Computing and Informatics, Vol. 34, 2015, 959–972

QUANTIFYING PRODUCTIVITY OF INDIVIDUAL SOFTWARE PROGRAMMERS: PRACTICAL APPROACH

Mehmet Suleyman Unluturk

Department of Software Engineering, Yasar University
Universite Cad. No. 37-39 Bornova, 35100, Izmir, Turkey
e-mail: mehmet.unluturk@yasar.edu.tr

Kaan Kurtel

Department of Software Engineering, Izmir University of Economics
Sakarya Cad. No. 156 Balcova, 35330, Izmir, Turkey
e-mail: kaan.kurtel@ieu.edu.tr

Abstract. Software measurement is a crucial part of a good software engineering. Software developers quantify the software to see if the use cases are complete, if the analysis model is consistent with requirements and if the code is ready to be tested. Software project managers assess the software process and the software product to determine if it is going to be finished on time and within budget. Customers evaluate the final product if it meets their needs. Overall, the main purpose of software engineering is to make software systems controllable and foreseeable, activities with a solid method rather than intuitional, complicated or unprincipled. Software measurement studies are about quantifying the software engineering entities and attributes, both of which aim to support software development efforts and quality improvement. In this paper, we quantify a set of relationships using the current size, defect and object-oriented software metrics practically and pragmatically. Our paper proposes a method to measure the productivity of individual software programmers. Furthermore, this method provides a common opinion for understanding, controlling and improving the software engineering practices.

Keywords: Programmer productivity quantification, personal software process, software measurement

1 INTRODUCTION

According to Tom DeMarco [1], "*you cannot control what you cannot measure*". Many industrial product companies should measure their own productivity and then, optimize their operations accordingly. Despite the importance of measuring programmer's productivity in the software industry, software companies are showing little enthusiasm to spend effort and allocate less resource to measurement. The major reason for the project leaders to be reluctant in this subject are possible difficulties and potential unsuccessful results. In detail, during the software development, we can encounter some typical problems regarding measurement and evaluation as given below [2–5]:

1. Measurement is a process, and thus it is time consuming, not well understood, and difficult to apply.
2. When the measurement process is prolonged, the already obtained information quickly loses its value and as a result continuous improvement cycle slows down.
3. Software programmers consider that software development is more important than measurement and collecting data.
4. It is in the nature of the software industry to produce software programs and launch them to the market very quickly.

To conclude, according to our observations that are related to the software development practices, the engineers do not want or are not very enthusiastic about measuring their software products, except for some of the critical system applications such as defense, health-care, and finance. It is due to these dynamics that the concept of measurement in software development may sometimes be ignored.

These dynamics encourage us to find a way to measure the performance of a software programmer in a practical and relatively cost-effective way. In other words, the results must provide motivation and volition to the software project leaders.

In this paper, we will try to answer the following question: How do we measure the programmer's productivity efficiently and effectively? In order to answer this question, we present a new judgmental-based measurement function as a fresh approach where we pragmatically estimate the productivity of an individual software programmer.

This study includes three sections. The second section talks about the planning of the measurement process that includes the measurement function for a software developer's productivity, and the third section represents empirical calculations into the model projects. The last section depicts the conclusion and the future work.

2 PLANNING THE MEASUREMENT PROCESS

The measurement process defines three data collection points and related measurement activities for four key measures during a single project that are strongly related

by a part of the software development life cycle. The three data collection points are represented in Roman numerals and different color pattern elements in Figure 1. At the first data collection point, we capture the LOC and the net task hour's data during the coding effort of software developers that includes code changes, deletes and reuses. At the second data collection point there are defects that might appear during in-house testing after the delivery of codes to the testing team. At this stage, the testing squad tests the codes, and at the end of the testing process, the number and/or types of defects are collected from the software initial and intermediate releases. These three measures (LOC, time and defects) are iteratively obtained and cumulatively calculated. Fourth and the last measure, named weighted methods, are collected as the weighted methods just before the eventual release of the project (Figure 1) which shows the delivered codes' difficulty level.

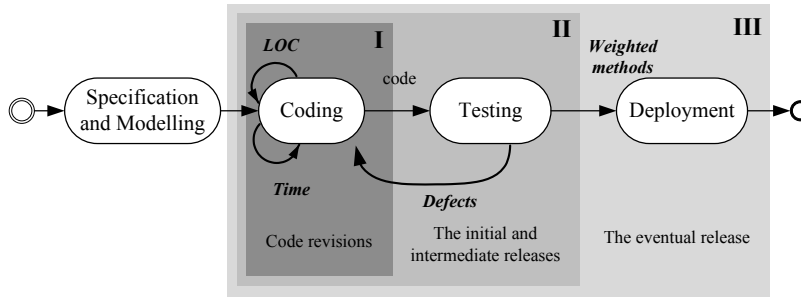

Figure 1. The iterative measurement activities and the data collection points

Input: Net Task Hours

The Net Task Hours for an individual programmer is represented in formula (1) that calculates the difference between hours spent for coding and hours spent for interruption time consisting of hours spent for answering and composing e-mails, calling, etc. anything other than coding.

$$\text{Net task hours} = \text{Task hours} - \text{Hours spend for interruption time} \quad (1)$$

Output 1: Lines of Code

The line of code (LOC) is the first simple measure in our work. The LOC counts provide information about the source size of a software program and it is a valuable metric for comparing the amount of effort as well as estimating the software programmer's productivity. The LOC is the oldest and the widely used method in complicated, real time or embedded systems. Furthermore, the LOC is used when organizations have large LOC-based historical data and the developer is comfortable with the LOC-based measurement [6, 7]; in addition it allows a simple comparison with data from many other projects. This is especially important for companies where they can easily compare current project

data with the earlier project data or can hold a record about its programmers' efforts on project. These development effort records can also be kept for later use and the LOC is a stable productivity measure for any company.

Output 2: Weighted Defects

The second output is the weighted defects (Figure 1). A software defect is defined as a product anomaly, such things as omissions and imperfections found during the early life cycle phases and faults contained in software sufficiently mature for test or operation [8].

We identify and classify defects during the testing phase. We do not collect any defect data during the construction, because a software programmer makes many errors or faults in the construction phase and fix these errors during the coding phase. If the defects are too many, then time for collecting these measures might increase. Consequently, it affects negatively the productivity of a programmer. Another benefit of measuring defects in the deployment phase is that the programmer can eliminate the potential negative impacts of data collection processes during the coding; he/she can focus on the quality of the codes.

In this proposal, the software defects are classified into three categories, namely serious, medium and trivial, according to [8] (Table 1).

Defect type	Weights
Serious	10
Medium	3
Trivial	1

Table 1. Relative defect weights

We will calculate the weighted defects according to the below equation:

$$\begin{aligned} \text{Weighted\_Defects} \;=\; & 10 * \text{total \# of Serious\_Defects} \\ & + 3 * \text{total \# of Medium\_Defects} \\ & + \text{total \# of Trivial\_Defects.} \end{aligned} \tag{2}$$

Then, we use this quantity and divide it by the LOC to find the weighted defects density:

$$\text{Weighted\_Defects\_Density} = \text{Weighted\_Defects/LOC.} \tag{3}$$

We define the quality of code created by this individual programmer as:

$$\text{Quality} = 1 - \text{Weighted\_Defects\_Density.} \tag{4}$$

If Quality calculation becomes negative because the programmer's weighted defects are more than his/her LOC, then we will accept that this calculation yields to zero value. We also further define the LOC as the quantity factor:

$$\text{Quantity} = \text{LOC}. \tag{5}$$

However, we derive the cumulative formula in the below Equation (6) when the iterative software development life cycle (in Figure 1) is taken into consideration:

$$A = \sum_{i=1}^{n} (\text{Quality}_i * \text{Quantity}_i) / \text{Net Task Hours}_i \tag{6}$$

where n is the number of iterations. The iteration includes the developer's coding phase and the testing team's testing phase of the developer's code (Figure 1). During the iteration i, the net task hours spent for the coding phase for an individual programmer is given as Net Task Hours_i.

Output 3: Weighted Methods

The third output that we want to focus on is the number of methods that are created by an individual programmer (Figure 1). We calculate the weighted methods of an individual programmer which is important for project managers to know how difficult or complicated the created code is. An important point for this measure is the collection point. We collect data just before the deployment process. The methods created by an individual programmer can be divided into five classifications [9]:

- Constructors – methods which instantiate an object.
- Destructors – methods which destroy an object.
- Modifiers – methods that change the state of an object. A modifier method will contain references to one or more of the properties of its own class or another class.
- Selectors – methods that access the state of an object but make no changes to this state. These are the methods used to provide public access to the data that is encapsulated by the object.
- Iterators – methods that access all parts of an object in a well-defined order. These may be used to visit each member in a collection of objects, performing the same operation on each member.

We will use the below relative weighting table to calculate the weighted methods for the methods created by an individual programmer (Table 2). Where N is the total number of methods for an individual programmer.

$$\begin{aligned} \text{Weighted\_Methods} \;=\; & (3 * (\text{total\# of constructors} + \text{total\# of destructors}) \\ & + 5 * \text{total \# of selectors} \\ & + 9 * \text{total \# of iterators} \\ & + 15 * \text{total \# of modifiers}) * (1/N) \end{aligned} \tag{7}$$

where N is the total number of methods for an individual programmer.

Method type	Relative Weights
Constructor	3
Destructor	3
Selector	5
Iterator	9
Modifier	15

Table 2. Relative method weights

The Productivity Measurement Formula

The productivity measurement function is the combination of A (Equation (6)) and Weighted_Methods (Equation (7)) and is given below:

$$\text{Software Engineering Productivity Index} = A * \text{Weighted\_Methods}. \quad (8)$$

One can deduce from the above equation that the productivity can be further improved by decreasing the # of weighted defects. If # of weighted defects gets close to zero, then (1 − # of Weighted_Defects/LOC, Equation (4)) gets close to 1 which is the maximum value for the Quality, then A (Equation (6)) will be increased. The next section provides empirical data on actual use. Two senior projects were used to do the productivity calculations. In the final section, the conclusion and the future work are presented.

3 EMPIRICAL CALCULATIONS

3.1 Model Projects

After providing the productivity formula in Section 2, the empirical calculations were carried by the senior project group of students under the supervision of the authors. Two projects were done by the same programmer and the productivity of this programmer for these projects was calculated by three senior level students. Due to the difficulty of getting expert opinions (we did not need a high level of engineering experience to do the productivity calculation of these two projects), we decided to use three senior level students. Moreover, in literature, there are many projects in which students were used to do empirical calculations [10–12].

The first project was developed as an M.Sc. senior project in the game programming domain that was named 1453, and it was inspired by the early centuries of the Ottoman Empire, which depicts the conquest of Istanbul in 1453. The game runs on MS Windows, and was developed using C++ programming language. The second project was also developed by the same student as a term project in the health care domain that was named 'Hasta (Patient) la vista'. The aim of this project is basically to manage the records of health care institutions and their journal system. The project was also developed using C++ programming language for MS Win-

dows operating systems. These two projects were coded by the same programmer and were written in the same programming language, C++.

3.2 Data Collection

In this phase, we have defined the data collection and analysis procedures under the guidance of the ISO/IEC 15939 Software engineering measurement process standard [13]. During this phase, three senior level students worked in the measurement process of these two projects.

The time, LOC, defect and method recording logs were obtained under the guidelines presented by the PSP [14]. The following 8 tables present the time, LOC, defects and methods data for both projects. The LOC data was counted automatically by Imagix 4D software tool [15] as each individual part of the program was committed by the programmer, and the screenshot for the software tool is also shown in Figure 2. In [16], a neural network methodology is utilized in prediction of defect density of subsequent software product releases.

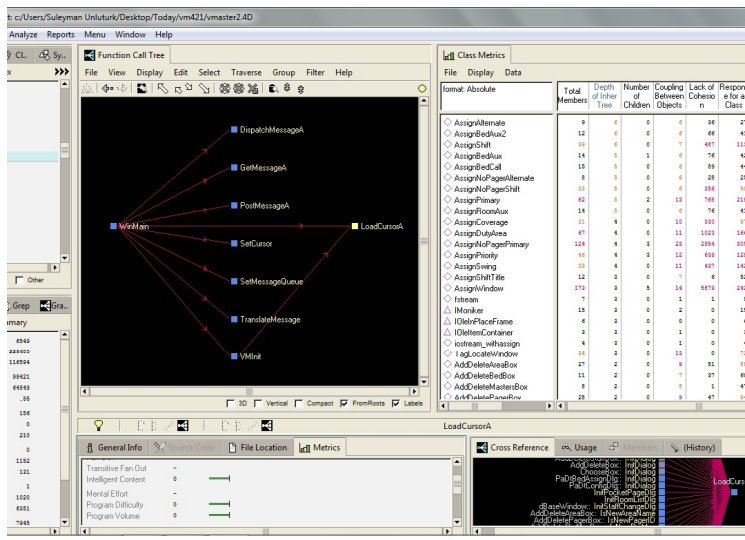

Figure 2. An Imagix 4D screenshot for the project 1453

In Tables 3 and 4, time recording sample logs are given. Interruption time was included because phone calls or checking e-mails and so on come accross during the projects development. We can notice that the same programmer spent more time in the game development than in the hospital information system application. Neither the programmer, nor the authors had previous experience with the game development and this increased the number of hours spent for the 1453 project.

In Tables 5 and 6, the LOC counts were given for each project. Hasta la vista project was written from scratch and the programmer did not reuse any lines of code

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2006/05/05–2006/05/31
Platform: MS Windows	Project name: Hasta la vista
Programmer: Kaya Oguz	Language: C++

Commit Name	Date	Task Hour	Interruption Time	Delta Time
R45	2006/05/12	60 min	9 min	51 min
R46	2006/05/13	30 min	5 min	25 min
R47	2006/05/13	30 min	5 min	25 min
R49	2006/05/13	30 min	5 min	25 min
R50	2006/05/13	30 min	5 min	25 min
R52	2006/05/13	30 min	5 min	25 min

Table 3. The PSP time recording log for project Hasta la vista

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2007/03/06–2007/03/10
Platform: MS Windows	Project name: 1453
Programmer: Kaya Oguz	Language: C++

Commit Name	Date	Task Hour	Interruption Time	Delta Time
R17	2007/03/06	120 min	18 min	102 min
R20	2007/03/06	120 min	18 min	102 min
R21	2007/03/08	120 min	18 min	102 min
R22	2007/03/08	120 min	18 min	102 min
R23	2007/03/10	180 min	27 min	153 min
R24	2007/03/10	120 min	18 min	102 min

Table 4. The PSP time recording log for project 1453

from anywhere else. On the other hand, in the game development, he reused some third-party game development libraries. These LOCs are shown in Table 6 (LOC-Reused: 1207). Even if there were third party libraries available, the programmer still ended up coding as many lines of code as he did in the Hasta la vista project. As the complexity of the program increases, the LOC count for the program increases as well.

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2006/05/13–2006/05/13
Platform: MS Windows	Project name: Hasta la vista
Programmer: Kaya Oguz	Language: C++

Commit Name	Date	LOC-Added	LOC-Reused	LOC-Modified
R46	2006/05/13	67		171
R47	2006/05/13	64		75
R49	2006/05/13	24		7
R50	2006/05/13	35		111
R52	2006/05/13	48		28

Table 5. The PSP LOC recording for project Hasta la vista

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2007/03/05–2007/03/11
Platform: MS Windows	Project name: 1453
Programmer: Kaya Oguz	Language: C++

Commit Name	Date	LOC-Added	LOC-Reused	LOC-Modified
R17	2007/03/06		1 207	
R20	2007/03/06	1 889		
R21	2007/03/08	649		4
R22	2007/03/08			22
R23	2007/03/10	125		77
R24	2007/03/10			99
R25	2007/03/11			20
R27	2007/03/11			127
R28	2007/03/11	3 381		

Table 6. The PSP LOC recording for project 1453

In Tables 7 and 8 depict some of the defects in each project. The programmer had some problems with the database programming and Hasta la vista project used the database heavily; on the other hand, 1453 project did not use a database at all. As a result, most defects have come from the database as indicated in Table 7. In 1453 project, most defects were related to the screen (Table 8), and the 1453 game should be able to draw the screen objects in real-time.

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2006/05/05–2006/05/30
Platform: MS Windows	Project name: Hasta la vista
Programmer: Kaya Oguz	Language: C++

Number	Beginning Date	Fix Time	Priority	Description
1	2006/05/07	2006/05/11	Serious	Compilation problem
2	2006/05/07	2006/05/11	Medium	Windows setting problem
3	2006/05/14	2006/05/17	Trivial	Field information
4	2006/05/14	2006/05/18	Medium	Search error
5	2006/05/18	2006/05/20	Medium	History error
6	2006/05/19	2006/05/20	Trivial	Invalid character problem
7	2006/05/19	2006/05/22	Medium	Lack of delete bottom problem
8	2006/05/28	2006/05/30	Medium	Toolbar problem

Table 7. The PSP defect recording log for project Hasta la vista

In Tables 9 and 10, the method types for each project are given. We can see that modifiers and selectors are nil for the game development since they all had come from the third party software components. On the other hand, since the programmer did everything by himself and did not use any third party component for the Hasta la vista project, modifiers and selectors have corresponding values in Table 9 for Hasta la vista.

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2007/04/10
Platform: MS Windows	Project name: 1453
Programmer: Kaya Oguz	Language: C++

Number	Beginning Date	Fix Time	Priority	Description
1	2007/04/03	2007/04/10	Medium	Combobox problem
2	2007/04/03	2007/04/10	Medium	TextView scroollbar problem
3	2007/04/03	2007/04/10	Trivial	Scrollbar color problem
4	2007/04/03	2007/04/10	Medium	Interruption problem
5	2007/04/03	2007/04/10	Medium	Mouse cursor problem

Table 8. The PSP defect recording for project 1453

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2006/05/05-2006/05/31
Platform: MS Windows	Project name: Hasta la vista
Programmer: Kaya Oguz	Language: C++

Function Name	Constructor	Destructor	Modifier	Selector	Iterator
Database			2	6	
Main				1	
mainWindow			2		
statsTabs				1	
WindowInspection				1	
WindowQuickRecord				2	
WindowPatient				2	
historyTabs				2	
searchTabs				3	
linkTabs				2	

Table 9. The PSP method recording for project Hasta la vista

Student(s): D. Eskici, I. Erdonmez, T. B. Kurtoglu	Date: 2007/05/24
Platform: MS Windows	Project name: 1453
Programmer: Kaya Oguz	Language: C++

Function Name	Constructor	Destructor	Modifiers	Selector	Iterators
Application	1				1
BaseBuilding	1				
BaseGraphObject	1				
BaseObject	1				
BaseUnit	1				1
Buildings	4				
BOJanizary	2	1			
BOHome	1				

Table 10. The PSP method recording for project 1453

The measurement results of the productivity measures – details are given in the Equations from (1) to (8) – are presented in Table 11. It is noticed that the net task hours is less for the 1453, because of the third party software components used by the programmer in the 1453 project. Since the game development was complicated, the LOC count was still high for the game development. Because of the weaknesses in the database programming area, the number of defects in the Hasta la vista project was higher than that of the 1453 project. Hasta la vista had also more method count than that of the game development since the programmer did the Hasta la vista from scratch and did not use any third party software components.

Base Measure	Unit	Data Collection and Calculation Method	Hasta la vista	1453
B1-Net task hours	Hour	Declaration of programmer. PSP used.	3 264	1 574
B2-Lines of code	Line	By using Imagix 4D and PSP together.	8 504	5 869
B3-Weighted defects	Defect	The defects are classified into three categories and counted by using PSP. (Equation (5))	42	30
B4-Weighted methods	Method	The methods are classified into five categories and counted by using PSP. (Equation (10))	3.23	6.82

Table 11. The base measures obtained from the projects

We computed the derived measures taken from the measurement results exhibited in Table 12. Following that, the final values obtained are shown in Table 12. We took i as one while calculating A (Equation (6)). The programmer did not go through any other iteration to fix the defects in these two projects since the authors were satisfied with the performance of the programs.

Derived Measure	Unit	Data Analysis and Interpretation	Hasta la vista	1453
D1-Weighted Defect Density	B3/B2	Close to 0 is better	0.0049	0.0051
D2-Quality	1-(B3/B2)	Close to 1 is better	0.9951	0.9949
D3-Productivity	D2*B2*B4/B1	The higher is the better	8.38	25.29

Table 12. Derived measures

3.3 Evaluate of the Measurement

According to Table 12 we can see that the programmer was three times more productive in the second project (1453) than in the first project (Hasta la vista). Reusing third party software components in the 1453 project caused the programmer to have less number of net task hours, LOC, weighted defects and methods compared to those of the Hasta la vista project. This increased the productivity of the programmer for the 1453 project. Furthermore, according to Table 7 (Defect recording log), the programmer had some defects in the database programming, and he needed some training on the field.

4 CONCLUSIONS AND FUTURE WORK

Measurement of the software productivity provides an important information about the software product and the productivity of an individual software engineer. The benefits of the formula are given below:

- Both embedded and big software companies can use this formula.
- This measure provides some advantages such as it can be done with different people and still generate the same result.
- Companies can compare the existing productivity position and the future position, because they usually use the same high-level languages in their projects.
- This formula can be used in any object-oriented software development project.
- This formula can give information about the reliability of the software product.

For future work, the formula will be applied by two well-known companies in Turkey. These companies are specialized in the application software area. The first company is in the home appliances sector, and about 50 programmers provide service to the international electronic manufacturers for embedded systems. The second cooperation provides us a number of services that add value to its customer's logistics services by 35 software professionals; also, this company changes its hardware platforms and software applications, and manages a large scale migration project. Each company has development, testing, maintenance and measurement teams, and uses the Microsoft platform and tools, and also C++ programming language. This collaboration will provide us an opportunity to improve the applicability, practicability, accountability, and cost-effectiveness of the formula to further measure the programmer's productivity.

Acknowledgments

We would like to thank Ph.D. candidate Mr. Kaya Oguz for sharing his projects. We would like to thank Engineers Demet Eskici, Irem Erdonmez and Tomris Beril Kurtoglu for their kind assistance with the data. We would also like to thank Mr. John Blattner from Imagix 4D for sharing the software.

REFERENCES

[1] DeMarco, T.: Controlling Software Projects. Yourdon Press, New York, 1982.

[2] IEEE Std. 1061. IEEE Standard for a Software Quality Metrics Methodology. IEEE, NY, USA 1988.

[3] Ebert, C.—Dumke, R.—Bundschuh, M.—Schmietendorf, A.: Best Practices in Software Measurement. Springer-Verlag, Berlin, 2005.

[4] Zuse, H.: A Framework of Software Measurement. Walter de Gruyter, Berlin, 1998.

[5] Abran, A.: Software Metrics and Software Metrology. Wiley-IEEE Press, 2010.

[6] Galorath, D. D.—Evans, M. W.: Software Sizing, Estimation, and Risk Management. Auerbach Publications, 2006, pp. 63.

[7] Olek, L.—Ochodek, M.—Nawrocki, J.: Enhancing Use Cases with Screen Designs. A Comparison of Two Approaches. Computing and Informatics, 2010, Vol. 29, No. 1, pp. 3–25.

[8] IEEE Std. 982.1-2005, IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.

[9] Minkiewicz, A.: Measuring Object-Oriented Software with Predictive Object Points. Proceedings Applications in Software Measurement (ASM '97), Atlanta, 1997.

[10] Basili, V. R.—Shull, F.—Lanubille, F.: Building Knowledge through Families of Experiments. IEEE Transactions on Software Engineering, Vol. 25, 1999, No. 4, pp. 456–473.

[11] Kitchenham, B.—Pfleeger, S.—Pickard, L.—Jones, P.—Hoaglin, D. C.—El-Emam, K.—Rosenberg, J.: Preliminary Guidelines for Empirical Research in Software Engineering. IEEE Transactionson Software Engineering, Vol. 28, 2002, No. 8, pp. 721–734.

[12] Cuadrado-Gallego, J. J.—Sicilia, M. A.: An Algorithm for the Generation of Segmented Parametric Software Estimation Models and Its Empirical Evaluation. Computing and Informatics, Vol. 26, 2007, No. 1, pp. 1–15.

[13] ISO/IEC 15939: 2007, Systems and Software Engineering – Measurement Process, International Organization for Standardization-ISO, Geneva, 2007.

[14] Humphrey, W. S.: PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley, Reading, MA, 2005.

[15] The Imagix 4D web site. Availaible on: `http://www.imagix.com`, May 15, 2014.

[16] Kumar, V.—Sharma, A.—Kumar, R.: Applying Soft Computing Approaches to Predict Defect Density in Software Product Releases: An Empirical Study. Computing and Informatics, Vol. 32, 2013, No. 1, pp. 203–224.

Mehmet Suleyman Unluturk got his M.Sc. and Ph.D. degrees in electrical and computer engineering from Illinois Institute of Technology, Chicago, USA in 1992 and 1997, respectively. He worked as a software engineer for Panasonic (Chicago) and General Electric (Chicago) for 11 years. He has expertise in the field of nurse call systems, electronic medical records, RFID, and real time location systems. In his free time, he works on neural network techniques for the organic-conventional food classification, bio-crystallization, and detection of antibiotics in milk products. Currently, he is Associate Professor at the Department of Software Engineering of the Yasar University.

Kaan Kurtel received his Ph.D. degree in computer science from Trakya University, Turkey, in 2009 studying the software product measurement and maintenance. Currently, he is Assistant Professor at the Department of Software Engineering of the Izmir University of Economics. His research interests focus on software quality, software measurement, software maintenance, web services engineering, and health care systems.