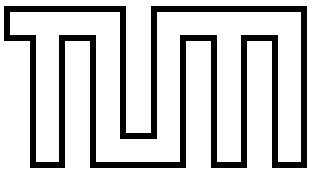


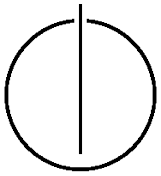
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Seminar Thesis

Sense and Nonsense of Software Complexity Metrics

Jona Neumeier



Contents

1 Introduction

Today, nearly everything is powered by software, resulting in large software projects with a great code basis. Since such enormous systems can be in place for several years or even decades, constant maintenance is required. Consequently, the original developer or new developers, spend a lot of time on maintaining the system. Maintaining a large software project can often times cost the same or even more than the original development [HMKD82]. For this and also for comparing purposes developers and researchers try to evaluate the complexity of a software project by calculating different metrics. Defining the complexity as the measure of how hard it is to fully comprehend the functionality of a program to be able to work and extend it [HMKD82]. A concrete correlation between the complexity of a program and the time needed to maintain this system can be observed. Researchers like McCabe in [McC76] or Halstead in [Hal77] were among the first ones to develop concrete metrics, that can be applied to software projects. Since then many more complexity metrics have been developed and used.

In this paper we want to give an overview over some of these metrics. We review their definition, as well as discussing some of their individual advantages and disadvantages. Followed by an in-depth review using and comparing the before explained metrics on several open source projects. The goal of this research is to provide an overview and examples of the metrics used to gain a better understanding of when and how to use these metrics.

1.1 Motivation

Our motivation for this research is manifold. Following is an outline of the most important points.

First and foremost, we want to give an overview of the wide variety of code metrics that are available today. Because of the important role software plays in our modern society, it is critical to be able to estimate the complexity for various software projects, which is why a lot of metrics and different metric categories were developed. However, when confronted with the task of evaluating the complexity of a software project it can be overwhelming to choose the right metric or multiple metrics for the job at hand. Categories for metrics for various scenarios, have been established, objected oriented metrics or size metrics are but two examples of categories for metrics. Chapter 2 will go into more depth with these categories and the metrics belonging to these categories.

Secondly, we want to evaluate how practical these metrics are for a real-world use case. We will calculate the metrics for a variety of open source software projects. We want to determine how difficult it is to calculate each metric as well as the message behind each one.

Finally, complexity metrics can be used to define how maintainable a software project is. The maintenance of a project can take up the most time during the software's lifecycle. Therefore it is critically to measure how maintainable a system is. To be able to maintain a system one has to first comprehend it, which is easier if the program is less complex. For this reason complexity metrics can be used to measure how maintainable a system is. We want to investigate how good the complexity metrics are for this purpose.

1.2 Organization

The remaining part of this paper is structured as follows: The next chapter will introduce all metrics used in our research. Grouped into four categories, we will define every metric. After that we will regard the metrics under four different aspects: Informative value, programmer perceptions, maintainability and computation time. We will evaluate each metrics with regard to these aspects, by computing and comparing the metrics for different open source projects. Concluding we will give a brief summary of our conducted research.

2 Metric definitions

We now want to define some of the most relevant and most used metrics. We will categories these metrics in four different categories, depending on the aspect of the measurement the metric focuses on. We want to note that these are just a small subset of metrics that exist. Also, we will use a general definition of these metrics as most of them can be used in different granularities (focusing on methods, classes, files, a complete project, etc.). It has also been studied how and which metrics correlate with each other, which will not be discussed broadly in this research.

2.1 Size metrics

Size metrics are one of the most fundamental metrics, mostly concentrating on counting different aspects of a software component. Lines of code is arguably the most known metric and will be discussed in the following. Metrics such as the percentage of comments the number of functions/classes/files or the physical size of a software component are further examples.

2.1.1 Lines of code

Lines of Code (LOC) is one of the first and most basic software metrics [SCG17]. It is calculated by adding up the number of lines, in which source code is written, disregarding how many statements an individual line contains. [SCG17, GK91]. Generally speaking, a software code is regarded as more complex the more lines of code it has. Two variations of this metric exist [GK91]:

Counting commented lines. This variant counts the number of lines with source code, as well as the lines containing comments.

Not counting commented lines. Known as the non-commented source lines of code or source lines of code (NCSLOC/SLOC), this variant only counts the lines of actual source code without counting lines that are commented out.

2.1.2 Halstead metrics

In [Hal77] Halstead introduced a number of different metrics, which are now considered fundamentals. We will review the basic concept behind Halstead's findings and metrics.

The author defines the following constants for a software program:

- η_1: The number of unique operators.
- η_2: The number of unique operands.
- N_1: The total number of operators.
- N_2: The total number of operands.

With this, the program vocabulary η is defined as:

(2.1) $\eta = \eta_1 + \eta_2$

and the program length N as:

(2.2) $N = N_1 + N_2$

Using this, Halstead defined the length HL as:

(2.3) $HL = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

and the volume HV as:

(2.4) $HV = N \log_2 \eta$

Adding to that the Halstead difficulty HD is defined as follows:

(2.5) $HD = \frac{\eta_1}{2} * \frac{N_2}{\eta_1}$

It is used to calculate the Halstead effort HE:

(2.6) $HE = D * V$

2.2 Cyclomatic complexity, a control flow metric

Control flow metrics focus on how the different statements within a programs source code are structured, for example in what sequence are the *if* and *for* statements arranged. Introduced by Thomas McCabe in 1976 the cyclomatic complexity (CC) metric is one of the first and widest known software complexity metrics [McC76, Hum14]. The following explanations are in accordance with [McC76, Hum14]. To retrieve this metric the source code is looked at in the *control flow graph* representation. The cyclomatic number of a graph G, as McCabe calls it, is calculated as follows:

(2.7) $v(G) = e - n + p$

with:

- e: The number of edges.
- n: The number of vertices.
- p: The number of connected components.

within the graph G.

McCabe further concludes that in a strongly connected graph the cyclomatic number $v(G)$ is equal to the maximum number of linearly independent circuits. In structured programming languages the cyclomatic number is roughly the count of *if*-statements and *loops* plus 1. Listing 2.1 has a cyclomatic complexity of 3 for instance.

Listing 2.1: Simple example program calculating a sum.

```
1 // calculates the sum of two positive integers
2 public int sum (int a, int b){
3     if(a > 0){
4         if (b > 0){
5             return a + b;
6         }
7     }
8     return 0;
9 }
```

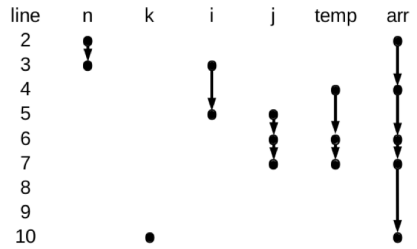
2.3 Data flow metrics

As the name explains, metrics in this category focus on how different data points behave within a programs lifecycle.

2.3.1 Lifespan

The first metric we want to look at is the Lifespan metric, as described in [Els76, KK12]. The central idea of this metric is that a program is more complex if there is a long distance, measured in lines of code, between different occurrences of one variable. The declaration of a variable is not counted in. The Lifespan for one method is the sum of all variable spans within this method. To retrieve the complexity of a whole program Lifespan is calculated as the average over all spans of all variables in a method.

In figure 2.1 we can see an example of this metric for listing 2.2. Variable $temp$ occurs in line 4, 6 and 7, therefore we have two spans in between these occurrences: $span_1(temp) = 2$ and $span_2(temp) = 1$.



(a)

variable	spans
arr	2, 2, 1, 3
n	1
i	2
temp	2, 1
j	1, 1
S	16

(b)

Figure 2.1: LifeSpan example for listing 2.2 [KK12]

Listing 2.2: Bico example [BF10].

```
1  int bico(int n, int k){
2      int[] arr = arrInit(n+1);
3      for(inti = 0; i <= n; i++){
4          int temp = arr[0];
5          for(int j = 1; j < i; j++){
6              arr[j] = arr[j] + temp;
7              temp = arr[j] - temp;
8          }
9      }
10     return arr[k];
11 }
```

2.3.2 Dependency degree

The dependency degree (DepDegree) measure is introduced by Beyer et al. in [BF10]. The authors use the control flow graph of a software application to define a *use-def* graph, which represents the dependencies from each use of a variable to its previous use or the initial definition. An example of a use-def Graph taken from [BF10] can be seen in figure 2.2, which represents the code fragment in listing 2.2.

The metric is then defined as the number of edges the use-def graph has. For figure 2.2 the Dependency Degree is equal to 28.

The authors argue that the more states a program has, the harder it is to understand it. Their metric tells the number of different predecessor operations we need to take into consideration when looking at a statement and therefore understanding the programs state.

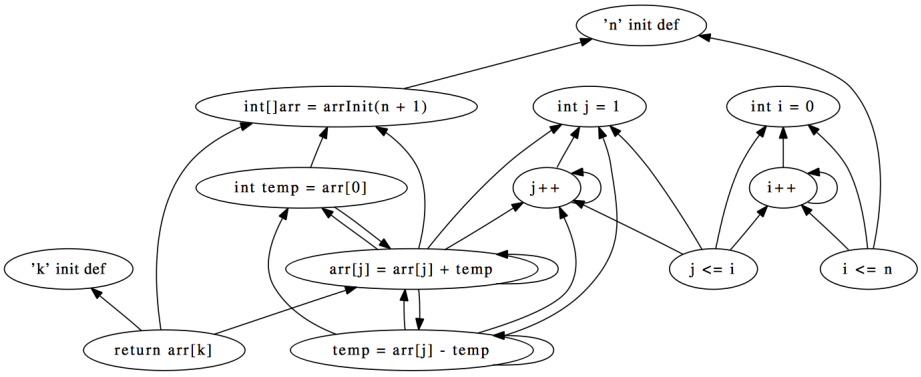


Figure 2.2: Def-Use graph for listing 2.2 [BF10]

2.4 Object orientated metrics

In accordance with Kemmerer et al. [CK94] we will now discuss metrics specifically for object orientated code design. Object orientation has been a major change from the traditional function orientated programming style. Object orientated design is a relational system with object elements and relationships between them. Since most modern software applications are written in object orientated code, a measure for this code design is needed. Kemmerer et al. outlines four major steps for building software in object orientated design, these steps also represent the major points code metrics should capture:

- Identification of classes (and objects)
- Identify the semantics of classes (and objects)
- Identify relationships between classes (and objects)
- Implementation of classes (and objects)

With this top down methodology a problem is first decomposed into its individual components (classes and objects). After that the exact use for each class is defined, followed by the relationships the classes and objects share. The last and most detailed step is the implementation itself for each class.

The authors introduced the following six metrics as a suite for a comprehensive review of object orientated design and code:

- Weighted methods per class (WMC)
- Depth of inheritance tree (DIT)
- Number of children (NOC)
- Coupling between objects (CBO)
- Response for a class (RFC)
- Lack of cohesion in methods (LCOM)

Following, we will review the definitions those metrics except 'weighted methods per class' and 'response for a class'.

2.4.1 Depth of inheritance tree

Inheritance is a common schema in object orientated design. With this one class can inherit another class, so that the subclass can use the methods of the superclass. This allows to reuse code in multiple classes without having to copy code. However, in complex projects inheritance can lead to complex dependency relationships between classes, especially if multiple inheritance is used. Depth of inheritance tree tries to measure these dependencies between classes. The depth of inheritance for one class is the maximum length of the node to the root node. Seen in figure 2.3 the classes B and C both inherit from class A. Class B inherits from both A and E, which makes up a multi-inheritance structure. This metrics value for class D is 2, the one for class B and C is 1.

2.4.2 Number of children

The next metric we want to discuss is the number of children for a class. Children are the immediate subclasses of a class. Like depth of inheritance tree this metric tries to classify the complexity by looking at inheritance. By observing how many children are going to inherit methods from the superclass one can observe how good the object orientated design is implemented, so that code does not have to be copied, but can be re-used. However, too many children also indicate bad design. In figure 2.3 NOC for class A is 2, whereas B and E have a NOC of 1 and C of 0.

2.4.3 Coupling between objects

A class is coupled to another class if a class uses a method or a variable of the other class. Coupling between objects (CBO) counts the number of classes that are coupled, via the use of a variable or method, to one specific class. CBO represents how modular a project is as well as how good a class is encapsulated.

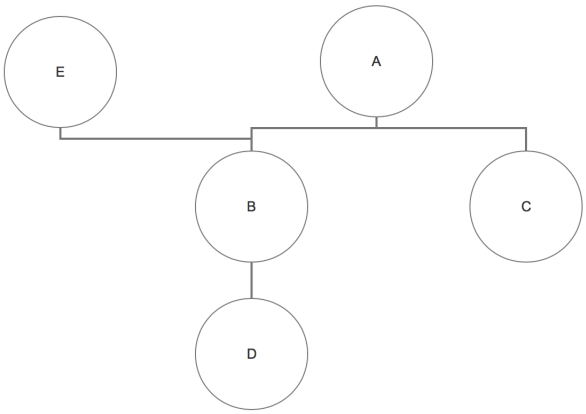


Figure 2.3: Example inhertiance tree

2.4.4 Lack of cohesion in methods

Lack of cohesion in methods analyzes how many methods in a class use the same variables of this class and how many methods are the only ones to use certain variables. Therefore making a prediction about how similar the methods in one class are. The formal definition is:

(2.8) $LCOM = |P| - |Q|,$ if $|P| > |Q|, 0$ otherwise

with:

- $M_1, M_2, ..., M_n$ the methods of a class C
- $I_j =$ set of instance variables used by method M_i
- $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$
- $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$

The following example is taken from [CK94]:

"Consider a class C with three methods M_1, M_2, and M_3. Let $\{I_1\} = \{a, b, c, d, e\}$ and $\{I_2\} = \{a, b, e\}$ and $\{I_3\} = \{x, y, z\}$. $\{I_1\} \cap \{I_2\}$ is nonempty, but $\{I_1\} \cap \{I_3\}$ and $\{I_2\} \cap \{I_3\}$ are null sets. LCOM is the (number of null intersections - number of nonempty intersections), which in this case is 1."

High cohesiveness in methods is an indicator for good encapsulation. Low cohesion in methods of a class increases the complexity of the project and indicate, that the class should be split into more classes.

2.5 Software maintainability index, a hybrid metric

Hybrid metrics, combine multiple different metrics to one, trying to factor out the faults from each individual metric. An example for this category of metrics is the software maintainability index, which will be discussed in the following part of this paper.

The software maintainability index (MI) is calculated as follows [OH92, Nie16]:

(2.9) $MI = 171 - 5,2 * \ln avgHV - 0,23 * avgCC - 16,2 * \ln avgLOC + 50 * \sin \sqrt{2,4 * perCM}$

with:

- HV: Halstead Volume
- CC: Cyclomatic Complexity
- LOC: Lines of Code
- $perCM$: Percentage of commented lines.

It tries to produce a single number indicating how difficult it is to maintain an existing software project [OH92, Nie16]. It can be easily seen that this metric combines previously mentioned metrics, from different categories. Unlike all other metrics discussed here a higher score for the MI metric is better than a smaller one.

3 Discussion

In the following we want to review the previously defined metrics according to different points of criteria. The goal is to show the strengths and weaknesses of the metrics, with examples from literature and open source projects. Weyuker in [Wey88] defines eight properties for evaluating software complexity metrics. Although we will not use all eight defined properties, it is worth mentioning the following ones.

- A complexity measure should not rank all programs equally complex and should not only have small set of predefined complexity classes (too broad).
- A complexity measure should not rank all programs in their own complexity class (too fine).
- Two programs can compute the exact same functionality, but the details of implementation define the complexity.

With these in mind we want to study the metrics under the following aspects:

- Informative value.
  Here we want to evaluate what the central message of the metric is. Additionally we will provide examples from open source projects, as well as points of criticism with corresponding examples.
- Programmer perceptions.
  Human beings are still the ones to develop and maintain software projects, therefore it is very important of how useful and intuitive they think a measurement is.
- Metrics and maintainability.
  One of the most often occurring scenarios a complexity metric is used for is to rate how hard it will be to maintain a software project. For this reason, we want to investigate the value of the individual metrics with regards to that.
- Computation difficulty.
  The best metric is worthless if it is too hard to compute, which is why we will briefly discuss the computational cost for each metric.

3.1 Informative value

We start our study by discussing the informative value, advantages and disadvantages of each metric.

Size metrics. As stated before, the LOC metric is one of the most rudimentary metrics and also one of the most often used ones [SCG17, GK91]. However it also has disadvantages: Every line of code, however long or complex, has the same weight on the result [SCG17]. Furthermore, an unexperienced programmer might need more lines of code to achieve a task an experienced programmer would have achieved in fewer lines of code. Therefore the metric can be distorted. As an example we want to use the django[1] and the linux[2] open source project. As of this writing the django project currently has a total of 319.000 LOC, from which 223.500 are SLOC[3]. The diversity for different files ranges from 3 SLOC for the main file to about 1.400 SLOC for a single file. In comparison to that the linux kernel has about $25.000.000$ lines of code [wik]. From this alone we can clearly see, that the linux project is by far a much bigger project than the django project. However, looking into the django project, most files have about 60 to 300 LOC, which makes it very hard to label those files in terms of complexity by looking at the lines of code alone. For example, the file *utils.py* in the db directory of django has about 212 SLOC, but is mostly made up from exception declarations, getter and setter methods and therefore relatively easy to understand for a programmer. The file *loader.py* in the *db/migrations* path has about 198 SLOC, so fairly similar

1 https://github.com/django/django
2 https://github.com/torvalds/linux
3 Obtained with the Teamscale tool (https://www.cqse.eu/en/products/teamscale/landing/)

to the utils-file. But this file is arguably much more harder to understand, as it has much more nestings and much longer methods. This all is not considered by the LOC metric.

The next set of metrics we want to look at are the measurements proposed by Halstead. With a python tool called *radon*[1] we calculated the averages for the Halstead length to $61,303$, the volume to $95,908$, the difficulty to 2.143 and the effort to $205,5177$ for the django project. The following values we found as the maximum values in the file *db/models/sql/compiler.py*. For the length $4189,892$, the volume $7932,471$, the difficulty $10,529$ and the effort $83521,2484$. We see that in an rather small project such as the django project, compared to the linux kernel the Halstead values can differ drastically. Gray in his Ph.D. thesis computed the Halstead volume and effort for the sum of all functions inside the linux kernel with the following result. The volume rose from about $4M$ in the 2.00.x kernel series to about $20M$ in the 2.04.x kernel series. Similar to that the Halstead effort rose from about $120,0$ to $420,0$. This shows a rather drastically growth in size between these kernel series, which is also represented in the LOC in these kernels. Different to the LOC metric Halsteads metrics to not count every line of code as equally complex.

All together, we can argue, that the size metrics do not provide specifics about the complexity, but the quantitative indicator these metrics give are a good estimator. As the general rule applies that a larger code basis is more complex than a small one.

Control flow metrics (cyclomatic complexity). As described in [Hum14] we see two major problems with this metric: In his original publication McCabe recommends seeing a conjunction in a conditional statement as two separate branches, which results to add 2 to the overall metrics score. Despite McCabes recommendation many modern source code analysis tools count the conjunction as one branch, therefore adding not as much to the overall score as in the first variant. This results in two different metric scores for one identical source code which is not desirable and has to be mitigated when comparing software applications or different components. Furthermore, when looking at a switch statement with many cases for instance, we have a high cyclomatic complexity as a switch statement is nothing else than many conditional statements. Yet in most cases one switch statement is very easy to comprehend for a human developer as the conditional logic is clearly structured. Listing 3.1 and 3.2 show a compelling example of two methods with the same cyclomatic complexity. However we argue that nearly every programmer would see the second example as far more complex and harder to understand than the first one.

Again looking at the django project, we used the radon tool to compute the cyclomatic complexity for every method and every class in the source code, disregarding the *test* directory. The average complexity for this project is $2,81$, which is a really low complexity. The method with the highest complexity we found is the *BaseDatabaseSchemaEditor._alter_field* method under *db/backends/base/schema.py* with $CC = 84$, the lowest score for some methods was 1.

The authors of [JMF12] investigated high CC functions within the linux kernel. They found the highest functions to have a CC of 112 to 587 most of these functions originated from the drivers part of the kernel (version 2.6.37.5). Most functions in the kernel had a CC between 50 and 100. They concluded that often times complex functions are justified, as the alternative would be to have a large number of very small functions, which is arguably not much better for the overall complexity. McCabes cyclomatic complexity also does not distinguish between a program with high computational cost and one which has little computational cost, given that both programs have the same decision structure [Wey88].

Nearly all of our above-mentioned sources agree that the CC does not provide much additional information to the LOC metric, to which is it highly correlated. We think that the metric can be used as an indicator: The more loops and conditional statements a function is composed of the more complex it is. Nonetheless, this metric clearly has disadvantages and inconsistencies and should therefore never be used as a standalone measure.

Listing 3.1: Sample method one with $CC = 5$ [Hum14].

```
1 String getWeight(int i) {
2     if (i <= 0) {
3             return "no weight";
4     }
5     if (i < 10) {
6             return "light";
7     }
8     if (i < 20) {
```

1 https://pypi.python.org/pypi/radon

```
9               return "medium";
10      }
11      if (i < 30) {
12              return "heavy";
13      }
14      return "very␣heavy";
15  }
```

Listing 3.2: Sample method two with $CC = 5$ [Hum14].

```
1   int sumOfNonPrimes(int limit) {
2       int sum = 0;
3       OUTER: for (int i = 0; i < limit; ++i) {
4               if (i <= 2) {
5                       continue;
6               }
7               for (int j = 2; j < i; ++j) {
8                       if (i % j == 0) {
9                               continue OUTER;
10                      }
11              }
12              sum += i;
13      }
14      return sum;
15  }
```

Data flow metrics. In [Els76] 120 PL/I programs are evaluated with the Lifespan metric. 13 % of all spans of variables exceeded 100 LOC. If a program has many large spans it may be an indicator that it is too long and should be more modularized, to bring down the LOC between variables, by clustering similar code together in on module/class. One point of criticism are constants. Often times programs declare specific constants at the very beginning or even in a separate file/class. These are then used throughout the program, which can lead to a high lifespan metric value. Additionally the lifespan indirectly measures the LOC metric, therefore it is closely correlated to that [HMKD82]. This measure can help to understand how readable and comprehendible a program is. It is harder to memorize all variables, if the spans between the usage are large and especially if this is true for many variables. A huge lifespan metric therefore indicates higher complexity.

Beyer et al. presented three different example functions each with two different implementation variants. Table 3.1 represents their findings for DepDegree compared to the LOC and CC metric. The results clearly show that there can be a difference in the DepDegree measure even though the LOC and CC stay exactly the same. As the table suggests the second variation of the implementation of the function seems to be easier to understand in this example. Although as there are not further studies provided, this cannot be said in general. We were not able to find more literature on this specific metric than the original paper.

We used a tool called DepDigger[1] to analyze a java open source project called Timber [2]. With this tool we found the highest DepDegree for a method within the project of about 130, the lowest with 0. Although the tool does not allow us to calculate the average of all methods, it seems to be between 10 and 30.

Method	LOC	CC	DD
swap 1	3	1	6
swap 2	3	1	3
bico 1	9	3	28
bico 2	9	3	24
equals 1	10	3	11
equals 2	11	4	8

Table 3.1: Indicator values for differnet function implementations [BF10].

Object orientated metrics. In [CK94] different viewpoints for each object orientated metric are given. In the following we want to examine these viewpoints for each metric.

Depth of inheritance tree

1 https://www.sosy-lab.org/ dbeyer/DepDigger/
2 https://github.com/naman14/Timber

	DIT	NOC	CBO	LCOM	LOC
Max.	33,00	1.214,00	70,00	55.198,00	9.371,00
Min.	0,00	0,00	0,00	0,00	0,00
Median	2,00	0,00	6,00	21,00	57,00
Mean	3,13	0,92	7,80	364,66	183,27
Standard deviation	3,42	21,65	8,35	1.875,40	425,31

Table 3.2: Obrained metrics [GFS05].

- The deeper a class in the hierarchy, the more methods it has inherited. This makes the class more complex.
- Deeper trees constitute a higher complexity.
- The deeper a class is in the hierarchy the more code it reuses.

The object orientated design promotes inheritance, it is one of the fundamental concept of this type of programming. With this a programmer can dictate certain interfaces and code does not have to be duplicated. Following this criteria a programmer should strive for inheritance wherever it seems fit. However, this at the same time leads to more complexity in the program, according to this metric.

In [GFS05] the authors used the object orientated metrics proposed by Kemmerer. They analyzed a Mozilla software and computed a maximum for the DIT at 33 and the minimum at 0. From this alone we can see that the metric varies a lot for a lot of classes. The median they computed is 2. Their complete findings can be seen in table 3.2 Following the above stated points the class with a DIT equals to 33 should be much more complex than the base class. However, this does not have to be the general case, as often times the classes that form the root of the inheritance tree are complex classes to form the general basis, subclasses then may only change an insignificant portion of the source of a few methods.

Number of children.

- The greater the NOC value the greater the code reuse.
- The greater the NOC value the more probably it is that the superclass is not well defined.
- The NOC value gives an approximation of how much influence a class has.

Looking at the first and second statement it becomes obvious that for a proper use of this metric a certain threshold has to be defined which distinguishes when a class has too many children. However, this is hard to do. Each team of programmers has to define this threshold on their own. With this the meaning of this metric can be disrupted easily. Furthermore in a large software project, the NOC number for various classes can differ drastically, as shown in [GFS05]. The maximum value the author computed for their analyzed project was 1.214 for one class, the project holds a total of about 3.000 classes. The next biggest number for NOC was just 115. So the metrics on the one hand says that this large class is good because it distributes code and on the other hand is bad at the same time as it indicates an improper use of sub classing.

Coupling between objects.

- Excessive coupling counters the idea of modularity.
- The higher the coupling, the more sensitive a system is to changes.
- The higher the coupling, the more testing has to be done, as many modules interact with other modules.

This metric essentially gives an overview of the enforcement of the encapsulation principle in object orientation. If the number is high it also points to a bad architecture for the software system. The authors in [CK94] also show that in most projects there are one or more classes that are usually more coupled with other classes than others, which indicates that these classes should be investigated more closely. Again, a programmer has to define a certain threshold to which he things coupling is reasonable and when it becomes excessive. Described in the next part utility classes do also worsen this metric, however they are necessary.

Lack of cohesion in methods

- Cohesiveness in methods promotes encapsulation.
- Lack of cohesion implies that the design of the classes is not best.

File	CC	LOC	LCOM	DIT	NOC
DB.java	17	163	0,939	6	0
DB2.java	15	216	0,789	6	0
Transfer.java	19	163	0,737	6	0
MainMenu.java	14	214	0,86	6	0
StandardCash.java	16	102	0,625	6	0
Information.java	3	47	0,5	6	0
Cash.java	2	113	0,929	6	0
CardTransaction.java	1	62	0	6	0

Table 3.3: Findings in [SPR12].

- Low cohesion implies more complexity.

In our opinion this measurement should not be used to evaluate a whole sofware project, but only to evaluate each and every class on its own. However, we have an exception here again. If we take a utility class for instance, which is a class for various not closely related convenience or small methods used throughout the project, the LCOM score is typically bad for such classes. In [SPR12] the authors analyzed different source code files with regard to object orientated metrics producing table 3.3. The authors came to the following conclusions:

- $DIT > 6$ indicates good re-use of classes but to an increased effort needed for testing. Furthermore it indicates great complexity.
- If NOC has a high value the same as for the DIT measure applies.
- High CBO complicates testing. Developers should strive for low CBO.
- High LCOM points to high complexity in the design of the class, therefore it should be more modularized, by splitting it in two or more classes for instance.

Interesting to note here is that Kemmerer et al. claims that low LCOM implies a higher complexity whereas Suresh et al. claim that high LCOM implies a higher complexity. We have not found any other literature supporting either of both claims.

We want to point out that the object orientated metrics are closely correlated and therefore trying to achieve a better score for one metric has to be most exactly considered. For instance, if the classes with a high LCOM are splitted into subclasses the NOC and DIT measure increases as well if the splitted classes inherited from a parent class.

Software maintainability index. This metric has an obvious disadvantage which is the dependence of the percentage of the commented lines. Taking this into consideration, while computing the metric makes it very easy to distort the result of the metric. One could simply add or delete a lot of commented lines to improve or worsen the result of the metric. Of course there are commenting guidelines, especially for large software projects, which should unify all results for this metric. However, the chance is that, in a big source code basis there are exceptions to the guidelines, therefore distorting the result. If one really wanted to get a good score from this metric he could just add commented lines. We want to give an example of the calculation of this metric and how we could adapt the percentage of the commented lines to distort the result.

We assume $avgHV = 80$, $avgCC = 3$, $avgLOC = 1000$ and $perCM = 0,25$. With that we get the following MI.

$$MI = 171 - 5,2 * \ln 80 - 0,23 * 3 - 16,2 * \ln 1000 + 50 * \sin \sqrt{2,4 * 0,25} = 70,59 \tag{3.1}$$

Now we want to better our result and therefore increase the number of commented lines, so that $perCM = 0,5$. Because we increased the comment percentage we also incrased the $avgLOC$, assuming that for example we shortened the code for the existing functionality and added a lot more comments. With this we get a significantly better MI:

$$MI = 171 - 5,2 * \ln 80 - 0,23 * 3 - 16,2 * \ln 1100 + 50 * \sin \sqrt{2,4 * 0,5} = 78,53 \tag{3.2}$$

As stated in [Nie16] it is not easy to understand the calculation of this metric and it can be very easily manipulated. It can be used for comparing different projects/files with each other, given the same calculation principles. But this is something one can accomplish with every metric.

3.2 Programmer perceptions

In [KK12] the authors discuss various metrics and how their value differentiates from developer opinions. The result of a metric is often not congruent with the opinion of one or more programmers looking at a source code. Showing how difficult it is to bound the comprehension of human beings into a number the authors calculated matching indices M_m for each metric and their associated opinion of programmers, this can be seen in figure 3.1. The opinions were obtained by letting participants of the study fill out a questionnaire. Their results show that the Lifespan metric has a higher accordance with programmers opinion than the Dependency Degree measurement. But the results of the previous two mentioned metrics were better than the CC measurement. This indicates that data-flow metrics predict the opinions of programmers more than control-flow metrics. The score for the CC metric was even worse than a useless metric *Foo* that the authors put in. These more or less renders the CC metric useless compared to how programmers see it. Even though the Dependency Degree metric and the Halstead difficulty had around the same score as the useless metric, showing that those metrics do not really reflect the developers opinion either. A very primitive metric NOES (number of occurrences of a character) that the authors also analyzed had a similar performance than most of the much more computational expensive metrics questioning the validity of these metrics in the eyes of human beings.

Figure 3.2 taken from [JMF12] shows a scatter plot of the perceived complexity from 1 to 10 (10 being the most difficult) and individual functions with different CC values from a linux kernel. This shows that the perceived complexity and the CC value share very little correlation. Assessing how comprehensible a program is, is another task complexity metrics try to achieve but fail [FALK11].

We conclude that most complexity metrics do not reflect the opinion of human developers. Some of them do even less correlate with the developer's opinion than a random and useless metric. Size metrics are the only reasonable indicators that can estimate a programmer's opinion on the complexity of a program and even these are often times very loosely correlated.

CyclCompl	*Npath*	*DepDeg*	*Lifespan*	*HalDiff*
0.6167	0.5791	0.6962	0.7433	0.7169
LOC	*CFS*	*NOES*	*Foo*	*SR*
0.6854	0.6541	0.7428	0.7243	0.2755

Figure 3.1: Matching indices. Calculated means [KK12].

3.3 Metrics and maintainability

During the life cycle of a software applications the maintenance of the source code can take up 40% to 70% of the total expenses [HMKD82, CALO94]. This shows that the maintenance of a software, especially when it is meant to be used for a extended period of time is at least as important as the initial development. For this reason, it is important to analyze and quantify the how hard it is to maintain a system.

In agreement with [CALO94] we define the following two terms:

- **Maintenance:** The process of modifying a (part of a) software system to improve performance, correct faults or adapt the software to other circumstances after the delivery of the software.
- **Maintainability:** The ease with which a (part of a) software system can be modified for maintenance.

Software complexity metrics which measure how difficult it is to comprehend a given program are the primary measurements used to classify the maintainability of a project [HMKD82]. Since a program or a component of a program first has to be understood before a developer is able to make changes and maintain or improve the functionality. We want to point out that measuring software complexity with complexity metrics only acts as an indicator for the maintenance. The metrics do not provide any value of why the software might be hard to maintain. Another key

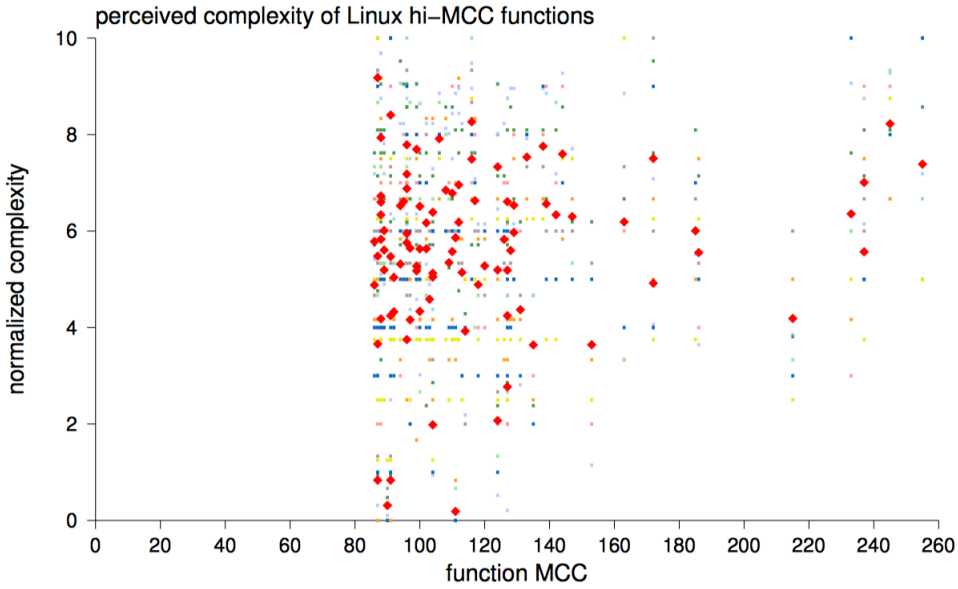


Figure 3.2: Scatter plot showing relationship between measured CC and perceived complexity [JMF12].

element for maintaining a software project is the level of experience the developer team have when it comes to maintaining a system [Kem95].

Given the fact that complexity metrics are used to determine the maintainability of a software project we want to analyze our metrics with regards to maintainability.

Size metrics. For the purpose of estimating the time and cost to maintain a code basis size metrics are good indicators as they present a quantitative measurement [HMKD82]. A project with 5.000 SLOC will require much less time than one with 100.000 SLOC, due to the quantity of source code lines a programmer or a team of programmers has to go through and understand. That said it has to be pointed out that there is no absolute correlation between the maintainability and the size of a software project. Especially, when the size difference between two components of systems to compare decreases other measurements become more important [HMKD82]. Experienced programmers often times need less LOC to achieve a certain task than less experienced programmers. Leaving the shorter source code seen as more complex, as less experienced programmers often use the most intuitive way of programming a task, without regard to performance.

Size metrics, such as the Halstead metrics or LOC, generally provide a good indication of how much time, and cost for that reason is needed to maintain a software system.

Control flow metrics. The cyclomatic complexity metric will be our primary metric regarded in this section, as it is throughout this paper. Previously conducted research such as [GK91, Kem95] came to the conclusion that the CC is strongly correlated to the SLOC metric. Kemmerer et al. has shown that more complex metrics, such as CC, often times essentially measuring the size of a program in one form or another. Therefore he comes to the conclusion that such complex metrics often times only provide little more information as the SLOC metric for instance. This makes the size metrics more favorable as they are much easier to compute and substantially provide the same information as more complex measures. This can be seen in data collected in [GK91] seen in table 3.4. The authors collected different data points before the maintenance was made, as well as the size of the change during the maintenance. Productivity is the LOC added divided by the time (in hours) spent in coding, testing and upgrading the software. It can be seen that generally the productivity decreases as the CC increases. The only exception being are the data points with index six. The initial CC as well as the SLOC are low compared to the other data points, which should conclude in a high productivity. But the productivity is the lowest of all data points with $0,3$. When looking at index four and five we can also see that the initial CC and productivity of index five is higher than the ones from index 4. These points contradict the hypothesis that the higher the CC is, the harder it is to maintain a software system. We want to point out that it is highly controversial to calculate a programmers productivity in this way and that these results are also highly dependend on the developer's who have done the change. However we can argue

Index	Productivity	SLOC added	Labor Hours	Initial CC	Initial SLOC
1	10,1	451	45	882	6682
2	10,3	3143	305	958	7133
3	3,6	395	110	1054	5343
4	2,0	347	174	1126	5738
5	6,1	341	56	1176	6085
6	0,3	221	737	310	2036
7	19,0	2147	113	370	3435

Table 3.4: Summary of maintenance data values [GK91].

that the CC measure is not a a crucial factor for calculating the time needed for maintenance with this study.

Another example is the following: A program with five branches will always have a cyclomatic complexity of five, regardless of how the branches are arranged [HMKD82]. However, the arrangement of the branches can make a huge difference for a human being to understand it.

Overall, we conclude that the cyclomatic complexity is not a good indicator of the maintenance cost of a software project and that size metrics should be used preferably.

Data flow metrics Unfortunately, we could not find any studies using data flow metrics for the prediction of the difficultness of maintenance. We argue that metrics in this category are not suitable for an overview over a complete software project. However if one has to analyze a specific function or class, these metrics can give an estimate on whether the given class/functions is less or more complex compared to another class within the project.

Object orientated metrics. We did not find studies exclusively looking at object orientated metrics for the sake of predicting the cost of maintenance. We think that these metrics can add value to the maintenance prediction. This is because these metrics generally concentrate more on the overall (object orientated) structure of a software system and not on the details of implementation itself, as most other metrics do.

In [Nan07] the open source project Blender is analyzed between the years 2002 and 2007 using object orientated metrics. The DIT measure went from around 380 to 550, the NOC measure went from ca 250 to 450 and the CBO measure did not change as much from about $0,35$ to $0,55$. The LOC metric also rose in a similar linear way as the DIT and NOC measure. One could argue that the object orientated metrics are indirectly measuring the size of a software project. Nonetheless, we think it does make sense to compute these metrics, if an object orientated project is given.

Hybrid metrics. We will use the software maintainability metric as the representative for the hybrid feature category. As the name suggests this metric is meant to measure the maintainability of a software project. However, it remains undefined which kind of maintenance is meant explicitly [Gra08]. Therefore we see it as a generic measure for the maintainability. Gray in [Gra08] studied different versions (2.00.x, 2.02.x and 2.04.x) of the linux kernel, computing different metrics for it. In the following we want to represent his findings regarding the MI metric. The study calculated the MI for the 2.00.x kernels to be between 55 and 101, for the 2.02.x kernels to be between 107 and 108 and for the 2.04.x kernels to be between 110 and 113. This suggests that the variability between the kernel series is much greater than in between the series. The biggest problem Gray points out is that MI is not sensitive to the number of functions. The author calculated that according to the MI the entire linux kernel is as maintainable as a program consisting of a single function containing 8 to 13 SLOC, a Halstead volume of 140 to 320 and comments on about 40% of the SLOC. This clearly shows that the MI should not be used exclusively as a single indicator to analyze the maintainability of a program. Furthermore the MI only allows to calculate an average value, which does not allow to seek out extreme outliers in a software project, be it in the maximum or minimum.

3.4 Computation difficulty

How long it takes to compute a certain metric is an important indicator on whether to use this specific metric or not. Today nearly every metric can be calculated with the code editor, IDE used or an external tool such as Teamscale, for example. Nevertheless, we think the complexity of

the calculation is an important aspect of each metric to understand how the complexity that this metric defines is made up and how it can be influenced.

Lines of Code. Arguable the easiest metric to compute. Nearly every code editor automatically shows the number of lines next to the text input. Therefore the computational cost for this metric is very low.

Halstead metrics. Since all of the Halstead metrics are based on counting the unique operands and operators, this metric is also easy to compute for a code analysis tool and can even be done by hand, for smaller projects.

Cyclomatic complexity. Since this metric relies on the control flow graph of an application the graph first has to be computed. After the graph is computed, the calculation is done easily. The cyclomatic complexity is more expensive to compute as previously mentioned size metrics, because of the computation of the control flow graph. To visualize a control flow graph only from seeing the source code, is also hard to do for the human mind.

Lifespan. This metric is again rather easy to compute and also possible to compute by hand. It is simple searching for multiple occurrences of a variable, then calculating the number of lines in between them.

Dependency degree. Similar to the cyclomatic complexity the control flow graph has to be computed first for this metric. Further, the control flow graph has to be transformed into a use-def graph. Because of these two transformations of the graph, this metric might be the most complex metric to compute in our research, except the hybrid metrics.

Object orientated metrics. Different from the aforementioned metrics these metric can only be calculated by looking at the whole source code in contrast to only looking at one component (file/class/object/method). Therefore these metrics are harder to compute, as they focus on relationships between modules rather than looking at the module itself.

Software maintainability index. This metric is by far the most expensive one to compute in our research, since it relies on various other metrics that have to be computed beforehand.

4 Conclusion

In this study we looked a number of different software metrics measuring the complexity of a software project. We categorized the metrics into four categories and presented the most relevant and widest known members in each category. In the second part of this paper we studied the previously defined metrics under different aspects. We conclude that most of the time no complex metric, such as the cyclomatic complexity or the maintainability, is needed, as they do not add much value to simpler metrics such as different size metrics. Furthermore opinions of programmers about a software's complexity often times differentiates widely from the complexity measured by the metrics. The more intuitive and easier to understand a metric is the more it reflects the opinion of programmers. The size metrics, especially the LOC metric emerges as one of the most useful metric, while at the same time also one of the easiest to compute. Every programmer or team has to define for themselves when a metric is too high and should be acted upon. We suggest to never rely on just one metric, but to have a set of metrics that have to be evaluated together. We also think it does not make sense for a software project to try to strive for a perfect score in one or more of the used metrics. A metric should not be used as an absolute measure, rather than as a general indicator which shows where some investigation needs to be done. As of this writing there is no clear standard metric suite to evaluate different software projects, which would help to define and compare software projects even better. We think this would be a worthy goal to achieve in the future in the field of software complexity metrics.

Bibliography

[BF10] D. Beyer and A. Fararooy. A simple and effective measure for complex low-level dependencies. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 80–83, June 2010.

[CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[Els76] J. L. Elshoff. An analysis of some commercial pl/i programs. *IEEE Transactions on Software Engineering*, SE-2(2):113–120, June 1976.

[FALK11] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner. Exploring software measures to assess program comprehension. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 127–136, Sept 2011.

[GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, October 2005.

[GK91] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, Dec 1991.

[Gra08] Thomas Lawrence Gray. *An analysis of software quality and maintainability metrics with an application to a longitudinal study of the linux kernel. Ph.D. thesis*. PhD thesis, Vanderbilt University, 2008.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[HMKD82] W. Harrison, K. Magel, R. Kluczny, and A. DeKock. Applying software complexity metrics to program maintenance. *Computer*, 15(9):65–79, Sept 1982.

[Hum14] Benjamin Hummel. Mccabe's cyclomatic complexity and why we don't use it, 2014. `https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/`, visited on 2017-11-19.

[JMF12] A. Jbara, A. Matan, and D. G. Feitelson. High-mcc functions in the linux kernel. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 83–92, June 2012.

[Kem95] Chris F. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, Dec 1995.

[KK12] B. Katzmarski and R. Koschke. Program complexity metrics and programmer opinions. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 17–26, June 2012.

[McC76] Thomas J. McCabe. A Complexity Measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–. IEEE Computer Society Press, 1976.

[Nan07] Volker Nannen. Measuring and tracking quality factors in Free and Open Source Software projects. master's thesis. Master's thesis, University of Helsinki, 2007.

[Nie16] Rainer Niedermayr. Why we don't use the software maintainability index, 2016. `https://www.cqse.eu/en/blog/maintainability-index/`, visited on 2017-11-19.

[OH92] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, Nov 1992.

[SCG17] Rajender Singh Chhillar and Sonal Gahlot. An evolution of software metrics: A review, 08 2017.

[SPR12] Yeresime Suresh, Jayadeep Pati, and Santanu Ku Rath. Effectiveness of software metrics for object-oriented system. *Procedia Technology*, 6(Supplement C):420 – 427, 2012. 2nd International Conference on Communication, Computing amp; Security [ICCCS-2012].

[Wey88] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, September 1988.

[wik] Linux (kernel). `https://de.wikipedia.org/wiki/Linux_(Kernel)`, visited on 2017-12-22.