

Measuring software complexity for early estimation of development effort

De Tran-Cao¹, G. Lévesque¹ & J.-G. Meunier²

¹*Computer Science Department,*

University of Quebec at Montreal, Canada

²*Philosophy Department, University of Quebec at Montreal, Canada*

Abstract

This paper introduces a new method for measuring software complexity. The term “software complexity” refers to the difficulty derived from the problem that is resolved by software. The method is specification-based as Function Point Analysis (FPA) and its variations. However, unlike other previous proposals that focus on “size” aspect, the method of this paper investigates the “complexity” aspect.

Software complexity is considered as the complexity of the task that must be fulfilled by software. Therefore, it can be analyzed after task complexity models. The task complexity model of Wood is introduced as a theoretical guide for establishing a framework and measures of software complexity. These measures capture the complexity in the input, output data, in data manipulation and in relationships between software components.

An empirical investigation done with 15 software projects shows that the proposed measures are relevant for measuring software complexity. They can be used to predict the effort of development with a fairly good precision. The empirical results confirm the validity of the functional complexity framework and the efficiency of the proposed measures for early effort estimation.

Keywords: software size, software complexity, complexity measurement, effort estimation, functional complexity measurement, task complexity.

1 Introduction

Software size is a key measure for many cost and effort estimation models. Models such as SLIM [20] and COCOMO [8] are based on lines of code (LOC).



However, LOC cannot be measured early in the software development process. Furthermore, evidence suggests that LOC can be very inaccurate because they depend on language and development tools [11].

The functional approach for software sizing and estimating development effort was proposed by Allan Albrecht in 1979 [5]. The measure of Albrecht - Function Point Analysis (FPA) - is well known because of its great advantages: independent of programming language and technology, easy to understand for client and user and applicable at an early phase of the software life cycle. This method measures software size in terms of *function points* that is the amount of functionalities of software delivered to users. The measurement process to quantify software size begins with identifying the elements (-type) of software from the software specifications. Then, these elements are weighted according to their complexity. Therefore, software sizing cannot be independent of software complexity measurement. Unfortunately, software complexity has not been precisely addressed in the method. The evaluation is basically subjective. It makes it difficult to be applied coherently and repeatedly. In addition, historically, FPA was developed in and designed for Management Information System (MIS). It is not appropriate to size software of other types, including real-time, scientific and embedded software. There have been many other proposals [3],[16],[21],[23],[27] that try to overcome these two weaknesses. These works focus on how to reduce the subjectivity in evaluating complexity and how to adapt FPA to capture new kinds of complexity associated with new types of software. However, no method explicitly proposes a model of software complexity. It is likely that these improvements are necessary but not sufficient because it is not clear what software complexity is. It is not sufficient because the quality of effort estimation from function points is still not good enough in practice. Many investigations [1],[4],[18],[24] have reported an average error rate of 40% or more. Researchers also call for a well-grounded theory approach for software complexity measurement [6],[17].

This paper focuses on how to improve software complexity quantification for better estimation of software development effort at the analysis phase. It considers software as a task – the task specified in the software specifications – and incorporated into software. Then, software functional complexity is defined as the complexity of the task and analyzed after a task complexity model. The task complexity model of Wood [25] is used as a theoretical guide to establish a software complexity framework and to propose complexity measures.

Wood's task model, on one hand, provides a global view from which software complexity can be analyzed systematically. That leads to define software complexity in both component complexity and system complexity. The component complexity refers to the complexity of the input, output data and the data manipulation of software components, while system complexity refers to data dependency between software components.

On the other hand, the model of Wood provides an objective view that allows analyzing task complexity from task characteristics. As a result, software complexity is analyzed and quantified objectively from the task that must be fulfilled by software, that is software functionality.



2 Background

2.1 Software complexity

The term “software complexity” is still not well defined. However, it is widely accepted that there are two main categories of software complexity: computational and psychological [26]. Computational complexity refers to algorithm efficiency in terms of the amount of time and memory needed to execute a program. While psychological (or cognitive) complexity refers to the human effort needed to perform a software task such as design, development or maintenance. In other words, psychological complexity is known as the difficulty experienced in understanding software or performing a task on software. It is interpreted by Zuse ([26], p.1) as “*the difficulty to maintain, change and understand software*”. It is often measured via the software characteristics that are associated with the resources needed to perform a task on software. In this research, software complexity is the complexity of the problem or the task that must be resolved by software. The task is defined in the specifications of software and is known as the functionality of software. If the task is complex, the software to execute the task is complex and it needs more effort to develop. In others words, software complexity refers to the difficulty of software functionality and it manifests in software development effort. A software complexity measure should be consistent with software development effort.

2.2 Function Point methods

In the literature, there are some measures that try to quantify the difficulty of software functionality in order to evaluate the development effort. Function Points Analysis (FPA) [4] is a “de facto” standard for software functional sizing. It claims to measure “software size” in terms of *function points* (FP), which is a dimensionless value derived from software functional specifications and is used as a measure of the relative size and complexity of software at an early stage of development - analysis and design. Software size is determined by identifying the software components as seen by the end-user: the inputs, outputs, inquiries, interfaces to other systems, and logical internal files. The components are classified as *simple*, *average*, or *complex*. These symbolic values are then scored and the total scores expressed in Unadjusted Function Points (UFP). After that, 14 general system characteristics (GSCs) are identified and weighted on an ordinary scale from 0 to 5. The sum of weights of these 14 factors is called the total degree of influence (DI). Then, the total function points of software (i.e. software size) are calculated by the formula: **FP = UFP * (0.65 + 0.01*DI)**.

From the measurement process of FPA, software size is evaluated via:

- The *complexity of software components*: Inputs, Outputs, Inquiries, Internal Files, and External Files, and
- The *complexity of system* characterized by 14 GSCs.



However, complexity is weighted subjectively from the experience of measurers. FPA does not propose explicitly a software complexity model. Many investigations for this problem have taken several directions:

- Establish a guideline to help determine complexity. This attempt aims to reduce subjective evaluations (e.g. [15]).
- Change the structure of FPA to objectively count complexity (e.g. Mark II [23], COSMIC-FFP [2],[3]).
- Add more characteristics into the set of system characteristics to adapt FPA to real-time, scientific and communication software (e.g. Feature Points [16], ASSET-R [21], 3D Function Point [27]).

The final goal of these improvements is to obtain a good measure that can well characterize the relative size and complexity of software and can be used to make a better prediction of software development effort. However, the lack of a functional complexity framework (or a functional size framework, if you want) makes it difficult to determine what needs to be measured. For example, Feature Points adds *algorithm type* to the five elementary types of FPA, while Mark-II adds six new system characteristics to the fourteen GSCs of FPA. Is there anything else to add? Obviously, a general framework that helps to systematically analyse software functional complexity is necessary to answer such a question. Moreover, it is also necessary to establish a measurement method that is well grounded on theory.

2.3 Task complexity

To study software complexity, software is taken as a task to be done. In fact, software can be conceptualized as an information processing system to fulfill a task in the real world. If a task is complex, the software to do the task is complex. This view is also adopted in function point methods and the research of Banker [7] on software complexity and maintenance effort.

Many investigations (see [9]) have proposed that task complexity may be characterized by information cues, which are information units to be treated while performing a task. Wood's model [25] is a sample of objective task complexity that is based on information cues. This model is built on three essential elements of a task: *products*, *required acts* and *information cues*. A task is conceptualized as an information processing in which input information (i.e. information cues) is manipulated in realizing the required acts to produce task outcomes (i.e. products).

Wood's model considers task complexity as a phenomenon with multiple dimensions including component, coordinative and dynamic complexity (Figure 1). *Component complexity* refers to information cues processed in the task and is quantified by the number of information cues. *Coordinative complexity* takes into account the forms and the strength of relationships (or interdependencies) between information cues and product outcomes. It is measured by either the number of alternatives to reach the task outcomes from



the information cues (interpreted as *Turning points*) or the number of previous acts that must be fulfilled before an act. *Dynamic complexity* arises from the changes of component and coordinative complexity over time.

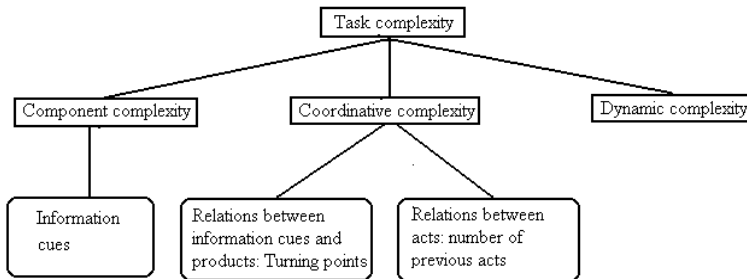


Figure 1: Wood's task complexity model.

3 Software model and software functional complexity framework

3.1 Software model

From a functional point of view, COSMIC (The Common Software Measurement International Consortium) considers software as a set of functional processes (see Figure 2), each of which is characterized by:

- Data movements that move data into and out the functional process, and
- Data manipulation that manipulates inputs to produce outputs.

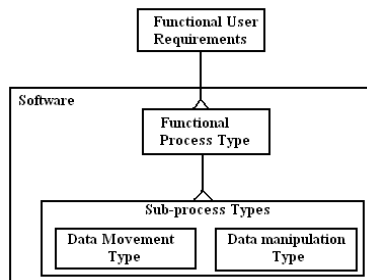


Figure 2: COSMIC software model [2].

In more detail, four types of data movements are defined: Entry, Read, Write and Exit. Each data movement moves one data group, which is a set of attributes of an entity, across the boundary of software into or out of a functional process. A data group coming into software from a user or a device is considered as an Entry. A data group moved out of software toward a user or a device is

considered as an Exit. A data group moved into or out a storage device is defined as a Write or a Read, respectively. COSMIC proposes a measure, called COSMIC-FFP, that quantifies software size by counting the number of data movements. However, it doesn't take into account in any way data manipulation. The COSMIC-FFP Measurement Manual [2] points out that COSMIC-FFP was not designed to take into account complex algorithms and that the method is only suitable for sizing "movement-rich" software type. Our research reuses the software functional model of COSMIC, but we try to measure objectively complexity in both data movement and data manipulation. In addition, the complexity of a whole system derived from relationships between functional processes is also investigated.

3.2 Software complexity framework and measures

The software model in Figure 2 can be mapped to Wood's task model. In fact, a functional process is taken as an elementary task that receives input information, manipulates the input to produce output information. Such a model of functional process totally coincides with Wood's task model, where the inputs are information cues of the task; the manipulations replace the required acts and the outputs are task product. From this mapping, software complexity can be considered as task complexity and, logically, it can be analysed after a task complexity model.

In this paper, Wood's task complexity is used as a framework for software complexity. However, the dynamic complexity is ignored because software is modelled as a set of static tasks that are described in the software specifications. The dynamic aspect of the task in its executions could be not taken into account.

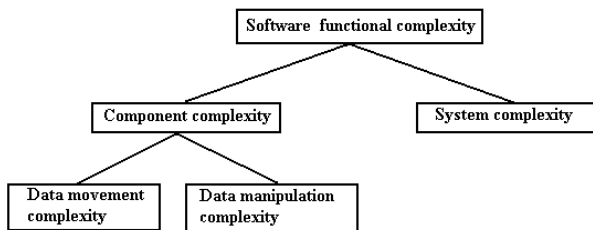


Figure 3: Software functional complexity framework.

The two remaining dimensions in the Wood's task model are used to define a general framework for software functional complexity as in Figure 3. Software functional complexity is characterized in three dimensions: Data movement complexity, data manipulation complexity and system complexity.

3.2.1 Data movement complexity

Data movement complexity corresponds exactly to component complexity in Wood's task complexity model (Figure 1). It refers to the inherent difficulty of the data coming into the system. Wood measures this kind of complexity by the number of information cues. In the software context, we consider a data

movement that introduces a new data group to the functional process as an information cue. Therefore, the number of new data groups moved in is defined as the data movement complexity of a functional process. The total number of new data groups in all functional processes is defined as data movement complexity of software. This measure is similar to COSMIC-FFP, but it does not count all data movements. It counts only the data movements that introduce something new to the functional process. An unchanged data group related to many data movements is counted only one time. The following rules can be used to identify and count the number of data groups:

- Use COSMIC-FFP rules [2] to identify functional processes and data movements. A movement can be an Entry, a Read, a Write or an Exit.
- Count one for each data group related to an Entry or a Read if these types of movement are assumed to bring something new into the process.
- Count one for each data group related to a Write or an Exit if the data group related to this movement has been modified since its last movement. If any movement of these types happens to a data group that has already been counted in a previous movement without any modification by the process; hence, it is not counted.

The data movement complexity is defined as the total number of data groups counted on all functional processes, herein, noted by NOD.

3.2.2 Data manipulation complexity

Data manipulation complexity refers to the inherent difficulty of the process that manipulates the inputs to produce expected outputs. This kind of complexity refers to the coordination between information cues and products in Wood's task complexity model. Wood uses the term *coordination* rather than manipulation. Here the term *manipulation* is used to conform to the software model in Figure 2. However, this term refers to the coordination between inputs and outputs rather than the algorithm used to manipulate inputs.

At the analysis phase, the data manipulation is modeled as a black box. The algorithm that manipulates input data to produce the desired outputs is not known. However, the specification of software must elicit what are the outputs desired, and the conditions to obtain such an output. In other words, the space of inputs (may be infinite) is partitioned into a finite number of classes. Each class corresponds to a desired output and is associated with a condition (membership condition of class). For example, consider the function that solves the equation " $ax+b=0$ ". The input space is partitioned into three classes with respect to the conditions: " $a=0$ and $b \neq 0$ ", " $a=0$, $b=0$ " and " $a \neq 0$ ". These three classes need to be manipulated separately to produce different outputs. Therefore, the number of classes or the number of conditions gives an indication of data manipulation complexity that refers to the coordination between inputs and outputs of the functional process.

Therefore, the number of conditions is defined as a measure of data manipulation, herein, noted by NOC. In practice, such a condition will become

one (or some) decision node in the flowchart of the program. As a result, NOC provides, as early as possible, an outline about the structure of the program. It can be seen as an “early” Cyclomatic number. In the literature, the Cyclomatic number of McCabe [19] is considered as a good measure for the complexity of a program structure. This measure is strongly related to the number of defects and difficulties encountered in understanding a piece of software. Therefore, an “early” Cyclomatic complexity measure like NOC is worth to consider for measuring complexity.

3.2.3 System complexity

System complexity refers to relationships between functional processes. Wood studied the coordination between *acts* in the task and proposed that the number of acts that must be achieved before an act is a measure of complexity. He also stated that, generally, this kind of complexity should deal with essential relationships between components of the task [25].

In a software context, it is believed that the data dependency between components is an essential element of system complexity. In Function Point Analysis [4], the complexity in relationships between software components can be found in some characteristics of system, for example “data communications” and “distributed processing”. However, they are evaluated subjectively. Another approach to measure system complexity is based on the information theory. Davis and LeBlanc [12] have developed an entropy-based measure for program structure. Data dependency (data connection) and data control (control connection) between “chunks” of code are studied separately. They found that data dependency and data control have a strong effect on development time.

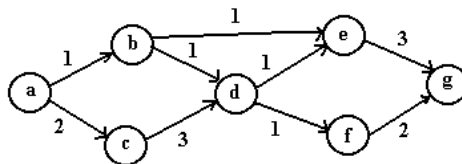


Figure 4: An example of a directed-weighted graph representing the data dependency between functional processes.

Here, we propose an entropy-based measure of system, noted by EOS, that is similar to the measure of Davis and LeBlanc for system complexity. The data dependency between functional processes is modeled as a directed weighted graph. Two processes have a data dependency if an output of one process becomes an input of another. A node on the graph is a process. An arc represents a data group in data dependency between two processes, and the weight of arc represents the number of data groups in the data dependency. Figure 4 gives an example of such a graph.

To calculate entropy, all functional processes are classified in equivalent classes. Two processes belong to the same equivalent class if and only if they have the same in-degree and out-degree, which is the sum of the weight of arcs coming into and out of the process, respectively. For example, the graph in

Figure 4 has the following equivalent classes: {a}, {b,f}, {c,e}, {d}, {g}. Entropy is then defined by: $EOS = -\sum p_i \log(p_i)$, where $i=1..n$; n is the number of equivalent classes; and p_i is the possibility of a process falling into the i^{th} class. For example, the entropy of the system modeled as the graph in Figure 4 is:

$$EOS = -[1/7 \log 1/7 + 2/7 \log 2/7 + 2/7 \log 2/7 + 1/7 \log 1/7 + 1/7 \log 1/7] = 2.24.$$

According to information theory [22], the entropy of system reflects the disorder of the system. If the degree of disorder is high the system is more complicated.

4 Empirical investigation

In this section, an empirical investigation which aims to test if the proposed framework and the three complexity measures in the previous section are valid is reported. The term “valid” means that the proposed measures (NOD, NOC and EOS) are relevant for measuring software complexity that manifests in development effort. The test will be based on 15 software maintenance projects provided by a professional software company. These projects are of the same type: Management Information System (MIS). The homogeneity of the sample data, on one hand, may allow generalizing the result of the experimentation or applying the result in a similar environment. On the other hand, it reduces the effect of development environment on effort. The data provided by the company are the software specifications and the effort (the total man-months) paid to realize the maintenance projects.

The three measures will be tested to see whether they are well correlated with development effort and whether they can be used to make a good prediction of development effort. The correlation between real development effort and NOD, NOC and EOS is analyzed. The linear regression and multiple regressions will be used for this analysis purpose. A measure is relevant for software complexity if it is a single indicator of development effort or it is combined with other measures in a compound value, which is well correlated with development effort.

For each project in the data collection, we applied the rules proposed in section 3 to count NOD, NOC and EOS. However, only **new** or **modified functions** in the maintenance project were counted. **Deleted functions** and **re-test functions** are not counted. This restriction aims to generalize the result of the experimentation, which is based on maintenance projects, for new development projects. Due to the lack of detail in specification documents, it was difficult to establish relationships between functional processes of some software. The system complexity (EOS) of those are marked by ** (two stars) in Table 1.

Table 1 provides the results of the counting of 15 maintenance projects used in the experimentation. *Effort* column represents the real effort (man-months) to execute the maintenance projects. It was provided by project managers in the software company. *NOD*, *NOC* and *EOS* are the three new measures investigated in this paper. They were counted by the first author of this paper and revised by other members in the research team.



Table 1: The data set of 15 projects.

#	Effort (Man-Months)	NOD	NOC	EOS
1	1.6	18	3	0
2	1.79	12	4	0
3	2.56	24	0	**
4	2.72	23	3	0
5	6.3	39	12	0.67
6	6.45	31	17	**
7	7.15	51	0	**
8	7.8	60	10	**
9	7.84	36	6	0.44
10	9.3	10	74	**
11	9.5	54	26	0.48
12	12.6	106	0	**
13	14.4	116	8	**
14	20.7	71	41	0.68
15	21.7	119	24	0

Table 2: (Multi-)correlations between effort and others variables.

Correlation	Number of observations	Correlation determination (R^2)
Effort vs. NOD	15	65%
Effort vs. NOC	15	18%
Effort vs. EOS	8	11%
Effort vs. NOD, NOC	15	89%
Effort vs. NOD, NOC, EOS	8	95%

Table 2 shows the results of analyzing the correlation between effort and other variables, including NOD, NOC and EOS. We can see that there is a weak correlation between effort and any single measure. However, effort is strongly correlated with NOD and NOC ($R^2=89\%$). It is also strongly correlated with NOD, NOC and EOS ($R^2=95\%$). The multiple regression formula is written in (1) and (2), respectively.

$$\text{EFFORT} = 0.14 \cdot \text{NOD} + 0.15 \cdot \text{NOC} - 0.94 \quad (1)$$

$$\text{EFFORT} = 0.15 \cdot \text{NOD} + 0.21 \cdot \text{NOC} + 0.75 \cdot \text{EOS} - 1.4 \quad (2)$$

The test of null hypothesis at the level of signification of 5% confirms that the correlations between of NOD, NOC and effort in (1) are significant; the correlation between EOS and effort in (2) is not significant. However, note that the correlation (2) is established on only 8 observations.

By analyzing the performance of these prediction models in terms of *magnitude of relative error* (MRE) and *mean magnitude of relative error* (MMRE), we can see that NOD and NOC are good measures of the software complexity that is manifested by effort of development. The first line in Table 3 shows that 73% of software projects can be estimated with an error rate of less than 25%; 93% of projects have an error rate of estimation of less than 35%; and none of the projects have a predicting error rate of more than 50%. The mean magnitude of relative error is very low, 15% only. The performance of model (2) is nearly the same as that of (1). However, the error rate tends to fall to less than 35% in 100% of cases.



Table 3: Analyzing performance of the prediction model (1) and (2).

Estimation model	Pred(0.25)	Pred(0.35)	Pred(0.50)	R ²	MMRE
(1) using NOD and NOC	73%	93%	100%	88%	15%
(2) using NOD, NOC and EOS	62%	100%	100%	95%	18%

The analysis on the data sample above leads to some conclusions:

- A single measure NOD, NOC or EOS is not enough for measuring software complexity. However, combinations of the three measures are relevant for measuring complexity. In fact, they can be combined together in a multiple correlations with development effort.
- NOD and NOC are good enough for predicting development effort. The combination of them in (1) makes fairly good prediction. The performance of this model is much better than those obtained using FPA or COSMIC-FFP. The average error rate on the data sample is 15% only. While the average error rate of other models using FPA or COSMIC-FFP is more than 40% [1], [4],[24].

Moreover, the quality of the prediction model (1) reaches the threshold acceptable in practice. Conte [11] propose that an acceptable prediction model must provide an estimation near to the real value, with a difference of less than 25% in 75% of cases (i.e. $PRED(0.25)=75\%$). The prediction model (1) has reached 73%. This performance is significantly better than those of prediction models using FPA or COSMIC-FFP reported in [1], [4], [24], in which $PRED(0.25)$ is just about 50%.

- Integrating EOS with NOD and NOC may provide a better correlation (correlation coefficient $R^2=95\%$). It can make a stable prediction because the error rate tends fall to less than 35%. Actually, due to the fact that data sample used to analyze EOS is small, it makes it difficult to draw a conclusion about the effect of this measure on development effort. However, if the trend of error rate above holds, using EOS along with NOD and NOC might give a good and stable prediction of development effort. It is worth to continue investigations on EOS.

5 Conclusion

This paper has developed a software complexity model after a task complexity model. The Wood's task complexity model was introduced as a theoretical framework for measuring software functional complexity. Three measures were proposed with respect to three aspects of software: data movement, data manipulation and relationships between system components. These three measures also correspond to three aspects of task complexity in Wood's model: component complexity, coordination between information cues and products, and coordination between acts.



Software complexity is considered as a vector of 3 dimensions rather than a single value. It was not intended to establish a single value called “complexity of software” because the relationship between the three measures is not clear. Nevertheless, a combination of NOD, NOC and EOS can be used as an abstract measure of “software complexity” that refers to an attribute of software. For example, the combination of NOD, NOC and EOS in formula (2) can be considered as a “complexity measure” that refers to software development effort. The sum in the right hand side of formula (2) is not development effort. It is a measure of complexity and it well correlates with development effort. In other words, it is a complexity measure used for estimating development effort.

The empirical test in the paper has confirmed that NOD, NOC and EOS are relevant measures of complexity. They can be used to make a good prediction of development effort. NOD and NOC are also good enough for this purpose. In addition, EOS may be used associated with NOD and NOC to make a better prediction. However, due to the fact that the size of the data sample used was small, it is suggested that more empirical test on EOS are required.

References

- [1] Abran A., Silva I., Primera L., Field Studies Using Functional Size Measurement in Building Estimation Models for Software Maintenance, *Journal of Software Maintenance and Evolution: Research and Practice*, 2002, Vol.14, pp. 31-64.
- [2] Abran, A.; Desharnais, J.-M.; Oligny, S.; St-Pierre, D.; Symons, C., COSMIC FFP - Measurement Manual – the COSMIC Implementation Guide for ISO/IEC 19761:2003, January, 2003.
- [3] Abran A., Desharnais J.M., Maya M., St-Pierre D., Bourque P., Design of a functional size measurement for Real-Time Software, research report No 13-23, 1998.
- [4] Albrecht A., Gaffney J., Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, November 1983.
- [5] Albrecht A., Measuring Application Development Productivity, presented at the IBM Applications Development Symposium, 1979.
- [6] Baker A.L., Beiman J.M., N. Fenton, Gustafson D.A., Melton A., Whitty R., A Philosophy of Software Measurement, *Journal of System Software*, Vol. 12, pp. 277-281, 1990.
- [7] Banker R.D., Davis G.B, Slaughter S.A, Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study, *Management Science*, vol. 44, no 4, 1998, pp. 433-450.
- [8] Boehm B.W., Clark B., Horowitz et al., Cost Model for Future Life Cycle Processes: COCOMO 2.0, *Annals of Software Engineering* 1 (1), 1-24, Nov. 1995.
- [9] Campbell D.J., Task Complexity: A Review and Analysis, *Academy of Management Review*, Vol. 13, pp. 40-52, 1988.



- [10] Card D.N., Glass R.L., Measuring Software Design Quality, Prentice Hall, 1990.
- [11] Conte, S.D., Dunsmore, H.E., and Shen, V.Y., Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, CA, 1986.
- [12] Davis J.S., LeBlanc R.J, A study of the applicability of complexity measures, IEEE Transactions on Software Engineering, Vol.14, No.9, September 1988, pp. 1366-1371.
- [13] Harrison, W., An Entropy-Based Measure of Software Complexity, IEEE Transactions on Software Engineering, Vol. 18, No. 11, 1992, pp. 1025-1029.
- [14] Henry S.M., Kafura D.G., Software Structure Metric Based on Information Flow, IEEE Transactions on Software Engineering, no. 5, pp.545-522, September 1981.
- [15] IFPUG (International Function Point Users Group), Function Point Counting Practices Manual, Release 4.1.1, 2001.
- [16] Jones C., Applied Software Measurement – Assuring Productivity and Quality, New York, McGraw-Hill Inc, 1991.
- [17] Kearney, K.J., Sedmeyer R.L, Thompson W.B, Gray M.A., Adler M.A., Software Complexity Measurement, Communications of the ACM, Vol. 29, No. 11, November 1986.
- [18] Kemerer C.F., An Empirical Validation of Software Cost Estimation Models, Communication of ACM, Vol. 30, No. 5, pp. 416-429, 1987.
- [19] McCabe T.J., A Complexity Measure. IEEE Transactions of Software Engineering, Volume SE-2, no. 4, pp.308-320, December 1976.
- [20] Putnam L.H, A general empirical solution to the macrossoftware sizing and estimating problem, IEEE Transactions of Software Engineering, SE-4(4), pp. 345-361, 1978.
- [21] Reifer D.J., Assert-R: A Function Point sizing Tool for scientific and Real-Time Systems, Journal of systems software, vol. 11, pp. 159-171, 1990.
- [22] Shannon C.E., Weaver W., The Mathematical Theory of Communication, University of Illinois Press, Urbana, 1949.
- [23] Symons C., Function Point Analysis: Difficulties and Improvements, IEEE Transactions on Software Engineering, Vol. 14, No. 1, January, 1988.
- [24] Tran-Cao D., Levesque G., Maintenance Effort and Cost Estimation Using Software Functional Sizes, International Workshop on Software Measurement, Montreal, Canada, September 2003.
- [25] Wood R.E., Task Complexity: definition of the construct, organisational behaviour and human decision processes, Vol.36, 60-82, 1986.
- [26] Zuse H., Software Complexity Measures and Methods, Walter de Gruyter, Berlin – New York, 1991.
- [27] Whitmire S.A., An Introduction to 3D Function Points, Software development, pp. 43-53, 1995.

