TECHMIND RESEARCH
ISSN: 2291-2118
International Journal of Research in Business and Technology
Volume 6 No. 3 June 2015

Software Size Estimation: Practical Models and their Applications in Various Phases of the SDLC

Mohammad A. Rob
University of Houston-Clear Lake
United States

Abstract- *Estimating software size has been one of the highly researched topics over the past few decades. Due to the unpredictable nature of many factors associated with software development projects and ever-changing technological complexity, the topic still draws widespread attention within the software engineering domain. This paper briefly discusses few estimation models that can be practically applied to estimate software metrics such as size, effort, and time, without getting involved into mathematical complexity by a project manager. We also discuss the pros and cons of these models. We further map the applicability of these models in various phases of the software development life cycle. Finally, we implement two of these models to develop a user interface that can be used repeatedly or as necessary to estimate the software metrics during the major stages of its developmental life cycle, and adjust the schedule as necessary.*

Key Words- *Software; Size; Estimation; Effort; Function Point; Lines of Code; COCOMO; WBS*

1. INTRODUCTION

The art and science of software project management can be described as an adjustment among three important factors: the size of the software, the time required to complete the project, and the cost of the project as well. These three metrics are interdependent levers that a project manager controls throughout the life of a project [1]. Whenever one lever is manipulated, subsequently, the other two levers are affected to some degree. However, the cost of the project is the most important factor for a project sponsor, followed by the time required for completion; and both of them in turn, depend on the size of the project. Many software project failures are due to unrealistic expectations based on inaccurate estimations [2][3].
The bulk of the software development cost is due to human effort, which depends on the size of the project. Effort is often measured in person-months of the developers such as programmers, analysts, and project managers. Due to many interrelated human and technological factors associated with a software development project which cannot be defined with certainty, size can only be estimated. Most software estimation models attempt to generate an estimation of effort, which can then be converted into project duration and cost [4]. Although effort and cost are closely related, they are not necessarily related by a simple transformation. Therefore, at the beginning of a project, a manager needs to estimate the size and depth of a project. Once the size is appreciated, the effort and time required to develop the software can be assessed, which in turn, can lead to a more accurate appraisal of cost. Thus, most of the software estimation models focus on assessment of the size of a project. However, the approximation of size depends on many factors, such as: human interaction with the system, process-to-process interaction within the system, as well as the technological complexity of the system. A key factor in selecting an estimation model is assessing the accuracy of its estimates.
Many methods have been proposed to obtain a close approximation of the size of a software project; however, most of them are too complex, or not practical, or not accurate. The number of methods grows daily as others in the field attempt to create more precise estimations based on their experiences [1][4][5][6]. However, not many methods can provide all the metrics of size, effort and time. Many times more than one methods are used to estimate the software size and other metrics. The most important factor is to get the best possible estimation of size at the early stage of the software developmental life cycle, so that a project manager can create the best possible plan for the project. Furthermore, some estimation models are more suitable in certain phases of a software development life cycle than others, and some can be used repeatedly throughout the life cycle to revise the project plan as necessary. The latter is desirable by any project manager.
In this paper, we briefly discuss the commonly-mentioned estimation models in the literature that can be practically applied to estimate software metrics such as size, effort, and time, without getting involved into mathematical complexity by a project manager. We also discuss the pros and cons of these models. We further map the applicability of these models in various phases of the software development life cycle. Finally, we implement two of these models to develop a user interface that can be repeatedly used to estimate and revise the software metrics at the major stages of its developmental life cycle. This kind of user interface will help a project manager to quickly and repeatedly calculate the software metrics and

update the project plan while the developmental effort continues.

2. AN OVERVIEW OF PRACTICAL ESTIMATION MODELS

In the following, we briefly review the methods or combination of methods that can provide an estimation of the size of a software project within the early stages of its development. We also provide pros and cons for each of these methods.

- **Industry Standard Percentage Estimation:** With this approach, the time spent in (or estimated for) the planning phase is used to calculate estimates for the other software development life cycle (SDLC) phases [1][6]. Industry standards (or percentages from an organization's own experiences) suggest that a typical business application system spends 15% of its effort in the planning phase, 20% in the analysis phase, 35% in the design phase, and 30% in the implementation phase. This would suggest that if a project takes 4 months in the planning phase, then the rest of the project likely will take a total of (4*100/15 – 4) = 22.67 person-months. Knowing the productivity of the development team, one can allocate the time and staff needed for to complete the rest of the project. The limitation of this approach is that it can be difficult to take into account the specifics of an individual project in an early stage, which may be of varying complexity in comparison to a typical software project. Furthermore, mature software development companies may have enough experience to use this projection than a start-up or less proven company. Nevertheless, the method gives ball-park figures of software metrics at the early stage of its development.
- **Functional Decomposition Diagram or WBS:** Functional Decomposition Diagram (FDD) or Work Breakdown Structure (WBS) is typically used as a technique for separating a business operation into its functional components or dividing a larger piece of work into smaller components or tasks [6]. A large or complex function is more easily understood when broken down through its functional decomposition. Decomposition is the process of starting at the high level of a system and subsequently subdividing that into smaller and smaller related components. As such, it forms an inverted tree-like structure, with each leaf defining a function. Typically FDD uses rectangles to display a function. The top rectangle represents the system or software as a whole and it is decomposed into first-level components identifying their functions. Next, the first-level components are decomposed into second-level, and so on, until sufficient level of detail is achieved.
  Once an FDD or WBS is created, the project manager assigns a time allotment for each function of a software development project. All functional (developing a form, report or database) and non-functional requirements (conducting and interviews, or writing requirements) need to be considered. Expert judgement of the project manager and the development group is often called upon to estimate time for each function. A weighted average formula can also be used calculate time or duration, T for each function or task: T = (B + 4P + W)/6, where B stands for best-case estimate with one time weight, P stands for probable-case estimate with four times weight, and W stands for worst-case estimate with one time weight [8]. Wide-Band Delphi method can also be used to obtain a better estimation of time, which uses time estimation from multiple team members of a project [9].
  An FDD or WBS shows the hierarchical structure of the components or tasks, but it does not show the sequence of the tasks to be developed. Thus a new WBS is created with a list of tasks along with their sequences defining time and resource (staff) needs to complete each task. Tools like PERT or Gantt chart can be used to create this sequence or develop a schedule, and as such an estimation of project completion time can be obtained [8]. Software like Microsoft Project can aid in the schedule development, resource assignment and estimation of staff need.
  Nermin [10] recently used the FDD technique to measure functional complexity of a computer system and investigated its impact on system development effort. Later, it examines effects of technical difficulty and design team capability factors in order to construct the best effort estimation model. With using traditional regression analysis technique, the study develops a system development effort estimation model which takes functional complexity, technical difficulty and design team capability factors as input parameters.
- **Constructive Cost Model or COCOMO:** The length of the software program codes can be used as a predictor of project characteristics such as size and effort. The Constructive Cost Model (COCOMO) is an algorithmic model developed by Barry Boehm to calculate effort, time and staff requirement of a software development project based on the number of lines of code to be developed [5][11]. The model used a basic regression formula with parameters that are derived from 63 historical projects ranging in size from 2,000 to 100,000 lines of code with a variety of programming languages. The COCOMO formula calculates software development effort (in person-months) if the program size is known. Program size is expressed in estimated thousands lines of source code (KLOC) to be delivered.
  A line of code (LOC) is defined as a logical line, not necessarily a physical line. For example, in C and C++ it is common to count the number of semi-colons and use that as the line count. In this manner, placing three logical statements on one physical line still counts as three lines of code. COCOMO formula varies depending on the type of software project to be developed: organic, semi-detached, and embedded. The organic projects fall in the category of small teams with good experience working with less than rigid requirements. The semi-

detached projects fall in the category of medium teams with mixed experience working with a mix of rigid and less than rigid requirements. The embedded projects are developed within a set of tight constraints. For a semi-detached project, effort, E in person-months is calculated as: $E = 3.0\ (KLOC)^{1.12}$.

The most important part of COCOMO model is the counting of source lines of code for the expected software to be developed. Although many techniques for counting lines of code for various languages have been developed, but the underlying philosophy is the same. It is the oldest and the most widely used method for software size estimation. Even though other techniques have made major in-roads in the world of software development, LOC remains popular as a technique for many code intensive applications.

However, there is a series of arguments against LOC due to the fact that the delivered functionality per line of code will vary based on the language being used, and it is very difficult to count the number of lines of code at the early stage of a software development project. Furthermore, in today's software environment of graphical user interface (GUI), it does not work well by itself, but it can be used along with other techniques to estimate the total software as discussed in the following. Although COCOMO II has been proposed that addresses the issue through object points, but other methods as defined below remain popular [6].

- **Function Point Analysis:** Function Point Analysis was an attempt to overcome the difficulties associated with lines of code as a measure of software size, and to assist in developing a mechanism to predict effort associated with software development. The method was first published by Allan J. Albrecht of IBM in 1979, then later in 1983[12] [13]. In 1984 Albrecht refined the method and since then many refines of the method were proposed since then [14]. It quantifies the functionalities contained within software in terms that are meaningful to the users such as inputs, outputs, queries, files, data and etc. Using a standardized set of basic criteria, each of the business functions is given a numeric index according to its type and complexity. The measure relates directly to the business requirements that the software is intended to address. It can therefore be readily applied across a wide range of development environments. It is independent of programming language and deals quantitatively with complexity. However, in order to calculate effort, one needs to know the productivity rate of the project team, which varies from company to company, and only the established company will have the necessary data. Effort is calculated by dividing the Function Points (FP) by the delivery rate or productivity, P. For example, a project with 1000 FP and with a productivity P = 15 FP per person-month, the Effort = 1000/15 = 67 person-months. As effort depends on the productivity of the software development team, which can be widely varied, the FP approach is not complete in and of itself to estimate the size or effort, but in combination with COCOMO and other models it can provide a very good estimate of software size in the beginning of its developmental life cycle. As such, it remains as one of the most widely used methods in the software industry, and we use them to develop our user-interface to proof the concept as outlined in the following.

3. Mapping the Estimation Models with the Software Developmental Life Cycle

A software development project typically goes through different stages or cycles. The widely accepted model known as the Software Development Life Cycle or SDLC, typically goes though certain sequential phases such as: planning, analysis, design, and implementation. Each phase is further subdivided into multiple activities and there are major outcomes of each phase. During the planning phase, a project is initiated and a feasibility study is performed that provides the basic information about the requirements or functionality of the software. The more detailed the requirements that can be gathered at this stage, the better will be the estimation of the size of the overall project. Based on the requirements in this early stage of the development, a project manager needs to estimate the size of the project, create a schedule, and plan accordingly for the amount of staff required to implement the rest of the project development phases. These early estimates act as baselines, which are typically modified or adjusted as more is learned about the functionality of the software while we progress through the subsequent phases. The fact that these estimates are required very early in the software development project makes it a formidable task. Most software development efforts fail due to a poor estimate of its size and other metrics right at the beginning of its life cycle. However in practicality, not all estimation models mentioned above can be used at this early stage. As such, it is important to apply appropriate techniques to obtain a realistic estimation of software size not only at this stage but also in other stages of its development.

In Figure-1 below, we provide a framework to map the various estimation models that should be used in various phases of the systems development life cycle. As detail requirements are not know in the planning phase, we propose to use the **industry standard** method towards the end of the planning phase to obtain a ball-park estimate of the software metrics. We use the PERT or Gantt chart to develop a project plan. During the early stage of the Analysis phase, detailed requirements will start to emerge, and we propose to use the **Functional Decomposition** or **WBS** Diagram to get a better estimate of the metrics, and accordingly, we update the PERT or Gantt chart. As we move further into the Analysis phase, more detailed requirements will emerge, and we propose to use **Function Point along with COCOMO** to obtain further accuracy of the estimations. While we continue with the PERT and Gantt charts to monitor and control the developmental effort, we compare the overall progress with the updated estimates by Function Points and COCOMO. This process continues until the end of the developmental effort.

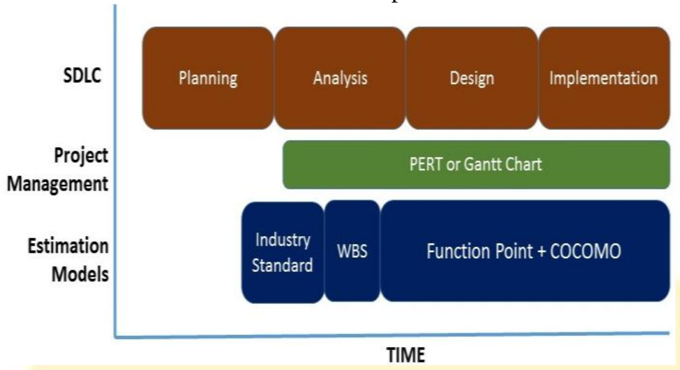


Figure-1: Mapping Estimation Models with SDLC Phases along with Project Schedule

4. Development of a User Interface to Estimate Software Size

In the following, we outline the steps of developing a user interface that can be repeatedly used to estimate software size during the major stages of its developmental life cycle as mentioned above. We use various attributes outlined in the Function Point Analysis and COCOMO Model to calculate the major matrices of a software development project such effort, time, and the number of people needed to develop the software. We also discuss the necessary attributes of the two analytical models and outline the steps necessary to develop and demonstrate our simple software package. Dennis et al. [1] outlined a three-step process to estimate effort, time and staffing requirement of a software development project, and we follow it in order to develop our user interface.

4.1 The Five Major Functional Components of a System

One of the primary goals of the Function Point Analysis is to evaluate system's capabilities from the user's point of view. To achieve this goal, the analysis is based on various ways users interact with an information system. From the user's point of view, a system assists them in doing their job by providing five basic functions. Two of these address the data requirements of an end user and are referred to as *Data Functions*. The remaining three functions address the user's needs by accessing internal data and are referred to as *Transactional Functions*. Table 1 briefly describes these functions [6].

Table 1: Five Major Components of the Function Point Analysis

Functional Components	Brief description
External Inputs	A process in which data crosses the boundary of a system from outside to inside. The data can be either business information or control information. User inputs or inputs from other systems are examples.
External Outputs	A process in which data passes across the boundary of a system from inside to outside. The data are used for reports or output files sent to other applications.
External Inquiry	A process with both input and output functionalities that require data retrieval from one or more internal logical files or databases.
Internal Logical Files	A user identifiable group of logically related data that resides entirely within the applications boundary and is maintained through external inputs.
External Interface Files	A group of logically related data that is used for reference purposes. The data resides outside the application and is maintained by other applications. For example I/O routines, sorting procedures, math libraries, run-time libraries, and etc.

In addition to the five functional components described above, there are two adjustment factors that need to be considered in Function Point Analysis: *Functional Complexity* and *Value Adjustment Factor*.

4.2 The Functional Complexity

The first adjustment factor considers the Functional Complexity for each unique functional component. Functional Complexity is determined based on the combination of data groupings and data elements of a particular function and can be rated as low, average or high complexity. Each of the five functional components has its own unique complexity matrix as shown in Table 2 [1][6]. For example, for External Input Functions, the complexity factors are: Low=3, Average=4, and High = 6.

Table 2: Steps of Calculating Function Points

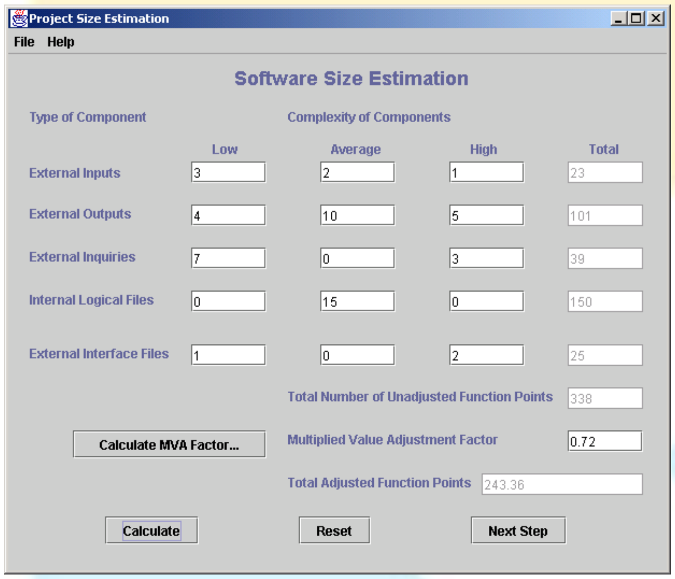
Type of Component	Complexity of Components		
	Low	Medium	High
External Inputs	_x 3 = _	_x 4 = _	_ x 6 = _
External Outputs	_x 4 = _	_x 5 = _	_ x 7 = _
External Inquiries	_x 3 = _	_x 4 = _	_ x 6 = _
Internal Logical Files	_x 7 = _	_x 10 =_	_ x 15 = _
External Logical files	_x 5 =_	_x 7 = _	_ x 10 = _
	Total Unadjusted Function Points (UFP)		
	Multiplied Value Adjustment Factor (MVA)		
	Total Adjusted Function Points (AFP)		

4.3 Calculating the Unadjusted Function Points

After the components or functionalities have been classified as one of the five major components, a ranking of low, average or high is assigned for each component. Refer to Table 2. The following steps can be used to calculate the size of a project as Unadjusted Function Points (UFP):

- Count or estimate all the occurrences of each type of functional component.
- Assign each occurrence a complexity weight.
- Multiply each occurrence by its complexity weight, and total the results to obtain a function count for each component.
- Add up all the function points for all five functionalities or components.

The main screen of our user interface used to calculate the Total Number of Unadjusted Function Points is shown in Figure-2. As we enter the number of components in each of the five functional areas with various complexities, the total unadjusted functional point is automatically calculated.



Software Size Estimation

Type of Component	Low	Average	High	Total
External Inputs	3	2	1	23
External Outputs	4	10	5	101
External Inquiries	7	0	3	39
Internal Logical Files	0	15	0	150
External Interface Files	1	0	2	25

Complexity of Components

Total Number of Unadjusted Function Points: 338

Multiplied Value Adjustment Factor: 0.72

Total Adjusted Function Points: 243.36

Figure-2: Main Screen - Calculation of Unadjusted and Adjusted Functions Points

4.4 The Value Adjustment Factor

The Unadjusted Function Point count is multiplied by the second adjustment factor called the Value Adjustment Factor. This factor considers the system's technical and operational characteristics and is calculated by answering 14 questions as shown in Table 3.

Table 3: Fourteen General Characteristics of a System

General System Characteristic	**Brief Description**
Data Communications	The data and control information used in the application are sent or received over communication network.
Distributed Data Processing	Distributed data or processing functions required by the application.
Performance	Application performance objectives, stated or approved by the user, in either response or throughput, influencing the design, development, installation and support.
Heavily Used Configuration	A heavily used operational configuration, requiring special design considerations.
Transaction Rate	The transaction rate is high and influences the design, development, installation and support of the application.
On-line Data Entry	On-line data entry and control information functions are required by the application.
End-User Efficiency	The on-line functions provided emphasize a design for end-user efficiency.
On-line Update	The application provides on-line updates to the internal logical files.
Complex Processing	Complex processing is a characteristic of the application requiring multiple data access and manipulation.
Reusability	The application and the code is designed, developed and supported to be usable for other applications.
Installation Ease	An application can be easy to convert and install. A conversion and installation plan or tools are provided to test the application.
Operational Ease	Operational ease can be like effective start-up, backup and recovery procedures of the application.
Multiple Sites	The application has been specifically designed, developed and supported to be installed at multiple sites of an organization.
Facilitate Change	The application has been specifically designed, developed that facilitate change.

The 14 General System Characteristics is rated on the scale of 0 (not important) to 5 (very important) and summed [1][6]:

$$N = \sum_{i=1}^{14} F_i$$

A Complexity Factor, C is then calculated using the formula: C = (0.65 + 0.01 x N).
Figure 3 illustrates the calculation of Complexity Factor using a simple scale for each of the 14 system characteristics – ranging from 0 to 50 for simplicity (which equivalents to 0 – 5). This screen is accessed from a button titled **Calculate MVA Factor** on the main screen as shown in Figure-2 After the calculation of the Complexity Factor, it can be closed and the result can be entered on the main screen in the appropriate text box next to the above mentioned button as shown in Figure-2.

4.5 Calculating the Adjusted Function Points

The Adjusted Function Point (AFP) Count is obtained by multiplying the complexity factor and the Unadjusted Function Point (UFP): AFP = UFP * C, as can be obtained by pressing the **Calculate** button as shown on the Main Screen in Figure-2.

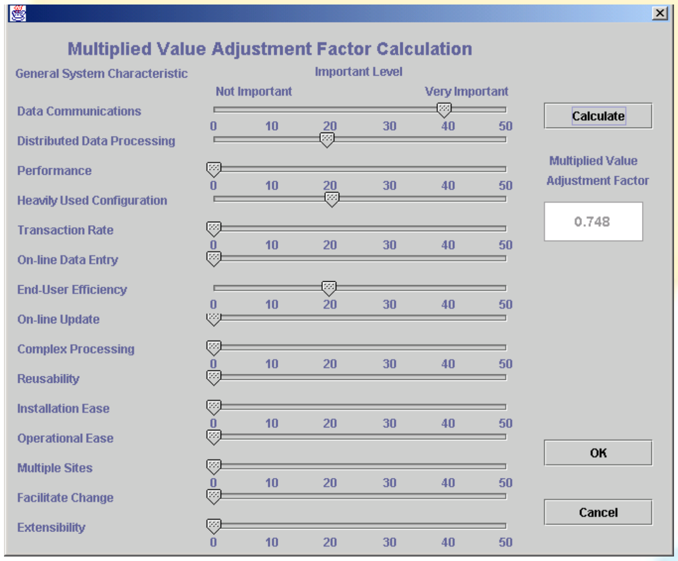

Multiplied Value Adjustment Factor Calculation
General System Characteristic
Important Level
Not Important
Very Important
Data Communications
Distributed Data Processing
Performance
Heavily Used Configuration
Transaction Rate
On-line Data Entry
End-User Efficiency
On-line Update
Complex Processing
Reusability
Installation Ease
Operational Ease
Multiple Sites
Facilitate Change
Extensibility
Calculate
Multiplied Value Adjustment Factor
0.748
OK
Cancel

Figure 3: Calculation of Value Adjustment Factor using 14 System Characteristics

4.6 Calculating the Lines of Code

Once we know the adjusted functions points, the lines of code can be calculated depending on the programming language used. Jones[15] provided a conversion factor from Function point to lines of codes for various languages as shown in Table-4 [1][6]. For example, in Visual Basic language one function points will yield 30 lines of code. In our simple software, this step is performed by pressing the **Next Step** button on the Main Screen as shown in Figure 2 and selecting a programming language in the third screen as shown in Figure 4.

Table 4: Conversion of Function Point to Lines of Code in Various Languages

Language	Approximate Number of Lines of Code per Function Point
C	130
COBOL	110
Java	55
C++	50
Turbo Pascal	50
Visual Basic	30
Power Builder	15
HTML	15
Excel, Access Packages	10-40

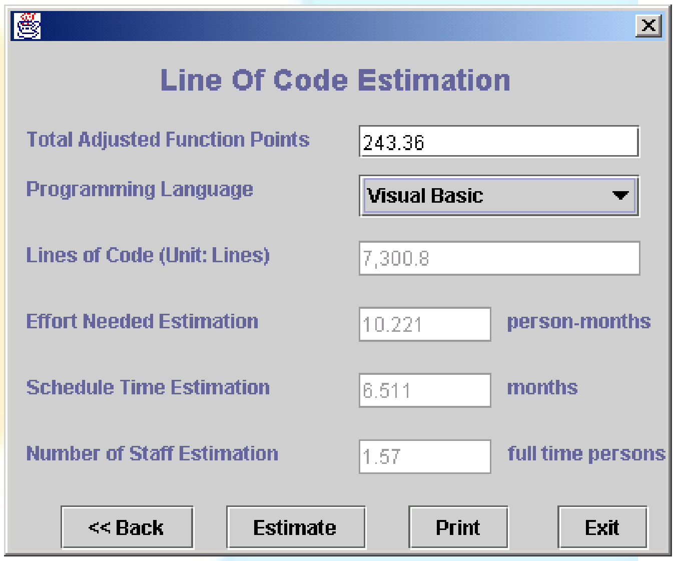

Line Of Code Estimation
Total Adjusted Function Points: 243.36
Programming Language: Visual Basic
Lines of Code (Unit: Lines): 7,300.8
Effort Needed Estimation: 10.221 person-months
Schedule Time Estimation: 6.511 months
Number of Staff Estimation: 1.57 full time persons
<< Back
Estimate
Print
Exit

Figure 4: Calculation of Effort, Time and Staff using a Particular Language

4.7 Calculating the Effort, Time and Staff

Once the numbers of lines of code (LOC) are estimated, effort, time and staff required can be estimated using COCOMO Model [11]. As it was mentioned before, COCOMO computes software development effort as a function of program size, and program size is expressed is estimated in thousands of source lines of code (KLOC). For small to moderate-size projects, effort can be calculated as [1]:

Effort (person-months) = 1.4 x KLOC.

Once the effort is estimated, scheduled time for the project can be calculated using the formula:

Time (months) = 3.0 x (person-months)$^{1/3}$.

The estimated number of staff needed to complete the project is calculated as: Effort/Time. Figure 4 illustrates the calculation of all three values using our interface.
Once we have an understanding of these three project metrics at the early stage of the software development life cycle, we can use other tools like WBS along with the PERT or Gantt chart to break the project into multiple tasks and create a baseline schedule for the project.

5. CONCLUSION

Software size estimation is a crucial input for the cost estimation process at the early stage of its developmental life cycle. Many software projects fail due to poor or inaccurate size estimation. Accurate and early estimation requires proper identification of the problem domain, including functional size and complexity, and then application of proper techniques to estimate the size. We have discussed four different models or techniques that are found to be practical in estimating software size and other metrics during the various stages of its developmental life cycle. We have also discussed the pros and cons of these models. Each method provides different levels of details and accuracy. We thus mapped the applicability of each of these models with the various phases of the software development life cycle. We have proposed that a project manager should use the industry standard method towards the end of the planning phase, followed by the Functional Decomposition or WBS at the beginning of the Analysis phase. During the middle of the Analysis phase, one should use the Function Point Analysis along with the COCOMO model to obtain an accurate estimation of the software metrics. And this latter methods should be used repeatedly or as necessary during the rest of the developmental life cycle to help with the monitoring and controlling of the project.

We have also elaborated the steps of Function Point Analysis and COCOMO Model, and developed a simple user interface that can be repeatedly used by a project manager to calculate the software size in terms of effort, scheduled time, and staff needed during the major stages of its developmental life cycle. A simple user interface can help automate the estimation process, compare the results with the actual work progress, and adjust the schedule as necessary.

6. REFERENCES

[1] A. Dennis, B. H. Wixom, and R. M. Roth, *Systems Analysis & Design*, Publisher: John Wiley & Sons, Inc., Hoboken, NJ (2009).

[2] Bill Meacham, "Estimating Software Size," http://www.bmeacham.com/Estimating/Estimating.htm. Accessed February 1, 2015.

[3] M. A. Rob, "Project Failures in Small Companies," *IEEE Software*, November/December issue, pp. 94-95 (2003).

[4] Hareton Leung, "Software Cost Estimation," https://www.st.cs.uni-saarland.de/edu/empirical-se/2006/PDFs/leung.pdf. Accesses February 15, 2015.

[5] B. Boehm, C. Abts, and S. Chulani, "Software Development Cost Estimation Approaches – A Survey," *Annals of Software Engineering*, Vol. 10, pp. 177-205 (2000).

[6] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Publishing (2014).

[7] M. Jorgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, pp. 33-53 (2007).

[8] G. B. Shelly and H. J. Rosenblatt, *Systems Analysis & Design*, Course Technology (2013).

[9] *The Delphi Method: Techniques and Applications*, edited by Harold A. Linstone and Murray Turoff (2002).

[10] N. Sökmen, "Functional Decomposition Based Effort Estimation Model for Software-Intensive Systems," *International Journal of Computer, Information, Systems and Control Engineering*, Vol. 8 No. 9, pp. 1505-1509 (2014).

[11] B. Boehm, *Software Engineering Economics*, Prentice-Hall Publishers, Englewood Cliffs, NJ, (1981).

[12] A. J. Albrecht, "Measuring Application Development Productivity," *IBM Application Development Symposium*, pp. 83-92 (1979).

[13] A.J. Albrecht, and J. E. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. On Software Engineering*, Vol. 9. No. 6, pp. 639-648 (1983).

[14] J. Desharnais and A. Abran, "Approximation Techniques for Measuring Function Points," *Proceedings of the 13th international workshop on software measurement (IWSM 2003)*, pp. 272-286 (2003).

[15] C. Jones, *Estimating Software Costs*, McGraw-Hill Publishing (1998).