A Study on the Difficulty of Accounting for Data Processing in Functional Size Measures

Luigi Lavazza Sandro Morasca Davide Tosi
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
{luigi.lavazza, sandro.morasca, davide.tosi}@uninsubria.it

***Abstract*—The most popular Functional Size Measurement methods adopt a concept of "functionality" that is based mainly on the data involved in functions and data movements. Functional size measures are often used as a basis for estimating the effort required for software development. However, Functional Size Measurement does not take directly into consideration the amount of data processing involved in a process, even though it is well-known that development effort does depend on the amount of data processing code to be written. Thus, it is interesting to investigate to what extent the most popular functional size measures represent the data processing features of requirements and, consequently, the amount of data processing code to be written. To this end, we consider three applications that provide similar functionality, but require different amounts of data processing. These applications are then measured via a few Functional Size Measurement methods and traditional size measures (such as Lines of Code). A comparison of the obtained measures shows that differences among the applications are best represented by differences in Lines of Code. It is likely that the actual size of an application that requires substantial amounts of data processing is not fully represented by functional size measures. In summary, the paper shows that not taking into account data processing dramatically limits the expressiveness of the functional size measures. Practitioners that use size measures for effort estimation should complement functional size measures with measures that quantify data processing, to obtain precise effort estimates.**

Keywords- functional size measurement; Function Point Analysis; IFPUG Function Points;COSMIC method.

I. INTRODUCTION

Functional Size Measurement (FSM) methods aim at quantifying the "functional size" of an application. Such size should represent the "amount of functionality" provided to the user by a software application. It is quite reasonable to expect that the "amount of functionality" is to some extent correlated to the amount of data processing performed by the application. In this respect, there are some doubts that FSM methods properly account for the amount of data processing when sizing software applications [1].

In fact, the most popular FSM methods adopt a concept of "functionality" that is based mainly on the number of operations that can be performed by the users via the software application and the amount of data managed by the application. More precisely, the most popular FSM methods take into account

- the processes, named Elementary Processes (EP) in IFPUG and Functional Processes (FPr) in COSMIC;
- the data that cross the boundary of the application being measured or that are used (i.e., read or written) in the context of a process.

In this paper, we consider the most widely known and used FSM methods:

- IFPUG (International Function point User Group) Function Points [2][3], which were originally proposed in 1979 [4] and are widely known and used today;
- Mark II Function Points [5][6], which were proposed to improve Function Points;
- COSMIC (Common Software Measurement International Consortium) [7], which aims at further improving the characteristics of functional size measures;
- Use Case Points [8], a method that was proposed for usage with the Objectory process [9] (which was then incorporated into UML).

Quite noticeably, none of the mentioned methods satisfactorily considers the amount of data processing involved in a process. As a matter of fact, some methods propose an adjustment of the size based on the characteristics of data processing, but quite imprecisely and ineffectively, as discussed in Section VIII, while other methods do not take the amount of data processing into account at all.

The goal of the paper is to provide evidence, based on examples, that not considering data processing dramatically limits the expressiveness of functional size measures.

The core of the paper can be described as follows:

- Three applications are specified. These applications are similar with respect to the aims and functionality offered to the user, but they are very different in the amount of data processing required.
- The considered applications are modeled and measured according to four different functional size measurement methods.
- It is highlighted that the applications have the same functional size measures, even though the amount of functionality to be coded is dramatically different.
- When measured via Lines of Code, it is apparent that the implementations of the applications have quite different

sizes. The reason is that –quite obviously– more data processing requires more code.

It is unlikely that the additional code required for additional data processing requires a negligible additional amount of development effort. Thus, using only the functional size to estimate development effort for applications that require a substantial amount of data processing may lead to large and dangerous effort underestimations.

Currently, development effort is commonly estimated based on the functional size and possibly some other environmental factors, but without taking in due consideration the amount of data processing required. Sometimes this practice is justified by the fact that the application to be developed is estimated using productivity models derived from the analysis of previous projects in the same application domain. There is an underlying assumption that applications in the same domain require approximately the same amount of data processing. In this paper, we show that the contrary is true, by measuring programs that belong to the same domain.

The difficulty to quantitatively represent the amount of data processing appears as an intrinsic –though not generally recognized– limit of FSM methods. It should be noted that this paper does not aim at proposing a method to account for data processing in functional size measures. Instead, we aim at providing some evidence of the problem, to raise the awareness of the limits of FSM methods and solicit research efforts towards working out solutions. At the same time, we warn practitioners about the risks connected with assuming that the amount of data processing is somewhat automatically incorporated in traditional functional size measures, as such assumption could lead to severely underestimating the actual size of the application to be developed.

The paper is structured as follows. Section II reports a few basic concepts of functional size measurement. Section III illustrates the case studies used in the paper. Section IV describes the models and measures of the considered applications: the collected measures are then compared in Section V. In Section VI, additional examples showing the limitations of FSM methods in accounting for data processing are given. Section VII discusses the alternatives that should be considered for complementing standards functional size measures with measures that represent data processing. Section VIII accounts for related work. Finally, Section IX draws conclusions and briefly sketches future work.

This paper is an extended version of a previous paper [1]. Here, we use two additional sizing methods (namely Use Case Points and Mark II Function Points): this allows us to generalize the presented results. Moreover, we considered an additional application in the board games with artificial intelligence domain, which confirms the results given in [1], thus increasing the reliability of our conclusions. To this end, a discussion of different domains has also been added in Section VI.

II. FSM CONCEPTS

Functional Size Measurement methods aim at providing a measure of the size of the functional specifications of a given software application. Here, we do not need to explain in detail the principles upon which FSM methods are based. Instead, for our purposes it is important to consider *what* is actually measured, i.e., the model of software functional specifications that is used by FSM methods.

A. Function Point Software Model

The model used by Function Point Analysis (FPA) is given in Figure 1. Briefly, Logical files are the data processed by the application, and transactions are the operations available to users. The size measure in Function Points is computed as a weighted sum of the number of Logical files and Transactions. The weight of logical data files is computed based on the Record Elements Types (RET: subgroups of data belonging to a data file) and Data Element Types (DET: the elementary pieces of data). The weight of transactions is computed based on the Logical files involved –see the FTR (File Type Referenced) association in Figure 1– and the Data Element Types used for I/O.

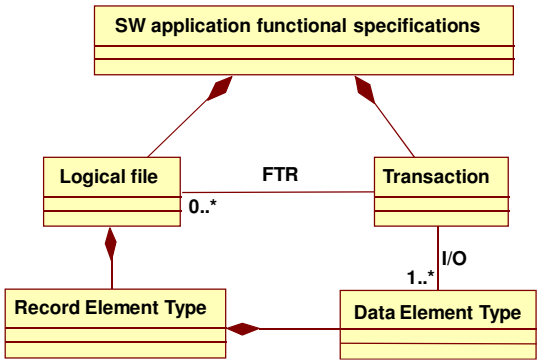


Figure 1. The model of software used in Function Point Analysis.

It is possible to see that in the FPA model of software, data processing is not represented at all.

B. COSMIC Software Model

The model used by COSMIC is given in Figure 2.

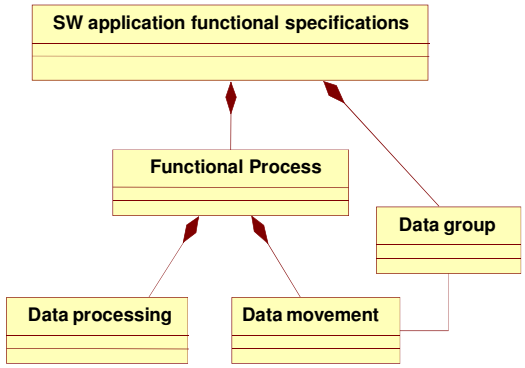


Figure 2. The model of software used by the COSMIC method.

The size of the functional specification expressed in COSMIC function points (CFP) is the sum of the sizes of functional processes; the size of each functional process is the number of distinct data movements it involves. A data movement concerns exactly one data group.

Although represented in Figure 2, neither data groups nor data processing are directly used in the determination of an application's functional size. In particular, data processing is not measured, since the COSMIC method assumes that a fixed amount of data processing is associated with every data movement; however, this is not the case in the examples considered in this paper.

C. Mark II FP Model

Symons proposed Mark II Function Points as an improvement of Albrecht's FPA in 1988 [5].

The application to be measured is modeled (see Figure 3) as a set of "logical transactions," which are essentially equivalent to IFPUG FP transactions and COSMIC functional processes. Each logical transaction is characterized in terms of the number of input DET, the number of output DET, and the number of Data Entity Types Referenced. In the Mark II FP model, DET have the same meaning as in the IFPUG FP model, while entities replace logical files (however, today the meaning associated with logical files is the same as that of Symons's entities).

The functional size in Mark II FP is the weighted sum, over all Logical Transactions, of the Input Data Element Types (N_i), the Data Entity Types Referenced (N_e), and the Output Data Element Types (N_o).

So the Mark II FP size for an application is:

$$Size = W_i \times \Sigma N_i + W_e \times \Sigma N_e + W_o \times \Sigma N_o$$

where 'Σ' means the sum over all Logical Transactions, and the industry average weights are $W_i = 0.58$, $W_e = 1.66$, and $W_o = 0.26$ [6].

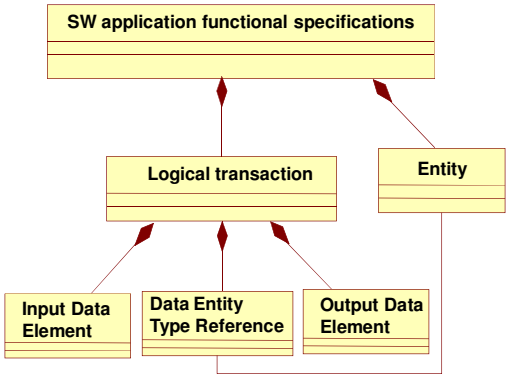


Figure 3. The model of software used in Mark II FP measurement.

D. Use Case Points Software Model

Use Case Points (UCP) were proposed by Karner to measure the size of applications specified via use cases [8]. Thus, the model of software considered for UCP measurement is centered on the concept of use case [9], as shown in Figure 4.

The UCP measurement process involves two phases. In the first one, the given application is measured in Unadjusted Use Case Points (UUCP). In the second one, the size expressed in UUCP can be "corrected" with the Technical Complexity Factor (TCF), which represents how difficult to construct the program is, and the Environmental Factor (EF), which represents how efficient our project is.

To compute size in UUCP, the considered factors are the application's users, the use cases, the transactions carried out in each use case, and the "analysis objects" (i.e., (interface, control, and entity objects, as defined in Objectory process [9]) used to realize the use case.

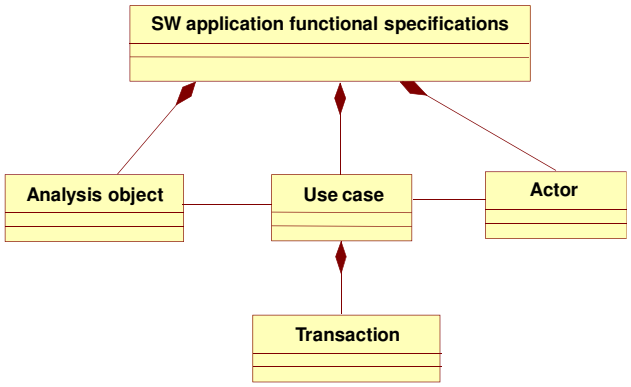


Figure 4. The model of software used in Use Case Points measurement.

The functional size in UUCP is the weighted sum of the Actors and the Use Cases. Both use cases and actors are weighted according to their "complexity." The complexity of actors is determined by nature of the actor (human or external system) and the type of interaction (e.g., via a GUI, or a command line interface). The complexity of use cases is determined by the number of involved transactions and the number of analysis objects needed to implement the use case.

The size in Use Case Points is calculated as follows.

$$TCF = 0.6 + 0.01 \times \Sigma FC_i \times W_i$$

$$EF = 1.4 - 0.03 \times \Sigma FE_i \times W_i$$

$$UCP = UUCP \times TCF \times EF$$

Where FC_i are 13 factors contributing to complexity and FE_i are 8 factors contributing to efficiency. W_i are the weights (integer value in the [0,5] interval) assigned to factors.

The details of the measurement can be found in [8].

III. Case Studies

In this section, we describe the functional specifications of the software applications that will be used to test the functional sizing ability of FPA and COSMIC.

The chosen applications are programs for playing board games against the computer. They are similar as for the functionality they provide, but they require different amounts of data processing.

The specifications that apply to both applications are as follows.

- The program lets a human player play against the computer.
- The program features a graphical interface in which the game board is represented.
- The player makes his/her moves by clicking on the board. Illegal moves are detected and have no effect. As soon as the human player has made a move, the computer determines its move and shows it on the board.
- When the game ends, the result is shown, and the player is asked if he/she wants to play another game.

The use case diagram of the considered applications is shown in Figure 5. From the point of view of the player, two main operations are available: to initiate a new game and to perform a move. In the latter case, the program will also compute its move. In both cases, the board is updated and displayed. A minor functionality of the program allows the player to show a few pieces of information concerning the application and its authors.

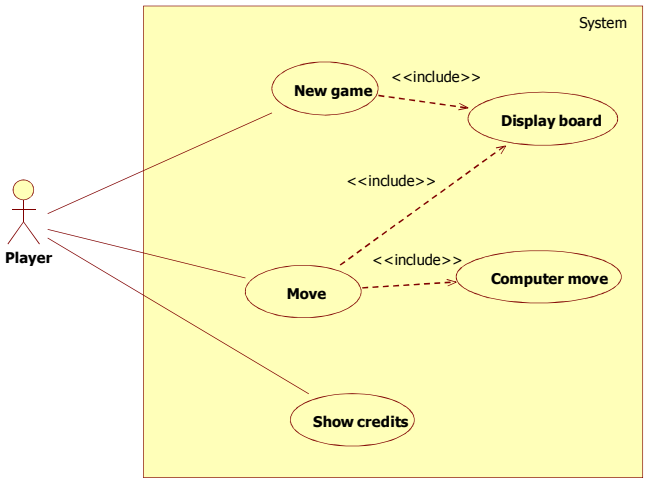


Figure 5. Use case diagram of the considered applications.

It is worth stressing that the use case diagram in Figure 5 describes all the considered applications, which differ from each other only for the implemented game, hence for the logic employed to compute the moves.

A. A Software Application to Play Tic-tac-toe

Tic-tac-toe is a very simple, universally known game. It is played on a 3×3 board, as shown in Figure 6.

Each player in turn puts his/her token in a free cell. The first player to place three tokens in a line (horizontally, vertically, or diagonally) wins. When the board is filled and no three-token line exists, the match is tie.

Playing Tic-tac-toe is very simple. In fact, to play optimally, a software program has just to evaluate the applicability of a short sequence of rules: the first applicable rule determines the move.

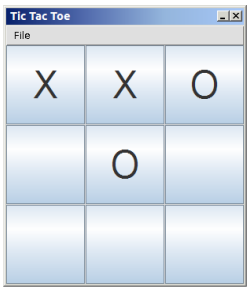


Figure 6. Tic Tac Toe playing board.

There are a few possible rule sequences: the one implemented in the considered application is the following:

1) If there is a line (row, column, or diagonal) such that two cells contain your token and the third cell X is empty, put your token in the free cell X, to win.
2) If there is a line in which your opponent has two tokens and the third cell X is free, put your token in the free cell X, so to prevent your opponent from winning at next turn.
3) If there is a move that lets you gain a winning position, make it.
4) If there is any move such that the adversary will not be able to gain a winning position at next turn, make such move. If possible, put the token in central cell.
5) If there is any cell free, put your token there.

A position is a winning one for a player when there are two lines each occupied by two tokens of the player, while the third cell is free.

The code that implements the playing logic described above is very simple and very small: we can expect that a few tens of lines of code are sufficient to code the game logic.

B. A Software Application to Play "five in a row"

Five in a row (aka Gomoku) can be seen as a generalization of Tic-tac-toe. In fact, it is played on a larger board (typically 19×19, as in Figure 7) and the aim of the game is to put five tokens of a player in a row (horizontally, vertically, or diagonally).

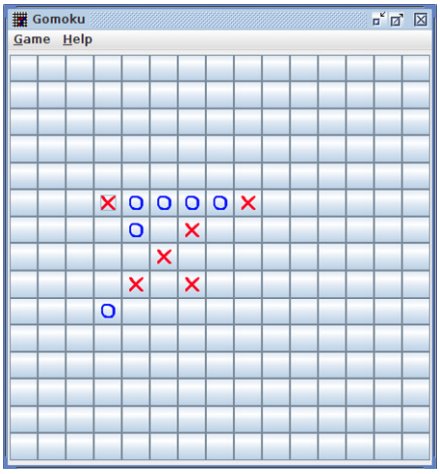


Figure 7. Gomoku playing board.

The functional specifications of Gomoku are exactly the same as the specifications of Tic-tac-toe, except that the size of the board is larger and the number of tokens to put in a row is 5 instead of 3.

The combinations of tokens and free cells that can occur on a Gomoku board are many more than in a Tic-tac-toe game. Accordingly, a winning strategy is much more complex, as it involves considering a bigger graph of possibilities.

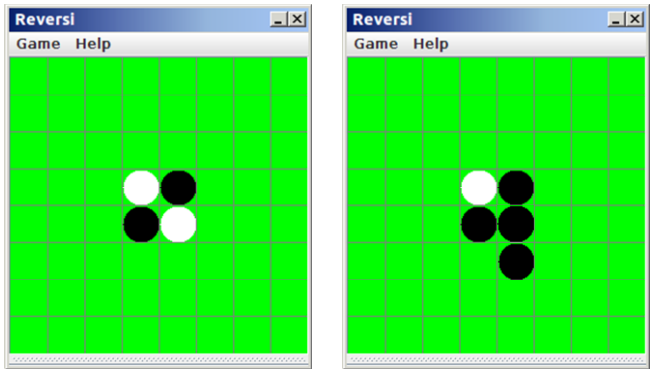
As a matter of fact, Gomoku has been a widely researched artificial intelligence research domain, and there are Gomoku professional players and tournaments.

Accordingly, we can safely state that Gomoku is a much more complex game than Tic-tac-toe and requires a large amount of processing, so that the machine can play at a level that is comparable with that of a human player.

On the contrary, Tic-tac-toe is a very simple game: you do not need to be particularly smart to master it and always play perfectly.

C. A Software Application to Play "Reversi"

Reversi (aka Othello) is played on an 8×8 board. The initial configuration is shown in Figure 8 a). Suppose player A has black tokens. At his/her turn, player A has to put its token in a position so as to form a horizontal, vertical, or diagonal line of adjacent tokens that has black tokens at the extremes and includes only white tokens (at least one). As an effect of the move, the white tokens between the black extremes become black. For instance, in the situation shown in Figure 8 a), player A could place his/her black token below the rightmost white token: such token is between two black tokens and becomes black, as shown in Figure 8 b). The game is named "Reversi" because usually the tokens are black on one side and white on the opposite one, so to change the color of a token you reverse it.



a) b)

Figure 8. Reversi playing board.

The strategy required to win a Reversi game is definitely more complex than the strategy required to play Tic-tac-toe. However, it is simpler than the strategy required for playing Gomoku, as the move search space is smaller.

IV. Application Sizing

A. Measurement of the Tic-tac-toe Application

Let us apply the FSM methods described in Section II to measure the Tic-tac-toe specifications given in Section III.A above.

1) Measuring Tic-tac-toe with IFPUG Function Points.

The software model to be used includes just a Logical data file: the board, which is a matrix of cells, each having one of three possible values (circle, cross, free). So, it is easy to see that there is only one Logical data file (the board), which is a simple Internal Logical File (ILF), contributing 7 FP.

The software model to be used involves the following elementary processes:

- Start a new game
- Make a move
- Show credits.

Start a new game is a simple External Input (EI), contributing 3 FP. Make a move is a simple external output, contributing 4 FP. One could wonder whether this operation should be considered an input (because the move involves inputting a position) or an output (because of the computation and visualization of the move by the computer). We consider that the latter is the main purpose of this transaction, which is thus an external output. Show credits is a simple External Query (EQ), contributing 3 FP.

In summary, the FPA size of the Tic-tac-toe application is 17 FP.

2) Measuring Tic-tac-toe with the COSMIC Method

The COSMIC functional processes of the application are the same as the FPA elementary processes. When measuring the application using the COSMIC method, we have to consider the data movements associated with each functional process:

- Start a new game involves clearing the board and possibly updating it, if the computer is the first to move (a Write) and showing it (a Read and an Exit). Therefore, this functional process contributes 3 CFP.
- Make a move involves entering a move (an Entry), updating the board with the human player move (a Write), reading it (a Read), and then updating it again with the computer move and showing it (an Exit). In addition, if a move concludes the game, the result is shown (an Exit). Therefore, this functional process contributes 5 CFP.
- Show credits involves the request to show credits (an Entry), reading the credits (a Read) and outputting them (an Exit). Thus, this functional process contributes 3 CFP.

In summary, the COSMIC size of the Tic-tac-toe application is 11 CFP.

3) Measuring Tic-tac-toe with Use Case Points

The Tic-tac-toe application has one user, who interacts with the system through a graphical user interface. According to UCP rules, such user is a complex actor, with weight 3.

The Tic-tac-toe application has three use cases, as shown in Figure 5. All of these use cases have 3 or fewer transactions and can be realized with less than 5 analysis objects; hence they are simple and their weight is 5.

So, the size of Tic-tac-toe is 3+5+5+5=18 UUCP.

TCF and EF involve several factors. However, only the "Complex internal processing" factor of TCF is relevant for our study, so we assume that all the factors considered in the TCF and EF have value 3, i.e., average relevance. As a consequence TCF×EF is $0.99 + 0.01 \times CIP$, where CIP is the value of the Complex Internal Processing.

The Complex Internal Processing factor is supposed to represent the complexity of the processing that is carried out in the application. It is rated on a scale 0, 1, 2, 3, 4, and 5. Unfortunately, in the original definition, Karner did not provide criteria to rate Complex internal processing; therefore, different persons could rate the same application differently. Tic-tac-toe surely is a very simple application, but it is very difficult to say if its Complex Internal Processing factor should be rated 0 or 1. So, we can conclude that the size of Tic-tac-toe is $18 \times (0.99 + 0.01$ CIP), that is, either 17.82 or 18, depending on the value assigned to CIP.

4) Measuring Tic-tac-toe with Mark II Function Points

To size the Tic-tac-toe application using Mark II FP, it is first necessary to identify the involved entities and Logical transactions. This is very easy, since we have only two entities (the board and the credits) while the logical transactions correspond to IFPUG FP transactions, COSMIC functional processes and UCP Use Cases (i.e., New game, Move, Show credits).

The size is computed according to the number of input data, entities referenced and output data as shown in TABLE I. While New game and Show credits have just one input (the event that triggers the operation), Move has two inputs: the row and column where the player puts his/her token.

New game and Show credits also have just one output (the board and the credits' text, respectively); Move outputs the board and the users' tokens, or a diagnostic message (when the player clicks on an already occupied cell).

New game and Move access the board entity, Show credits accesses the credits entity.

TABLE I. MEASURES OF TIC-TAC-TOE APPLICATION IN MARK II FP

Logical transaction	***Ni***	***Ne***	***No***	***MKII FP***
New game	1	1	1	2.5
Move	2	1	3	3.6
Show credits	1	1	1	2.5
Total				**8.60**

In conclusion, the application to play Tic-tac-toe has size 8.60 MKII FP.

5) Tic-tac-toe Code Measures

Since we are also interested in indications concerning the amount of computation performed by the application, we selected an open source implementation of Tic-tac-toe and measured it.

To evaluate the "physical" size of the Tic-tac-toe application, we looked for an open source application that implements the specifications described above. Two such applications are the programs available from [10] and [11]. To make the considered program functionally equivalent to the other applications, we performed a merge of the code from [10] and [11]. The main measures that characterize the obtained code are given in TABLE II.

TABLE II. MEASURES OF THE TIC-TAC-TOE APPLICATION CODE

Measures	**Tic-tac-toe**	
	Total	***AI part***
LoC	286	146
Number of Java statements	187	101
McCabe (method mean)	3.6	4.5
Num. classes	2	1
Num. methods	26	13

In TABLE I (and in TABLE II), column "AI part" indicates the measures concerning exclusively the part of the code that contains the determination of the computer move.

We reported both the number of lines of code and the number of actual Java statements: the latter is a more precise indication of the amount of source code, since it does not consider blank lines, comments and lines containing only syntactic elements, like parentheses. We also reported the mean value of McCabe complexity of methods.

B. Measurement of Gomoku Application

Let us measure the Gomoku specifications given in Section III.B above

1) Measuring Gomoku with IFPUG Function Points and COSMIC

The functional size measures of the Gomoku application are exactly the same as the measures of the Tic-tac-toe application. In fact, the specifications of the two applications are equal, except for the board size and winning row size, which do not affect the measurement, because both IFPUG FPA and COSMIC consider data types, not the value or number of instances.

2) Measuring Gomoku with Use Case Points

Gomoku has the same actor and use cases as Tic-tac-toe. Therefore, the size of Gomoku measured in UUCP is equal to Tic-tac-toe's.

As for Tic-tac-toe, we assume that the factors that determine TCF and EF are all average, except for the CIP; therefore, TCF×EF is $0.99 + 0.01 \times CIP$.

Gomoku is definitely a much more complex game than Tic-tac-toe; therefore, the Gomoku playing program has to perform quite complex processing to achieve an acceptable playing level. We can therefore assign the Complex Internal Processing a high rating, even though it is not clear whether we should set CIP=5 or CIP=4. In conclusion, the size of Gomoku is 18.54 or 18.72, depending on the value of CIP.

3) Measuring Gomoku with Mark II FP

The Gomoku application is characterized by the same actors, use cases, transactions and entities as the Tic-tac-toe application. Therefore, they have the same size measure expressed in Mark II FP.

4) Gomoku Code Measures

As for Tic-tac-toe, we selected an open source implementation of Gomoku and measured it. More precisely, we looked for a program capable of sophisticated "reasoning" that lets the program play at the level of a fairly good human player. One such application is the Gomoku Java program available from [12].

The main measures that characterize the code are given in TABLE III.

TABLE III. MEASURES OF THE GOMOKU APPLICATION CODE

Measures	Gomoku [12]	
	Total	*AI part*
LoC	859	373
Number of Java statements	419	212
McCabe (method mean)	2.6	5.4
Num. classes	17	3
Num. methods	83	25

Measures in TABLE III were derived using the same tools and have the same meaning as the measures in TABLE II.

C. Measurement of Reversi Application

1) Measuring Reversi with IFPUG Function Points and COSMIC

The functional size measures of the Reversi application are exactly the same as the measures of the Tic-tac-toe and Gomoku applications. In fact, the specifications of the three applications are characterized by the same basic functional components.

2) Measuring Reversi with Use Case Points

Reversi has the same actor and use cases as Tic-tac-toe and Gomoku. Therefore, the size of Reversi measured in UUCP is equal to Tic-tac-toe's and Gomoku's.

As for the other applications, we assume that the factors that determine TCF and EF are all average, except for the CIP; therefore, TCF×EF is $0.99 + 0.01 \times$ CIP.

Reversi is definitely more complex than Tic-tac-toe, but less complex than Gomoku. We can therefore assign the Complex Internal Processing a high rating, but not as high as Gomoku's. So, it is probably reasonable to set CIP=4 or CIP=3. In conclusion, the size of Reversi is 18.36 or 18.54, depending on the value of CIP.

Note that the value assigned to CIP is largely subjective: this is due to the fact that the definition of UCP does not provide precise guidelines for determining the values of TCF and EF factors.

3) Measuring Reversi with Mark II Function Points

When measuring the Reversi applications with Mark II FP, the same considerations reported for Tic-tac-toe and Gomoku apply. The only difference is that when the New game logical transaction is performed, the initial situation of the board is not empty, therefore we have 2 additional output DET associated to New game. This is shown in TABLE IV.

TABLE IV. MEASURES OF THE REVERSI APPLICATION IN MARK II FP

Logical transaction	Ni	Ne	No	MKII FP
New game	1	1	3	3.02
Move	2	1	3	3.6
Show credits	1	1	1	2.5
Total				**9.12**

In conclusion, the application to play Reversi has size 9.12 MKII FP.

4) Reversi Code Measures

Like with the other applications, we selected an open source Java implementation of Reversi [13] and measured it.

More precisely, the implementation of Reversi that we found [13] was richer than the implementations of Tic-tac-toe and Gomoku in functionality (e.g., it features a help function, the possibility of choosing the playing level and the dashboard color, etc.). To make the Reversi application comparable to the others, we simplified the implementation, deleting all the additional functions and the corresponding code.

The main measures that characterize the resulting code are given in TABLE V.

TABLE V. MEASURES OF THE REVERSI APPLICATION CODE

Measures	Reversi [13]	
	Total	*AI part*
LoC	419	218
Number of Java statements	290	180
McCabe (method mean)	3.1	4.4
Num. classes	6	4
Num. methods	36	17

Measures in TABLE V were derived using the same tools and have the same meaning as the measures in TABLE II and TABLE III.

V. COMPARISON OF MEASURES

The measures reported in Section IV and summarized in TABLE VI show that a few applications may have the same functional size, but very different code size: for instance, the Gomoku application is twice as big as the Tic-tac-toe application. Considering the nature of these applications, the difference in code is largely explained by the different amount of processing required. In the case of Tic-tac-toe, the number of possible moves is very small, as is the number of different possible configurations that can be achieved by means of a move: hence, every move computation has to explore a very small space. The contrary is true for the

Gomoku application. The consequence is that Gomoku requires an amount of code devoted to move computation that is more than twice as much as the code required by Tic-tac-toe. Reversi requires more data processing than Tic-tac-toe and less processing than Gomoku; accordingly, its implementation is bigger than Tic-tac-toe's and smaller than Gomoku's.

The collected measures are summarized in TABLE VI. Both measures concerning the complete application (column Total) and measures concerning the artificial intelligence part of the application (column AI) are given. It should be noted that the functional size makes sense only concerning the complete application, since it is not allowed in FSM methods to measure only a portion of the application (this would actually be possible with the COSMIC method, but the resulting measure would not be comparable to those of the complete applications, though).

TABLE VI. SUMMARY OF THE APPLICATIONS' MEASURES.

	Tic-tac-toe		**Reversi**		**Gomoku**	
	Total	***AI***	***Total***	***AI***	***Total***	***AI***
Data proc.	Average	Very low	Average	Medium -high	Average	High
Java statem.	187	101	290	180	419	212
McCabe	3.6	4.5	3.1	4.4	2.6	5.4
Classes	2	1	6	4	17	3
Methods	26	13	36	17	83	25
IFPUG FP	17		17		17	
CFP	11		11		11	
UUCP	18		18		18	
UCP	17.8–18		18.4–18.5		18.5–18.7	
Mark II FP	8.60		9.12		8.60	

These observations suggest a few important considerations, which are reported below.

A. *Functional Size and Data Processing*

The definitions of IFPUG FP, COSMIC FP, Mark II FP, and UUCP do not properly take into account the amount of processing required by software functional specifications. If we plot the three considered applications in a Cartesian plane, where the axes represent the amount of required data processing and the functional size (expressed via any functional size measure), we get the situation described in Figure 9 (note that the y axis is not in scale). It appears that there is no relationship that links the functional size and the amount of processing required.

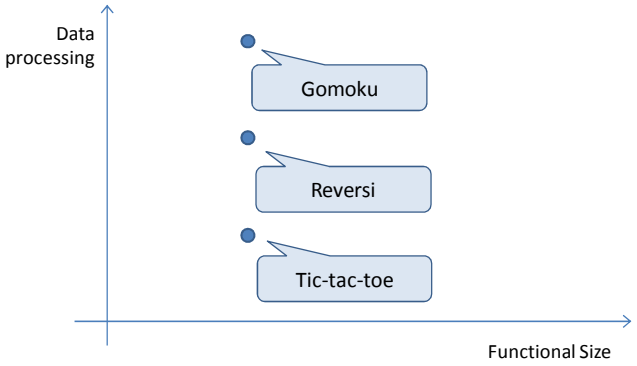


Figure 9. Plot of data processing vs. functional size for the considered applications.

For the sake of precision, we must note that Mark II FP and UUCP measures are not equal for all the applications, but are only slightly different, while the differences in terms of data processing are fairly large.

If we plot the three considered applications in a Cartesian plane, where the axes represent the amount of required data processing and the physical size (expressed in LoC, or number of statement, or in number of methods, etc.), we get the situation described in Figure 9 (note that axis scales are just indicative). It is possible to see that there is a clear relationship between the physical size and the amount of processing required.

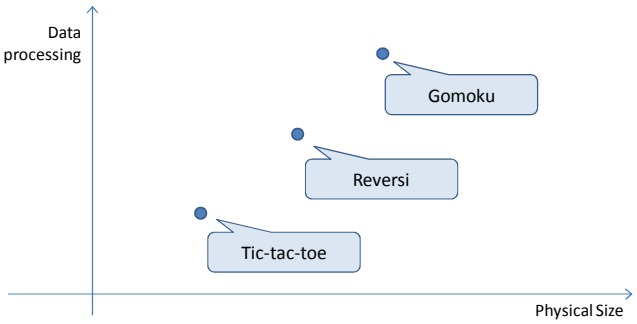


Figure 10. Plot of data processing vs. physical size for the considered applications.

If we assume –as is generally accepted– that the effort required to implement a software application is related to the number of Lines of Code to be written, the possibility of having widely different sizes in LoC for applications that have the same functional size implies that functional size is not a good enough predictor of development effort.

B. *Mark II FP Measures*

In the considered cases, Mark II FP size measures indicate that Reversi is marginally bigger than the other applications. This is misleading when considering that the Gomoku application is actually much bigger than Reversi. In addition, the difference with respect to Tic-tac-toe's size does not give a proper idea of the actual difference: the implementation of Reversi is 55% bigger than Tic-tac-toe's, while the functional size in Mark II FP is only 6% bigger.

C. Effort Required for Non-coding Activities

The observation reported in Section V.A above does not apply only to the coding phase. In fact, the difference in the number of classes and methods (shown in TABLE VI) suggests that also the effort required by design and testing activities is better estimated based on measures that represent the size of the code structure –like the number of classes– rather than the functional size.

D. The Explanation Power of TCF

In the analyzed cases, the correction to UUCP due to the TCF appears largely insufficient. In fact, assigning to CIP the biggest possible value (i.e., 5) for Gomoku and the smallest (i.e., 0) for Tic-tac-toe causes the size of Gomoku (18.72 UCP) to be only 5% bigger than the size of Tic-tac-toe (17.82 UCP). Such difference does not appear to be able to predict the difference in terms of Java statements to be written, as the AI code of Gomoku is twice as big as the AI code of Tic-tac-toe.

We should note that in this paper we considered Unadjusted Function Points, as defined in the ISO standard [3]. However, the IFPUG also defines "adjusted" Function Points, which are obtained by applying to the unadjusted measure a value adjustment factor (VAF) that is based on a few characteristics of the application being measured, including "Complex Processing" [2]. Actually, the definition of UCP's TCF and EF were inspired by Function Points' VAF. The considerations reported above concerning the representativeness of TCF apply to VAF as well. The "Complex Processing" component of VAF affects the size by less than 8%: too little to explain the observed differences in the considered applications' code size.

E. The Explanation Power of McCabe Complexity

As a final remark, we can observe that mean McCabe complexity is fairly similar in all the considered applications. The mean McCabe complexity of the AI part of the applications increases with the amount of data processing required by the games, but the differences are very small: from 4.5 of Tic-tac-toe to 5.4 of Gomoku. This means that applications dealing with more complex games (like Gomoku) do not need code that is much more complex (in McCabe's sense), but just more code. In other words, it is the difference in the amount of data processing, not in the complexity of the processing that is relevant, and that existing functional size measures fail to represent.

VI. ADDITIONAL EVIDENCE

The problems described above are at the level of elementary processes (alias transactions, alias functional processes). Namely, the problem with the considered board games is located in the Move process, which has the same functional size in all applications, but requires quite different data processing in the three considered applications.

Readers might wonder whether the described problem is due to the nature of the considered applications, which involve the usage of artificial intelligence. Actually, the same type of problem can be found in different application domains. Let us consider the measurement of source code. Several processes that are frequently found in measurement programs share the same set of properties, namely:

- Inputs: the request to measure and the name of the source code file to be measured.
- Output: the value of the measure.
- Data read: the code file.

Examples of such processes are the measurement of LoC, the measurement of Non commenting LoC (i.e., LoC not including comments), the measurement of McCabe complexity and the measurement of the coupling between objects (CBO) [14].

It is easy to see that these processes have the same functional size, whatever measure they compute. More precisely, they all have the same functional size if IFPUG FP, COMSIC FP or Mark II FP are used. If UCP are used, the sizes could differ of up to 5%, because of differences in the "Complex Internal Processing" factor.

However, different measures require different amounts of data processing:

- Total LoC: the processing is extremely simple, as it just involves counting the number of 'new line' characters.
- Non commenting: the required processing is more complex than in the former case, but still rather simple. In fact it is sufficient to recognize the beginning and end of comments and exclude lines that are entirely included in comments.
- McCabe complexity: the processing is more complex than in the previous case, since syntax analysis is required to recognize functions (or procedures or methods, depending on the programming language) and decision statements (if, while, for, etc.). The computation of McCabe complexity is usually performed by first parsing the code to obtain an abstract syntax tree, and then visiting the tree to count the relevant syntactic elements (if, while, etc.).
- CBO: the processing is still more complex. In fact, semantic analysis of code is also required, in addition to syntax analysis. For instance, when a statement like `a = b.new_class(x,y,z);` is found in class C, it is necessary to understand the type (class) returned by method new_class, to properly count the number of dependencies of class C.

So, a program that measures McCabe complexity and CBO has the same functional size as a program that counts total LoC and non-comment LoC; however, it is quite clear that implementing the former program is much more demanding in terms of development effort, since a greater amount of data processing has to be implemented.

Similar examples in different domains are easily found. For instance, in the statistical domain, a few processes have to show a series of data concerning a time period, via different representations. All the processes have the same inputs, read similar data, and output similar information (although using different graphical styles): accordingly, all the processes have the same functional size, since graphical styles are irrelevant. However, some of these process could require a very small amount of data processing. For instance, the process that shows data via a bar chart (Figure 11)

consists of a simple loop: at every iteration a value is read and a bar of length proportional to the value is drawn. On the contrary, a process that shows the data via a "smooth" interpolation line, e.g., a lowess (locally weighted scatterplot smoothing) curve [15] (Figure 12) has to perform a definitely greater amount of data processing, since the computation of weighted linear least squares regression is required.

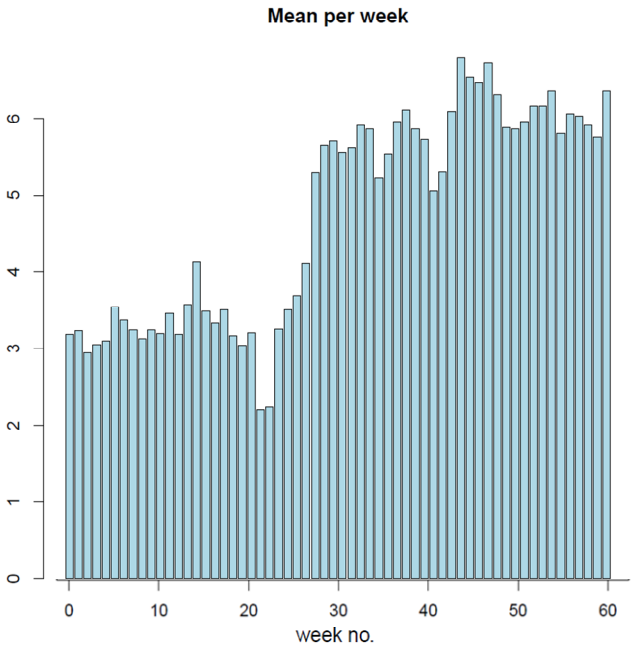


Figure 11. Output of a transaction that represents a time series via an histogram.

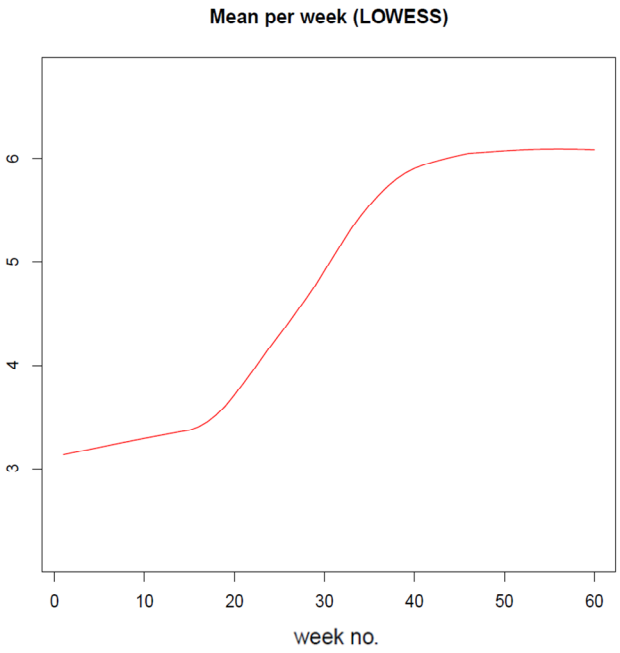


Figure 12. Output of a transaction that represents a time series via a LOWESS curve.

Summarizing, there are many examples of transactions (alias elementary processes, alias functional processes, etc.) whose functional size measure does not appear effective in representing the functionality delivered to the user, since the –quite variable– amount of data processing is not accounted for.

VII. Discussion: What Solutions Are Possible?

The usefulness of the evidence given in this paper stems from a few well-known facts:

- We need to estimate, during the early phases of a project, the overall software development effort.
- Development effort has been widely reported to be directly related to the size in LoC of software. Unfortunately, the size in LoC is not available in the early phases of projects, when estimates are most needed.
- Therefore, we need FSM methods, i.e., we need measures of functional specifications, because specifications are available in the early phases of projects.
- In this paper, we provided some evidence that current FSM methods appear limited in representing the amount of data processing required by functional specifications. Therefore, we need to somehow enhance FSM methods to remove such limitation.

So, we are facing the following research question: how can we improve FSM methods so that the delivered functional size measures account for the amount of data processing described or implied by the functional specifications?

This is an open research question, which calls for a substantial amount of further studies. In the following sections, we report a few observations, ideas, and evaluations that could be useful considering when tackling the problem.

A. *Software Models*

FSM methods –like any measurement method– are applied to models of the object to be measured. Hence, a rather straightforward consideration is that data processing must be represented in the model that describes the software application to be measured.

We can observe that the conceptual model of software proposed in the COSMIC method includes data processing, but no criteria or procedures for measuring data processing are given in the context of the COSMIC method.

In COSMIC, data processing is a sub-process of a functional process. Therefore, functional processes should be described in a manner that makes it possible to identify and measure the extent of data processing that occurs within a functional process.

Given the similarity of COSMIC functional processes and FPA elementary processes (or transactions) any technique used to enhance the expressivity of COSMIC models as far as data processing is concerned should be readily applicable to FPA models as well.

B. Software Specifications

A question that should be considered is if the information required for identifying and measuring data processing is always available from the software specifications that are derived from user requirements.

FSM methods use models of functional specifications: if functional specifications do not include information on data processing, neither do their models, and FSM methods will not be able to account for data processing.

So, another open question is the following: is it necessary to go beyond user requirements related specifications to be able to represent data processing? In other words: should elements of design be anticipated, to get better measures of the amount of data processing to be implemented?

C. Qualitative Knowledge

Current FSM methods are inherently quantitative. Even though some measurement activities –like deciding if two sets of data should be two RET of a unique logic file or they should belong to separate logic files– involve some subjectivity, they are always meant to provide measures (the number of ILF, RET, etc.) at the ratio level of measurement.

One could wonder if the use of more qualitative knowledge, derived through inherently subjective evaluations and expressed via ordinal scales, would be more suitable for expressing the relevant information concerning data processing.

For instance, after talking with stakeholders, an analyst could easily classify the functional process "Make a move" of the Tic-tac-toe application as very simple, while the same process of the Gomoku application could be classified as very complex.

D. Towards a Measure of Data Processing

As mentioned above, proposing a solution to the problem outlined above is very difficult. Here, we outline a few directions to be considered when addressing the problem.

A first consideration concerns the level of description of data processing. At a high level, the "variability" of the processes in terms of number of different cases to be considered could easily determine the amount of data processing required. Consider for instance a process that starts by identifying users: if the specifications indicate that the user can be identified in three different ways (e.g., by name, by social security number, and by email address) it is likely that it will have to process three times as much data as a process that identifies users in a single way.

Another observation concerns how to differentiate functionalities. A possibility is to account for the internal states a function has to deal with. In the case of tic-tac-toe, the number of states in which the game can be is quite small; on the contrary, the states of a Gomoku game are very numerous. Accordingly, the amount of computation could be proportional to the number of states, since the function has to properly deal with all states. However, the quantification of data processing could be further complicated by the presence of equivalent states, i.e., sets of states that are managed in the same way, so that having N or N+1 states in such sets would not affect the amount of processing required. For instance, a date increase function has to account for months having 28 (or 29), 30, or 31 days: the fact that there are 7 months having 31 days and just one having 28 days is irrelevant. In complex cases, identifying the relevant states could be very difficult; for instance, in Gomoku several token patterns can be identified, and each pattern calls for a specific strategy. So, the interesting states are the token patterns, but imagining in advance all the possible patterns is quite hard. A qualitative indication concerning the number of states would probably be more appropriate, in this case.

VIII. Related Work

Although several FSM methods (e.g., Mark II FP, NESMA and FiSMA) have been proposed as extensions or replacements of Function Point Analysis, very little attention has been given to the measurement of data processing.

A noticeable exception is the proposal of Feature Points by Capers Jones [16]. In this functional size measure, algorithms were added to the set of FPA basic functional components (ILF, EIF, EI, EO and EQ), and each component type was assigned a unique value, i.e., the notion of complexity was removed. The method was soon abandoned, mainly due to the difficulty of identifying algorithms, which are typically not documented in functional specifications.

Function Point Analysis and other methods –like Use Case Points [8]– introduce a mechanism for "adjusting" the size measure to take into account additional complexity factors that are likely to increase the effort required for implementation. In fact, among FPA value adjustment factors (VAF) we find "Complex Internal Processing," which represents to what degree the application includes extensive logical or mathematical processing. This mechanism is similar to what we need, but has a few shortcomings, including:

- In FPA the considered VAF's value increases the application size by 4% to 8%: at least one order of magnitude less than needed in the Tic-tac-toe vs. Gomoku case.
- The VAF applies to the whole application, so that it is not possible to distinguish simple and complex processes.

Noticeably, only the definition of unadjusted Function Points was standardized [3].

The Path measure [17][18] represents the complexity of processes in terms of the number of execution paths that are required for each process. Although this measure proved fairly effective in improving effort estimation based on functional size measures, it is not applicable in cases like those considered in this paper, since the alternative courses of the specified processes are not known.

IX. Conclusions

In this paper, we have shown by means of examples that FSM methods fail to represent the amount of data processing required by software functional specifications.

One could wonder how general are the results reported in the paper. As for this issue, we showed in Section VI that the

limits of FSM methods discussed in the paper apply to several application domains.

The work reported in the paper indicates that we need a measure that can complement traditional FSM methods to represent the amount of data processing that is needed to provide the required functionality.

We are interested in representing and quantifying the amount of data processing not because of an abstract interest in the definition of functional size measures, but because –as shown in the paper– data processing is logically related to code size, which in its turn is linked to the amount of development effort required to build a software application.

How to measure the amount of data processing required by the specifications of a software application is an open research question of great practical interest that should receive much more attention than it currently does.

ACKNOWLEDGMENT

The work presented here has been partly supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission and by project "Metodi, tecniche e strumenti per l'analisi, l'implementazione e la valutazione di sistemi software," funded by the Università degli Studi dell'Insubria.

REFERENCES

[1] L. Lavazza, S. Morasca, and D. Tosi, "On the Ability of Functional Size Measurement Methods to Size Complex Software Applications," 9th Int. Conf. on Software Engineering Advances - ICSEA 2014, October 12-16 2014, Nice. pp. 404-409.

[2] International Function Point Users Group. Function Point Counting Practices Manual - Release 4.3.1, January 2010.

[3] ISO/IEC 20926: 2003, Software engineering – IFPUG 4.1 Unadjusted functional size measurement method – Counting Practices Manual, Geneva: ISO, 2003.

[4] A. J. Albrecht, "Measuring Application Development Productivity," Joint SHARE/ GUIDE/IBM Application Development Symposium, 1979, pp. 83-92.

[5] C. R. Symons, "Function point analysis: difficulties and improvements," IEEE Transactions on Software Engineering, 14.1, 1988.

[6] ISO/IEC 20968:2002, "Software engineering – Mk II Function Point Analysis – Counting Practices Manual," 2002.

[7] COSMIC – Common Software Measurement International Consortium, The COSMIC Functional Size Measurement Method - version 4.0 Measurement Manual, April 2014.

[8] G. Karner, "Resource estimation for objectory projects," Objective Systems SF AB, 17. 1993.

[9] I. Jacobson, G. Booch, and J. Rumbaugh. "The Objectory Software Development Process," Addison Wesley, 1997.

[10] http://algojava.blogspot.it/2012/05/tic-tac-toe-game-swingjava.html, last accessed 15 May, 2105.

[11] http://sourceforge.net/projects/tictactoe-javab/files, last accessed 15 May, 2105.

[12] http://sourceforge.net/p/gomoku/, last accessed 15 May, 2105.

[13] https://reversi.java.net/, last accessed 15 May, 2105.

[14] S.R. Chidamber and C.F. Kemerer. "A metrics suite for object oriented design," IEEE Transactions on Software Engineering 20.6, 1994.

[15] W.S. Cleveland, "LOWESS: A program for smoothing scatterplots by robust locally weighted regression," American Statistician, 1981.

[16] C. Jones, "The SPR Feature Point Method," Software Productivity Research, 1986.

[17] G. Robiolo and R. Orosco, "Employing use cases to early estimate effort with simpler metrics," Innovations in Systems and Software Engineering, 4(1), 2008, pp. 31-43.

[18] L. Lavazza and G. Robiolo, "Introducing the evaluation of complexity in functional size measurement: a UML-based approach," ACM-IEEE Int. Symp. on Empirical Software Engineering and Measurement, September 2010.