Manual software estimation

Hu You
2631052
Chen Junfu
2630516

Lyu Fangzheng
2644757

***Abstract*— In now a day, estimation is an essential part of software development to make whole project controllable. Automatic estimation is now wildly used in commercial projects. However, manual estimating is not out of date, it can provide quick estimation for small projects and magnitudes of work for business development without formal estimating procedures. In this paper, there are several rules of thumb which guide estimating and confine the scope of estimation. The main idea of these kinds of estimation is simple enough to calculate in short time.**

Keywords— manual estimation; rules of thumb; software metrics

I. INTRODUCTION

To get total control of whole project especially big commercial development, the project team prefer to utilize formal and automatic methods to predict the cost of the entire work. Although it will cost extra budget, the error can be reduced to an acceptable level. In 75% of time, the total errors are higher than 35% and the maximum errors for both costs and schedules exceed 50%. In contrast, in 45% of time, the errors can be controlled within 30% and only cost 5% of budget by using automate estimates.

The biggest disadvantage of manual estimates is errors. Thus, it is hard to be used as main estimating method for software project, but it does not mean that there is no place for manual estimation. As it is very simple to use, it can be utilized in small projects for quick development or estimating before requirements are specified. And it also does not mean developers cannot use it in formal work. Here is an example to show how you use manual estimating method inadvertently.

If there is a development plan for a commercial project, the information shows below.

- Requirements of software
  - 5000 LOC
  - 150 Function Points
  - Military use
- Personal and schedule
  - 10 programmers
  - 2 months
  - 20000$ budget

Within only few seconds, the unreasonable place can be found: the combination of personal and schedule cannot be true in real development. When you calculate the number and found the personal and schedule cannot fit to the budget, you are doing some kinds of manual estimating. Reversed, if the requirements are known, of course there are also some unreasonable things in the requirements in this example, the magnitudes of these numbers can be roughly figured out.

The numbers shown in the example are too extreme. In daily development, the combinations of key numbers are more complex, and numbers won't be such extreme to make problems sensed. In this paper, several rules of thumb will be introduced and based on these rules personal and schedule can be quickly estimated no harder than calculate by using packet calculator.

II. RULES OF METRICS

A. Rules of thumb based on lines-of-code metrics

In the first few decades, manual estimating methods are based on the lines-of-code (LOC) metrics, and the popular programming languages were all procedural type. It is usually used to estimate how much is the coding work, and combined with radios and percentages, the total effort of project can be estimated. The table below shows six indexes related to the size of program. And to make it more intuitive, the table is transferred to figures.

As the figure 1 shows, the larger the size of program is, the lower the efficiency of coding will be. And the coding effort increases faster than the growth of size. And figure2 illustrates the relationships between non-coding works and coding effort. The percentages of non-code and testing effort are becoming larger and larger, when the project is big enough these efforts will exceed the coding effort.

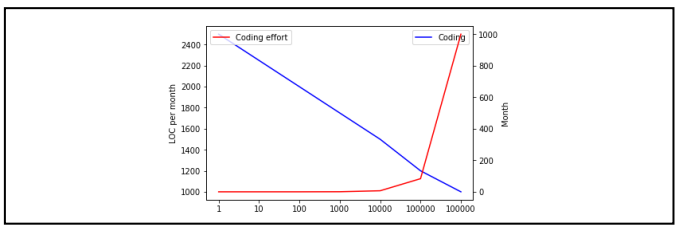

Figure.1. relationship between coding effort and coding

TABLE 1 RULES OF THUMB BASED ON LOC-MERTICS FOR PROCEDURAL LANGUAGES

Size of program, LOC	Coding, LOC per month	Coding effort, months	Testing effort, %	Non-code effort, %	Total effort, months	Net LOC per month
1	2500	0.0004	10.00	10.00	0.0005	2083
10	2250	0.0044	20.00	20.00	0.0062	1607
100	2000	0.0500	40.00	40.00	0.0900	1111
1000	1750	0.5714	50.00	60.00	1.2000	833
10000	1500	6.6667	75.00	80.00	17.0000	588
100000	1200	83.3333	100.00	100.00	250.0000	400
1000000	1000	1000.0000	125.00	150.00	3750.0000	267

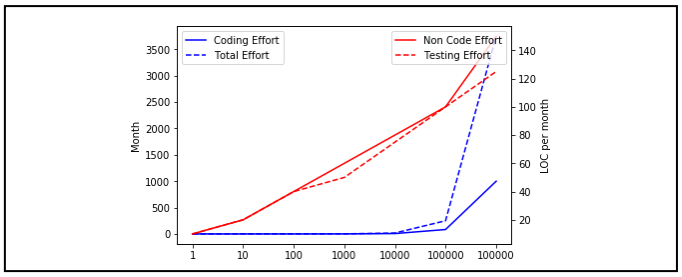

Figure.2. relationships between non-coding works and coding effort

In fact, coding work is only rank 4 of in large system software cost elements, and it is the part that easiest to estimate.

TABLE 2 RANK ORDER OF LARGE SYSTEM SOFTWARE COST ELEMENTS

1	Defect removal (inspection, testing, and finding and fixing bugs)
2	Producing paper documents (plans, specifications and user manuals)
3	Meetings and communication (clients, team members, and managers)
4	Programming or coding
5	Project management
6	Change management

B. Rules of thumb based on ratios and percentages

By using the LOC model, when the size of program fixed, the total effort can be estimated roughly. But as we all know, the non-code works are hard to predict, and it depends on many of factors. For instance, for an end-user project, there is all coding work even if the project may be over thousands of LOC. Here, the ratios for different class and size are shown in the table 3 and table 4.

If you set a fixed ratio for different projects, it will get a huge error. Besides, the combination of class and size will make it more complex, for example, how do we calculate the percentages of 100-FP military projects?

Beside of class and size, programming language, reusable materials and methodology are also the import elements related to ratios. With the new technology coming up, the LOC based rules are dropping from use.

C. Rules of thumb based on function point metrics

Function point metrics are now most widely used in software development, since it was published in the late 1970s. Compared to LOC, function point is defined in higher level and because it is abstractive the influence caused by the different programming languages and reuse will become smaller.

The function point consists of five aspects of software:

- The types of inputs to the application
- The types of outputs that leave the application
- The types of inquiries that users can make
- The types of logical files that the application maintains
- The types of interfaces to other applications

After the function point is defined, the size of software can be measured by function points number. By using the scope, class and type of taxonomy, the approximate size of software can be guessed. Here, assuming IFPUG V4 is the counting method. To get the number of function points:

$$FP = (Scope + Class + Type)^{2.35}$$

For example, an application with the following three attributes:

- Scope: standalone program
- Class: internal—single site
- Type: client/server

According to the IFPUG V4, this application gets scores for each attribute in 6, 4, 8. Then sum them up, get score of 18. Raising 18 to the 2.35 power yields a value of 891. So, we can assume this application contains about 891 function points. Considering a client/server application, consisting of about 900 function points seems reasonable.

TABLE 3 PERCENTAGES OF DEVELOPMENT EFFORT BY SOFTWARE CLASS

Activity	End-user projects	MIS projects	System projects	Commercial projects	Military project
Requirement definition	0	7	8	4	10
Design	10	12	15	10	15
Coding	60	25	18	25	18
Testing	30	30	30	36	22
Change management	0	6	10	5	12
Documentation	0	8	7	10	10
Project management	0	12	12	10	13
Total	100	100	100	100	100

TABLE 4 PERCENTAGES OF DEVELOPMENT EFFORT BY SOFTWARE SIZE

Activity	10-FP projects	100-FP projects	1000-FP projects	10000-FP projects	100000-FP projects
Requirement definition	5	5	7	8	9
Design	5	6	10	12	13
Coding	50	40	30	20	15
Testing	27	29	30	33	34
Change management	1	4	6	7	8
Documentation	4	6	7	8	9
Project management	8	10	10	12	12
Total	100	100	100	100	100

An import inference is that the larger number has more significance than the smaller numbers. As the formal shows:

$$(Scope + Class + Type)^{2.35}$$

$$\approx (Max(Scope + Class + Type))^{2.35}$$

In the example shown behind, the type get scope of 8, so we mainly consider it as a client/server application.

Besides of this formula, sizing by analogy is also a good way for measure the size of application. With more and more software are developed, the development data are becoming more and more rich. Project team can directly estimate the software by consult similar projects.

III. SIZE DELIVERABLE ITEMS AND DEFECTS

A. Sizing source volumes

Actually, function points are widely useful for sizing source code instead of lines of code in the word. The figure 2 shows the difference of cost per function Point versus cost per Lines of cost.

It could be found that, cost per Line of Code rises with the level of language escalates but cost per function Point has a notable decline. So, in the financial aspect, using function point is a better way.

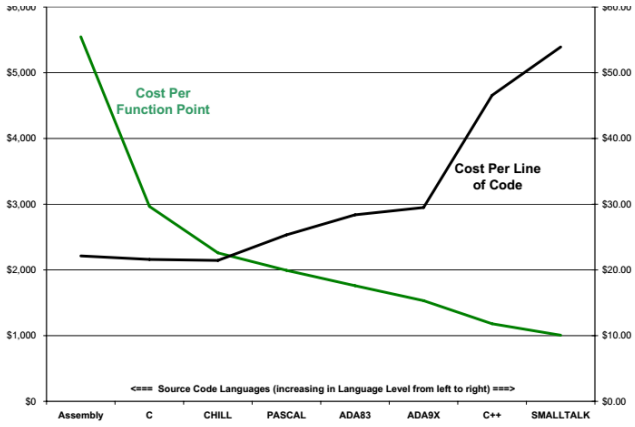

Figure.2. cost per FP versus cost per LOC

Different language converts source code to function points has various levels of efficiency. Table 5 shows empirical ratios that converting source code statements to one function point.

TABLE 5 RATIOS OF SIZING SOURCE CODE

Size of FP	Language
One Function Point	320 statements for basic assembly language
One Function Point	128 statements for the C programming language
One Function Point	53 statements for the C++ language
One Function Point	50 statements for the JAVA language
One Function Point	15 statements for the Smalltalk language

Because of the low language level, 320 statements of assembly could code one function point. As language levels increases, like object-oriented language, fewer statements are required to code a single function point. It is helpful to enhance efficiency.

But for some aging legacy applications, specifications are missing, and source codes are the only remaining artifact, the backfiring that is used for describing converting between source code statements and function point is an important key to arrive at function point values.

In the aspect of small enhancement and maintenance projects below about 15 function points, the weighting factors have lower limit and it creates a floor at about 15 function points, so normal method cannot work efficiently. Nevertheless, backfiring method has no lower limit, it could provide an effective way to calculate as small as a fraction as one function point.

B. Sizing paperwork volumes

In fact, there are kinds of paper deliverable type need to be created during the software development, such as requirements, cost estimates, development plans, functional specifications, change requests, logic or internal specifications, project status reports, user manuals, test plans, bug or defect reports and so on. Paperwork volumes are related to the size of software, small project produces fewer paper documents than larger system produces.

Furthermore, the document standard is popular in the U.S, like 400 words each paper, single-spaced type, using New Times Roman type, so it is easy to estimate the number of the whole English words.

It is noteworthy that paperwork is such a crucial element of software schedule, so it should not be ignored.

C. Sizing creeping user requirements

Creeping user requirements might change uncontrollably and cause several severe problems, because there are no visible penalties in internal information system. Software contracts and outsourcing agreements are used to using function point metric to avoid risks.

Empirical evidence shows that size of function point would grow 2 percent of the initial software FP value every month from designing through coding. For example, there is a 1000 function point project and its schedule from designing to coding phase is 8 months. The final productivity would be 1160 function points because of this rule.

As for the price of single function point, table 6 shows the specific contract. As the time goes on, the price per function point would grow, and the final cost per function point is $1200 that is more than twice of it one year ago.

TABLE 6 ESCALATING COST SCALE FOR FEATURE ADDED

Initial 1000 function points	$500 per function point
3 months after contract signing	$600 per function point
6 months after contract signing	$700 per function point
9 months after contract signing	$900 per function point
12 months after contract signing	$1200 per function point

D. Sizing defect potentials

In fact, most of defect potentials might be found in requirements errors, design errors, coding errors, user documentation errors, bad fixes, or secondary errors introduced in the act of fixing a prior error. The thumb of rule could predict size of potential defects for enhancement: Function point increased to the 1.27 power. Also, function point metric could divide the whole range of defect potentials. Table 7 illustrates that defects outside coding aspect are more than that within it.

TABLE 7 THE U.S AVERAGE FOR SOFTWARE DEFECTS

Defect origin	Defects per function point
Requirements	1.00
Design	1.25
Code	1.75
User documents	0.60
Bad fixes	0.40
Total	5.00

The rules of thumb do not contain two other types because of the lack of empirical evidence: (1) errors in databases, and (2) errors in test cases themselves. It is found that there are no data point metrics to normalize size of database and test case. Commercial companies found that defects in test case are more than that in databases.

E. Sizing test case defect removal efficiency

Function point metrics could help to assume approximately time for each test case would be run or executed in the software cycle, also it could predict the efficiency each test case would find and remove defects.

Commercial software companies use a typical set of test steps to find defects, like unit testing by developers, new function testing, regression testing, performance testing, system testing, and acceptance testing. Normally, there would be 30 percent of errors that are present might be found and removed, the total reduction could be 88 percent of the total. So, it is significant to do design and code inspections and various levels of testing from unit testing through system testing.

The software inspection was firstly proposed in 1974 by IBM engineer Fagan. He thought testing team should find mistakes and defects from paper documents from design through code phase. All of the inspectors need to attend kick-off meeting and then try to find typical in documents errors by comparing with checklists.

Although formal code and design inspection are not inexpensive, major software producers prefer to use them, because formal inspection could reduce increase test speed and reduce test maintenance cost. Between 85 percent and 95 percent of defect potential would be found and removed after testing.

F. Sizing Post-Release Defect-Repair rates

Maintenance repair case has been around software industry for more than 30 years and still seems to work, and maintenance participants could repair 10 bugs per staff month.

IV. Rules of Thumb for Schedules, Resources, and Costs

A. Estimating Software Schedules

This is Rule 9 which can estimate the software schedules from the start of requirements until the first delivery to a client by using the function point. The rule is that function point raised to the 0.4 power to predict the approximate development schedule in calendar months. But the value 0.4 is not fixed, it need adjustment when the project is for military or there's some standard which add complexity or extra paper work like DoD 2167 and ISO 9001. And also, the easier project will match the lower power and the bigger and more complex project will match the lager power.

Rule 9 is one of the more useful by-products of function point because the schedule usually has the highest priority in the topic between clients, project managers and software executives. And like other manual thumb rules, the result is only approximate. So, it can't be use in some serious business occasions like contract, but it's fairly useful as a sanity check.

B. Estimating Software Development Staffing Levels

This is Rule 10 about estimating software development staffing levels. We can use this rule to estimate how many personnel we need to develop a certain application by dividing the function point over 150. And the personnel here are consisting of analysis, design, technical, writing and programming etc. But the value 150 is approximate here, it also depends on the skill and experience of the team and other factors. By the way, if the managers are only interested in programming, they can use the value of 500 to predict the number of programmers they need.

C. Estimating Software Maintenance Staffing Levels

Rule 11 is about maintenance, used to predict how many personnel to keep the application updated. It assumes that one person can perform minor updates and keep about 750 function points of software operational. The value varies from different programs and different programming languages, some programs can perform up to 3500 function points per staff during maintenance. These always mean the programs are well-formed and geriatric, including some special tools to make it easy to maintenance. Last but not least, the experience of personnel is also an important factor, those who are really experienced can perform double efficiency than normal staff.

D. Estimating Software Development Effort

This is a hybrid rule multiplying the results of rule 9 by rule 10 to predict the approximate number of staff months of effort. It's a combination of Rule9 and 10. The hybrid rules are more complex than single rules, but at least this hybrid rule includes the two critical factors of staffing and schedules.

V. Rules of Thumb Using Activity-Based Cost Analysis

In the process of estimating the productivity rates. It is easy to find that military software projects have much lower productivity rates than civilian software projects and large systems usually have much lower productivity rates than small project. So why this productivity differences occur? Our thumb method must be able to answer this question. The table in figure 3 has shown the different activities performed in six general kinds of software: (1) end-user software, (2) management information systems (MIS), (3) outsource projects, (4) commercial software vendors, (5) systems software, and (6) military software. And from (1) to (6), the activities they perform have become more and more.

Activities performed	End user	MIS	Outsource	Commercial	Systems	Military
01 Requirements		X	X	X	X	X
02 Prototyping	X	X	X	X	X	X
03 Architecture		X	X	X	X	X
04 Project plans		X	X	X	X	X
05 Initial design		X	X	X	X	X
06 Detail design		X	X	X	X	X
07 Design reviews			X	X	X	X
08 Coding	X	X	X	X	X	X
09 Reuse acquisition	X		X	X	X	X
10 Package purchase		X	X		X	X
11 Code inspections				X	X	X
12 Independent verification and validation						X
13 Configuration management		X	X	X	X	X
14 Formal integration		X	X	X	X	X
15 Documentation	X	X	X	X	X	X
16 Unit testing	X	X	X	X	X	X
17 Function testing		X	X	X	X	X
18 Integration testing		X	X	X	X	X
19 System testing		X	X	X	X	X
20 Field testing				X	X	X
21 Acceptance testing		X	X		X	X
22 Independent testing						X
23 Quality assurance			X	X	X	X
24 Installation and training		X	X		X	X
25 Project management		X	X	X	X	X
Activities	5	18	21	20	23	25

Figure.3. Typical Activity Patterns for Six Software Domains

The activity-based cost analysis can easily tell the answer why these occur: Because the military software and large system perform more activities than civilian and little system. And another interesting phenomenon is that most of the observed differences of productivity between projects and fields are due to the different set of activities they use to build the software.

This give activity-based measurements strong advantages that can lead to significant process improvements. Data measured, and Data based on rudimentary phase structures can

only get to the project level, which are inadequate for any kind of in-depth economic analysis or process comparison.

Of course, variation in activity patterns is not the only factor that causes productivity differences, the skill and experience of the team, the available tools, the programming code and methods and other factors are also important. But these factors cannot be understood so well unless the cost and effort data for the project is accurate and is also granular down to the level of the activities performed. Because only in this way can the effects of these factors be analyzed rather than vague estimates.

VI. Summary and Conclusions

A. Simple rules of thumb are not so accurate but popular. And they can't substitute formal estimating methods. Rules of thumb are not accurate enough for business but still the most widely used for software projects, and also useful as a sanity check in business.

B. The rules of thumb have high margin of error. The function point is not the only example and normalization metric. There are more for different phenomena in the process of manual software estimation.

C. Work sheets are more accurate than simple rules, but they need more work to use and also require more work when assumptions change. Automated spreadsheets can eliminate some of the drudgery but neither of these two methods can deal with dynamic estimating situations.

D. There are still some problems to be addressed and solved in order to use collected data for process improvements or industry benchmark comparisons.

E. Activity-based software costing is not common now but the need for precision is already critical in the software industry.

A. Manual estimating methods have the largest literature of almost any project management topic. There are a number of books offering algorithms, rules of thumb. But in the process of estimating, the manager should use the data from their own company or organization to make the estimation adapt to their environment instead of using the data of industry averages.

[1] Capers, Jones. Programming Languages Table[R]. U.S:Software Productivity Research, Inc, 1997.

[2] E, P, DOOLAN. Software Practice and Experience[J]. shell Research, 1992, 22(3): 173-182

[3] SPR Programming Languages Table[R]. Software Productivity Research, LLC, 2007. 1-7

[4] Robert, B, Grady, and, Tom, Van, Slack. Key Lessons in Achieving Wide Spread Inspection Use [J]. IEEE SOFTWARE, 1994, (4): 46-57

[5] Fagan. Dsign and Code Inspection[R]. U.S:IBM SYST J, 1976.