(b 1 1)
            (s(n 1 1)
              (n 1 1)
            )
          )
          (n 1 1)
          (n 1 1)
        )
      )
    )
  )
```

The average structural complexity ***c = 1.97***

The program length ***l = 39***, Overall complexity ***e = 76.98***

5th Program

```
Program ex5;
var index, count: integer;
checkerboard: array[1..8]of array[1..8] of integer;
value: array[1..8,1..8] of integer;
begin
index:=1;
while index<=8 do begin
   count:=1;
   while count<=8 do begin
      checkerboard[index,count]:=index+3*count;
value[index,count]:=index+2*checkerboard[index,count];
      count:=count+1;
   end;
   index:=index+1;
end;
writeln('Output of checkerboard');
writeln;
index:=1;
while index<=8 do begin
   count:=1;
   while count<=8 do begin
      write(checkerboard[index,count]:7);
      count:=count+1;
   end;
   writeln;
   index:=index+1;
   end;
count:=1;
while count<=3 do begin
writeln;
count:=count+1;
```

```
(s(n 1 1)
  (n 1 1)
  (n 1 1)
  (n 1 1)
  (s(n 1 1)
    (do(b 1 1)
      (s(n 1 1)
        (do(b 1 1)
          (s(n 1 1)
            (n 1 1)
            (n 1 1)
          )
        )
        (n 1 1)
      )
    )
    (n 1 1)
    (n 1 1)
    (n 1 1)
    (do(b 1 1)
      (s(n 1 1)
        (do(b 1 1)
          (s(n 1 1)
            (n 1 1)
          )
        )
        (n 1 1)
        (n 1 1)
      )
    )
  )
 )
 (n 1 1)
```

```
end;
writeln('output of value');
writeln;
count:=1;
while count<=8 do begin
   index:=1;
   while index<=8 do begin
      write(value[count,index]:7);
      index:=index+1;
   end;
   writeln;
   count:=count+1;
   end;
end.
```

```
    (do(b 1 1)
      (s(n 1 1)
        (n 1 1)
      )
    )
    (n 1 1)
    (n 1 1)
    (n 1 1)
    (do(b 1 1)
      (s(n 1 1)
        (do(b 1 1)
          (s(n 1 1)
            (n 1 1)
          )
        )
        (n 1 1)
        (n 1 1)
      )
    )
  )
)
```

The average structural complexity ***c = 2.27***
The program length ***l = 36***
Overall complexity ***e = 81.81***

As a first observation, when comparing the results obtained for the 4th program with those for the 5th program, we can state that even the length of the program 5 is lower that the length of program 4 (36<39), the average complexity of program 5 is greater than that of program 4 (2.27>1.97). This is true and intuitively explained by the fact that the density of complex structures in program 5 is greater than that in program 4. The difference in size (39-36) is explained by the three assignment statements (value[3,5]:=-1; value[3,6]:=3; value[3,7]:=2;) which appear only in program 4 and have a lower value for complexity.

This shows that taking just length as a complexity measure is not a correct way to evaluate the quality of a software product of being complex or not. Even though the length is high, it is possible that the program is easily readable and easy to maintain, if it consists of few complex and nested structures and much many simple statements.

6th Program

```
void merge(apvector<int> &a, int first, int mid, int last) {
        int aPtr=first, bPtr=mid+1, cPtr=first;
        int total=last-first+1, loop;
        bool doneA = false, doneB = false;
        apvector<int> c(a.length());
        for (loop=1; loop<=total; loop++) {
                if (doneA) {
                c[cPtr] = a[bPtr];
                        bPtr++;
        }
                else
                        if (doneB) {
                        c[cPtr] = a[aPtr];
```

```
(s (n 1 1) // aPtr = …
   (n 1 1) // bPtr = …
   (n 1 1) // cPtr = …
   (n 1 1) // total = …
   (n 1 1) //doneA = false
   (n 1 1) // doneB = false
   (n 1 1) // constructor call
   (n 1 1) // loop = 1
   (do (b 1 1)
      (s (if (b 1 1)
            (s (n 1 1)
               (n 1 1)
            )
```

```
                        aPtr++;
                    }
                        else
                    if (a[aPtr] < a[bPtr]) {
                       c[cPtr] = a[aPtr];
                            aPtr++;
                          }
                          else {
                           c[cPtr] = a[bPtr];
                             bPtr++;
                         }
                    cPtr++;
            if (aPtr > mid) doneA = true;
                    if (bPtr > last) doneB = true;
            }
          for (loop=first; loop<=last; loop++)
             a[loop] = c[loop];
}
```

```
                (if (b 1 1)
                    (s (n 1 1)
                       (n 1 1)
                    )
                    (if (b 1 1)
                        (s (n 1 1)
                           (n 1 1)
                        )
                        (s (n 1 1)
                           (n 1 1)
                        )
                    )
                )
            )
            (n 1 1)
            (if (b 1 1)
               (n 1 1)
            )
            (if (b 1 1)
               (n 1 1)
            )
            (n 1 1)
        ) // close do
        (n 1 1) // loop = first
        (do (b 1 1)
            (s (n 1 1)
               (n 1 1) // loop++
            )
        )
) // close s
```

The average structural complexity ***c = 2.3***.
The program length ***l = 30***.
Overall complexity ***e = 69***.

7th Program

```
void quickSort (apvector<int> &list, int first, int last){
        int g = first, h = last;
        int midIndex, dividingValue;
        midIndex = (first + last) / 2;
        dividingValue = list[midIndex];
        do {
           while (list[g] < dividingValue) g++;
           while (list[h] > dividingValue) h--;
           if (g <= h) {
             swap(list[g], list[h]);
                  g++;
                  h--;
           }
        } while (g < h);
        if (h > first) quickSort (list,first,h);
        if (g < last) quickSort (list,g,last);
}
```

```
(s (n 1 1)
    (n 1 1)
    (n 1 1)
    (n 1 1)
    (do (b 1 1)
        (s (do (b 1 1)
               (n 1 1)
           )
           (do (b 1 1)
               (n 1 1)
           )
           (if (b 1 1)
               (s (n 1 1)
                  (n 1 1)
                  (n 1 1)
               )
            ) // close if
          ) // close s
    )  // close do
) // close s
```

The average structural complexity ***c = 1.93***.
The program length ***l = 17***.
Overall complexity ***e = 32.8075***.

8th Program

```
void mergeSort(apvector<int> &list, int first, int last) {
        int mid;
        if (first == last)
                last++;
        else
            if (1 == last - first) {
                if (list[first] > list[last])
                    swap (list[first], list[last]);
             }
             else {
                  mid = (first+last) / 2;
                mergeSort (list, first, mid);
                mergeSort (list, mid+1, last);
                merge (list, first, mid, last);
             }
}
```

```
(if (b 1 1)
    (n 1 1)
    (if (b 1 1)
        (if (b 1 1)
            (n 1 1)
        )
        (s (n 1 1)
           (n 1 1)
           (n 1 1)
           (n 1 1)
        ) // close if
    ) // close if
)// close if
```

The average structural complexity ***c = 1.79***.
The program length ***l = 9***.
Overall complexity ***e = 16.12***.

9th Program

```
void insertionSort (apvector<int> &list) {
    int pos;
    for( int i=1; i<list.length(); ++i ) {
                pos = i;
        while( (pos>0) && ( list[pos-1]>list[pos] ) ) {
                        swap( list[pos-1], list[pos] );
                pos--;
        }
    }
}
```

```
(s (n 1 1) // i=1
    (do (b 2 1) // function call
        (s (n 1 1)
           (do (b 2 1)
               (s (n 1 1)
                  (n 1 1)
               )
           )
           (n 1 1) // ++i
        ) // close s
    ) // close do
) // close s
```

The average structural complexity ***c = 1.76***.
The program length ***l = 9***.
Overall complexity ***e = 15.83***.

10th Program

```
void screenOutput (const apvector<int> &nums) {
    cout << setiosflags( ios :: right );
    for( int x=0; x<nums.length(); ++x ) {
                if( x%12 == 0 )
                cout << endl;
                cout << setw(6) << nums[x] << " ";
    }
```

```
(s (n 1 1)
    (n 1 1) // x=0
    (do (b 2 1) // function call
        (s (if (b 2 1) // length = 2
                    // 1 from x%12
            // 1 from x%12 == 0
            (n 1 1)
```

}	) // close if
	(n 3 1) // display 3 times
	(n 1 1) // ++x
	) // close s
	) // close do
	) // close s

The average structural complexity ***c = 1.8035***.
The program length ***l = 11***.
Overall complexity ***e = 19.8385***.

Some of the last five programs are procedures which implement different kinds of sorting (quick sort, insertion, sort, merge sort). We have to mention that in the last five programs we did not take into consideration the variables declarations when calculating the complexity. Also when we transformed "for" structure in "do" structure we have followed the rule:

```
for (i=0; i<n; i++) equivalent with (n 1 1) // i = 0
                                    (do (b 1 1)
                                        ....
                                        (n 1 1) // i++
                                ) // close do
```

We have obtained results which are intuitively correct, that is we have obtained higher values for average structural complexity and total complexity for the programs which "look" more complex than the others, not only in terms of length of program but also in terms of the contained structures.

References

1. Bache R. **Graph Theory Models of Software**, PhD thesis, South Bank University, London, 1990.
2. Fenton N.E. and Pfleeger S.L. **Software Metrics - A Rigorous & Practical Approach**, International Thomson Computer Press, London 1997 (Second edition)
3. Gill G. K. and Kemerer C.F. **Cyclomatic Complexity Density and Software Maintenance**, IEEE Transanctions on Software Engineering, SE-17: 1284-1288, 1991
4. IEEE – IEEE Standard 1061-1992 **Standard for a Software Quality Metrics Methodology**, New York: Institute of Electrical and Electronics Engineers, 1992
5. McCabe T. **A Software Complexity Measure**, IEEE Transactions on Software Engineering SE-2(4): 308-320, 1976
6. Törn A., Andersson T. and Enholm K. **A Complexity Metrics Model For Software**, South African Computer Journal 24, November 1999, 40-48
7. Woodward M.R., Hennell M.A. and Hedley D. **A measure of control flow complexity in program text**, IEEE Transactions on Software Engineering, SE-5(1): 45-50, 1979

[1] Adrian COSTEA hold a PhD from Turku Centre for Computer Science
Research interests:
Data Mining Techniques for Decision Support
Financial Benchmarking
Economic/Financial Performance Classification Models
Economic/Financial/Process Variable Predictions
Further interests:
Software reliability models