Measuring LOC and other basic measurement

Seminar on Software Cost Estimation, WS 2002/03

erstellt von

Emanuel Weiss

Angefertigt am
Institut für Informatik der
Universität Zürich

Prof. Dr. M. Glinz
Betreuer: Arun Mukhija

Contents

Summary

The process of measuring program size needs some basic metrics and units. Lines of Code (LOC) is the widest used. It is characterized by the difficulty to define it precise enough to be a highly comparable measure and to get realistic productivity rates. But the careful use of it makes it still one of the top measurement methods. From a large choice of alternative measures there are just a few who really can compare with LOC, either by even counting the tokens of line or by distancing from lines of code and counting functions. The process of measuring the time period of software development makes a difference between small and large projects because of different factors to consider. Productivity, size divided by time, can play an important role for software development, but it can also be misleading when wrongly interpreted.

1 Introduction

The reason why to write about this topic is that we want to measure the effort and the cost of developing software. The question is how to do that? One thing we often do, when we measure effort in general is to measure the size of work. When we are doing homework, say calculations, we can measure the effort of our work by counting the number of calculations we made so far. Now with software development it is the same. Independent of how a program is written, it has a size. We will see later that this size can be expressed in various units, but first let us state that every program has a size.

There are at least three reasons to measure the effort of developing software by measuring the size of the program:

- It is easy to do.
- Size is a basic measurement for a number of estimation methods, where COCOMO is probably the most known.
- We generally measure productivity by size.

As mentioned, there are a number of metrics for measuring size and I want to present some of them, beginning with the most popular one: Lines of Code.

2 Measuring Lines of Code

2.1 Why do we count lines?

We all probably remember situations where our program did not work and we did not know why. We were searching the error, changing some parameters, inserting some new commands and deleting others just to find out, what is wrong with the program. And most of us could tell of at least one situation where the error was hidden in one single line. At least then we realize that every line of code is work and requires effort. And this is the basic idea: The more lines of code I write, the more work I do. Of course this statement is vague and neglecting lots of aspects as we will see later. But the main idea of counting the lines of code of a program to measure development effort remained the same until today.

The error searching example above shows also an other thing: We wrote lines that were correctly compiled the first time and others that were more tricky or at least needed more time to write, not just because of their length but because of the logic implemented in this line. This leads to a second statement, that one line requires more effort than another. A conditional clause – an if statement and the following Boolean expression – requires more effort than an output statement in most cases. So we need a size metric that makes a difference between simple lines of code and complex lines of code.

The first alternative is by counting tokens instead of lines, the second is by grouping lines of code and only count the groups. We will discuss these methods when we are familiar with the Lines of Code metric.

The unit for this metric is LOC, the abbreviation of Lines of Code, and its symbol is S_s. Because of the length of programs we also use the unit KLOC for one thousand lines of code. This unit has the symbol S.

Another problem of this metric is that there is no precise agreement about which lines to count and which not. It is important to clearly define which kind of lines you count in an organisation, before collecting masses of data.

2.2 Comments

First you have to decide about counting comment lines or not. We all agree that writing comment lines for documentation purposes needs time and is effort. But we also agree that not every method of documentation needs the same time and that code lines normally need more effort to write than comment lines[1]. It makes a difference in time and effort whether I write a line of comment consisting a precondition or whether I write a line of comment like `/* End of While Clause */` . Further the amount of comment varies because of the developers programming style, the documentation rules of an organisation and the language used.

It is important to know that the decision, whether to count lines of comment or not, could have an influence on the code written. If you count lines of comment, then programmers could be tempted to write much more comment line than needed, only to enhance their productivity rate. On the contrary, if you do not count lines of comment, programmer would appear more efficient, if they omit to document their code. If we look for a moment beyond the development process, the cost of maintenance and enhancement will increase because of missing documentation and the overall cost will be higher than in the case where programmer had taken the time to document their code.

2.3 Non-executable code

A second decision has to be made for non-executable code. With regard to effort, we could argue again that non-executable code like constant and variable declarations, function headers or labels should not be count as lines of code. They either do not contain the real logic of the program or when they do, they are so easy to write that you cannot say that it was an effort to write this line. Often we make a difference between function calls and function headers as well as between the go to statements and their labels. While function calls consist of program logic and are counted as lines of code, the function headers are bounded to the function calls. We do not have to think another time about the same program structure, the statements are inseparable. Therefore we do not count function headers. The same is true for go to statements and their labels. This question will come up again when it comes to the counting of tokens and Halstead's Software Science (see Chapter 3.1).

2.4 Non-delivered code

The third decision goes in the same direction: What about non-delivered code? Do we count the lines of codes of temporary programs like test drivers, that may require upto 5% of the total software effort? Stevenson [Ste95] writes about large projects where the non-delivered code from tools, utilities and even compilers is four times the amount of delivered code. He recommends to avoid a lot of this

[1] It is important to see that this is not about documentation and whether this is good or not. The necessity of documentation should be clear to everyone who reads this document.

"throw away code" by developing top down to omit writing test drivers and by standardization of tools to use more than just once. But the main question about non-delivered code remains unanswered. In my opinion we should differentiate between wrong code from inexperienced programmers which has to be rewritten and should not be counted and non-delivered code which is essential for and is part of the software project like test drivers and should be counted as regular effort.

2.5 Statements and Lines

The fourth thing to consider is what to do with lines of code that have two or more statements. Or to put it in other words: Do we really count lines of code or do we count statements terminated by a delimiter (a semicolon in most cases)? Remember that in certain languages we then would count an if-else-clause as one unit.

2.6 Definition of LOC and Productivity

Now that we have defined some aspects of the metric Lines of Code we could dare a definition. Conte [Con86] states:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

There are some more open questions about the metric Lines of Code that are mentioned next sub-section. We could also say that these are disadvantages in the sense that the metric is not precise and handled differently in various organizations. On the other hand it is important to know that most of these open questions can be answered by precise definitions and standards.

The first questions are all dealing with the problem of productivity. So let me quickly introduce this term: Productivity in software cost estimation is the output quantity divided through a period of time. In this case the output quantity is equal to the amount of LOC and the period of time is the time to develop a program.

> Productivity = output quantity / period of time

2.7 Disadvantages

Comments and productivity

Counting lines of code can be a motivation for programmers to write lengthy instead of lucid, not only with regard to comments (see above) but also to the code itself. The reason is that without needing much more time we write a lot of additional lines of code which again seem to increase our productivity.

Experience and productivity

The experience of a programmer can make a difference in code length up to 20%. This reduction of code could be misinterpreted as inefficient compared with an inexperienced programmer who writes much more lines per day. Again we have a seeming increase of productivity.

Efficient code and productivity

Fewer lines of code leads to higher productivity seen through the eyes of sales management because efficient code is sold more often. This fact shows the importance of looking beyond the development process when considering the number of LOC.

Languages and productivity

Counting lines of code has another problem with productivity when it comes to the comparison between high- and low-level languages. The power of high-level languages can be seen in the number of LOC for one and the same program. High-level languages have much less lines than low-level languages and the projects can be finished earlier. Now suppose a manager is responsible for this three implementations of the same program and wants to know how the productivity is for the single implementations (see Table 1). Inexperienced as he is, he divides the amount of LOC through the total project months and finds out, that low-level languages have a significantly higher productivity than high-level languages because of a rate of 555 LOC per month for macro assembly, 350 LOC per month for Ada 83 and only 333 LOC per month for C++. A possible reaction to these results could be: "Let us code only in low-level languages from now on" or "If my programmers write more than 500 LOC of assembler code in one month and C++ is a much more powerful language, we will estimate 700 LOC of C++ code for our next project." Capers T. Jones [Jon98] called this phenomenon the mathematical paradox. In fact, the high-level languages have a higher productivity. To prove this we could divide the amount of LOC through the time we really spent with coding. We get 2000 LOC per month for macro assembly, 2300 LOC per month for Ada 83 and 2500 LOC per month for C++.

Table 1: Implementation of the same program in three languages

	Macro assembly	Ada 83	C++
Source code required, LOC	10,000	3,500	2,500
Effort per activity, staff month			
Requirements	1.0	1.0	1.0
Design	3.0	2.0	0.5
Coding	5.0	1.5	1.0
Testing	4.0	1.5	1.0
Documentation	2.0	2.0	2.0
Management	2.0	1.0	0.5
Total project, months	18.0	10.0	7.5
Total project, LOC/staff months	555	350	333
Coding, % total project	28%	15%	13%
Total project, LOC/staff months of coding	2000	2300	2500

An interesting thing we can observe in this table and will discuss later is that most of the time of a software project is used for other things than coding. We will also have to answer the question whether counting lines of code is a reliable estimation method for the effort of the whole software development process (not only the coding process) or not.

There are two reasons why there is so much struggling with the mathematical paradox:

1. The historical data from different projects used for estimation are not granular enough. We often know the time period of the whole project and the amount of LOC for the program but we have no idea of the time period of the single activities of this project.
2. Counting lines of code is not useful when it comes to cross-language-comparison. We cannot compare the amount of LOC from two programs, not knowing in which language they are written. Modern tools are able to identify the language and use adjustment factors for the estimation process, saying for example that for one C++ statement we need an average of five assembler statements. Here comes in an idea of Stevenson [Ste95], who considers to count machine instructions instead of LOC or tokens.

The other way is to use an estimation method which is independent from languages and therefore free from LOC distortions. The function point method is probably the most known today. So if we would divide the number of function points, which is an estimate for effort and the same for the three implementations Macro assembler, Ada 83 and C++, through the total project time, we would receive

a rate which represents the productivity of high- and low-level languages (see Table 2). Assuming that the program contained 50 function points, we would have rates of 2.8, 5.0 and 6.6 function points per month.

Table 2: Implementations and Function Points

	Macro assembly	Ada 83	C++
Function Points	50	50	50
Total project, months	18.0	10.0	7.5
Total project, Function Points/staff months	2.8	5.0	6.6

Modularisation

The resulting number of LOC strongly depends on the degree of modularisation. If we use to split up our programs using a modularisation by functions we add upto 25% to the actual length of our program. Using a modularisation by data type adds about 53% of code and a super-modularised program where this principle is carried to extremes contains 73% additional lines of code (see also Table 3). We further can state that the more we practice structural programming the more LOC we will have.

Table 3: Modularisation and additional LOC

Modularisation type	Additional LOC in %
Modularisation by functions	25%
Modularisation by data type	53%
Super-Modularisation	73%

Strongly typed languages

Programs that are developed with strongly typed languages like Pascal or Java have usually more lines of code compared with programs containing the same functionality but coded with weakly typed languages like Fortran.

Reused code

Another problem of this estimation method is that of reused code. Do we count lines of code that are written by someone else? Perhaps we only have to make some adaptation to the code copied, how do we calculate this effort? We probably all agree that we neither can fully count those parts that are reused nor can we only count these lines that we wrote because of the effort we made to read and understand the code, adapt the interface or recode some functions to finally put those parts together. The total lines of Code S_e consists of an amount of lines of new code S_n and an amount of lines of reused code S_u. S_e is a function of these parts S_n and S_u. Conte [Con86] shows three ways, how this function could look like. All of them are additions of S_n and S_u, but the adaptation of S_u is different.

The first formula has an adaptation factor which is influenced by the amount of design modifications, code modifications and integration modification. These three parameters between zero and hundred represent the effort invested in the single modifications. The higher the modification factors, the more lines of reused code are counted as total effort. This formula was the idea of Boehm and is used in his estimation model COCOMO.

$$S_e = S_n + a/100\ S_u\ , \qquad a = 0.4(DM) + 0.3(CM) + 0.3(IM)$$

DM = Design Modification

CM = Code Modification

IM = Integration Modification

The second formula is from Bailey and Basili and has also this adaptation factor, which is fixed to 0.2.

$$S_e = S_n + k\, S_u , \qquad k = 0.2$$

The third formula from Thebaut uses the parameter k as power, where k is smaller than one. The more code is reused the less you will count these lines in the total S_e. Integrate reused code needs some general effort, but it is not so important whether the reused code consists of thousand lines of code or a million lines of code.

$$S_e = S_n + S_u^{\,k} , \; k<1, \; k \approx 6/7$$

Other disadvantages

Another weakness of this metric is that it does not consider complexity, reliability or quality of the software. We all have written simple loops that took more of our time than a lot of sequential statements. We cannot measure these efforts with counting lines of code.

The metric LOC is not suitable for technological aids such as utilities like file reformats or prints and many fourth-generation software products like screen painters.

Counting lines of code does not take into account the effort of non-programming activities such as getting requirements, user training, administration and clerical jobs, installation planning. But they can be included in models based on the metric LOC.

As seen above there are different measuring methods. So it could easily happen that we are using data about projects where we know the functionalities and the size in lines of code, but we do not know anything about the measuring methods that were used to collect these data. The lack of a clear and worldwide definition of measuring LOC is problematic.

At least we cannot make any prediction with the method of counting LOC. It can only be used if a program is written and we can really count the lines of code.

2.8 Advantages

We have seen that this metric has a lot of disadvantages or at least weaknesses and problems. Of course the question comes up why we still estimate software development effort like this? After mentioning some reasons why we do so, we will make some concluding statements about counting lines of code.

We use lines of code because...

- ... it is one of the most widely used techniques in cost estimation.
- ... it is the basic metric underlying several cost estimation models by Boehm.
- ... due to the widespread use, it allows a simple comparison to data from many other projects, the historical ones included.
- ... the alternative methods to the counting of LOC are also fighting with problems and weaknesses
- ... it is an easy method to measure effort
- ... in spite of its unreliability for individual programs, it gives reliable average results, which is crucial especially for huge projects.
- ... it is also reliable in small projects when we quantify the method defining the programming language and the rules how we count lines of code.
- ... the discrepancies caused by including or excluding Job Control Language, administration or clerical work are usually small.
- ... it considers in fact the higher productivity of high-level languages (as seen above).
- ... we can avoid lengthy code by organizing reviews, increase, as Stevenson [Ste 95] proposes, the pressure of work or introduce adjustment factors for especially verbose programmers.

- ... cost estimation models like COCOMO, which are based on lines of code, show a close agreement between predicted and actual effort.
- ... there is a strong correlation between lines of code and effort.
- ... and there is also a strong correlation between lines of code and alternative metric or in other words: it is possible to switch from lines of code to another metric and vice versa.

2.9 Qualifying Lines of Code

In addition it is an interesting fact that programmers write a certain amount of LOC per day regardless of the programming language. This implies that using a language that needs 30% fewer statements for a given functionality has a productivity that is 30% higher. Again we see the importance of qualifying a method. Arthur put it rather well (as quoted in [Ste95]):

> *"LOC is not a good productivity measure because it penalizes high-level languages: Assembler programmers produce five statements to a COBOL programmer's one. But you should not compare COBOL to Assembler: they are as different as night and day. If you compare COBOL programs only to other COBOL programs, and PL/1 to PL/1, then LOC provides a stable comparison tool."*

2.10 Measuring quality and documentation

Of course we should never forget that we still in no way measure quality in code or documentation. We even could argue that a line of code can only be counted if fully debugged and S_s is the amount of successfully tested lines of code. This would probably lead to far and take a measuring effort that is not worth. But we keep in mind that a amount of LOC includes neither amount of maintenance nor guaranty of quality in code structure or documentation.

After gained an understanding of the problems of LOC and why this metric is still widely used, let us have a look at some alternative metrics. They are never playing the important role that LOC does in its simplicity and efficiency, but are worth looking at.

3 Alternative basic measurement

3.1 Counting tokens

At the beginning we mentioned another method for measuring size, namely to count tokens. To put it simple we just sum up the tokens in the program code. Doing so we take into account that lines of code with many tokens, that at least look more complicated than those with few tokens, required more effort.

Halstead developed in his Software Science a weighting-scheme (as described in [Con86]), that works as described above and additionally makes some interesting statistics. We divide all tokens in a program in two classes: The operators and the operands. By simplifying we can say that operators are all defined keywords of a language like String, for, if, while, the brackets and the character to terminate a statement. Operands are all the words and numbers that programmers add to write a program, for example the names of variables or numbers. We call the number of all operators in a program N_1 and the number of all operands N_2. The sum of N_1 and N_2 is N and is equal to the count-tokens-size mentioned above.

$N = N_1 + N_2$

N_1 = number of all operators

N_2 = number of all operands

Again Halstead did not count declarations, function headers or labels in his weighting-scheme. He even omitted to count input- and output-statements, probably also with regard to the purpose, only to count lines which require a certain effort to write.

But Halstead went on, and defined η_1 as the number of different operators appearing in the program and η_2 as the number of different operands appearing in the program. There are again several rules how to count the operators and operands. For example think of the operator "-". We use it in most languages as both unary and binary operator and have to define whether we should count it once or twice.

The sum of η_1 and η_2 is η and describes somehow the vocabulary of a program. Using a language-dependent constant c, we can switch between S and N, dividing N trough c and so link these two metrics. Halstead also defines the metric Volume V, which is defined as the number of tokens N multiplied with the dual logarithm of the vocabulary η. To put this formula into words: Each of the η words of our vocabulary is represented by a binary number. The dual logarithm tells us, how many digits this binary number has. When we multiply this number of digits with the number of tokens we need for our program, then we get the volume V, we could say a digital representation of our code.

$S = N / c$

Vocabulary $\eta = \eta_1 + \eta_2$

Volume $V = N \times \log_2 \eta$

η_1 = number of different operators

η_2 = number of different operands

c = language-dependent constant

The most interesting thing is that over about one thousand programs, the metrics S, N and V were linearly related. Or in other words: You can switch from one metric to another. In addition, changing the counting rules does not effect the metrics N and V. They are robust. However, this method also has definition problems like the counting of LOC and does not consider aspects like experience of programmers, use of tools or modern programming practices, which in turn have a significant influence on software development effort.

3.2 Counting modules

Another method to measure size is by counting groups of lines. Now every program of a certain size has to be split up in parts for the sake of overlooking the whole program and reduce the complexity in the parts. We call this modularisation and it is obvious that we look out for a possibility to use modules as units for this kind of measurement. The advantage of counting modules and take the result as an indicator of the size of a program is, that it is simple. But there is a problem that makes this method not useful: We cannot estimate the effort for a module. How long do we have to work for a module? How many lines of code would this be? Studies show that the average size of a module is about hundred lines of code. But the same studies also show that the size of a module varies from ten lines of code up to ten thousand lines of code. This makes an estimation of effort for a number of modules impossible. Therefore we search another possibility to split up programs in groups of lines of code that are more or less equal with regard to effort and find this possibility in functions.

3.3 Counting functions

In Programming we use functions as a way of thinking. The principles are the same as for modularisation and this thinking is not bound to the programmer's job but is present in our daily activities. We drive a car using a lot of functionalities where we know what they do but not how they do. We write an essay splitting up the whole theme in sections and chapters to concentrate on some

aspects of the theme. We do it because of the limitation of our thinking to overview everything at one time. We are limited to cope with a program of more than hundred lines of code without using abstractions. But we do well with a function of ten lines of code.

Now of course every programmer according to his programming style splits up his program in different ways. And after a refactoring process a program with the same functionality will probably have more functions than before. But it is a fact that the variation of lines of code for functions is significantly smaller than the variation of lines of code for modules. This was shown in an interesting study: A large number of students had the task to write a program with the same defined functionality. The resulting programs were similar in their number of functions, but not at all similar in their number of modules.

The idea of counting functions are still near to the basic one of counting lines of code, of course. But as seen above in discussing language productivity, to put functions at the beginning of a measuring process is a real alterative to LOC.

3.4 Other basic measurement

Metrics by Boehm

There are some other metrics proposed by Boehm (mentioned in [Ste95]) that I would mention shortly. They all focus on measuring the size of a program, but very different ways. A possible metric could be:

- a combination of parameters such as the number of routines, operators, operands, files or master files and inputs/outputs.
- the number of variables of the program
- the amount of documentation
- the number of paragraphs in the requirements specifications
- the number of structure-charts in the software design specifications
- the number of lines of documentation written in a program design language
- a subjective estimate of difficulty
- the number of machine instructions, that would be independent from languages and would make possible cross-language comparison.

Boehm rejected this metric later because lines of code would stronger correlate with effort. This makes sense for we spend effort on a statement and do not care about machine instructions. Statements that are easy to write for us can contain a lot of machine instructions while other statements that we write with much effort are some single machine instructions.

Metrics by Jones

Jones also listed some alternative metrics that are based on non-coding activities (mentioned in [Ste95]). Again it is an attempt to measure effort without struggling with language problems and the following mathematical paradox. These metrics have a similar point of view as the Function Point Method. Jones proposes to count:

- the number of pages for the description of software design
- the number of tests
- the number of repairs

Especially the number of tests could be a nice attempt to measure effort with non-coding activities assuming that for every functionality described in the requirements specification there also exists a description for testing this functionality. What misses is the distinction between tests for complex functionality and tests for trivial functionality, the weighing aspect somehow.

3.5 Comparison with LOC

All these alternatives in general were not used. The reason lies in definition and normalization problems like "What is a small and a large routine?" or "What is an important or a nice-to-have requirement?". And Jones put it like that (as quoted in [Ste95]):

> *"The same practical counting difficulties which hamper use of line of code also hamper taking measures of operators and operands."*

In fact, the variability of counting operators and operands actually is higher than the variability of counting lines of code.

We could mention alternatives like measuring the number of personnel, the number of systems or programs used, but they all fail because of definition problems and an unacceptable variability. A metric which would not be based on code and even would take into account the salary and the abilities of each programmer is cost. However, this metric is so difficult to compare over time, regions and countries, that it gets attention no more.

4 Collecting data

After all we see how clearly lines-of-code metric exceeds its alternatives. With that in mind I would briefly comment on the process of collecting data. We need figures about cost and effort for two reasons:

- for valuating an estimator and
- for managing projects

Even if we were not at all interested in estimation, we had to know about profitability, feasibility and productivity of our projects to lead them successfully. But the knowledge of past projects makes new projects easier to plan and is part of many estimation methods. It is important to know how data will be used in the future before collecting them. So before investing in large data material, we should question the purpose of it.

4.1 Collecting data in small projects

We can collect data in small organizations or from experiments. In most cases we are using units like hours or days for time periods. Due to this small units the precision of measurement is very important. A variation of time caused by measuring errors would have a significant impact on the productivity rate. Where are the problems in this measuring process:

Assuming the programmers have forms where they can make notes about the time spent on a project. One problem is the interruption in work through telephone-calls or mail, another is the intentional bias: A programmer rather writes down how much work he should have done than how much work he actually did. An important question is what to do with time spent on non-programming activities like administration or defining requirements. Imagine you have to implement a quick-sort-algorithm in a programming language of your choice. Now either you are one of these guys who say: "Ah, quick sort, I could solve this problem with threads!" and you have finished your code in 30 minutes. Or you are on the other side: You have to search for a description of the algorithm, you make some examples only to get an idea how it works, you have to think about the implementation idea and what kind of features your language should have and so on. You easily can spend half of the day or more for such a task. So one rule for the measuring process would be: Do not mix working-time and education-time.

This kind of measurement is useful for controlled experiments to gain a better understanding of the isolated factors being tested. Note also that there is still a difference between experiments and real projects in organizations.

4.2 Collecting data in large projects

Collecting data in large projects is different with regard to personnel, time or method. We are measuring the period of time in person-months or person-years and do so not for one or two programmers but for whole teams. There is a lot more of non-programming work as administration, communication and clerical work. In fact programmers are only coding four to five hours on an eight-our day and do not forget the days when they are sick. To collect the data by tools instead of forms lets the programmer concentrate on his work. If a programmer has to fill the form himself, he could be tempted to overestimate and making the impression he is overworked or to underestimate and making the impression he works very efficient.

With regard to the fact that we need data material that is comparable to former projects it is important not to change the measuring process. That means for example to define if overtime is counted or not or if we calculate with the normal salary of a programmer or with the burdened average salary, which includes costs for hardware, paper, travel or fringe benefits.

After all, it is unwise to use this data material for both project management and performance evaluation of an employee. In contrary we should free employees from fear about productivity measurement and do our best to avoid distortion factors or cheating. No programmer should have to "pad" his code with non-essential lines only to get a higher salary. Nevertheless, Stevenson [Ste95] recommends to use productivity measurements for selecting good programmers and find the bad ones.

We finish this section mentioning that cost and effort measures have always to be relevant for management goals and listing up three advantages of measuring productivity:

- It provides information, which tools and techniques are the best, why they are and under what circumstances.
- It improves the precision of deadline prediction
- It improves the productivity of programmers

5 Conclusion

Measuring LOC and other basic measurement seem to have a lot to do with weighting the pros and cons. In most cases we cannot say which method is the only one to use. Of course correlation is a good indicator but also are experience, simplicity or comparability. There are so many studies that show one thing and also the opposite that perhaps only time will decide about what is useful to use and what is not. Personally to learn about these methods and to consider different aspects of cost estimation was the useful thing.

6 Bibliography

[Con86] Conte, S.D.; Dunsmore, H.E.; Shen V.Y.: Software Engineering Metrics and Models, Menlo Park, Ca.: Benjamin/Cummings, 1986

[Jon98] Jones, T.C.: Estimating Software Costs, New York: McGraw-Hill, 1998

[Ste95] Stevenson, C.: Software Engineering Productivity, London, etc.: Chapman&Hall, 1995