A Reasonable Software Maintenance Effort Estimation Technique

ESHETU DERESU DISASA

A Thesis Submitted to

Department of Computing

School of Computing and Electrical Engineering

Presented in Partial Fulfillment of the Requirement for the Degree of Master of Science in Software Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama

October, 2017

A Reasonable Software Maintenance Effort Estimation Technique

ESHETU DERESU DISASA

Advisor:

Dr. MESFIN ABEBE

A Thesis Submitted to

Department of Computing

School of Computing and Electrical Engineering

Presented in Partial Fulfillment of the Requirement for the Degree of Master of science in Software Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama

October, 2017

DECLARATION

I hereby declare that this MSc Thesis is my original work and has not been presented for a degree in any other university, and all sources of material used for this thesis have been duly acknowledged.

Name: Dr. MESFIN ABEBE

Signature: _____

This MSc Thesis has been submitted for examination with my approval as thesis advisor.

Name: ESHETU DERESU

Signature: _____

Date of submission: October 03, 2017

DEDICATION

I dedicate this entire thesis report to my beloved mother and father for their tireless support they accorded to me ever since I was children. I thank them for the spirit of hard work, courage and determination they had instilled in me throughout my school days till today. I also honor and owe my dear sisters and brothers for the happiness and appreciation for the guidance protection and idea support they offered me.

ACKNOWLEDGEMENT

God is good all the time, He keeps answering all my pray beyond my expectations. I prayed for one, He answered me with overflow. I can't express my feeling; just I keep saying Hallelujah, Glory to God. Here I am only because of You. Thank you GOD.

Interdependent is a higher value than independence. This work is a synergistic product of many minds. I am grateful for the inspiration and wisdom of many technology thinkers and for the trans-generational resources and roots of this technology wisdom. Here, I want to thank the people that have had a positive influence on my personal life and thereby also contributed to this thesis.

I am deeply indebted to my advisor, Dr. Mesfin Abebe, for his encouragement, support, and constructive advice. I have learned not only from his deep knowledge and experience in software engineering but also from his remarkable personality. He has been much more than just an advisor; he has been a tutor, mentor and teacher at the same time. He has taught me what doing research is all about, and how to tell the story that emerges from the research.

And also I am grateful for many students, classmates, friends at Adama Science and Technology University and others who have tested this study works and have given feedback and encouragement.

Finally, I thank my parents, my brothers, my sisters, and other family and my friends for their support. I want to especially thank my parents for teaching me to strive for maximum output – to which I've always silently added 'with minimal input'. But most of all I am grateful to my mother for her continuous love and support. I am doing my very best to maintain it.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

COCOMO	Constructive Cost Model
DBMS	Database Management System
FP	Function Point
FPA	Function Point Analysis
FPC	Function Point Count
IEC	International Electro-technical Committee
IEEE	Institute of Electrical and Electronic Engineers
IFPUG	International Function Point Users Group
INSA	Information Network Security Agency
IRBC	Improved Requirement Based Complexity
KLOC	Kilo Line of Code
LOC	Line of code
MAF	Maintenance Adjustment Factor
MCF	Maintenance Change Factor
MMLR	Multi nominal Multi-variate Logistic Regression
NASA	National Aeronautics and Space Administration
OS	Operating System
RBMC	Requirement Based Maintenance Complexity
RBMEE	Requirement Based Maintenance Effort Estimation
RSMEET	Reasonable Software Maintenance Effort Estimation Techniques

SCM	Software Configuration Management
SDLC	Software Development Life Cycle
SI	System integration
SLOC	Source Line of Code
SMPEEM	Software Maintenance Project Effort Estimation Model
SRS	Software Requirement Specification
SSE	System Slice Encoding
SU	Software understanding
SWEBOK	Software Engineering Body of Knowledge
UFP	Unadjusted Function Point
UNFM	Programmers' unfamiliarity
VAF	Value Adjustment Factor

ABSTRACT

Software development efforts result in the delivery of a software product that satisfies user requirements. Accordingly, the software product must change or evolve. Once in operation, defects are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle begins following a warranty period or post-implementation support delivery, but maintenance activities occur much earlier.

Any activity after the delivery of the system or software is considered as maintenance, if so we have to know how much effort is needed to make the change. But at first it is known that the effort to be assigned has a direct relation with the size of the software to be maintained. So it is much better if we measure the size of the system or software to be maintained before proceeds to the assignment of the effort. In this study we have given a detail attention to the size measurement of the system or software to be maintained for the accuracy of the effort estimation is only as good as the accuracy of the size measurement of software to be maintained. We have used LOC approach which is adopted from the basic COCOMO developed by Barry Boehm and FP approaches which is again adopted from IFPUG organization. The Line of code is understandable at the developer side but more difficult to communicate with the client who has no understanding of the complexity behind the software because the client judge the software based on the functionality of that software itself.

In order to make measured size used as same unit we have used IFPUG conversion table to convert the size measured in LOC to UFP. After the conversion of the LOC to FP the new concept of VAF is applied to adjust the unadjusted FP.

According to the Basic COCOMO the effort is measured in terms of size of the software to be maintained since the effort required to maintain the type of the requested maintenance type is highly depend up on the size of the system to be maintained. As the size of the maintenance system increased the effort also increased in the same rate. Then we conclude our model as the effort required to fix the requested type of maintenance have an exponential factor to account for the relative economies or diseconomies of scale encountered as a software project increases its size.

Keyword: *Effort Estimation; Maintenance; Maintenance Estimation; Software Estimation; Software Maintenance*

CHAPTER ONE

BACKGROUND OF THE STUDY

1.1. Introduction

It is better if it given a detail attention to the software maintenance effort estimation as other software development life cycle since its cost is as doubled as development cost which means, in every software spent two dollar for its development, it needs four dollar to maintain it [1]. So we have to care about the maintenance stage of the software. Think when the cost of the development is high the cost of maintaining that software also increased with that rate at least.

In the study of this thesis we have set our focus on the size of the software to maintained for the size of the software is highly affect the number of efforts needed to fix or maintain the requested maintenance type. So in order to achieve the thesis intention which is the software maintenance effort estimation, it needs to careful when sizing the software to be maintained because committing an error in software sizing were result error estimating for software to be maintained. So to avoid such error in the software maintenance sizing we have decided to use the hybrid approaches to measure the software to be maintained. We have used the function point approach and line of code approaches in order to size the software to be maintained.

Under this chapter we deal with different topic which is categorized under different section within the chapter. Section 1.2 discusses about the general information on the given study area. It gives a brief description about background information on the study area. Section 1.3 of the chapter discusses the motivation toward the thesis of the researcher/author. Section 1.4 is about the statements of problem which is aimed to be solved by the research result. Section 1.5 discusses about the objectives of the research which is categorized as general objectives and specific objectives. In General objectives sub section the main objective or main target of the research is discussed. Under specific objectives sub section the detail path toward the main target is discussed.

1.2.Background of the study

Software maintenance is defined as the process of modifying, for update or repair, existing operational software, but leaving its primary functions intact. Simplified, software maintenance can be understood as any work that is undertaken after delivery of a software system [1]. The Institute of Electrical and Electronics Engineers (IEEE) defines software maintenance as follows:

Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment [2].

Software development efforts result in the delivery of a software product that satisfies user requirements. Accordingly, the software product must change or evolve, so this change or evolution is considered as maintenance. Once in operation, defects are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle begins following a warranty period or post-implementation support delivery, but maintenance activities occur much earlier.

Software maintenance is an integral part of a software life cycle. According to the definition stated on the SWEBOK Guide V3.0, software maintenance is defined as the totality of activities required to provide cost-effective support to software. Activities are performed during the pre-delivery stage as well as during the post-delivery stage. Pre-delivery activities include planning for post-delivery operations, maintainability, and logistics determination for transition activities and Post-delivery activities include software modification, training, and operating or interfacing to a help desk.

Whenever a change is made to a piece of software, it is important that the maintainer gains a complete understanding of the structure, behavior and functionality of the system being modified. It is on the basis of this understanding that modification proposals to accomplish the maintenance objectives can be generated. As a consequence, maintainers spend a large amount of their time reading the code and the accompanying documentation to comprehend its logic, purpose, and structure. Available estimates indicate that the percentage of maintenance time consumed on program comprehension ranges from 50% up to 90% [6]. Program comprehension is frequently compounded because the maintainer is rarely the author of the code (or a significant

period of time has elapsed between development and maintenance) and a complete, up-to-date documentation is even more rarely available.

The objective of software maintenance is to modify existing software while preserving its integrity. The international standard also states the importance of having some maintenance activities prior to the final delivery of software (pre-delivery activities). Notably, IEEE 14764 emphasizes the importance of the pre-delivery aspects of maintenance planning [3].

The categories of maintenance defined by ISO/IEC are as follows: [8].

- Corrective maintenance. Reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance. Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance. Modification of a software product after delivery to improve performance or maintainability.
- Preventive maintenance. Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

The ISO Standard on Software Maintenance [ISO14764] classifies Adaptive and Perfective maintenance as enhancements. It also classifies Corrective and Preventive maintenance as corrections. Preventive maintenance, the newest category, is defined as maintenance performed for the purpose of preventing problems before they occur. Preventive maintenance is most often performed on software products where safety is critical [8].

The main aim of the thesis is, since the maintenance is an integral part of software life cycle; we have had some model which is accurately estimate software maintenance effort. So the research intended to study the current states-of the-art of maintenance effort estimation and provide reasonable software maintenance effort estimation technique.

1.3.Motivation

Despite software maintenance is one of a core phase among software development life cycle (SDLC) many software company especially in our country does not recognize as it is. So there is still a gap of understanding about software maintenance concept.

Since the software maintenance have to considered as one of the SDLC phase in the software development environment we have to look forward on how about estimation effort of software maintenance. So we as a project manager have to have some model in order to estimate the maintenance effort which is how much effort will be assigned and enough to fix the type of maintenance requested.

Firstly, the current misunderstanding about software maintenance is among my reasons to be interested in the field which I am supposed to make our country developer aware of software maintenance, which I mean its original definition.

And secondly, there are so many model developed yet by different researchers. But every developed model has its own limitation like accuracy problem, applicability problem, availability of their requirements, depends on someone else and so on. So this study supposed to avoid those limitations if possible or to reduce them.

Generally, the above points and the gap or problem those are discussed in the statement of problem section motivated the author to focus on the area of software maintenance.

1.4.Statement of the Problem

So many models are developed for software maintenance effort estimation by different researchers. Among them; Basic COCOMO, Intermediate COCOMO, SMPEEM are the majors. There are also a good research works that aim at developing a model that improve estimating effort of software maintenance. Despite the fact that the model is working, till there is a problem in software companies to estimate accurately the maintenance effort, for COCOMO model is a model suitable for development management rather than for maintenance. COCOMO also considers the software product as a single homogenous entity but in reality most large systems are made up several smaller sub-systems and these sub-systems may have widely different characteristics [25].

It is known that to estimate accurately maintenance effort is very difficult since it depends on the maintenance team skill and their productivity rate but it is acceptable that to estimate nearly by using available resources such as developer experience, recorded previous data, documents, also based previous related software [26].

Software maintenance effort estimation models developed before has their own benefit and limitations. For example, Line of code based developed model is limited and varies depending on the nature of underlying programming language. SRS based approach is not applicable for open source software since their SRS document is not available. So we have to make a hybrid approach that may solve the limitation happen in the model developed using a single approach.

So the intention of this work is to fill the gap that exist in the software companies which is to improve the maintenance effort estimation with a reasonable way by assessing the models developed before by different researchers, especially on the model developed by using function point and Line of code approaches.

1.5.Research Questions

Up on the way developing a Reasonable Software Maintenance Effort Estimation Techniques, some questions will be answered at the end. These questions are described as follows:

RQ1: What are the constraints that may affect the size of the software maintenance projects?

RQ2: What are the direct and indirect factors that affect the cost of software maintenance project?

RQ3: What is the adjustment factors mostly used for estimating software maintenance effort?

1.6.Objectives

The thesis is intended to meet the following target or objectives. The specific objectives of the study lead us to the general objectives of the thesis.

1.6.1. General Objective:

- The general objective of the research is to develop a Reasonable software maintenance effort estimation technique.

1.6.2. Specific Objectives:

- To collect the metrics or constraint that may have a relationship with the software maintenance effort.
- To categorize the constraints based on directly and indirectly relation with the software maintenance effort.

- To identify the constraints those are highly correlated to the software maintenance effort.
- To rank the above identified constraints based on their level of relation to affecting software maintenance.
- To improve the accuracy of estimating software maintenance effort by using the above ranked constraints.
- To develop the reasonable software maintenance effort estimation technique as expected result of my research.
- To test the performance of developed model.

1.7.Scope and Limitations

The research is initiated with the assumption that the project or software that is going to be maintained is well-documented and organized projects. The developed model by the researcher may be inaccurate with not well documented software projects. So the research scope is limited to the well documented SRS of the software which is in understandable way. The model that is developed as the best fit result of the thesis is applicable for any type of software maintenance type since the model has been developed by taking account some metrics those may directly related and also those indirectly factors that may factors affect to the software maintenance effort like function point and line of code. The expected result of the thesis is as explained to develop a model that can be used to estimate software maintenance effort using a reasonable metrics by avoiding limitations happened in other estimation models.

The researcher faced some problem on data collection and availability of data on time when required at the company side if so the research will decide to continue with online data that may not be accurate. Also there may be another limitation like company may not be interested to apply the model in order to estimate the software maintenance effort; this may precede the problem like lack of project to test the model performance.

Generally to make precise what we have done is; a reasonable software maintenance effort estimation technique by using a hybrid approach which is composed of line of code approach and the function point approach. And the model measure the effort in terms of man-month, it mean that the amount of man requires per month to finish the work.

1.8.Evaluation and testing

The data collection completed in the case study, furthermore analysis, interpretation and summarization of the data performed and used as an input for main research area. The model is developed based on the data collected and the input of the developed model is calculated by using the regression analysis which is a non-linear regression for our expected result may not be always linear. Based on the survey result and literature design a model and finally test the model.

Based on the data gathered from different firm or site which is reserved for research purpose the developed model is tested and verified and also the performance of the model is tested. The testing of the developed model is taken place with the data which is left from the experimental activities.

1.9.Application of Results

The main contribution of this research is on finding an efficient method of effort estimation for Software Maintenance. This thesis work is believed to produce an effective approach for Software maintenance effort estimation. The thesis tried to put the two different approaches together and attempted to fill the gap currently present on the area of the software maintenance project. The two approaches are used the way they are essential which means they have been used as they appear or they have been modified as necessary. The developed model was tested in terms of performance, accuracy, other factors needed to be checked. Therefore, the model may have a significant contribution for the development of software maintenance effort estimation in the field of software engineering.

1.10. Thesis organization

The thesis is organized as follows; chapter two contains review of literature that shows the research direction and problems area that will be solved by this research. And also it contains related works that was identified based on literature review and preliminary survey.

Chapter three describes about the methodology used throughout the thesis works. The chapter discusses the detail theory and the modification made to adopt the approaches to reach the best fit model for making the prediction equation more accurate than before.

Chapter four of the thesis is about the architecture of proposed model which give the basic concept how the proposed model is going to work and how it is developed from the very beginning of the ideology. The chapter also includes not only the concept of proposed model architecture but also it includes the model architecture of previously developed model by different scholars or authors as a literature review.

The Fifth chapter covers about the activities of experimenting of the model and also validating the developed model. In this chapter we also include the idea of regression analysis in order to get best prediction model with best accuracy.

Finally, in the last chapter six of the thesis the conclusion made on the thesis result, the contribution of this research work and recommendation on possible future work related to the thesis are presented.

CHAPTER TWO

LITERATURE REVIEW

2.1. Introduction

Under this chapter we presented a topic which gives a highlight and a general concept about the approaches we used in order to obtain the result. So the first section of the chapter deals with a general overview about the Software Maintenance, the second section of the chapter is about types of software maintenance which is the aim of maintaining the given software whether it is to adapt the new environment, modify the existing system before an error is found or to correct the existing system to add a new feature to it. And the third section of the chapter discusses about the factors that may affect the software maintenance costs, which is categorized as direct factor and indirect factor. The direct factor is that factor that directly can make differences to the costs such as complexity, human capacity, quality of documentations and etc. In the other hands the indirect factors is that may affects indirectly the maintenance costs such as application experience, application time, staff stability and so on. And the last section of the chapter is about the methodology used to achieve the research hypotheses and discusses about the hybrid approaches which the core of the findings.

2.2. Overview of Software Maintenance

In the very beginning, software is developed by considering its life span of ten to fifteen years with some modification throughout its operations. But this is not always applicable according to the planned one. So, in order to meet the planned life span of the operational software we have kept it operational with minor or major modification as needed termed as maintenance, for we are in charge of keeping it operational as owner of the product. Unlike other physical products, software product exists only in binary form, which means that it is not subjected to wear or decay [3]. So in theory, software could run for years without modification. However, this does not occur in practice. Because it is the flexibility agent for most systems, software must adapt as the environment in which it operate changes. There are many reasons for which the software needs to be maintained [7], among them some of them are listed below:

- ***Enhancement***: - Enhancement is sometime considered as expansion of the system in different ways such as adding business, features and so on. Enhancement is not always

about to add features it may apply as cropping existing feature from the system. Some changes are often needed to add features and functionality of existing or operating system and also to alter the software to comfort with the changing business rule and keep the users happy. Changes may also be needed to remove or alter functionality, for example, when the structure of organization changed the system also needed to be changed to fit with the newly organization structure. So enhancing the software is categorized among the reason for maintaining operating software, no matter whether the bug found or not for enhancement is highly related to adding features or functionality to operating system [3].

- ***Environment upgrade***: - When we shift this concept from the system to human being it may be same as changing life standard, changing living cities and stabling to the new city, its culture, language and so many. These concepts also true with the system we may change the platform on which the system may keep operate on, or other. The software does not operate in a static world. The platform that it run on and the environment in which it operate needed to be periodically updated, upgraded and replaced. In fact, the external disruptions like machine and platform revisions frequently create an opportunity for change. During the environment upgrade the enhancement may also occurs because the existing system may not well function with the new environment. Some features or functionality may be upgraded too. For example, some feature of the existing system may not work well on the new version of platform or hardware [3].
- ***Repairs***: - This is those activities taken place after the discoveries of defects or malfunction within the operating system. Defects that must be fixed as the software application are used. When we say defect, there are different perspectives on the understanding of what the defect means. Some group says: defect is a misunderstanding of what the user wants, this is caused from the communication gap between the developer and the clients or it may be from unbalanced understanding level of the developer and clients. And the others say: it is a latent error embedded in the software that surface only through prolonged use. For example, error in mismatched interface layer with that of business layer. However; independent of the causes it needed to be fixed. In reality such fixes are often made as enhancements are made because it make sense technically and economically to do both together [3] [7].

- ***Performance Improvement***: - The performance of operational software or system may be down because of age of the system itself or platform or other factors, so the maintenance activity also needs to consider the performance issue on such side. After the software is in an operation for a time it may need to be tuned and optimized. The main thrust here is to improve speed, performance and access, especially when the system becomes mired down with the tasks that execute at the background and with the temporary files that clutter the storage devices. Again, performance improvements are done along with the other tasks as the software is updated and released to the field [3][7].
- ***Configuration Changes***: - It is assumed to be developed software or system must fulfill the minimum requirement of its user. Finally, when the software is moves from the development environment to the operational environment, different users might configure the software differently. Because they run on different platforms and have different hardware configurations, the software may have to be tailored specifically for them. As such, some changes to one configuration might be needed while others will go completely unchanged [3][7].

However, referring those discussed above changes without discussing who the change agents are, what processes are used, what the aligned tasks are, and what constitutes the maintenance environment leads one to the wrong conclusions relative to what constitute software maintenance. This infrastructure must also be discussed along with those who use it to process these change and update the software. The infrastructure includes those facilities, equipment, staff, specialized tools, and process used to make changes and manage the releases and distribution of updated products to the field. This infrastructure also include the effort undertaken to provide users training and support as they use the product and patch them in order to keep them operational. How many efforts may enough to made changes for requested maintenance activities. In response we have to consider the following four related software maintenance to correct this situation [7][9].

- **Maintainer**: - Refers to the person or organization that performs the maintenance activity. The workforce may be made up of contractors, employee or a mix. Generally those personnel involves in the activities of software maintenance of the requested change sent by the clients [9][7].

- **Maintenance environment:** - Refers to the set of facilities, equipment, software and other resources used to maintain the system. Such an environment differs from the one used development in that it uses actual operation equipment and interface rather than simulated one for we are commit the maintenance activity in the real operational environment of the running system [7][8][9].
- **Maintenance process:** - Refers to the set of processes used to maintain and sustain the software after it has been delivered. These processes involve more than development. It is the process set to be followed during the maintenance activity which is at organization level; in general we can call it a protocol. For example, they may address distribution control and adaptation to make the software work at different sites [8] [9].
- **Sustaining engineering:** - Refers to the continuing engineering and technical support that is needed to sustain maintenance operations. This activity my include such activities as user training and support, staffing and operating a help desk, keeping facilities and equipment up-to-date, performing configuration management, managing software licenses, performing quality assurance, managing networks and administrating security [7][8][9].

2.3. Types of maintenance

Software maintenance classification is varying as the organization set to categorize them varies. According to the categorization of International Electro-technical Committee activities suited with the needs of software maintenance, software maintenance is categorized in to five different categories [8], namely:

- Adaptability maintenance,
- Evolution of maintenance,
- Improvement of maintenance,
- Preventive maintenance and
- Repair of maintenance.

Hence the ISO/IEC further merges the five categories of software maintenance activities together according to their relation [9]. So as a merging strategy of ISO/IEC improvement of maintenance and evolution of maintenance is merged to Integrity maintenance, preventive maintenance and repair of maintenance is merged to corrective maintenance. In other word improvement and evolution are merged and becomes Integrity maintenance, preventive and repairs are merged and

become Corrective maintenance. Therefore, software maintenance is broadly categorized into the following three categories.

1. Adaptive maintenance
2. Corrective maintenance
3. Integrity maintenance

Adaptive maintenance

Adaptability maintenance, adapt the software to do the changing operating environment changes. It is enhancing the degree of adaptability of the running software or system within changing environment. The main changes include the impact of the system of rules or laws and rules, hardware configuration, data format or structure of the text volume, system software, etc.

Corrective maintenance

Correction of maintenance, the activities of fixing the errors after identified by either user or developer themselves. It is the operations for maintenance of the system running in the development process resulting in the testing and inspection did not find the error for the correction. It includes design errors, logic errors, coding errors, documentation errors, data errors, etc.

Integrity maintenance

Integrity maintenance is the activity intended mainly to make the user of the system or software satisfied. It is about expanding the functionality and improving performance and make changes and expansion to meet the changing needs of users. The main contents include the expansion or enhancements and changes made, to improve performance and make changes, for ease of maintenance and make changes, etc.

It checks whether the added feature correctly integrated with the running system and operates as the client needs it. It is not only about adding the new features but also do the removal of obsolete features from the system up on the request of user or client. So it has to be checked whether it is working as expected once the feature is removed from the operating system or software.

2.4. Factors affecting Software Maintenance Cost

There are so many factors that may affect the costs of software maintenance. Some of them are visible as soon as they applied and others are recognized by the help of other factors [27]. Software maintenance costs affect the main factors of indirect/ non-technical or direct /technical factors.

2.4.1. Indirect factors

The general indirect factors which affect software maintenance costs include Application experience, staff stability, application time, the external environment, support environment, user needs, organization standard, programmer/maintainer initiation etc.

- Application experience: If the application areas of software development experience, the definition of system requirements can be completely accurate, adaptive maintenance was relatively small. If a software in new areas, the demand is very different understanding and practice, to change the demand for adaptive maintenance workload will be enormous.
- Staff stability: If the developer responsible for maintaining part of their own development, no other software engineers to spend time to know the code, maintenance costs will be significantly reduced.
- Application time: Early in the software applications, coding errors when the concentration of the emergence of a larger correction of maintenance workload. With the software running, some of the older procedures, integrity and adaptability of the heavy workload, the more abandoned to maintain the structure of the more maintenance costs will increase accordingly.
- External environment: External environment here refers to the soft environment, namely, business rules, workflow, reports, and so on. If a software overreliance on the external environment, when the external environment changes, the software to do the appropriate changes. Adaptive maintenance workload will be enormous.
- Support environment: In the software life cycle, if the software running the support environment (including hardware, network and system software) changes, the software needs to change accordingly to accommodate the new support environment, adaptation to maintain the workload will be enormous.

- User needs: Users of the software is the gradual understanding of hair. As the software is running, users will deepen the understanding of the software, functionality and performance may be put forward higher requirements for perfection of the workload will increase.
- Organization standard: The standard of the organization may also affect the maintenance cost. The communication way may take some amount of time from the maintenance time or the frequency of meeting may also affect the maintenance.
- Programmer/maintainer initiation: The moral of the programmer also considered as the main factor that affect the maintenance. This may include the way the company threat the programmer may matter.

2.4.2. Direct factors

Technical factors include general software complexity, development of human capacity, documentation quality, configuration management technology, modern programming specifications, and database scale.

- Software complexity: If the software structure is more complicated to carry out maintenance software needs more time to read, modify the program also takes a long time, maintenance workload will increase significantly.
- Development of human capacity: Systems Analyst resistant full needs analysis; software maintenance workload will be reduced. Programmers have a good programming style, software, easy to read easy to understand, maintain the workload will be reduced. Testers using advanced methods, testing more fully, the software error will exist a significant reduction.
- Documentation quality: Documentation for software maintenance support. Maintenance personnel need a detailed description of the software system to help familiar with the system, identifying and correcting errors and improving the system to meet user requirements change or adapt to changes in the system environment. If there is a clear, complete and concise documentation to support, maintenance costs will decrease.
- Configuration management technology: Configuration management including configuration program, configuration identification, version management, change control, system integration, status reports, configuration auditing. Effective control of software

configuration management technology to maintain and strengthen the management of the maintenance work, it can help control maintenance costs.

- Modern programming specifications: Follow the systems engineering and software engineering principles and methods, the use of modern programming practices, including: documentation and variable Naming, prepare single-entry single-exit modules, to improve cohesion, reducing coupling, the use of structured development methods. If the software is a unified programming specification, can effectively reduce maintenance costs.
- Database scale: Software maintenance is a lot of work to maintain the database. The size of the database (including database objects and data) is an important factor affecting the maintenance workload. More data, you may want to re-organize the database and reconstruction, maintenance workload will increase significantly.

2.5. Methodology

2.5.1. The Line of Code

The most commonly used metric to represent software size is the Line of Code (LOC). This is more usually written as KLOC or one thousand lines of code **[6]**. Lines-of-Code is the oldest and the most widely used. First popularized by Barry Boehm in his Constructive Cost Model (COCOMO), it has since become the basis for a vast array of software cost estimating tools. Although other techniques have made major in-roads in the world of MIS applications, Lines-of-Code is still the standard for applications with lots of behind the scenes processing. This includes system programming, embedded programming, and most scientific programs. Even in the MIS world, it remains popular as a technique for many code intensive applications **[6].**

The basis of the Measure LOC is that program length can be used as a predictor of program characteristics such as effort and ease of maintenance. The LOC measure is used to measure size of the software. Techniques for counting Lines of Code have many variations but the underlying philosophy is the same.

A line of code 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. Another compelling series of arguments against Lines-of-Code revolves around the fact that the

delivered functionality per line of code will vary based on the language being used. It is strongly believed that this argument dooms Lines-of-Code as an effective measure of programmer productivity, and we discourage companies from measuring programmer performance using Lines-of-Code developed as a metric. On the other hand, the use of Lines-of-Code as a predictor of total project effort continues to be successful for a wide range of programming projects using a variety of languages [28].

Line Counting Rules

- Comments do not count.
- Blank lines do not count.
- Code that is included more than once (e.g., include files) counts only the first time it is included.
- Standard operating system include files do not count.
- Code lines count.
- Job control lines count.
- User defined includes files count (once).
- SQL statements count

Advantages of LOC

- It is straight forward (simple)
- Easily can be automated (plenty of tools are available)

Disadvantages of LOC

- Its Language dependent
- Penalizes the well-designed short programs
- Cannot easily accommodate nonprocedural languages
- Need a level of detail that may not be available at the early stages of development.

2.5.2. Function Point

Function Point Analysis was developed first by Allan J. Albrecht in the mid-1970s. It was an attempt to overcome 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 in 1979, then later in 1983. Function point (FP) is a measure which represents the functional size of application software and also we define the function point as a standardized metric that describes a unit of work product suitable for quantifying software that is based on what the end user requests and receives. Function Point Analysis (FPA) is a standard method for measuring software development and maintenance from the customer's point of view. Function Point Count (FPC) is the function point measurement of a particular application or project [30]. Function Point is a conceptual measure of software size based on functional requirements:

- Basis is the measured proportions of effort required to produce types of functionality.
- Derived through multiple regression analyses applied to actual project results.

Function Point Analysis

One of the initial design criteria for function points was to provide a mechanism that both software developers and users could utilize to define functional requirements [31]. It was determined that the best way to gain an understanding of the users' needs was to approach their problem from the perspective of how they view the results an automated system produces. Therefore, one of the primary goals of Function Point Analysis is to evaluate a system's capabilities from a user's point of view. To achieve this goal, the analysis is based upon the various ways users interact with computerized systems. From a user's perspective 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 addresses the user's need to access data and are referred to as Transactional Functions.

The Five Major Components

External Inputs (EI): - is an elementary process in which data crosses the boundary from outside to inside. This data may come from a data input screen or another application. The data may be used to maintain one or more internal logical files. The data can be either control information or business information. If the data is control information it does not have to update an internal logical file.

External Outputs (EO): - an elementary process in which derived data passes across the boundary from inside to outside. Additionally, an EO may update an ILF. The data creates

reports or output files sent to other applications. These reports and files are created from one or more internal logical files and external interface file.

External Inquiry (EQ): - an elementary process with both input and output components that result in data retrieval from one or more internal logical files and external interface files. The input process does not update any Internal Logical Files, and the output side does not contain derived data.

Internal Logical Files (ILF's): - a user identifiable group of logically related data that resides entirely within the applications boundary and is maintained through external inputs.

External Interface Files (EIF's): - a user identifiable group of logically related data that is used for reference purposes only. The data resides entirely outside the application and is maintained by another application. The external interface file is an internal logical file for another application.

After the components have been classified as one of the five major components (EI's, EO's, EQ's, ILF's or EIF's), a ranking of low, average or high is assigned. For transactions (EI's, EO's, EQ's) the ranking is based upon the number of files updated or referenced and the number of data element types. For both ILF's and EIF's files the ranking is based upon record element types and data element types.

Record Element Types: A record element type is a user recognizable subgroup of data elements within an ILF or EIF.

Data Element Types*:* A data element type is a unique user recognizable, non-recursive, field.

The counts for each level of complexity for each type of component can be entered into a table such as the following one. Each count is multiplied by the numerical rating shown to determine the rated value. The rated values on each row are summed across the table, giving a total value for each type of component. These totals are then summed across the table, giving a total value for each type of component. These totals are then summoned down to arrive at the Total Number of Unadjusted Function Points.

Table 1 Complexity multiplier

Types of components	**Complexity of components**			
	Low	**Average**	**High**	**Total**
External Input	___x3=___	___x4=___	___x6=___	
External Output	___x4=___	___x5=___	___x7=___	
External Inquiries	___x3=___	___x4=___	___x6=___	
Internal Logical Files	___x7=___	___x10=___	___x15=___	
External Interface files	___x5=___	___x7=___	___x10=___	
		Total Number of Unadjusted Function Points		_____
		Multiplied Value Adjustment Factor		_____
		Total Adjusted Function points		

Source: IFPUG Guideline.

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. The factors are:

Table 2 Value adjustment factor

General System Characteristic	**Brief Description**
Data Communications	The data and control information used in the application are sent or received over communication facilities.
Distributed Data Processing	Distributed data or processing functions are a characteristic of the application within the application boundary.
Performance	Application performance objectives, stated or approved by the user, in either response or throughput, influence (or will influence) the design, development, installation and support of the application.

Heavily Used Configuration	A heavily used operational configuration, requiring special design considerations, is a characteristic of the application.
Transaction Rate	The transaction rate is high and influences the design, development, installation and support.
On-line Data Entry	On-line data entry and control information functions are provided in 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 update for the internal logical files.
Complex Processing	Complex processing is a characteristic of the application.
Reusability	The application and the code in the application have been specifically designed, developed and supported to be usable in other applications.
Installation Ease	Conversion and installation ease are characteristics of the application. A conversion and installation plan and/or conversion tools were provided and tested during the system test phase.
Operational Ease	Operational ease is a characteristic of the application. Effective start-up, backup and recovery procedures were provided and tested during the system test phase.
Multiple Sites	The application has been specifically designed, developed and supported to be installed at multiple sites for multiple organizations.
Facilitate Change	The application has been specifically designed, developed and supported to facilitate change.

Source: IFPUG Guideline

The 14 General System Characteristics must be rated on the scale of 0 (not important) to 5 (very important) and summed, then multiplied by 0.01, and the constant of 0.65 added to create the influence multiplier weight.

The above fourteen value adjustment factors are often reduced to ten value adjustment factors and used accordingly. The reduced value adjustment factors are presented as follows.

Table 3 Reduced Value adjustment factors

	Factors	Notation
1	Knowledge of application domain	KAD
2	Familiarity with programming language	FPL
3	Experience with system software (OS, DBMS)	ESS
4	Structuredness of software modules	SSS
5	Independence between software modules	ISM
6	Changeability/readability of program language	CRP
7	Reusability of legacy software modules	RLS
8	Up-to-dateness of documentation	DOC
9	Comformity with software engineering standard	SES
10	Testability	TST

Based on the relation of these reduce VAF it has been categorized in to three broad groups as follows:

Table 4 Category of Reduced VAF

Characteristics group	Factors	Notation
Engineer's Skill	Knowledge of application domain	KAD
	Familiarity with programming language	FPL
	Experience with system software (OS, DBMS)	ESS
Technical Characteristics	Structuredness of software modules	SSS
	Independence between software modules	ISM
	Changeability/readability of program language	CRP
	Reusability of legacy software modules	RLS
Maintenance Environment	Up-to-datedness of documentation	DOC
	(9). Conformity with software engineering standard	SES
	Testability	TST

Advantages of Function Points

- Independent of programming language
- The necessary data is available early in a project
- Deals quantitatively with complexity
- Uses readily countable characteristics of the "information domain: of the problem
- Does not "penalize" inventive implementations that require fewer LOC than others
- Makes it easier to accommodate reuse and the trend toward object-Oriented approaches

Disadvantages of Function Points

- Hard to automate
- Built based on traditional data processing applications
- Ignores quality of output

2.6. Related Works

Mining Defect Reports Based Software Maintenance cost estimation model.

Rajni. J et al [10] shown that the software maintainability effort can be predicted in terms of 'total changes' which refer to the total number of Lines of Code (LOC) that are inserted, deleted or modified in moving from one version of the software to the next. Also text mining steps have been applied in the following order: pre-processing, feature selection, Tf-Idf weighting and finally normalization. All these words which are obtained after pre-processing are referred to as 'features'. Then InfoGain measure is used to extract some useful features from the feature set. Each term in the vector is weighted using a tf-idf approach which stands for Term Frequency Inverse Document Frequency.

Feature selection applied by using InfoGain measure, as it is one of the most widely used feature selection method which identifies those words from the document which aim to simplify the target concept.

[10] Have used Multi-nominal Multivariate Logistic Regression (MMLR) method which is a variant of logistic regression method. Logistic regression has been one of the most popular statistical methods being used in the literature so far. In MMLR method, all the independent variables together are used to predict different categories of the dependent variable. In other words, here the dependent variable is not binary and can have more than two values. For instance, the dependent variable in our study has four levels of category viz. very low, low, medium and high coded as 1, 2, 3 and 4 respectively. Now, in order to predict the model using MMLR, all the independent variables are used together on each of these abovementioned categories.

[10] Used some performance measure in order to check the performance of the mode is listed below:

1. Sensitivity which is defined as the ratio of correctly classified defect prone documents to the actual number of defect prone documents. It is also known as Recall.
2. *Receiver Operating Characteristics (ROC) analysis:* is one of the most popular techniques which is used to evaluate the performance of the predicted model. It is a plot of sensitivity values (also known as true positive) on the y- axis and 1-specificity values (also known as false positive) on the x- axis at different possible cut-off points.
3. *Validation method used:* Hold-out validation method (70-30 ratio) is used in order to divides the entire dataset into 70% training data and the rest 30% as testing data.

As we have seen the developed model by those authors the model needs the previous defect report as an input at the first stage and the it tries to filter the useful data from the defect report this causes the question what if the defect report is not available and the system or software is new and not maintained ever before or this maintenance request is the first maintenance request for the system. So the model fails under such environment and not applicable.

Function Point Based Software Maintenance cost estimation model.

Y. Ahn et al [11] Assumes that, in order to estimate the effort to maintain the requested maintenance we have to know first the two mostly related constraints to the effort which is the size of software to be maintained and the productivity rate of the maintenance team. The function point approach to measure the functionality of the software to be maintained and the productivity

is widely explained very well. Software cost estimation models often have an exponential factor to account for the relative economies or diseconomies of scale encountered as a software project increases its size.

The relation of the software maintenance effort and the function point is expressed as the exponential of some constant coefficients. So if the functionality of the software to be maintained is found we ready and have enough resources to calculate the software maintenance effort.

SRS Based Software Maintenance cost estimation model.

Aprna.T et al [12] it is known and yet studied that the code complexity of the software to be maintained and the amount of maintenance effort has a direct relationship. For the simplicity of understanding and accuracy of the software maintenance effort this study used an improved requirement based complexity (IRBC) which is developed by **Sharma et al** [13].

According to their assumption, it is better to invest more effort early in the SRS stage since the effort required repairing a defect in the analysis phase is much less than that of in design or implementation phase, it is advisable to invest more effort in early phase of SDLC to reduce maintenance cost.

The model called IRBC developed previously is composed of all attributes that are sufficient to compute function point for any SW. The function point measures include five parameter i.e. external input, external output, interfaces, file and inquiry we focused on function point under methodology topic.

Since every attribute of IRBC is functionally dependent on VAFs, so they take the IRBC and Value Adjustment Factor (VAF) as a bench mark to compute the complexity which based SRS called Requirement Based Maintenance Complexity (RBMC), Then RBMC is a product of IRBC and VAF.

The other attributes they needed to develop there is the size of the software to be maintained and the productivity of maintenance team in terms of Kilo Line of Code (KLOC). Since the RBMC is equivalent to the functional size of the system they used RBMC in calculating the size of the

system. And the productivity is directly depends up on size of the system, so they've used the system size in the calculation of productivity.

Finally they've developed the model called The Requirement Based Maintenance Effort Estimation (RBMEE) in Person-Month (PM) for proposed SW is expressed as the ratio of software size and productivity.

This model is only applicable and accurately works on the SRS developed based on IEEE Std 830-1998, So the model is lead us to failure when we were used other than this type of SRS. Again the model fail under the situation where the SRS of the software to be maintained is not found or available, this means the model is not work in the reverse engineering situation.

Program Slicing based Software Maintenance Effort Estimation

Hakam W et al. [14] used the program slicing method as a basis for estimating software maintenance effort estimation. In their study they have presented a case study of the GNU Linux kernel with over 900 versions spanning 17 years of history. For each version there are developed system dictionary by using a lightweight slicing approach and encodes the forward decomposition static slice profile to all variables within the system. Changes to the system are then modeled at the behavioral level using the difference between the system dictionaries of two versions. Also the three granularity of the slice is considered when aimed to slice the program which is line, function, and file is analyzed.

There are some beliefs which say that building an accurate maintenance effort estimation model should be derived from accurate maintenance effort data. But here it is difficult to get the maintenance data for the system is an open source system. So it better to deal with the indirect software maintenance effort measures like lag-time and source-code change to the system is a valid indirect effort estimation measures.

Lag-time: - Each version of the system has its own release date. The lag-time (*measured in days*) includes the duration from the date when a base version is released, until the date the evolved version is released. The maintenance request assumed to be started at release of base version and the task is accomplished when the evolved version of the system is released. That is, the lag-time is the sum of the individual times for each maintenance task in a version of a system. However,

using lag-time as an indirect maintenance effort measure indicator has its own problem of over reporting maintenance effort.

Source-code change: - Despite the maintenance effort for open-source systems is not given as the number of person-hours expended as the case in closed-source systems, it has been argued [14] that source code change in open-source system can be used as an indirect measure for estimating maintenance effort. When source-code changes are submitted using the Software Configuration Management (SCM) tools (e.g., Subversion, CVS, and Clear Case) best practice is for developers to commit a brief explanation of the change into the change log, which is saved collectively with the source-code deltas in the SCM repository. However, the quality of change logs varies, it depend on the commitment of the developer who submit the change logs. But it is known that this tracking data is not always available.

Many existing software metrics are computed only using syntactic information of the code and use that to model semantic information. For example, cyclomatic complexity is computed by counting the number of branch (i.e., conditionals) to infer semantic complexity. Semantic information is much more difficult to derive and model. For example, a semantic change in one function might create a ripple effect among other functions. In a maintenance context, the effort estimation is a function of the code that is to be changed. To help identify such problems, program slicers are often applied and are a valuable tool in determining side effects. It is possible to determine the parts with different behaviors by comparing the slices of the base and the evolved versions with respect to corresponding points.

The slice profile and system dictionary is constructed for tracking information of the slices. The approach computes a slice profile that contains all the relevant statements, from all possible slices, over a given slicing variable. We define our slicing criterion to consist of a file name, a function name, and a variable name. This slicing criterion is the triple (*f, m, v*) where *f* is a file in the system, *m* is a function/method in the file *f*, and *v* is a variable in the given function *m*. This definition of a slicing criterion does not require a precise reference to a statement number. Moreover, the slicing criterion (*f*) can be used to find all the slices of all variables in all functions in a given file. A system dictionary is built, referred to as (*F, M, V*), and includes all files in the system, all functions in each file, all variables in each function, and all global variables in the system.

The other thing is the process of encoding the slice information, for the encoding slice information an algorithm called System Slice Encoding (SSE) is developed. The basic process of the SSE algorithm starts with a single pass through the system dictionary, encoding each slice profile to a string value, which is then fed to a hash algorithm to produce the final results, the hashed slice encoding. In this algorithm we have to follow two steps in order to accomplish the encoding tasks. Step 1: Encode the slice profile- For a given function; the SSE algorithm encodes the slice profiles into a string value (denoted by *mdsES*). This string consists of two parts, the *function_Name* and the *slines* defined by the *mds* equation. The file encoding string (denoted by *fdsES*) is equal to *fileName;* {*fds*(*f*)}. Finally, the system encoding string (denoted by *gdsES*) is equal to *systemName*; {*gds*(*F, M, V*)}. Step 2: Hash the string value- This step maps the encoding string from Step 1 to a hash value using the MD5 hash algorithm.

The other concept needed is system behavioral change information, to compute behavioral change information across the entire version history of a system, we check out every pair of consecutive versions of the system from its subversion repository, use *src2srcml* to convert the source code into srcML format, use *srcSlice* to build the system dictionary with slice profiles for all the slicing variables in each version, and apply the SSE algorithm on them.

Generally, by considering every each detail of the system and slice them for better understanding and easily manageable way the maintenance effort is predicted based on the previous data or change log of the system compared to the change made to the system with in its successive version of the system.

Beside some understandability and reasonability of the model, it has some major exception that may cause the model fail under some circumstances where the system is completely new and has no previous version the model doesn't work perfectly for it needs previous data as an input and also the situation where the system has no log registry features.

Effort Estimation for Corrective Software Maintenance

Andrea .D et al [15] focuses on ordinary multivariate least squares regression as the modeling technique for effort estimation since this is one of the most common modeling techniques used in practice. The ordinary least squares modeling technique minimizes the sum of squared error, whereas they are evaluating the resulting model in terms of the relative error. With the aim to

obtain a simple and comprehensible model, they evaluated multivariate linear regression models including the number of tasks (total or of different types) and the size of the system.

Based on the requirement of the software maintenance they have categorized the maintenance type, as Type A, B, and C.

- Type A: the maintenance task requires software source code modification;
- Type B: the maintenance task requires fixing of data misalignments through database queries;
- Type C: the maintenance task requires intervention not comprised in the previous categories, such as user disoperation, problems out of contract, and so on.

The correlation matrix was used which shows the absence of strong correlations between the independent variables. The highest correlation value (0, 6458) is between the number of tasks of type A and C, while all remaining correlation value between the number of tasks of different type are not meaningful. There is a high correlation between the total number of tasks N and its components NB and NC, while there is no correlation between N and NA. A potential reason for this is that maintenance tasks of types B and C are more recurrent than maintenance tasks of type A.

The model developed is significant or necessary especially for the corrective software maintenance effort prediction. They collect the historical data, analyze the collected data in order to fit the data into the corrective maintenance and finally use the analyzed data for predictive purpose.

Table 16: Summary of Related Work

S.N	Name	Model	Approaches	Limitation
1	Defect Report Mining Based Software Maintenance Effort Estimation	$\text{Prob}(Y1, Y2, \ldots Y3 = \frac{e^{B_0+B_1Y_1+\cdots+B_nY_n}}{1+e^{B_0+B_1Y_1+\cdots+B_nY_n}}$	Text Mining Approach	the model needs the previous defect report as an input at the first stage and the it tries to filter the useful data from the defect report this causes the question what if the defect report is not available and the system or software is new and not maintained ever before or this maintenance request is the first maintenance request for the system. So the model fails under such environment and not applicable.
2	Function Point Based Software Maintenance Effort Estimation	Effort = a*FP^b	Function Point Analysis Approach	The model is only accurate for the small size projects and the accuracy of the software is decreased as the size of the software to be maintained is increased.
3	SR Based Software Maintenance Effort Estimation	$RBME = \frac{\text{KNCSLOC}}{\text{Productivity}}$	IEEE std 830-1998 Based SRS Approach	This model is only applicable and accurately works on the SRS developed based on IEEE Std 830-1998, So the model is lead us to failure when we were used other than this type of SRS. Again the model fail under the situation where the

				SRS of the software to be maintained is not found or available, this means the model is not work in the reverse engineering situation.
4	Program Slicing Based Software Maintenance Effort Estimation	E_{code} = c1+c2(locSize)+c3(fileSize) E_{slice}= c1+c2(sliceSize)+c3(hashSize) + c4(sCoverage)	Program Slicing Techoiques	Beside some understandability and reasonability of the model, it has some major exception that may cause the model fail under some circumstances where the system is completely new and has no previous version; the model doesn't work perfectly for it needs previous data as an input and also the situation where the system has no log registry features.
5	Effort Estimation for Corrective Maintenance	A: Effrot= a1N+a2Size B: Effort= a1NA+a2NBC+a3Size C: Effort= a1NA+a2NB+a3NC+a4Size	MDTs (Missing Data Techniques) Algorithm	The model has considered only the corrective maintenance and excluded other types of maintenance. Also the model doesn't consider about the personnel skill for it has its own impact on the maintenance effort.
6	Software Maintenance Cost of Influence factor analysis and Estimation Model	$Effort = ACT * SDT * \prod_{i=1}^{n} F_i$	Expert Judge Approach	The accuracy of the model is depend on the individual decision of the expert rather than actual data of the project.

CHAPTER THREE

METHODOLOGY

In order to answer the research question we have focused on the following methods or techniques.

3.1. Data Collection

Building an accurate maintenance-effort estimation model should be derived from accurate maintenance-effort data, which is recorded for different software systems. So obvious data is very useful, without data it is unthinkable to develop a methods to estimate the effort since the things to be estimated is a data itself.

From the beginning we may have assumed there are several data sources to accomplish our tasks. Firstly, in our country, there are many software company among them we have chosen the leading and the largest one INSA, and secondly, as an alternative there are also so many online data which is reserved for an experiment only by different international institution such as NASA, Apache, etc.

Since we haven't got any data regarding the software maintenance from the institution which are from our country, we have decided to use the data collected for the research purposes from different software company and different country. The data we have used for an experiment in order to find the best model that may fit the sample data is collected from 4 different software companies in Korea. The data is collected in the form of questionnaire which is categorized into two parts. The first part of the questionnaire is asked the respondents about the weighting of the VAF values. And the second part is about actual effort used in the process of maintaining the previous maintenance project. From those distributed questionnaire they have 57 respondents from part 1 of the questionnaire and 37 respondents from the part two of the questionnaire. So the regression analysis is held based on those results.

3.3. Tools and Techniques

Tools

In the development of the model different tools are used according to their necessity.

Name	Description
SPSS	For handling analysis data part
Microsoft Visual Studio	For developing prototype for the developed model
MS Word	For writing thesis report
MS Excel	For sketching graph and other simple analysis

Techniques/ Approaches

The study has selected the following two approaches, with or without modification as needed, in order to meet the intention of the thesis objectives. Those selected approaches are used to measure the size of the software to be maintained based on different point of view which is from developer point of view, we have Source Line of Code (SLOC) and the other is from client or user point of view, which is Function Point (FP). It has decided to use these two different approaches in order to make both party to have similar understanding on what is going to be done especially for user or client.

The maintenance effort estimation model can be classified in to three broad categories like Model based maintenance effort estimation, function point based maintenance effort estimation and code based maintenance effort estimation.

According to [5] they focus on the function point and Line of code based approach in order to estimate effort for software project development management. So the thesis follow the same approach but try to reformulate to adjust to make a sense with the concept of software maintenance effort estimation. According to [5], the two approaches are used in order to meet the requirement of both parties that means it help them to have relative understanding on what they are dealing on.

The study has selected the following two approaches, with or without modification as needed, in order to meet the intention of the thesis objectives. Those selected approaches are used to measure the size of the software to be maintained based on different point of view which is from developer point of view, we have Source Line of Code (SLOC) and the other is from client or user point of view, which is Function Point (FP). It has decided to use these two different approaches in order to make both party to have similar understanding on what is going to be

done especially for user or client. Regression analysis techniques is used to find the coefficient used in the model.

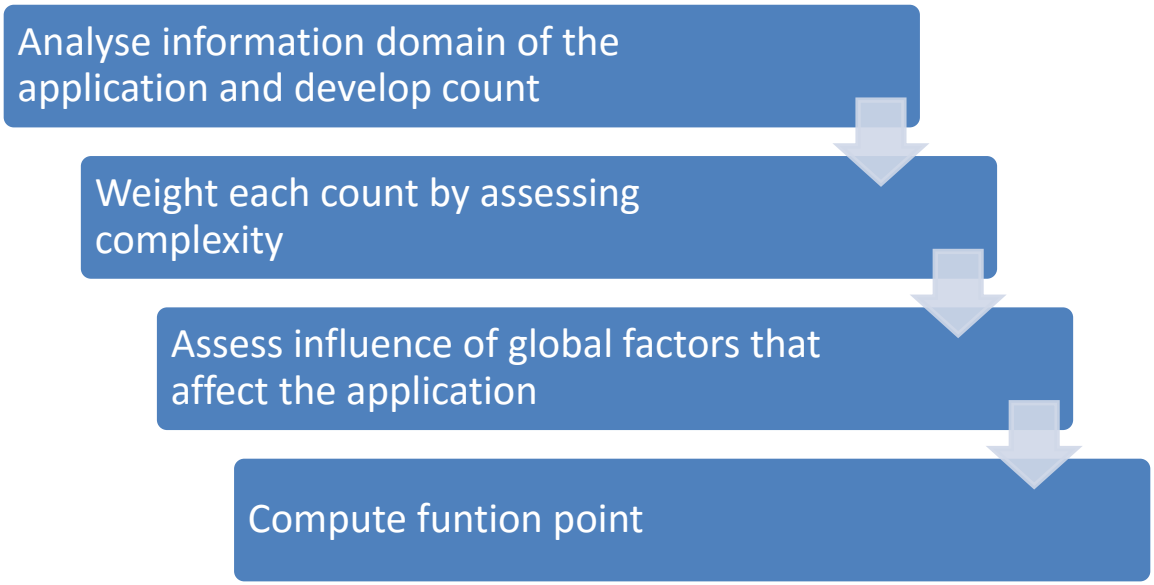


Figure 1 Summary of Function Point calculations

Hybrid Approach

Knowing the size of the software to be maintained is the most important steps before estimating as we cannot estimate what we can't measure [29]. So it is better to focus on the approach we used when we measure the size of the software. The more we are confident in the size of the software to be maintained, the more we are confident in estimating the required effort to maintain the system.

Here both approaches we have discussed above, which means the line of code approach and the function point approach has their own limitation in making the reasonable decision on the estimation of software maintenance effort as we have seen above. The Line of code is understandable at the developer side but more difficult to communicate with the client who has no understanding of the complexity behind the software because he judge the software based on the functionality of that software itself.

So to come up with the level of understandability of both party which means the client and the developer in order to make them to talk with relative understandability of the software to be

maintained in addition to the model accuracy, we have chosen the hybrid approaches to develop the Reasonable Software Maintenance Effort Estimation Techniques (RSMEET).

3.4. Testing and Evaluation of the Model

To validate the proposed Reasonable Software Maintenance Effort Estimation Technique model (RSMEET), the actual data were used which is collected by different scholars for we haven't found any data in regards of our country's software company we have decided to do with an online found data which is collected from a survey of the system management (SM) organizations of four system integration (SI) companies in Korea. If so we have decided only to use the summary of the data not the detail one in this work.

According to their survey result they have distributed the totally 64 questionnaire paper (which is categorized in two parts) to the different companies which is guided as a manager centered distribution system. From the distributed questionnaire they have collected 56 answers for part one of the questionnaire and 37 answers for part two of the questionnaire. To make brief about the general idea about the questionnaire, Part 1 of the questionnaire is asking about the VAFs' weight value covering the software maintenance project of most business applications, and Part 2 of the questionnaire is for the collection of actual data about size in both LOC and FP and maintenance efforts of past projects.

CHAPTER FOUR

MODEL DEVELOPMENT

4.1. Introduction

It is known that as the size of the software to be maintained is increase the effort requires to maintain the software also increases. So this is why the proposed model decided to focus on the size of the software to be maintained. In this chapter we have covered a lot of sections with their corresponding descriptions. Section 4.2 of the chapter deals with the related architecture of other model, the next section 4.3 of the chapter deals with the architecture of the proposed model, under this section as a subsection the subsection like the sizing concept of the software to be maintained is discussed very well by using different approaches like Line of Code, Function Point and the Hybrid approaches which is composed of LOC and FP.

4.2. Related Model

In Software development model, the COCOMO II reuse model was derived on the basis of experience and findings drawn from previous empirical studies on software reuse costs. [17] Performed an analysis of reuse costs of reused modules in the NASA Software Engineering Laboratory, indicating nonlinear effects of the reuse cost function. [18] Describe a formula to represent the number of interface checks required in terms of the number of modules modified and the total number of software modules, showing that the relationship between the number of interface checks required and the number of modules modified is nonlinear. The cost of understanding and testing the existing code could, in part, cause the nonlinear effects. [19] Found that the effort required to understand the software be modified takes 47 percent of the total maintenance effort.

The COCOMO II reuse model computes the equivalent SLOC for enhancing, adapting, and reusing the pre-existing software. This model takes into account the amount of software to be adapted, percentage of design modified (*DM*), the percentage of code modified (*CM*), the percentage of integration and testing required (*IM*), the level of software understanding (*SU*), and the programmer's relative unfamiliarity with the software (*UNFM*).

On the other hand, the framework of SMPEEM developed by [20] is based on the concepts shown in Figure 2. First, to estimate the size of the maintenance tasks, the concept of a FP model

is applied. Second, to adjust the counted FP considering the maintenance environment, they have introduced the new concept of the VAF. The VAFs of this model are related to the attributes of each characteristic group such as the engineer's skill, technical characteristics and the maintenance environment. Finally, they suggested that an exponential function model that can estimate how much effort is required for the software maintenance project. The procedure followed by the author is described in a brief way diagrammatically as follows:

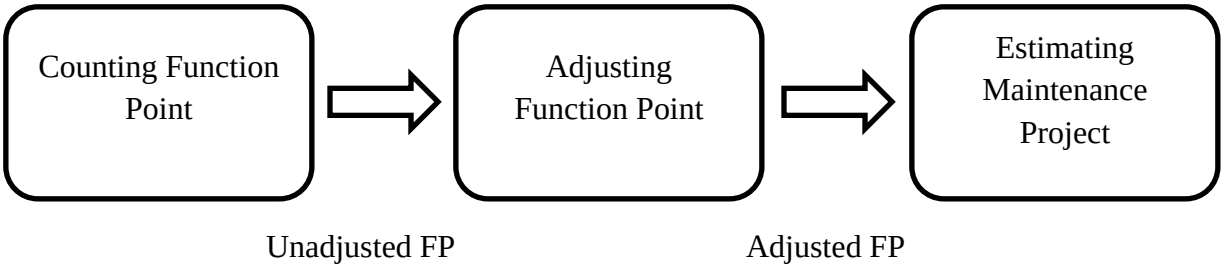


Figure 2 Concept of SMPEEM

4.3. Proposed Model

Software maintenance sizing methods

Software size has a direct effect on the estimation of effort required to maintain the requested software maintenance. So the size issue must be focused when we are about to estimate the software maintenance effort. The accuracy of the estimation is highly dependent on the accuracy of our size measurement. So in order to make our model more accurate than previously developed model we focused more on the size measurement of the software to be maintained.

The scope of software maintenance includes adding new capabilities and fixing or adapting existing capabilities. But it excludes major product rebuilds, changing over 50% of the existing software and development of sizable interfacing system requiring little rework of the existing system which is calculated and checked as:

$$\text{Maintenance Sizing} = \left(\frac{\text{Size Added+Size Modified}}{Total\ base\ Size}\right) * 100 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Based on the Equation 1 above, the size of feature added to an existing system and the size of feature modified feature is summed up and divided by the total size of the system, if the result from the equation is greater than 50, we recommend that it is better to develop a new system rather than maintaining existing system or software.

Unlike the new development, where the code for core capabilities does not exist, the software maintenance relies on the preexisting code. As a result, the maintenance sizing method must take into account different types of code and other factors, such as the effects of understanding the preexisting code and the quality and quantity of the preexisting code.

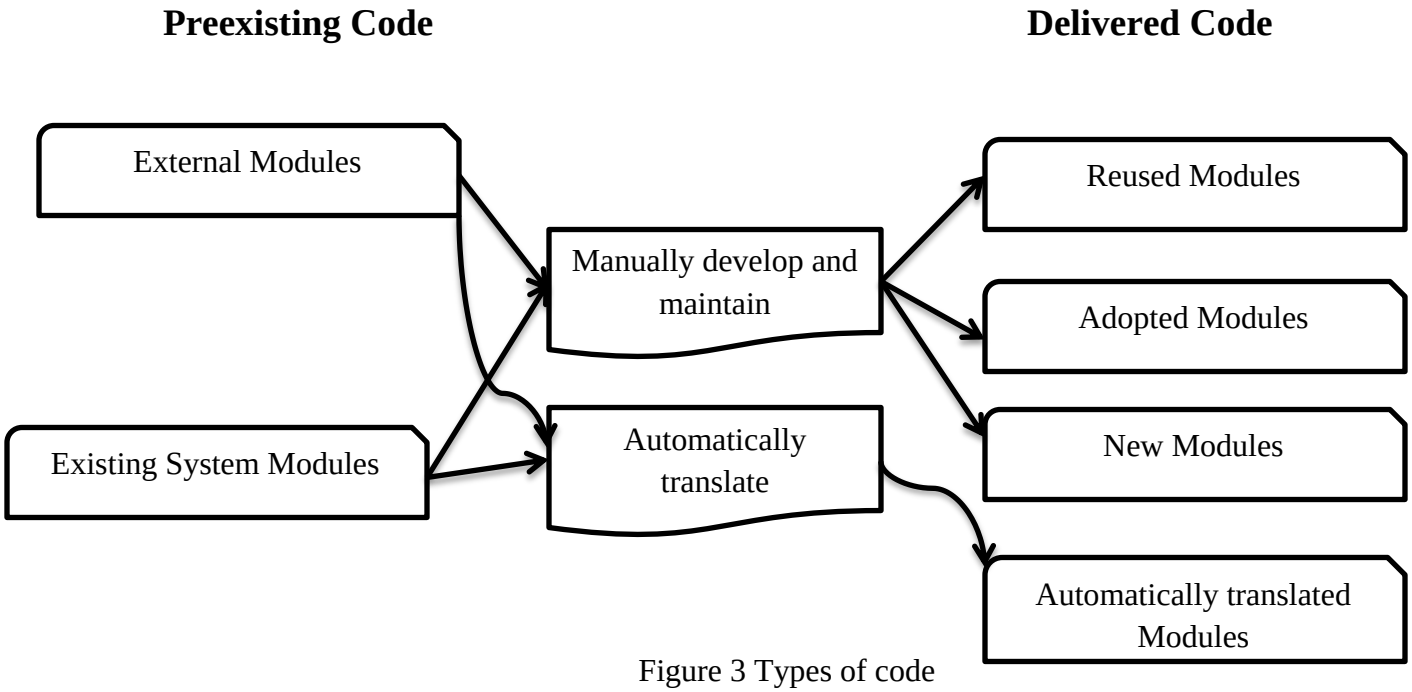


Figure 3 Types of code

The above figure shows that since we are talking about the software maintenance then we have some preexisting code which we may use as input for the newly delivered software which means the software released after the activities of maintenance. Among the preexisting code we have external modules which are taken from external sources, it may be from out of the system to be maintained and the other one is existing system modules which are taken from the previous system as it is or with modification as needed. They can be grouped into the following types:

- **External modules**: the code taken from a source other than the system to be maintained. They can be proprietary or open-source code.
- **Preexisting system modules**: the code of the system to be upgraded or maintained.
- **Reused modules**: the preexisting code that is used as a black-box without modifications.
- **Adapted modules**: the code that is changed from using the preexisting code. The preexisting code is used as a white-box in which source lines of code are added, deleted, modified, and unmodified.

- **New modules**: the modules newly added to the updated system.
- **Automatically translated modules**: the code obtained from using code translation tools to translate the preexisting code for use in the updated system.

In COCOMO, the automatically translated code is not included in the size of the maintenance and reuse work. Instead, the effort associated with the automatically translated code is estimated in a separate model different from the main COCOMO effort model. In the COCOMO estimation model for software maintenance, we also exclude the automatically translated code from the sizing method and effort estimation.

COCOMO II provides two separate models for sizing software reuse and software maintenance. The reuse model is used to compute the equivalent size of the code that is reused and adapted from other sources. The reuse model can also be used for sizing major software enhancements. On the other hand, the maintenance model is designed to measure the size of minor enhancements and fault corrections. In our model we excluded the reuse module from the sizing of the software to be maintained.

The main focus is on the sizing of the software going to be maintained, since it has a visible contribution to the accuracy of the estimation model.

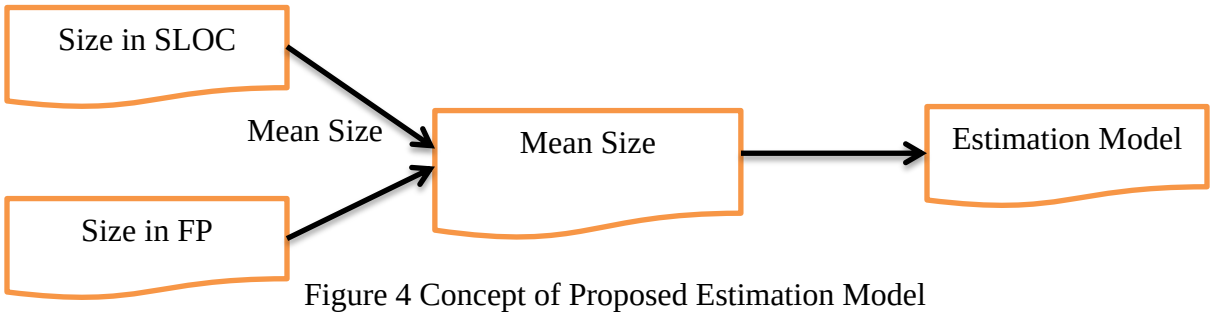


Figure 4 Concept of Proposed Estimation Model

We have used the hybrid approaches to measure the size of the software to be maintained which is the composition of Line of Code approach and Function Point approaches. As we have discussed in the previous section the accuracy of the estimation is only as good as the accuracy of the size measurement of software to be maintained.

4.3.1. Software Maintenance Sizing in LOC

[21] COCOMO II differs from COCOMO 81 in applying the COCOMO II scale factors to the size of the modified code rather than applying the COCOMO 81 modes to the size of the product

being modified. Applying the scale factors to a 5 million SLOC product produced overlarge estimates as most of the product was not being touched by the changes. COCOMO II accounts for the effects of the product being modified via its software understanding and unfamiliarity factors.

The maintenance size is normally obtained via Equation 2, when the base code size is known and the percentage of change to the base code is known.

$$\text{SLOC} = [(\text{Base Code Size}) * \text{MCF}] * \text{MAF} \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

The Maintenance Adjustment Factor (MAF) is discussed below. But first, the percentage of change to the base code is called the Maintenance Change Factor (MCF). The MCF is similar to the Annual Change Traffic in COCOMO 81, except that maintenance periods other than a year can be used. Conceptually the MCF represents the ratio in Equation 3:

$$\text{MCF} = \frac{\text{Size Added+Size Modified}}{Total\ base\ Size} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

There are several sources for estimating new lines of code. The best source is historical data. For instance, there may be data that will convert Function Points, components, or anything available early in the project to estimate lines of code. Lacking historical data, expert opinion can be used to derive estimates of likely, lowest-likely, and highest-likely size.

A simpler version can be used when the fraction of code added or modified to the existing base code during the maintenance period is known. Deleted code is not counted.

$$\text{SLOC} = (\text{Size Added} + \text{Size Modified}) * \text{MAF} \ldots\ldots\ldots\ldots\ldots\ldots (4)$$

The Maintenance Adjustment Factor (MAF), Equation 5, is used to adjust the effective maintenance size to account for software understanding and unfamiliarity effects, as with reuse.

Software Understandability

The Software Understanding increment (SU) is obtained from below Table 5. SU is expressed quantitatively as a percentage. If the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface-checking penalty is 10%. If

the software is rated very low on these factors, the penalty is 50%. SU is determined by taking the subjective average of the three categories.

Table 5 Rating Scale for Software Understanding Increment SU

	Very Low	**Low**	**Nominal**	**High**	**Very High**
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably Well structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data /control structures.
Application Clarity	No match between program and application world-views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptiveness	Obscure code; documentation missing obscure or obsolete.	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentation.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well organized, with design rationale.
SU Increment to SLOC	50	40	30	20	10

Source: COCOMO II Model Definition Manual

Programmer Unfamiliarity

The amount of effort required to modify existing software is a function not only understandability of the existing software (SU), but also of the programmer's relative unfamiliarity with the software (UNFM). The UNFM factor is applied multiplicatively to the software understanding effort increment. If the programmer works with the software every day, the 0.0 multiplier for UNFM will add no software understanding effort increment. If the programmer has never seen the software before, the 1.0 multiplier will add the full software understanding effort increment. The rating of UNFM is shown in below Table 6.

Table 6 Rating Scale for Programmer Unfamiliarity (UNFM)

UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

Source: COCOMO II Model Definition Manual

COCOMO II uses the Software Understanding (SU) and Programmer Unfamiliarity (UNFM) factors from its reuse model to model the effects of well or poorly structured/understandable software on maintenance effort.

$$MAF = 1 + (\frac{\text{SU}}{100} * UNFM) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

Relating SLOC to UFPs

Next, convert the Lines of Code to Unadjusted Function Points (UFP). The source lines of code have to be converted to unadjusted function points in the implementation language so we can get

the result of the same unit. The conversion is achieved by using the conversion ratio of LOC and FP given in the following Table 7.

Table 7 UFP to SLOC Conversion Ratios

Language Default	**SLOC / UFP**	**Language Default**	**SLOC/ UFP**
Access	38	Jovial	107
Ada 83	71	Lisp	64
Ada 95	49	Machine Code	640
AI Shell	49	Modula 2	80
APL	32	Pascal	91
Assembly - Basic	320	PERL	27
Assembly - Macro	213	PowerBuilder	16
Basic - ANSI	64	Prolog	64
Basic - Compiled	91	Query – Default	13
Basic - Visual	32	Report Generator	80
C	128	Second Generation Language	107
C++	55	Simulation – Default	46
Cobol (ANSI 85)	91	Spreadsheet	6
Database – Default	40	Third Generation Language	80
Fifth Generation Language	4	Unix Shell Scripts	107
First Generation Language	320	USR_1	1
Forth	64	USR_2	1
Fortran 77	107	USR_3	1
Fortran 95	71	USR_4	1
Fourth Generation Language	20	USR_5	1
High Level Language	64	Visual Basic 5.0	29
HTML 3.0	15	Visual C++	34
Java	53		

Source: IFPUG manual.

After SLOC is converted to the Unadjusted Function Point based the above conversion table, the Unadjusted FP is adjusted by Equation 8 discussed in the next section.

4.3.2. Software Maintenance Sizing in FP

Again in this section we've deal with another approach to estimate the size of the maintenance tasks, here to do that we have applied the concept of a FP model. And, to adjust the counted FP considering the maintenance environment, we need to use the new concept of the VAF introduced by [20]. The VAFs of this model are related to the attributes of each characteristic group such as the engineer's skill, technical characteristics and the maintenance environment as shown in Table 9. Finally, we got the size of the software to be maintained.

Table 8 Unadjusted FP count calculation table

Function Type	Weight value by functional complexity (W)		
	Low	Average	High
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

(*Source: IFPUG counting Practice Manual 4.1, 1999*)

Counting FP

FPs of each application type should be counted to estimate the size of the maintenance project. There are three types of application maintenance such as adding, changing, and deleting. The rule of counting the function point is used as in the International Function Points User Group (IFPUG) model [22].

An application maintenance type could be one of the five function types: ILF, EIF, EI, EO, or EQ. Each of these is individually assessed for complexity and given a weighting value that varies from three (for simple EIs) to 15 (for complex ILFs) as shown in Table 8.

The unadjusted function count (FC) is computed by multiplying each raw count by the estimated weight (*W*):

$$FC = \sum_{Function\ Type} Number\ of\ function * W \ldots\ldots\ldots\ldots\ldots (6)$$

Where FC is the counted function point and W is the complexity weighting value of each function type.

Table 9 Value Adjustment Factors

Characteristics group	Factors	Notation
Engineer's Skill	(1). Knowledge of application domain	KAD
	(2). Familiarity with programming language	FPL
	(3). Experience with system software (OS, DBMS)	ESS
Technical Characteristics	(4). Structuredness of software modules	SSS
	(5). Independence between software modules	ISM
	(6). Changeability/readability of program language	CRP
	(7). Reusability of legacy software modules	RLS
Maintenance Environment	(8). Up-to-dateness of documentation	DOC
	(9). Comformity with software engineering standard	SES
	(10). Testability	TST

Source: IFPUG Guideline

Adjusting FPs

Value adjustment factors

This FC is then further modified by the 10 VAFs. According to [20] model, the three types of VAFs are:

1) The engineer's skill, considered from the perspective of 'people';
2) The technical characteristics considered from the perspective of 'product';
3) The maintenance environment characteristics considered from the perspective of 'process'; as shown in Table 9.

The first characteristic group is the engineer's skill required for the software maintenance, which includes three factors such as knowledge of the application domain, familiarity with programming language, and experience with the system software (OS, DBMS). These factors

describe the capability level of maintenance staff. If the application domain is well understood, the system requirements can be analyzed easily [23]. Also application experience, language and tool experience, and platform experience factors measure the productivity of COCOMO 2.0 [24].

The second technical characteristic group contains four factors, i.e. the structuredness of the software modules, the independence of the software modules, the changeability/readability of the programming language, and the reusability of the legacy software modules. These factors are representative of program modules status. As the software is maintained, its structure is degraded [23]. Application age and reusability [24] factors are adopted as the technical productivity measures of software.

The last characteristic group is the maintenance environment which includes up-to-dateness of documentation, conformity with software engineering standards, and testability factors. These factors are more related to the maintenance process. Documentation matches with the lifecycle needs [24]; process maturity and testability [24] are referenced as the productivity factors to reduce maintenance costs.

Adjustment methods

The unadjusted FP is multiplied by the VAF to produce the final size in terms of FP. In [20], three characteristic groups of the 10 VAFs have a weighting value for each group and for each of the factors, respectively. The VAF of a software maintenance project could be calculated from the following equation:

$$VAF = \sum_{group=1}^{3} [\sum_{factor=1}^{k} (\frac{Scores * Wf}{100}) * (\frac{Wg}{100})] \dots\dots\dots\dots\dots\dots (7)$$

Where Wg and Wf are the weighting values of each characteristics group and their factors respectively, these weight values can be substituted from the value in the regression result which is discussed in the next chapter. Scores are measured by the software maintenance project manager and vary from 0 (the lowest degree) to 5 (the highest degree).

The VAF ranges from 0 to 50. This VAF can be used to adjust an unadjusted function count which was computed by Equation (6).

Alternatively, FPs can be computed by Equation (8), which is introduced for adjusting function counts as the unit of $\pm\delta\%$:

$$FP = FC * \left\{1 - \left(\frac{\delta}{100}\right) + \left(\frac{2\delta}{5000}\right) * VAF\right\} \ldots\ldots\ldots\ldots\ldots\ldots (8)$$

Where δ is the absolute value of the adjustment range. The adjustment range can be decided by the actual data and repeated experiments of the concerned organization. For example, if an organization intends to adjust the function counts within the range of ±30%, δ of Equation (8) can be substituted with an absolute value of ±30 to produce the following equation:

$$FP = FC * \{0.70 + (0.012 * VAF)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots (9)$$

In the case of VAF = 0, FP is calculated as 70% of the unadjusted function counts. In contrast, the maximum VAF (50) provides 130% of the unadjusted function counts to final FPs. Hence, Equation (8) gives us the adjusted FPs through the various adjustment ranges.

4.3.3. Software Maintenance Sizing in Hybrid Approach

Here we got the size of the software to be maintained but the size is only the part and other features that needs modification with the newly developed or existing features after the maintenance, in terms of source line of code (SLOC) and function points (FP). Now by using both above approaches we merge the size by applying the statistical model called Mean. Accordingly we have developed the mean size of the software to be maintained, this helps us to be more accurate in our size measurement. Based on the concept of statistics, the mean size formula is derived by using Equation 4 after conversion and Equation 9 as follows:

$$Mean\ Size = \frac{FP_{(SLOC)} + FP}{2} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (10)$$

From Equation (10), we obtain the mean size of the software to be maintained which can be used to calculate the estimated software maintenance effort.

5. Effort Estimation Model

According to the Basic COCOMO the effort is measured in terms of size of the software to be maintained since the effort required to maintain the type of the requested maintenance type is

highly depend up on the size of the system to be maintained. As the size of the maintenance system increased the effort also increased in the same rate. Software cost estimation models often have an exponential factor to account for the relative economies or diseconomies of scale encountered as a software project increases its size [24]. This factor is generally represented as the exponent *B* in the equation:

$$Effort = A * Size^B$$

The basic COCOMO uses the following in order to estimate the effort needed to maintain the requested maintenance type [21].

$$Effort = A * SLOC^B$$

And also according to the software maintenance effort estimation developed by [5], the model called SMPEEM; they've described the relationship between effort and Function point in terms of exponential factors as follows.

$$Effort = A * FP^B$$

The relation between the software maintenance effort and Mean Size can be represented in the following exponential equation

$$Effort = a * (Mean\ Size)^b \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots (11)$$

Where the constants ***a***, ***b*** are the coefficients which can be observed from the result of regression. So, if Mean Size and VAF of any maintenance project are known, the maintenance effort can be estimated by Equation (11).

CHAPTER FIVE

EXPERIMENT AND VALIDATION

5.1. Introduction

This chapter is composed of several sections and sub sections. Among them Section 5.2 reveals the information about data collections. It discusses the way data is found in order to make an experiment and validate the developed model. The other section is Data processing, it all about preparing the collected data for real experiment, in the other hand it is the process of making the data more meaningful for the developed model. The final section of the chapter is about the regression analysis on the collected data and validation of the model.

5.2. Experimentation

According to their survey result discussed earlier in the data collection part, they have distributed the totally 64 questionnaire paper (which is categorized in two parts) to the different companies which is guided as a manager centered distribution system. From the distributed questionnaire they have collected 56 answers for part one of the questionnaire and 37 answers for part two of the questionnaire. To make brief about the general idea about the questionnaire, Part 1 of the questionnaire is asking about the VAFs' weight value covering the software maintenance project of most business applications, and Part 2 of the questionnaire is for the collection of actual data about size in both LOC and FP and maintenance efforts of past projects.

5.2.1. Data Processing

Data needs to be processed in order to make a sense with intended result. So we make some processing activity to make a data in usable form in the equation. The SLOC is counted based on the counting rule discussed in chapter three of this document. And also the FP count is based in the IFPUG counting rule discussed in the chapter three.

Table 10: Reduced Value Adjustment

Project	Factors									
	Engineer's Skill			Technical Characteristics				Maintenance Environment		
	KAD	FPL	ESS	SSS	ISM	CRP	RLS	DOC	SES	TST
1	44	15	27	21	32	25	23	45	44	32
2	33	18	25	38	23	19	20	34	16	35
3	19	33	34	29	45	16	26	22	32	36
4	48	27	16	32	33	18	23	36	35	39
5	50	17	35	22	23	19	18	23	26	26
6	36	50	33	34	23	17	19	34	33	33
7	45	23	22	43	30	23	17	44	15	18
8	32	45	36	45	26	16	26	11	27	49
9	33	19	23	38	23	22	23	23	50	21
10	26	26	31	31	18	21	18	35	23	39
11	45	40	32	27	19	19	19	32	33	36
12	38	13	34	45	17	23	19	45	23	15
13	49	29	34	24	26	26	21	44	17	37
14	42	34	11	49	23	23	19	25	23	17
15	49	32	23	33	18	18	23	16	49	30
16	34	44	35	49	19	19	18	45	36	32
17	44	10	32	47	19	30	19	33	18	35
18	45	20	33	48	21	26	23	50	19	32
19	48	19	18	46	39	23	16	46	21	29
20	50	32	23	49	23	18	22	33	39	26
21	50	30	23	47	18	22	21	49	28	33
22	45	27	37	22	19	23	22	49	33	26
23	44	25	50	36	23	25	23	34	23	34
24	49	34	23	23	16	23	25	45	17	23
25	33	16	12	34	22	23	26	38	16	45
26	49	45	33	11	21	17	16	49	32	49
27	49	33	17	23	22	15	33	42	35	33
28	50	22	39	35	23	17	25	49	26	28
29	46	36	41	32	35	23	32	30	33	43
30	49	23	17	34	26	15	23	18	36	17
31	47	31	12	25	16	18	25	25	15	50
32	49	32	13	34	29	23	23	33	17	23
33	50	34	25	16	33	27	18	34	50	43
34	49	34	15	24	23	13	17	39	23	33
35	49	11	18	33	25	26	25	45	25	45

36	44	23	33	31	29	20	19	38	49	34
37	50	35	27	32	34	29	24	49	29	29
38	50	32	17	18	45	24	23	42	32	34
39	49	33	47	22	13	25	13	49	23	32
40	49	18	24	25	26	29	26	34	45	25
41	50	23	45	34	30	23	21	44	45	29
42	46	33	19	16	34	22	14	45	34	33
43	49	26	26	25	33	24	23	48	29	27
44	47	34	30	33	24	27	24	50	33	38
45	49	23	13	22	15	14	16	50	32	23
46	50	45	29	36	17	25	18	45	17	35
47	49	49	33	23	34	28	15	44	22	34
48	49	33	31	31	27	19	17	45	33	29
49	44	21	43	32	25	22	15	33	27	34
50	50	39	10	34	15	24	19	49	23	45
51	50	36	20	34	18	32	17	47	26	33
52	45	15	19	11	39	25	26	35	40	35
53	49	23	32	23	42	19	23	45	34	15
54	45	17	29	35	32	23	18	39	34	47
55	45	16	15	32	40	24	16	47	45	43
56	43	32	22	25	28	18	12	29	35	27

Source: http://www.spr.com/library/0Langtbl.htm

From the above data the weight of characteristics groups which was used in the VAF equation was calculated in the Table 11.

Table 11: Weight of characteristic groups (N = 56)

Characteristic groups	Weight (W_g)		*t* -test (*df* = 55)		95% Confidence interval of the difference	
	Mean (%)	Standard deviation	T	Significance	Lower	Upper
Engineer's skill	33.3	10.1	0.011	0.991	-3.4263	3.4638
Technical characteristics	25.0	8.6	0.016	0.987	-3.0721	3.1221
Maintenance environment	33.4	9.9	0.025	0.980	-3.6048	3.5173

In the above Table 11, the mean and standard deviation of the characteristic groups are shown which is based on the result found from the questionnaire. Each mean value was tested using the two-tailed one sample *t* -test at the 95% significance level. All significance probabilities (sig.) of

Table 11 are above significance level (0.05) and confidence intervals include zero. For instance, significance probability (sig.) of characteristic group 'Technical characteristics' is 0.987 > 0.05 and the confidence interval ranging from the lower (−3.0721) to the upper (3.1221) bound includes zero. The mean value (25.0) of this group can be supported. Likewise, these mean values of Table 11 could be used as the weight value of group ($\boldsymbol{W_g}$) in the VAF calculation of the proposed model.

Table 12: Value adjustment factors of proposed model (N = 56)

. Characteristics Groups	Factors	Weight (W_f)		*t* -test (*df* = 55)		95% Confidence interval of the difference		Cronbach's alpha	Factor's Loading
		Mean (%)	Standard Variation	T	Significance	Lower	Upper		
Engineers' Skill	KAD	45.0	13.7	0.000	1.000	-4.9316	4.9316	0.679	0.359
	FPL	28.3	8.7	-0.012	0.990	-3.1448	3.1073		0.985
	ESS	26.7	8.6	0.012	0.990	-3.0736	3.1111		0.924
Technical Characteristics	SSS	31.3	10.6	-0.027	0.979	-3.8809	3.7809	0.896	0.884
	ISM	25.9	6.8	0.031	0.975	-2.4043	2.4793		0.836
	CRP	21.9	11.2	-0.013	0.990	-4.0624	4.0124		0.912
	RLS	20.8	5.3	-0.020	0.984	-1.9127	1.8752		0.862
Maintenance Environment	DOC	38.6	15.5	-0.002	0.998	-5.5811	5.5686	0.643	0.873
	SES	29.1	11.8	-0.018	0.986	-4.2945	4.2195		0.758
	TST	32.5	12.2	0.000	1.000	-4.3919	4.3919		0.646

According to the result of the data found from the questionnaire we also find that all significance probabilities (sig.) of above Table 12 are above significance level (0.05) and confidence intervals include zero. For instance, the significance probability (sig.) of the 'knowledge of application domain (KAD)' is 1.000 > 0.05 and the confidence interval ranging from lower (−4.9316) to upper (4.9316) bound includes zero. Using the same approach, the mean values of 10 factors,

shown in Table 12, are calculated and can be used as the weight value (W_f) of adjustment factors after testing of the two-tailed one sample *t* -test.

For the result collected data from part 1 of the questionnaire distributed by them, reliability and validity tests were performed. Table 12 shows the Cronbach's alphas as they exceed 0.6. The factor's loading values are the result of factor analysis using the extraction method of principle component analysis.

5.2.2. Regression Analysis

Table 13 shows the summary description of the questionnaire result of 37 maintenance projects collected from different companies from part 2 of the questionnaires.

From the dataset of the questionnaire, we see the general description of these systems. These systems have an average of 169.1 source modules. The mean value of the age of the systems is 6.24 years which is the year of operation. The programming language of the 33 systems is C, C++ or COBOL. The other four systems are written in PowerBuilder.

Table 13: General descriptions of projects (N = 37)

.Item	Unit	Mean Value
System's volume	No. of source module	169.1
Programming Language	Project	4 (Power Builder) 33 (C, C++ or COBOL)
System's age	Year	6.24
Size/ Project	Function Point	33.8
	Line of code	2775.95
Maintenance Effort/Project	Man-Month	7.2

Also, the average function points and line of code per maintenance project are 33.8 and 2775.95 respectively with a minimum of 11 and 224, and a maximum of 140 and 8320 respectively. The average effort of maintenance project is 7.2 man-months, which is much smaller than the size of new development projects.

From part 2 of the questionnaire, unadjusted FPs of Table 14 Actual effort was estimated using the IFPUG model. VAFs were calculated according to the proposed model. The responded projects include add and change types of projects. Actual efforts were collected from the survey results and measured in man-month.

Table 14: Actual effort and VAF

Project	SLOC	UFP	Mean Size	VAF	Actual Effort	Estimated In LOC	Estimated In FP	Estimated In Hybrid	System's Volume	Programming Language	System's Age
1	1705	31	31	3.27	5.7	6.1	5.9	5.69	244	C++	2
2	990	18	22	3.56	3.6	4	3.9	3.6	90	C++	3.5
3	1540	28	28	3.58	5.7	5.7	5.1	4.8	121	C++	4
4	990	18	11	3.15	1.4	2.3	2	1.5	210	C++	7
5	1638	18	18	3.04	1.4	3.1	3.1	2.5	98	COBOL	2
6	1911	21	18	3.25	2.8	3.5	3.8	2.6	300	COBOL	8
7	1001	11	11	2.84	2.1	2.1	2.6	1.4	120	COBOL	4
8	4186	46	40	2.89	5	6	5.3	4.5	211	COBOL	5.5
9	2548	28	28	1.8	6.1	6.1	6	3.1	172	COBOL	8
10	2093	23	23	2.84	2.8	4.7	4.2	3.7	192	COBOL	3
11	2457	27	27	2.46	4	5	4.5	4	120	COBOL	9
12	2002	22	22	3.41	5.3	6.3	5.8	5.5	230	COBOL	5
13	6188	68	68	3.37	20.5	20.5	18	15	160	COBOL	5
14	5187	57	57	3.01	12.9	18	15	12.5	149	COBOL	7
15	1265	23	20	3.15	3.1	4	3.7	3.2	321	C++	2
16	1045	19	19	2.65	2.8	5	4	3	168	C++	4
17	1430	26	26	2.68	4.6	7	6.5	4.9	129	C++	6

18	2915	53	53	1.9	13.4	15	15.5	13	257	C++	9
19	7700	140	140	4.05	50.2	57	55	47	120	C++	7
20	1540	28	28	3.25	3	6.5	5.7	5	198	C++	7
21	715	13	13	2.66	2.7	2.7	3.5	1.7	155	C++	3
22	2048	16	16	3.77	1.7	3.8	4	2.5	120	C	1
23	5632	44	64	3.54	15	23	19	16	221	C	1
24	368	23	23	3.28	4	6	7.2	3.8	132	Power Builder	10
25	960	60	60	2.41	12.9	18	22	14	376	Power Builder	15
26	2176	17	17	2.79	2	5.1	4.3	2.5	109	C	6
27	224	14	14	2.29	1.9	3.8	4.7	2.1	98	Power Builder	18
28	2176	17	17	2.79	2.5	3.6	4.1	3	126	C	8
29	832	52	52	1.8	11.3	14	12.3	11	111	Power Builder	19
30	5504	43	43	3.45	8.75	10	9	8.75	109	C	8
31	3072	24	24	1.93	3.98	5.5	4.8	4	325	C	4
32	4992	39	39	3.13	7.68	9	8.9	7.5	80	C	4
33	8320	65	65	3.22	15.3	18	16.3	15	90	C	2
34	7936	62	62	3.07	14.4	17	15.9	14.8	175	C	3
35	1280	10	10	2.58	1.2	3	2.5	1.5	90	C	5
36	3200	25	25	2.57	4.2	5.6	5	4	120	C	7
37	2944	23	23	2.37	3.75	5.9	5.23	3.9	210	C	9

Source: http://www.spr.com/library/0Langtbl.htm

Table 15: Regression results by adjustment range

.Adjustment range (%)	*a*	*b*	R^2	Asymptotic 95% confidence interval			
				a		*b*	
				Lower	Upper	Lower	Upper
± 0	0.041	1.440	0.9654	0.0192	0.0626	1.3249	1.5554
± 10	0.048	1.394	0.9660	0.0233	0.0717	1.2848	1.5033
± 20	0.054	1.353	0.9664	0.0277	0.0811	1.2483	1.4572
± 30	0.062	1.311	0.9648	0.0318	0.0924	1.2087	1.4140
± 40	0.068	1.282	0.9639	0.0490	0.1006	1.1810	1.3834
± 50	0.073	1.256	0.9613	0.0370	0.1090	1.1543	1.3585
± 100	0.037	1.304	0.9647	0.1768	0.0570	1.2019	1.4058
± 150	0.022	1.302	0.9646	0.0096	0.0348	1.2003	1.4043
± 200	0.015	1.304	0.9649	0.0061	0.0241	1.2023	1.4060

5.3.Validation

To evaluate the proposed model, regression analysis is performed. Equation (10) was used for nonlinear regression analyses which describes the relation of actual efforts and mean size.

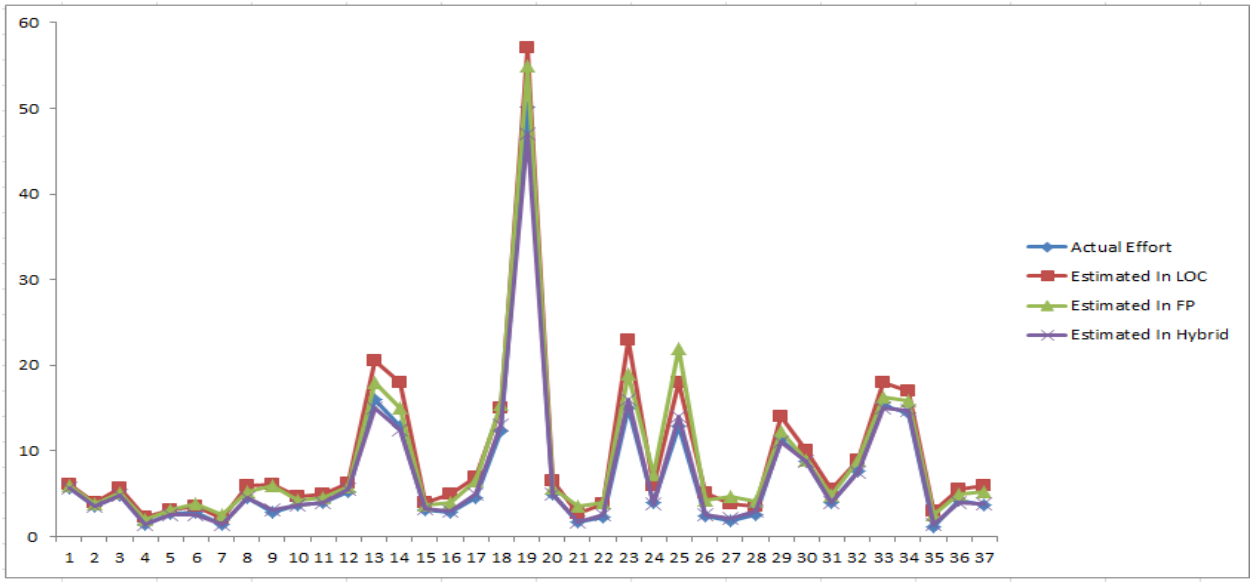


Figure 16: Comparison of different toward developed model.

The actual effort (man-month) and the Mean size values of Table 14 were used in Equation (10) for each project. The constants ***a, b*** are the coefficients which resulted from the regression. Table 15 shows a summary of the regression analysis results. We sought the best regression model by varying the adjustment range from ±0% to ±200%. Regression coefficients (R^2) of most models are above 0.9. Constants ***a, b*** are significant in the asymptotic 95% confidence interval. We obtain the highest R^2 value $R^2 = 0.9664$ in the model of an adjustment range at ±20%.

CHAPTER SIX

CONCLUSION AND FUTURE WORKS

6.1. Summary and Conclusion

The main aim of this research was to develop a reasonable software maintenance effort estimation technique. In order to estimate the efforts require maintaining the software we have to know the size of the system or software we supposed to maintain. The accuracy degree of the software size is highly related to the accuracy degree of the system or software to be maintained. So we highly focused on the size measurement in order to come up with accurate software maintenance estimation model.

The above model is designed to help the project manager in charge of software maintenance to calculate the estimated software maintenance effort. Software maintenance means repairing defects in order to correct errors and adding small new features to software applications. In this model, the method of counting the FP to estimate the size of the maintenance is referred to as the FP model of the IFPUG, and the method of counting the LOC to estimate the size of the maintenance is referred to as the SLOC model of the basic COCOMO.

The maintenance sizing is done by two different approaches; those are by using Function Points and Line of Code. The size measured by using LOC is at the end converted to an Unadjusted Function Point for it is needed to make the size measurement more accurate by applying the complexity factors categorized under three categories, and we need both measured maintenance size with the same unit in order to make them easy to measure the mean size of the maintenance size.

Ten new VAFs are used for adjusting the FPs considering the software maintenance complexity characteristics. Also it is considered as an adjustment for those converted FP from LOC. These VAFs are grouped into three categories, i.e. the engineer's skill (people domain), its technical characteristics (product domain) and the maintenance environment (process domain). Each weight of VAFs and the three characteristic groups is given by the mean value of collected data and validated by the *t*-test. The ten factors of the developed model are simpler and adequate for assessing the maintenance effort.

With experimental analysis of the ranges of adjustment, the most significant regression model could be selected. We found that Mean Size (which are a mean size calculated from both FP and LOC converted to UFP) are a good measure to estimate the effort of the maintenance project and that they can be adjusted in the range of ±20% by the 10 VAFs of the model. However, the 10 VAFs are not as influential as one expected.

We are aware that the collected data are only from small maintenance projects. Due to the limitations of the actual data, our developed model should be refined by collecting and analyzing more actual projects if this model is to be applied to an organization. The result is that the methodology used in the development of model may be used as a useful guide for the estimation of a maintenance project in any organization.

6.2. Future works

This study can be further extended by using the maintenance projects of larger size software systems. These and similar investigations will provide a better estimation of the maintenance efforts based on productivity factors.

Also it is better if the capability maturity model integration (CMMI) and capability maturity model (CMM) is considered for the future of the study for CMMI and CMM level of the organization itself has its own impact on the effort estimation of software maintenance. These are primary process improvement approaches that swamp the literature.

Finally, for the future work of the study we highly recommend that, the issues like the productivity rate of the maintenance team and Capacity Maturity Model Integration of the underlying organization on software maintenance. Additionally, we also recommend that the next study to expand the regression for large size data and take the maintainer skills into consideration.

REFERENCES

[1] Juan Carlos Granja-Alvarez and Manuel Jose Barranco-Garcia. “A Method for Estimating Maintenance Cost in a Software Project: A Case Study” Research, Spain.

[2] IEEE-SA Standards Board. “IEEE Standard for Software Maintenance” IEEE Std 1219-1998.

[3] Pierre Bourque, Richard E. (Dick) Fairley. “Guide to the Software Engineering Body of Knowledge” SWEBOK V3.0 Chapter 5.

[4] Thomas M. Pigoski. Chapter 6 Software Maintenance Technical Software Services (TECHSOFT), Inc. Book, Pensacola, Florida 32501 USA.

[5] Mr. K Koteswara Rao, Dr. G.S.V.P Raju, Mr. T.V.Madhusudhana Rao, “Effort Estimations Based on Lines of Code and Function Points in Software Project Management” Journal, Visakhapatnam 530003 India, 2008.

[6] Gerardo Canfora and Aniello Cimitile, Software Maintenance, Book, Palazzo Bosco Lucarelli, Piazza Roma, 82100, Benevento Italy.

[7] Donald J. Reifer. Chapter 2 Software Maintenance Success Recipes (CRC Press), Book, Boca Raton, FL 33487-2742 New York.

[8] Zhi YZ, “Software Engineering Standards Manual”, China Standards Press, 2004.

[9] Thomas M. Pigoski. Chapter 6 Software Maintenance Technical Software Services (TECHSOFT), Inc. Book, Pensacola, Florida 32501 USA.

[10] Rajni Jindal, Ruchika Malhotra, and Abha Jain. “Mining Defect Reports for Predicting Software Maintenance Effort” Journal, Delhi- 110042, India, 2015.

[11] Yunsik Ahn, Jungseok Suh, Seungryeol Kim and Hyunsoo Kim. “The software maintenance project effort estimation model based on function points” Journal, Seoul, Korea 2003.

[12] Aprna Tripathi, Bhuvnesh Kumar, Ashish Sharma, Dharmender Singh Kushwaha. “SRS Based Estimation of Software Maintenance Effort” Third International Conference on Computer and Communication Technology Journal, India, 2012.

[13] Sharma A., Kushwaha D.S., “Complexity Measure based on Requirement Engineering Document and its Validation,” Int’l Conf. on Computer & Communication Technology, 2010, pp. 608-615.

[14] Hakam W. Alomari, Michael L. Collard, and Jonathan I. Maletic. “A Slice-Based Estimation Approach for Maintenance Effort” Journal, USA, 2014.

[15] Andrea De Lucia, Eugenio Pompella, Silvio Stefanucci; “Effort Estimation for Corrective Software Maintenance” Journal, Italy, 2002.

[16] Dr Paul Vickers. An Introduction to Function Point Analysis, Book, Northumbria University, NE2 1XE UK.

[17] Selby R. (1988), Empirically Analyzing Software Reuse in a Production Environment, In Software Reuse: Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, pp. 176-189.

[18] Gerlich R., and Denskat U. (1994), "A Cost Estimation Model for Maintenance and High Reuse, Proceedings," ESCOM 1994, Ivrea, Italy.

[19] Zelkowitz M.V., Shaw A.C., Gannon J.D. (1979), “Principles of Software Engineering and Design”, Prentice-Hall.

[20] Yunsik Ahn, Jungseok Suh, Seungryeol Kim and Hyunsoo Kim; “*The software maintenance project effort estimation model based on function points*”, Seoul, Korea, 2003.

[21] COCOMO II, Model Definition Manual, Version 2.1 © 1995 – 2000 Center for Software Engineering, USC.

[22] IFPUG Enhancement project function point calculation, *Function Point Counting Practices Manual (Release 4.1)*, International Function Points Users Group: Mequon WI, 1999; 8.10–8.17.

[23] Sommerville I, *Software Engineering*. Addison-Wesley: Harlow, UK, 1996; 666–672.

[24] Boehm BW, Clark B, Horowitz E, Westland C, Madachy R, Selby R. Cost models for future software life cycle process: COCOMO 2.0. *Annals of Software Engineering Special Volume on Software Process and ProductMeasurement*, Arthur JD, Henry SM (eds.). Science Publishers: Amsterdam, 1995; 57–94.

[25] COCOMO Model. “Software Project Planning” Version 2, Module 11, CSE IIT, Kharagpur.

[26] Albrecht, A. J., and Gaffney, J. E., “*Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*”, IEEE Transactions of Soft Engineering, vol 9 no 6, pp 639-648, 1983.

[27] Barbara Kitchenham and Pearl Brereton, "*Problems Adopting Metrics from Other Disciplines*", Workshop on Emerging Trends in Software Metrics, May pp 1-7 2010.

[28] Jonas Verlodt, "*Software Cost Estimation Using COCOMO II model*", 2004.

[29] Pfleeger, Shari lawernce, "*Software Engineering theory and Practice*", Second Edition, Preice-Halll, Inc, New Jersey, 2001.

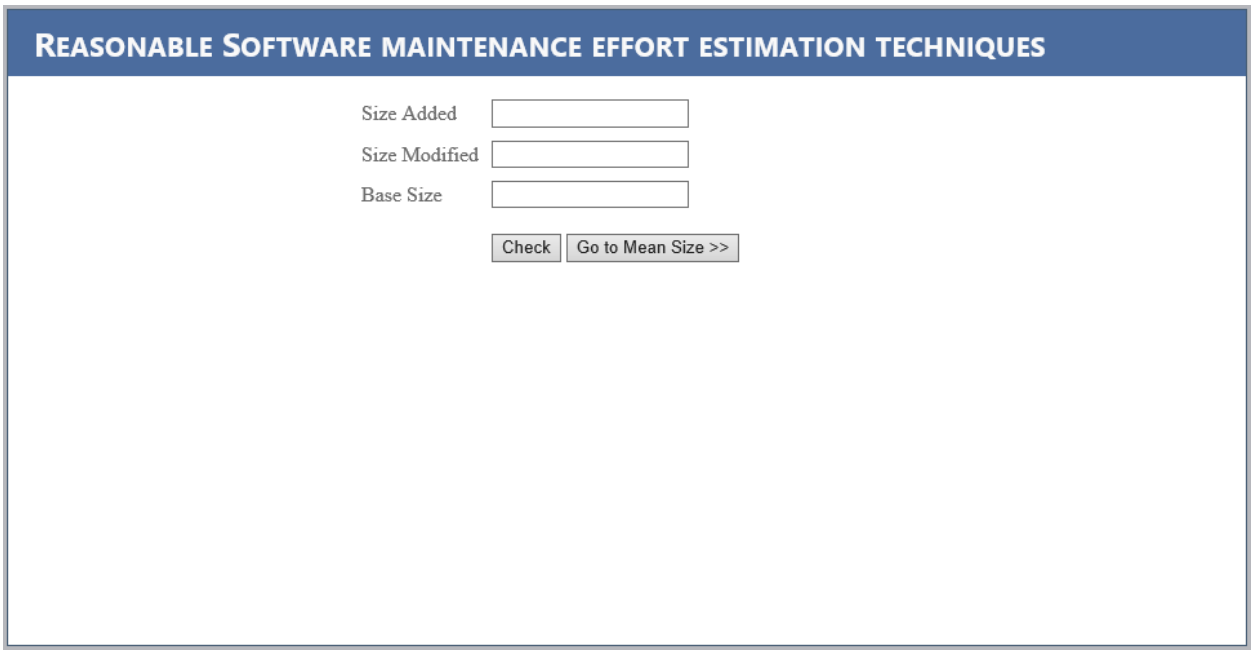
[30] IFPUG - Function Point Counting Practices Manual, Rel. 4.0 - January, 1994.

[31] IFPUG - Guidelines to Software Measurement, Release 1.1, Sep. 1996 - Chapt. 13.

APPENDICES

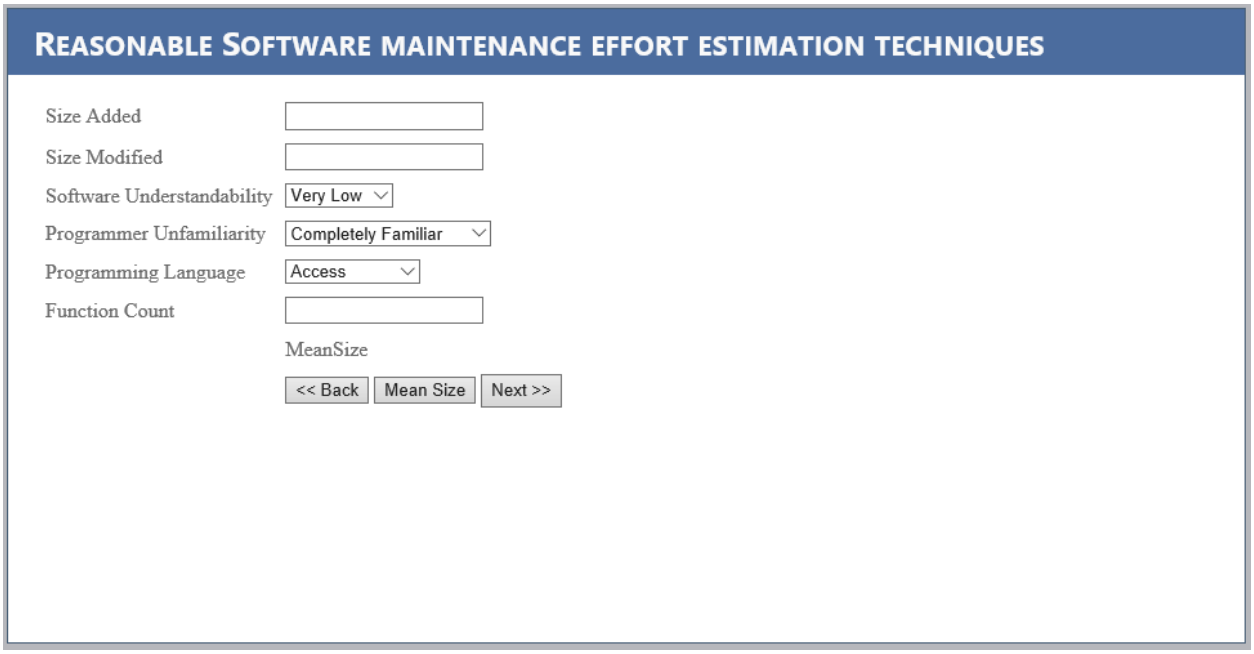
Page 1: Check Maintainability

This is to check whether the feature of to be maintained software or system is lesser than Software Base code. This interface implements Equation 1 and gives an appropriate recommendation based on the result. It accepts Size of Added feature, size of modified feature and add up the accepted input and divide by the Based code and multiplied by 100. If the result is greater than 50, it is better to develop new system or software and if it is less than 50 better to maintain existing system. And if it is equals to 50 let the maintenance manager and client decide what to do.



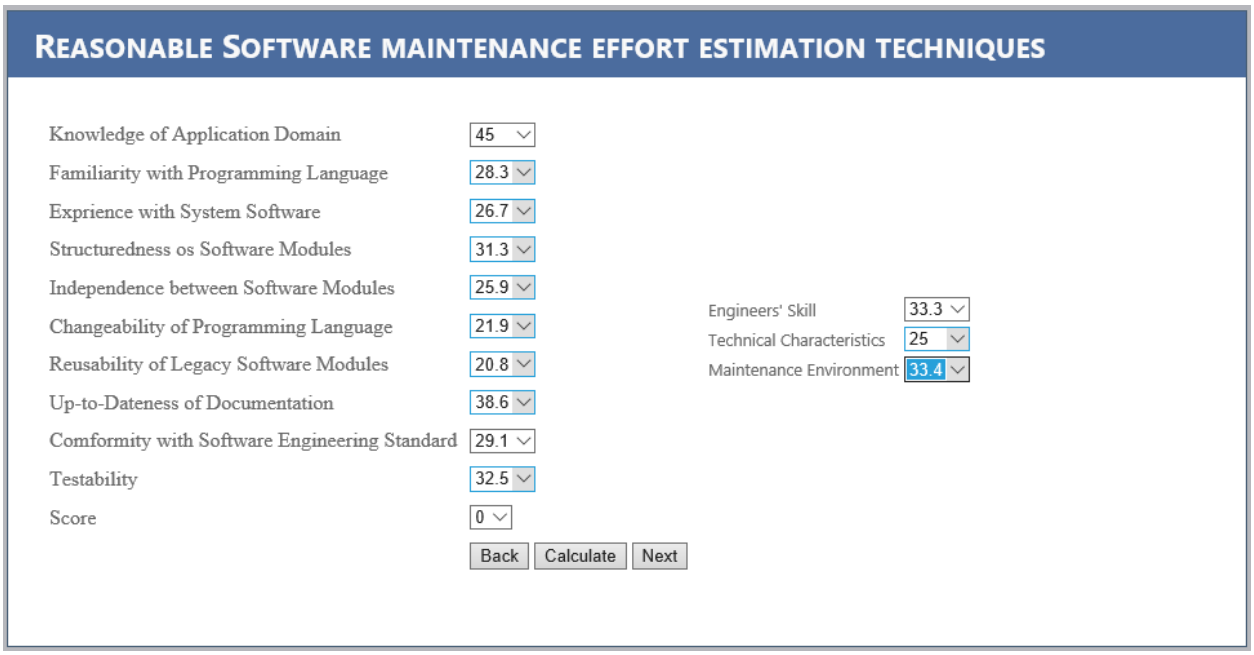
Page 2: Mean Size Calculation

The interface accepts size added and size modified and sum up and multiply by Maintenance Adjustment Factor (MAF) and convert the LOC into UFP based on selected Programming Language. The Interface also accepts the value of FC and calculates mean size by using converted UFP and FC.



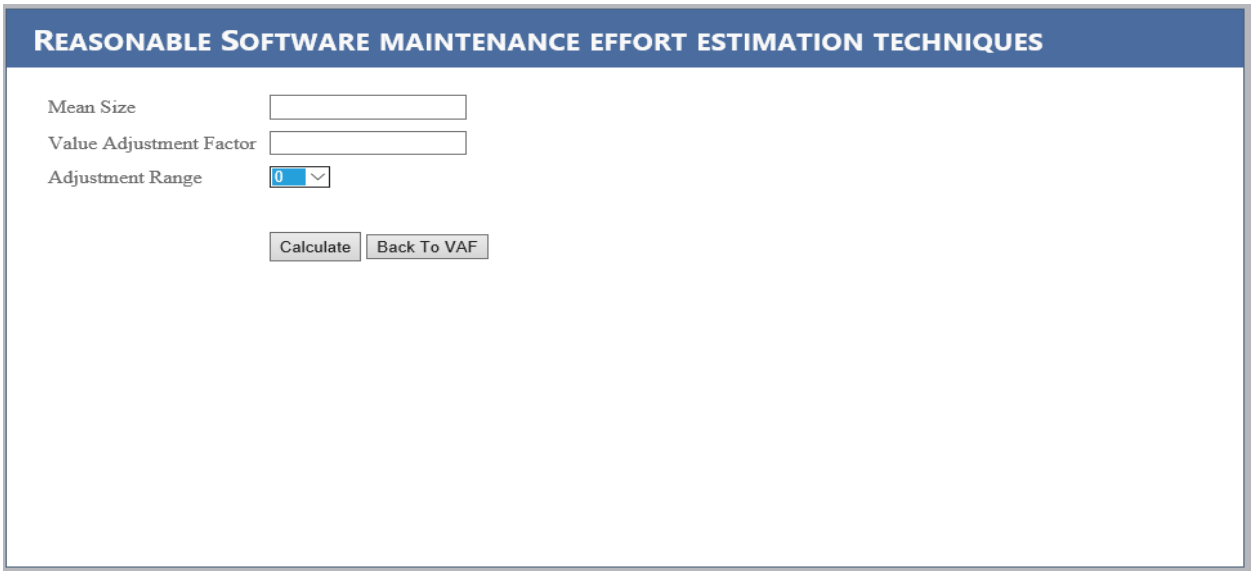
Page 3: VAF Calculation

This page calculates the VAF based on inserted data by the maintenance manager. Also the maintenance manager needs to select the score.



Page 4: Effort Estimation in MM

Finally the Mean Size and VAF calculated from the previous page is used to calculate the Effort but it needs an Adjustment range. Based on the selected Adjustment Range it retrieve the appropriate constant a, b.



```
<%@ Page Title="RSMEET Model" Language="C#" MasterPageFile="~/Site.master"
AutoEventWireup="true"
    CodeBehind="RSMEETModel.aspx.cs" Inherits="RSMEET.RSMEETModel" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">


    <div style="height: 147px">
    <table>
    <tr>
    <td>
    <table>
    <tr>
    <td>
    <table align="left" cellpadding="4" cellspacing="2"

          style="font-family: 'Times New Roman', Times, serif; font-size: medium; font-weight:
normal; font-style: normal; font-variant: normal">
    <tr>
    <td>
       <asp:Label ID="MSLabel" runat="server" Text="Mean Size"></asp:Label>
    </td>
    <td>
       <asp:TextBox ID="MSTxtBoxValue" runat="server" Text=""></asp:TextBox>
       <asp:RequiredFieldValidator ID="RequiredFieldValidatorMS" runat="server"
ControlToValidate="MSTxtBoxValue"
       ForeColor="Red" ErrorMessage="Mean Size Required"></asp:RequiredFieldValidator>
    </td>
    </tr>
```

```
<tr>
<td>
<asp:Label ID="Grand_VAF" runat="server" Text="Value Adjustment
Factor"></asp:Label>

</td>
<td>

<asp:TextBox ID="Grand_VAFTxtBxValue" runat="server" Text=""></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidatorVAF" runat="server"
ControlToValidate="Grand_VAFTxtBxValue"
ForeColor="Red" ErrorMessage="VAF Required"></asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td>
<asp:Label ID="AdjustRangeID" runat="server" Text="Adjustment Range"></asp:Label>
</td>
<td>
<asp:DropDownList ID="AdjustRangeList" runat="server"
DataSourceID="SqlDataSource1" DataTextField="Adjustment_Range"
DataValueField="b"> </asp:DropDownList>

<asp:SqlDataSource ID="SqlDataSource1" runat="server"
ConnectionString="<%$ ConnectionStrings:RSMEETConnectionString %>"
SelectCommand="SELECT * FROM [Regression_Table]"></asp:SqlDataSource>

</td>
</tr>

<tr>
```

```
<td>
    <asp:Label ID="Label1" runat="server" Text="Adjustment Range"></asp:Label>


</td><td> </td>
 <tr>
<td>
</td>
<td>
<asp:Button ID="Button1" runat="server" Text="Calculate" onclick="Button1_Click"
        style="height: 26px" />
    <asp:Button ID="Button3" runat="server" Text="Reset" />

</td>
</tr>
</table>

</td>

</tr>
 </table>
   </td>

   </tr>
   </table>


    <br />

</div>
```

```
</asp:Content>
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace RSMEET
{
    public partial class About : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Page lastPage = (Page)Context.Handler;
            MeanSize.Text = Convert.ToString(((Label)lastPage.FindControl("Label10")));
            //MSLabelValue.Text = ((Label)lastPage.FindControl("");
        }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                double KAD = Convert.ToDouble(KADList.SelectedValue);
                double FPL = Convert.ToDouble(FPLList.SelectedValue);
                double ESS = Convert.ToDouble(ESSList.SelectedValue);

                double SSS = Convert.ToDouble(SSSList.SelectedValue);
```

```
            double ISM = Convert.ToDouble(ISMList.SelectedValue);
            double CRP = Convert.ToDouble(CRPList.SelectedValue);
            double RLS = Convert.ToDouble(RLSList.SelectedValue);

            double DOC = Convert.ToDouble(DOCList.SelectedValue);
            double MET = Convert.ToDouble(SESList.SelectedValue);
            double TST = Convert.ToDouble(TSTList.SelectedValue);

            double Score = Convert.ToDouble(ScoreList.SelectedValue);

            double SUM_ENG = (KAD * Score) / 100 + (FPL * Score) / 100 + (ESS * Score) /
100;

            double SUM_TECH = (SSS * Score) / 100 + (ISM * Score) / 100 + (CRP * Score) /
100 + (RLS * Score) / 100;

            double SUM_ENV = (DOC * Score) / 100 + (MET * Score) / 100 + (TST * Score) /
100;

            double Factor_SUM = SUM_ENG + SUM_TECH + SUM_ENV;

            double ENGGroup = Convert.ToDouble(EGNList.SelectedValue);
            double TECHGroup = Convert.ToDouble(TECHList.SelectedValue);
            double ENVGroup = Convert.ToDouble(ENVList.SelectedValue);

            double Group_SUM = (ENGGroup / 100) + (TECHGroup / 100) + (ENVGroup / 100);

            double Grand_VAF = Factor_SUM * Group_SUM;

            //Server.Transfer("RSMEETModel.aspx");
        }
```

```
        catch (Exception exc)
        {
            Response.Write("Sorry Something went Wrong" + exc);

        }
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace RSMEET
{
  public partial class _Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {


        try
        {

          string sizeAdded= TextBoxSizeAdded.Text;
          string sizeModified = TextBoxSizeModified.Text;
           double SLOC = Convert.ToDouble(sizeAdded + sizeModified);
```

```
            double su = Convert.ToDouble(DropDownListSU.SelectedValue);
            double dev = su/100;
            //int dev = Convert.ToInt32(su) / 100;
            double unfm = Convert.ToDouble(DropDownListUNFM.SelectedValue);
            double MAF = 1 + (dev * unfm);

            double SLOCAdjusted = SLOC * MAF;
            //TextBox1.Text = Convert.ToString(SLOC);
            //Label10.Text = Convert.ToString(SLOCAdjusted);
            int conversion = Convert.ToInt32(DropDownList1.SelectedValue);
            double UFP = SLOCAdjusted / conversion;
            //Label10.Text = Convert.ToString(UFP);
            int FP = Convert.ToInt32(TextBoxFC.Text);
            double meanSize = (UFP + FP) / 2;
            Label10.Text = Convert.ToString(meanSize);

            //Server.Transfer("About.aspx");
            }
        catch(Exception ex)
        {
            Response.Write("Exception " + ex);
        }
    }

  }
}
```

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Configuration;
using System.Data.SqlClient;


namespace RSMEET
{
   public partial class RSMEETModel : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
      }
      protected void Button1_Click(object sender, EventArgs e)
      {
         try
         {
            string AdjustRange = Convert.ToString(AdjustRangeList.SelectedItem);
            double va = Convert.ToDouble(AdjustRangeList.SelectedValue);
            //AdjustRangeID.Text = Convert.ToString(va);

            int AR = Convert.ToInt32(AdjustRange);
            //AdjustRangeID.Text = Convert.ToString(AR);

            double FC = Convert.ToDouble(MSTxtBoxValue.Text);
            double VAF = Convert.ToDouble(Grand_VAFTxtBxValue.Text);

            double AdjustedFP = FC * ((1-(AR/100)) + (((2*AR)/5000) * VAF));
```

```
            //Label1.Text = Convert.ToString(AdjustedFP);

            string CS =
ConfigurationManager.ConnectionStrings["RSMEETConnectionString"].ConnectionString;
            using (SqlConnection con = new SqlConnection(CS))
            {
               string a = "Select a from Regression_Table where (b = " + va + ")";
                                        //string b = "Select b from Regression_Table where
                        Adjustment_Range = " +    AdjustRange;

               SqlCommand cmd1 = new SqlCommand(a,con);
               //SqlCommand cmd2 = new SqlCommand(b, con);

               con.Open();

               double constanta = Convert.ToDouble(cmd1.ExecuteScalar());

               double power = System.Math.Pow(AdjustedFP, va);
               double Effort = constanta * power;
               Label1.Text = Convert.ToString(Effort);
            }

         }
         catch(Exception exce)
         {
            Response.Write("Sorry For some error" + exce);
         }
      }
   }
}
```