A Framework for the Evaluation of Test Effort in Industrial Software Development

By
Godwin Oziegbe
Mälardalen University, Department Of Innovation Design & Engineering, Box 883, 72123

Supervisor/Examiner:
Daniel Sundmark
September, 2011.
Abstract

In software engineering there are methods for estimating the required effort for software development. Examples of such techniques are COCOMO (Constructive Cost Model) and FPA (function point analysis). However, these techniques cannot be used to estimate the required effort in carrying out testing. In this report I propose a framework or model based on previous scientific work regarding testing that seeks to merge the efforts needed in executing test cases and that of the required planning and related preliminary work to acquire test effort evaluation for a generic industrial application. In this proposed framework I compute the test execution effort from the complexity of test cases based upon the specification and this result would be used together with figures of the test planning effort to achieve our results. This method of estimating effort shows the amount of time, in man hours and the capacity of the testing team. Time begins when the test analyst starts analyzing specific requirements organizing them into different test levels and roles, the summation of times spent from this time up to the moment that a complete, debugged test script is developed and executed at least once for all requirements. Background work in this thesis gravitates around metrics popularly used in the industry regarding testing and also in software development effort.
Acknowledgement

First I thank all mighty God for granting me the privilege of writing this report and for His strength and courage through it all. I also thank my supervisor Daniel Sundmark for believing in me through the long time it took to complete this thesis and for the positive comments he made regarding my writing skills and performance. I greatly appreciate this. Many thanks also to my wife Blessing .N. Oziegbe for the support and encouragement I received through this time. Finally, I appreciate all the valuable concerns, comments and criticisms regarding the completion of this thesis, from family and friends, whose names are too numerous to write out. They know themselves and I want them to know that their concerns played a major role toward my successful completion of this thesis. Many thanks go to you all.

Västerås, August 2011.

Godwin Oziegbe
# TABLE OF CONTENTS

**Chapter 1 INTRODUCTION**

1.2 Structure of thesis……………………………………………………………………………….….…3
1.3 Research Questions....................................................................................4

**Chapter 2 THEORETICAL BACKGROUND AND RELATED TOPICS**

2.1 Categorization of different test Activities/Techniques-Test Levels.............7

2.2 Extent of measuring software testing.........................................................10

**Chapter 3 SOFTWARE DEV. EFFORT ESTIMATION METHODS/METRICS**

3.1 Software size estimation .................................................................12
3.2 Measuring software size.................................................................13
3.3 Software cost estimation techniques................................................17

3.3.1 Non-Algorithmic Methods..........................................................17
3.3.2 Analogy Costing.............................................................................17
3.3.3 Expert Judgement..........................................................................17
Chapter 1

Introduction

Testing is an activity that is carried out in software development to ensure quality. It is an attempt to find and remove errors that may have been introduced during software development and maintenance [1]. Carrying out testing may not prove the absence of errors therefore the two primary reasons for carrying out testing may be to locate and fish-out defects so the customer do not receive a flawed product secondly is to prove to the customer that the system satisfies all requirements according to the customer’s expectation [3]. Being a part or a resultant factor of software quality, proper testing can determine a product’s quality as well as how much patronage it can get in the competitive market. In fact testing is so important that test teams are allocated solely for testing in companies [2]. There are even companies whose sole job is to carry out software testing [3]. The importance of testing cannot be over-emphasized for instance the failure of life-saving software in the hospital during use for its critical purpose such as vital-signs monitoring can be catastrophic to a patient. So can be the failure of the navigation/communication systems in aircrafts, ships or military hardware used during combats including nuclear ammunitions. This is why testing is very important.

Test effort is the effort in carrying out the test process. According to the Wikipedia online encyclopaedia, “it is the expenses for (still to come) tests”. The process starts from the moment the test analyst starts evaluating test requirements up to the time a fully debugged test script is written and executed at least once. The first deliverable in test effort may be the test specification which is “a document that consists of test design specification, test case specification and/or test procedure specification” [1]. Test effort often means money because the time spent in testing must be paid for by the company this is why it is important to be able to calculate test effort either before testing or after testing. This thesis shall be about the later.

Test effort before testing in my view, means that effort is calculated before testing begins on a new project so that the cost and effort is calculated and known before testing. This means that a framework is followed for calculating this effort, on a new project. This is very good for budgeting and planning purposes. However, test effort after testing means that effort/cost is calculated after testing has been done. So in terms of reducing expenses in my view, test effort before testing is preferable because if cost and effort are known prior to testing, decisions can then be made by management to add or reduce human and material resources in planning, geared towards reducing expenses. Although this thesis is focused on evaluating test effort after testing is done, we have introduced the concept of “productivity” in the end of section 5 meaning that test effort before testing can be calculated for future related projects and this forms part of our motivation for this thesis.

In test effort evaluation, I believe, the effort in test planning and test execution must be summed up to arrive at the overall test effort evaluation. Test planning may include developing
test strategy and test plan. We should come across these terms more in the coming sections but continuing as an overview, another highly used term in testing, test execution effort seems to be related to the effort in running or executing a test case for the system under test. Before executing test cases, the test plan and strategy to be adopted should have been finalized into the test specification. Also required in order to execute test cases is the test case specification which is “a document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item” [1]. This thesis will be based on the broader picture of test effort evaluation as can be seen in the diagram in fig 1 below. On this picture we can see the main tasks to be performed during the test effort evaluation.

![Fig 1: Some tasks in performing our test effort evaluation](image)

The terms requirements and specification often overlap meaning they can be used interchangeably in my view. But more specifically speaking, requirements mean customer or user expectations or what the user want and when it becomes an agreement between the customer and the developers this becomes a specification. Requirements can be divided into explicit and implicit requirements the former is customer specific requirements while the later is derived by the software engineer. On the other hand specification is more of a formalised document written by the engineer and agreed by customer. We have several forms of specification such as design specification, test speciation, test case specification, requirement speciation etc. These are all documents in my view and in relation to this thesis work; the
customer can be the management of our establishment, other organisations or individuals. However, customer in this work is hypothetical and customer is my thesis supervisor.

It is important to be able to evaluate test effort so that resources utilized can be accounted for in terms of time, man power and budget and this can help the company to be able to estimate effort for future projects similar to the ones being evaluated thus learning from previous experience. Studies have shown that we spend about 40% of development time on testing [4] [5], this can be very expensive in terms of money but quality does not often come cheap.

Several methods and techniques have been developed over the years for measuring software development effort some of which are FPA(Function Points Analyses), UCP (Use Case Points Analysis), SLOC (Source Lines Of Codes Analysis), TPA (Test Point Analysis), COCOMO (Construction Cost Model), etc[2]. These methods are about system characteristics and not testing characteristics and cannot be used to estimate test effort [44] [7]. Hence the task of this thesis is to formulate a framework to evaluate or measure test effort and to answer specific questions such as: “what metrics (e.g. in man-hour)is to be appropriate for measuring test effort and why” but we cannot touch the issue of test effort without talking about testing itself therefore, as part of the scope of this thesis the next coming sections will be about frequently used terminology in testing and to find out about “what is the generic and complete set of measurements in test effort evaluation such as test levels, test case design, test case execution etc”. I shall also be writing about “how should data regarding the above be collected e.g. interviews, surveys, tools etc” in subsequent sections.

1.1) Motivation

Testing is highly capital intensive and since it is a compulsory task in software development towards achieving quality, it is important that cost/effort breakdown on different test activities in terms of time, money and labour utilized during testing are evaluated for budget purpose. Moreover, future similar projects can be estimated based on the experience and learning of previous evaluation. Furthermore, there are no widely known or popular methods out there for estimating test effort other than CoCoMo and the other methods mentioned above for estimating software development effort. This thesis hopes to be able to formulate a framework that demonstrates how test effort can be estimated.

1.2) Structure of This Thesis

In chapter two (2) below, which is about the theoretical background I shall be looking into some of the mostly used terminology in testing just to get the reader step into a higher tempo at the same time acquainting themselves with those useful terms that they will regularly come across while perusing through this report. Similarly, in chapter three (3), some of the most familiar software development effort estimation techniques such as COCOMO shall be discussed in limited details so as to get a grasp of their application, coverage, merits and demerits and
understand why they cannot be used to estimate test effort then in the subsequent chapter four (4), the specific question of metrics will be addressed. In it, we shall learn what metrics should or should not be used for testing which is one of our research questions that this thesis is meant to answer: “what metric(s) is appropriate for measuring test effort”, “how should data regarding testing be collected” etc. Thereafter in chapter five (5), the definition of a step by step process for performing the test effort evaluation at a generic industrial software development organization will be attempted. Words in italics that will regularly appear in most part of this thesis report are terminologies in testing.

1.3) Research Questions

The purpose of this thesis work is to investigate and seek to answer the following:

a) **Survey the research state-of-the-art in**
   1. Similar evaluation methods and frameworks
   2. Test effort estimation and related topics

b) **Based on current industrial practice in software testing, find suitable categorizations of different test activities** (e.g., test levels, test techniques, etc.) that are relevant for test effort evaluation.

c) **Describe, step-by-step, how to perform a test effort evaluation in an industrial software development organization.**

In order to satisfy number (a) above, I have studied similar areas of research relevant to software testing in general and also to software test effort evaluation specifically. After thorough study on this, the theoretical background work for this thesis is produced in chapter 2. In it, I define terms, methods etc that are needed to understand test effort and also later in section 2.1, for the research question (b) which is to categorize test activities, levels and techniques into their categories as best as possible is attempted. But further on in chapter 5, an approach for research question (c) is suggested.
Chapter 2

Theoretical Background and Related Topics

In this chapter, we shall be looking into various topics related to software testing in general. These topics are meant to provide a general background work for this thesis in order to give an introduction and guide the reader as best as possible to the theory relevant and leading to our research area. The topics that I have chosen to write about in this chapter will be familiar topics in the field and also, parts of the research questions will be answered in this chapter.

Functional Testing
This is testing that is based on analyzing the specification according to the functionality of a component or system under test [1]. In other words in functional testing, the system functionality is tested whether it complies with specification [3].

Non-Functional Testing
This is the testing of a component or system for its non-functional requirements [8]. In other words the system attributes that are not system functionalities are tested [1]. Examples are reliability, usability, maintenance, performance, portability etc. Some of these attributes overlap and they also comprise of other terms for example software performance can comprise of efficiency, reliability and scalability [8].

Defects
A defect is a flaw or fault in a component or system that causes it to behave adversely from expected behaviour [1]. This fault may be an incorrect addition or omission of a specific value in a statement. Some defects are caused by human errors such as a programmer failing to include a specific requirement into the system these are called requirement gaps. A common source of requirement gaps is non-functional requirements such as scalability, usability, performance and security [8]. If a defect is encountered during program execution, it may cause a component or system failure [1]. At different stages of software development, the earlier a defect is detected the cheaper it is to remedy the system or component from failure [8]. The figure below from Wikipedia online dictionary shows that if defects in the requirements were detected say in the construction stage, it will require 5 to 10 more time to find the defects than if it were detected in the requirements review stage [8]. A defect can also be referred to as a “bug”.
Fig 2: Finding faults early [8].

Failures
Failure is “the actual deviation of the component or system from its expected delivery, service or result” [1]. A software defect that goes unnoticed can result in a component or system failure. In a relatively large software system, every component is a potential point of failure [9] and these failures may go unnoticed until the integration testing phase. The cost of software failure can be huge, so much so that the delay in Denver airport’s opening on a particular day was due to software failure in the automated baggage handling system, a delay with costs as high as $1.1 million per day [10]. Software failure is also implicated during the Ariane 5 (which is a European expendable launch system designed to deliver payloads into orbit) explosion resulting to a cost or loss of $370 million [10]). Software reliability should be taken into account early in the system development process and a test plan that best solves the reliability question should be developed and this plan should be done early during the system design itself.

Validation and Verification
Software validation is about making sure the software is correct, specific to the user requirements. In validation the following question should be answered: “Is this the right specification?”[1], furthermore, the developers should ask this question: “Have we built the right software? Or is this what the customer wants?”[8] On the other hand, verification is the process of determining whether a system at a point in the development phase satisfies conditions imposed at the beginning of that phase [1]. In other words in verification the following questions can be asked: “Is the system correct to specification?”[1]. Or “Have we built the software right? Does it match the specification? [8].
**Static and Dynamic Testing**

*Static testing* is the testing carried out without executing the program. In static testing, the component or system is checked at specification or implementation level and the codes are not executed. Example is reviews, static code analysis [1]. Static testing is also known as *dry run* testing [11]. During static testing, programmers read codes manually to see if there are any defects [11]. Also see *white Box testing*. *Dynamic testing* is testing in which the software is actually executed with the aim of detecting any defects [1]. In dynamic testing the program is compiled and run and input values introduced to see the response. When a specific test case is executed, this is dynamic testing. Unit, integration, system, acceptance testing are all dynamic testing methodology [8]. A related terminology is *dynamic program analysis* where programs that are part of a software system are executed with inputs to study the behaviour of that system. Also *dynamic analysis* is a process of observing the behaviour of a system during execution for especially non functional characteristics like CPU usage, performance, etc.

2.1) **Categorizations of Different Test Activities/Techniques-Test Levels**

**Unit Testing**

Unit test is the smallest stand-alone test performed on a software unit which can be a component or system. A unit test is also called a module test [16]. Certain entry criteria must be met for unit testing to proceed and exit criteria also need to be stipulated in order for the testing to be approved. Some of the entry criteria can be:

- The code for the module is 100% complete.
- The development environment is ready and stable
- The customer requirement is complete or nearly complete and approved

While the exit criteria can be some of the following:

- There is no major defect that can prevent the testing from proceeding to other levels
- The testing is approved by management

A unit test validates a particular module based on the design specification. It tests functionality and reliability and where there are defects the system can be fixed prior to proceeding to other test levels [15]. Units are small building blocks in a program. In a programming language like C, on a broader level, units are the individual functions and unit testing is the process of validating such building blocks. Certain benefits accrue when we carry out unit testing, some of the major ones are as follows [17]:

- Defects can easily be detected and removed at a much cheaper rate than at later stages
- When we limit our search for bugs to a small area or unit, debugging is simplified.
• We are able to test internal conditions that cannot be detected by larger integrated systems such as exception conditions not easily encountered during normal operation

• A high level of structural coverage of the code can be reached.

Integration Testing
After each individual units (modules or components) is tested separately and approved, it will then be time to test and see how all these units work together to form the complete system. However, where there are still defects, the severity or type of the defects can dictate whether or not to proceed onto the integration phase, or to run both integration tests and some of the remaining unit tests in parallel [18]. For instance such a defect maybe due to incomplete system such as “database not found” Therefore, testing the interaction of parts of a complete system is known as integration testing [16].

Hardware integration with the software can also be tested during integration testing in order to see how the hardware components will function when using the software. The integration test plan is often written and followed prior to integration testing this plan can be contained in the overall test plan. A part of integration testing concerned with testing the interfaces between components or systems is called interface testing [1].

System Testing
System testing is the testing of the entire software system. In this test all components, parts both hardware and software are tested. The disparities between the specification and implementation are determined [21]. This testing involves both functional and non functional testing (see functional and non-functional testing above) involving the load, reliability, performance, and security etc [21]. The difference between system and integration testing is that system testing involves the entire system from a to z as a whole while in integration the interaction between units are tested in separate tests whereas system test is more of a top down approach testing the finished system.

In system testing, involvement with outside systems should be minimized in order to reduce externally induced problems [16]. But whereas integration testing in small form is only limited to testing the integration of interconnecting parts of a system or component to expose defects, another software testing term known as system integration testing concerns itself with testing how a complete system or package interacts, integrates or connects with other external systems. Examples are web applications, Electronic Data Interchange [1].

The first test usually performed on a new system is usually the “smoke test”, a quick run of a system going through its main functions without paying much attention to details. This term is borrowed from hardware testing, turning on a system for the first time to see if it will work successfully without smoking or bursting into flame [16].

One more difference is that in system test, when a defect is uncovered, previously executed parts of the system must be re-run to make sure the correction of the defect does not affect other parts of the system whereas in integration testing only the parts concerned are tested.

Acceptance testing also known as Beta testing or application testing or end user testing [16] is the testing that the test team and the customer perform together towards the satisfaction of the later. This test is done in a simulated/real environment.
Regression testing is testing when there is a bug or a fix, it happens throughout the system life cycle. Regression testing can be done at any time when there is need for some form of modification, a fix of a problem, or a change occurs [21].

Other types of testing are:

**Compatibility testing:** How compatible a system is to other system for instance in a web application to see if the system is compatible to different browsers [16].

**Performance testing:** Testing the performance of the system based on criteria such as efficiency, reliability etc [1]. In performance testing, the scalability of the system as to for instance how the system will behave in certain terms when for instance several users are added or during unusual conditions must be tested [16].

**Stress testing:** Testing to see what level of stress a system will fail. That is loading the system above normal and since catastrophic failure can mean great consequence depending on the system, stress testing is essential. During this test, a gradual reduction of performance is recorded and is the desired result [16] [1].

**Load testing:** Similar to performance testing but specific to load. This means determining at what load the system will fail e.g. number of simultaneous users or number of transactions [1]. In load test, the performance level is recorded to determine what level requirements can be accommodated as stipulated in the system requirements. In other words the capability of the system is tested [16] whereas stress testing is testing the application beyond the limit of stipulated requirements.

**Automated Testing Vs Manual Testing**

While manual test deals with carrying out testing manually by the test engineer, an automated test will be performed by machine. Some of the activities in testing in general still requires a human intervention meaning that manual testing will always be a part of any test effort [22] such as drawing up a test plan, test cases, test design, test environment, these are all activities of a human test engineer. In manual testing the engineer establishes test cases to execute according to the test design and plan with a detailed description of how to perform the tests, performs the tests step by step and determines if the test where successful or not, or complete and he records his findings. Manual testing is very useful in the early stage of software development when the system under test is still unstable. [22] See fig 5.0 for a diagram showing the process of manual testing.

On the other hand, automated testing is a testing technique where a testing tool is used to run test scripts automatically. The test engineer has to be grounded in programming to be able to write test scripts for automation and this is a time consuming job [22] because care must be taken to avoid introducing any errors that will affect result. Also, the environment should be prepared for this tool to be deployed. Examples of test tools are: autopilot, android, auto tester for windows etc [23].
2.2) Extent of Measuring Software Testing

One of the main challenges of this thesis work as stated earlier is to find the appropriate metrics for measuring software testing and since there are available techniques for measuring software development effort in general we can learn from these and come up with our own approach at the end. In the next session, “the traditional software development effort estimation techniques” will be discussed and after this we can begin to see similarities and differences in trying to relate these techniques to those of test effort evaluation. “The Extent of Measuring software testing” is a topic I coined myself after stumbling on a report by CEM KANER [24] titled “Measurement Issues and Software Testing” in that report I quickly learnt about some of the useful factors that could affect the extent of measurement e.g. software complexity, tester’s ability etc as I have listed them below with brief explanations. Therefore the quality of testing and/or the extent of it can be affected by any of these variables or factors and in the use of the term “The extent of measuring software testing” I mean the measure of the extent of it. In other words, the extent to which we can test a particular software system depends upon the depth of all or any of these factors. And also, in measuring testing or the act of carrying out testing, we have to be able to break down and show in man hours, the times and costs expended in the process.

In that report, (by Kaner), the variables or factors I have mentioned above are discussed and I summarize them as follows:

**Software Complexity** [24]: How large is the project, this can be measured by lines of codes as stated earlier or similar criteria, also, the view on complexity can be different from tester to tester sometimes regarded as more of a psychological metric i.e. how big does the tester see the project at hand, is it hard to maintain or understand? [24] [25].

**Tester’s ability or knowledge**: How efficient, productive etc. is a tester? This is a notion of human ability or performance.

**Coverage area or extent of testing**: To what extent of testing is considered enough testing? Is code coverage regarded as a measurement of test effort?

In general all these metrics are subjective and therefore hard to measure [24] or compare since they are subject from person to person or system to system.

How then do we then define measurement?

“Within the computing literature, measurement is often defined as the assignment of numbers to objects or events according to a clear cut rule.” [24].

This definition is rigid in the sense that even adding how many times the software is run or for testing can be said to be measurement so another better definition can be:-

“Measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory” [24].

**Product Coverage**: What percentage of the project is to be covered? A good enough testing should be determined that covers essential areas of the entire project. An infinite amount of testing cannot be attained therefore the product coverage should only reflect essential testing. After testing, the actual product coverage should be recorded against the initially planned coverage and this can be written as a percentage [24].
**Agreement-Base:** What are the tasks to be covered according to agreement? In reality, the actual tasks completed maybe more or fewer so, tasks could be added or removed as testing proceeds. Sometimes tasks never foreseen could come up during test such tasks should be reported and the agreement revised [24].

**Risk-Based:** This is the Assessment of risks or risk analyses and how they can affect the project. Where there are known risks or threats, they should be tested the extent of testing or areas to be covered during testing depends upon agreement which means effort, translating into money. The more the organisation is willing to spend the more areas should be covered during testing. The risks or threats under which testing was done must be stated [24].

**Project History:** Your project in comparison to similar project in history, try to learn from the later to estimate the amount of work to be done for the new project.

**Effort:** The effort should be reported in man hours. The test team should calculate the amount of hours spent on testing.

**Test Environment:** The test environment under which testing takes place must be taken into account such as the hardware, software tools, users etc. For example in a web-based application, you will need an internet or network connection, suitable hardware, integration software (for integration testing) and any other interface needed, the test team should report if the connection was available, and state the test environment that was available or not.
Chapter 3
Software Development Effort
Estimation Methods/Metrics

We cannot investigate the subject of test effort estimation without touching aspects of existing traditional software development effort estimation methods in my opinion. In this section, we are going to be studying two relatively broad topics, the software size estimation and software cost estimation. The software size is significant in determining the effort required for both the software development project [25] and test effort, while the software cost estimation will show how this effort is translated into financial terms, in man-power and other terms and by studying these, we can learn about these metrics and apply them in our research area of test effort. However, the methods/metrics we shall be looking into in this section shall be discussed in limited detail.

3.1) Software Size Estimation

The software size may be the one factor that affects the software cost the most [26] and invariably test effort. This is why we must estimate software size correctly and not too low in order to avoid cost and schedule overruns. But according to the author in the report just mentioned above, in reality, we often underestimate software size and so the actual resources and schedule figures are often higher than we initially predicted. To size software, we need to employ a variety of software sizing techniques and not to rely only on one method in order to avoid budget schedule risks from too low size estimates. However, the earlier the estimation is made, the less the software is known and the more it is prone to inaccuracy [25].

According to experts, software size can be determined from the following [25]:

- The number of functional requirements in both the requirement specification and the interface specification.
- The number of software units as contained in the software design description.
- The source lines of codes SLOC or the function point estimates.

Software size should be measured, tracked and controlled throughout the development of the software so that estimates can be compared to the actual size and to determine trends and progress [25].

Furthermore, it is very important to estimate software size early in the software development lifecycle so that early significant deviations can be detected. These deviations can be as a result of the following problems:

- Error in the model or logic used for in the development of the estimates
- Error in the requirement design, coding, process etc.
- Unrealistic or wrongly interpreted requirements and resource estimates used for the development of the software. Error in the development rate estimation

Perhaps most importantly, size based estimation should be compared to existing similar project’s size, scope and complexity.
3.2) Measuring Software Size

There are generally five main metrics used in measuring software size out of which only three can be said to be the main metrics most frequently used. Listed above are these three main metrics. But by tradition, the main method or metric is the SLOC. The other metrics as stated above are the use of the Function Points (i.e. size according to functionality) popularly used in the Management Information System (MIS) and the Feature Points as used in embedded systems. These techniques, used where applicable, are very useful in software development. Below are the main differences between the Function Points and SLOC techniques [25].

<table>
<thead>
<tr>
<th>FUNCTION POINTS</th>
<th>SOURCE LINES-OF-CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification-based</td>
<td>Analogy-based</td>
</tr>
<tr>
<td>Language independent</td>
<td>Language dependent</td>
</tr>
<tr>
<td>User-oriented</td>
<td>Design-oriented</td>
</tr>
<tr>
<td>Variations a function of counting conventions</td>
<td>Variations a function of languages</td>
</tr>
<tr>
<td>Expandable to source lines-of-code</td>
<td>Convertible to function points</td>
</tr>
</tbody>
</table>

Fig 3.1 Function Points Versus Lines-of-Code [25].

Source Lines-Of-Code (SLOC)

The source lines of code is the total number of lines in the software delivered that excludes blanks, comments and continuation lines [25] [26]. This technique is popularly known as LOC and it is the most important factor in determining the software cost. Estimating the LOC before the software is built is as difficult as estimating the software cost and it is mostly derived by expert judgement and a technique known as PERT [26]. “It involves experts' judgment of three possible code-sizes: Sl, the lowest possible size; Sh the highest possible size; and Sm, the most likely size. The estimate of the code-size $S$ is computed as” [26]:

$$S = \frac{S_l + S_h + 4S_m}{6}$$

[26].

PERT can be used to estimate the size of each individual component and then their summation should yield the estimated size of the software. Expert judgement can be used on the basis of comparing the new software with existing software of similar functionality we can achieve a better estimate using PERT [25].
This estimate is then also compared with the actual software on completion. This is the beauty of using the LOC. Other advantages are that the LOC is simple to use and it enables recording of actual software size for future use and it facilitates automated counting of actual software size. Another model the LOC can be used with the predictive model called Constructive Cost Model or COCOMO. In using the COCOMO, your estimates must be continually updated as new information becomes available and it is only through this reassessment that this model can be used to achieve actual cost estimates. The COCOMO model is used to further refine metrics using the quality and productivity equations of the model [25].

But the SLOC has also a few drawbacks. For instance, it is almost impossible to estimate cost from the initial requirements using the LOC without sufficient details. The planner must therefore estimate costs before the details arrive that will enable them to make a more accurate estimation. Also since the SLOC is language dependent, it is not easy to standardize the definition on how the SLOC are counted thus making size comparisons with applications written in different programming languages difficult [25].

Apart from PERT, another technique that can be used in addition to expert judgement is the Code Length and Volume metrics in the software science approached proposed by Halmstead [26]. The code length is as the name implies used to measure the length of code in the software thus:

\[ N = N_1 + N_2 \]

Where,

- \( N_1 \) = total number of occurrence of operators
- \( N_2 \) = total number of occurrence of operands

While the volume is the amount of required storage space and it is defined by:

\[ V = N \log(n_1 + n_2) \]

Where:

- \( V \) = volume
- \( N_1 \) = number of operators
- \( N_2 \) = number of operands in the software programme.

However there has been disagreement over this theory and it has been receiving decreasing support in recent years [26].

**Function Points**

The function point which is a frequently used metric to determine software size relates according to A.J. Albrecht [25] [26], to the weighted estimates of five different factors from the software requirements as listed below:

- User Inputs
- User Outputs
- Logic (or master) (or internal logic) files
- Inquiries
• **Interfaces.** [25]

This measure is based on the functionality of the software and it is derived by first counting the number of functions in each category as listed above in the software. They are then adjusted by adding complexity measurement to each type of function points to arrive at the sum of the total adjusted function points measure. This final count can be converted into a useful estimate for required resources of developing the software system [25].

The functions classes or categories are explained as below:

1. User inputs: these are controlled user input-types given from the user to the system.
2. User Output: the output to the user given the inputs.
3. Inquiries: the interactive input-types that require user response
4. Logic: the internal files or logical groups of information used and shared by the system itself
5. Interfaces: functions or files passed or shared with other “external” systems [26].
   - Each of these functions are then counted and multiplied by a given figure and then adjusted by a degree of complexity using expert judgement.
   - The process for complexity adjustment is highly domain specific and is affected by the following factors: “data communications, distributed data processing, performance, transaction rate, on-line data entry, end-user efficiency, reusability, ease of installation, operation, change, or multiple site use” [25].
   - One of three levels or degree of complexity; *simple, average* and *complex* is assigned them and then an established weighted figure given during this process [25] [26].

The unadjusted function count (UFC)=
\[ \sum (\text{The total number of elements of a given type})*(\text{the weight}) \] [28].

- The function point count is then adjusted by the complexity of the project [28].

Below is the summation of the above in table form:

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Average</th>
<th>Complexity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>3X</td>
<td>4X</td>
<td>6X</td>
<td>20</td>
</tr>
<tr>
<td>Outputs</td>
<td>4X</td>
<td>5X</td>
<td>7X</td>
<td>19</td>
</tr>
<tr>
<td>Inquiries</td>
<td>3X</td>
<td>4X</td>
<td>6X</td>
<td>0</td>
</tr>
<tr>
<td>Files</td>
<td>7X</td>
<td>10X</td>
<td>15X</td>
<td>10</td>
</tr>
<tr>
<td>Interfaces</td>
<td>5X</td>
<td>7X</td>
<td>10X</td>
<td>10</td>
</tr>
</tbody>
</table>

**UNADJUSTED FUNCTION COUNT (UFC) = 59**

Fig 3.2  Function point calculation [25].

FP can be used to estimate LOC [28] depending upon how many LOC per function point there is. The average number of lines of code maybe used for the calculation:

\[ \text{LOC} = \text{AVC} \times \text{number of function points} \]
Where AVC is the average number of codes per function point and it is highly language dependent.

An advantage of the FP based estimation is that it can help in making initial estimates based upon requirements specification during the early stage of development. This is why using the FP is more preferable to the SLOC at the early stage of software development after the SRS has been completed. However the technique has a few drawbacks just like the SLOC at the very beginning of development, it is difficult to obtain. Additionally the complexity factor is subjective and is based upon the judgment of the expert therefore automatic FP count is impossible to obtain and also two different projects are difficult to compare if the FP estimation is done by different engineers or experts [25, 28].

**Feature Points**

The feature point is an extension of the function point as stated above but the key difference is with algorithm. In feature point algorithms are set by mathematical rules e.g. Gaussian elimination method, tower of Hanoi algorithm, square roots etc. and are factored-in in addition to the function points to obtain computational values for the system. Therefore in addition to the five factors or parameters of the function point calculation, a sixth factor, algorithm is added for the feature points calculation and given a default weight of 3 from a range of 1-10 of which 1 meaning simple to 10 meaning sophisticated algorithms. And the feature point is the weighted total number of algorithms plus the function points.

The feature point is used more in real time system with high level of complexity but fewer inputs/outputs than those in the MIS e.g. in mathematical software, military, and surgical as well as in robotics [26][25]. For applications with more algorithms than logical files, feature point calculation produce more total number than function point calculation and vice versa. The table below summarizes the ratio between feature points and function points count for certain domains.

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>FUNCTION POINTS</th>
<th>FEATURE POINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch MIS Projects</td>
<td>1</td>
<td>0.80</td>
</tr>
<tr>
<td>On-line MIS projects</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>On-line database projects</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>Switching systems projects</td>
<td>1</td>
<td>1.20</td>
</tr>
<tr>
<td>Embedded real-time projects</td>
<td>1</td>
<td>1.35</td>
</tr>
<tr>
<td>Factory automation projects</td>
<td>1</td>
<td>1.50</td>
</tr>
<tr>
<td>Diagnostic and prediction projects</td>
<td>1</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Fig 3.3 Ratios of Feature Points to Function Points [25]
3.3) Software Cost Estimation Techniques

The other arm of our traditional software development effort evaluation study as we stated earlier is the software cost. Software cost estimation methods are grouped into two methods according to my research. These are the algorithmic and the non-algorithmic methods. The algorithmic models are based on the use of mathematical rules/algorithms such as statistical means, standard deviation, regression, differential equations etc, to calculate cost. Accuracy of these models are quiet mixed and cannot be used as off-the-shelf [26]. The non-algorithmic are non-mathematical in nature and rely upon human judgements based upon professional experience, previous similar projects, resources available and other forms and means for the estimation [25][26][27].

3.3.1) Non-Algorithmic Methods [25] [26] [27] [28]

This includes

- Analogy costing
- Expert judgement
- Parkinson’s law
- Price-to-win
- Bottom-up (or Engineering-build)
- Top-down

Analogy Costing:

This is a method whereby previous similar projects are compared to the current one reasoning is applied by analogy in the comparison to estimate the metrics of the new project. Estimates are relatively easy to achieve since for instance the actual costs, resources used, and time spent, etc for the old project is already known. The old and new projects must however be in the same domain area for a realistic comparison to be achieved [27] [28].

Expert Judgement

Expert judgement simply means that an expert is consulted he or she then uses his/her own experience/expertise to estimate the metrics. It is always better to get the expert assessment of several experts instead of just one for obvious reasons. Experts in both the new project and the project domain should be consulted. The inconsistencies can be harmonised by using the Delphi method or the PERT, iteration proceeds until an acceptable estimate is achieved. Drawbacks of this method are that highly experienced experts are hard to find for every new project and the proposed project may not always be of the same size as the old project among other constraints [26] [27] [28].

Parkinson’s Law

Parkinson’s Law, states that “work expands to fill the available volume” [26] [28] meaning that determination of cost is based upon available resources instead of on objectives. If 4 people are available and is to be delivered in 6 months the effort is estimated to be 24 persons / month.
This method can yield an unrealistic estimate and does not promote sound software engineering practice therefore it is not recommended [26] [28].

Price-to-win

The software development effort is estimated based on the best price to win the project. This means the cheapest i.e. what the customer can afford. For instance, if the cost is estimated reasonably to be 80 persons / month but the customer can only afford 50 person/month, the estimation will be modified to fit only “50 persons/month”. This is also not a good software engineering practice. This can lead to low quality system and the developers can be forced to work overtime among others [26] [28].

Bottom-Up (Engineering Build)

In this method, estimation must start from the grassroots or bottom units such as components with all the tasks and functionality breakdown done and the summation of all the efforts and schedules will be the total estimation for the project. This method requires a detailed knowledge of software engineering principles and architecture. This method is also labour intensive but it captures the entire scope of work needed and provides a better assurance than all other methods. A requirement for this system is that an initial design must be in place showing the breakdown of the system [25] [26] [28].

Top-Down

This is the opposite of the bottom-up approach. System level activities are broken down into various components or units. It starts from the overall system’s global properties and using algorithmic or non-algorithmic methods to derive cost or schedule estimates, this estimate is then shared among the different sub-levels such as components and other units. This method does not require knowledge of software architectural design and it is more suitable for costs and schedule estimate at the initial level of estimation of the project. However, the detailed-level activities can be under-estimated under this method [26] [28].

3.3.2) Algorithmic Methods [25] [26] [27] [28]

Algorithmic models are mathematical models that estimate costs given a number of variables known as cost variables. An algorithmic model is of the form:

\[ \text{Effort} = f(x_1, x_2, ..., x_n) \]

where \( \{x_1, x_2, ..., x_n\}\) = cost factors

Size is a cost factor but besides size, there are other factors that directly affect software cost like project, product and process attributes. These factors are also found in the COCOMO II model which is to be discussed briefly, later in this section [26]. Different models exist most models are generally similar expressing Effort functions with form:

\[ \text{Effort} = A \times \text{Size}^B \times M \]

Where A, B are cost factors and M is a multiplier of product, process or people attributes

However the values for A, B, and M are dissimilar for most models. Examples of models are:
• COCOMO model
• Linear model
• Multiplicative model

COCOMO
The CONstructive COST MOdel popularly called COCOMO is an empirical model that is based on project experience. The COCOMO is widely accepted and used in software engineering. In this model the Code Size is represented in thousands of LOC or KLOC and Effort is in Person-month or PM. The model was initially published in 1981 by Barry Boehm. Another version is called COCOMO 2 which takes into account different aspects of software development or object oriented programming such as reuse etc.

a) Basic COCOMO [26] [27].
The basic cocomo model is relatively simple to use because many cost factors are not included in the calculation therefore it is better to use this model as a rough estimate. This model uses three different categories or set of \( \{a, b\} \) respective to the level of complexity as below: understand

(i) for simple, well comprehended application systems, \( a = 2.4, \ b = 1.05 \);
(ii) for relatively more complex applications, \( a = 3.0, \ b = 1.15 \);
(iii) for embedded systems, \( a = 3.6, \ b = 1.20 \).

This can be summarized into the table below.

<table>
<thead>
<tr>
<th>Application Type</th>
<th>Value of a</th>
<th>Value of b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Application</td>
<td>2.4</td>
<td>1.05</td>
</tr>
<tr>
<td>Complex Application</td>
<td>3.0</td>
<td>1.15</td>
</tr>
<tr>
<td>Embedded System</td>
<td>3.6</td>
<td>1.20</td>
</tr>
</tbody>
</table>

Fig 3.4 Applications and their \( \{a, b\} \) values [27].

b) Intermediate and Detailed COCOMO [26] [27].
The main difference is the value of \( a, b \) thus:

<table>
<thead>
<tr>
<th>Application Type</th>
<th>Value of a</th>
<th>Value of b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Application</td>
<td>3.2</td>
<td>1.05</td>
</tr>
<tr>
<td>Complex Application</td>
<td>3.0</td>
<td>1.15</td>
</tr>
<tr>
<td>Embedded System</td>
<td>2.8</td>
<td>1.20</td>
</tr>
</tbody>
</table>

Cost Factors [26] [27]
Below are additional cost factors. The overall impact \( M \), of these factors is the products of all the cost factors. The cost factors are:
1. Product
Under product, the cost factors are:
   a. Required Software Reliability
   b. Database Size
   c. Product Complexity

2. Computer
The factors regarding machine used are the following:
   a. Execution Time Constraint
   b. Main Storage Constraint
   c. Virtual Machine Volatility
   d. Computer Turnaround Time

3. Personnel
Personnel means the staff involved be it analyst, the programmer etc. Below are the factors:
   a. Analyst Capability
   b. Programmer Capability
   c. Application Experience
   d. Virtual Machine Experience
   e. Language Experience

4. Project
   a. Software Tools
   b. Modern Programming Practices
   c. Development Schedule

(C) COCOMO 2
The major difference between early cocomo models and the cocomo 2 is the change in the variable b that changes according to the cost factors below:

1- Development Flexibility.
2- Architecture or Risk Resolution.
3- Team Cohesion [27].

4- Process Maturity [27].

Cocomo 2 is useful in providing support for different activities such as project planning and cost estimation.

**Summary of estimation models and their comparisons [27] [26].**

<table>
<thead>
<tr>
<th>Method</th>
<th>Merits or Strengths</th>
<th>Demerits or Weaknesses</th>
<th>Algorithmic/Non-Algorithmic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert Judgment</td>
<td>Fast estimation, Experts that have experience can provide good estimation.</td>
<td>Because it depends upon expert, it can lead to biasness</td>
<td>Non-Algorithmic</td>
</tr>
<tr>
<td>Analogy</td>
<td>It is based upon actual project data and previous experience</td>
<td>Inaccuracy in historical data, similar project may not be available</td>
<td>Non-Algorithmic</td>
</tr>
<tr>
<td>Parkinson, Price to Win</td>
<td>Good to win the contract</td>
<td>It is not a good software engineering practice, Large overruns</td>
<td>Non-Algorithmic</td>
</tr>
<tr>
<td>Top Down</td>
<td>Minimal requirements</td>
<td>Less accurate than other models</td>
<td>Non-Algorithmic</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>Based on detailed analysis, Better project tracking as compared to other models, a good sw eng. Practice</td>
<td>Need more estimation effort than top down, not suitable for early estimation</td>
<td>Non-Algorithmic</td>
</tr>
<tr>
<td>Algorithmic</td>
<td>Objective, results are repeatable, It facilitates better understanding of estimated methods</td>
<td>Subjective Inputs may not reflect the current state or environment. Too company specific, may not be generic.</td>
<td>Algorithmic</td>
</tr>
</tbody>
</table>

Fig 3.5 Summary of estimation models and their comparisons [27] [26].
Chapter 4
Typical Software measuring and metrics [25]

You may agree that one way to ensure the delivery of quality software products is to improve the management of software engineering practise or process [25]. In the report by CEM KANER, J.D., Ph.D. I have learned that industry metrics are used for software development management and are mainly concerned with the product or process through which it is developed ranging from everything from high-level software development effort/size measurement to the detailed software requirement measures and even personnel information [25] [31]. The software product can be thought of as the final result that originates from initial letters and words of requirement up to the finished software system including source and object codes, documentation and other resources of the system. As the models of measurement of these product and process are developed, metrics can be used to predict or estimate cost and schedule and also in measuring performance quality etc which can lead to improved result in the management of the software development process. [25] [31]. In that report, software metrics are grouped into product metrics and process metrics this i.e. according to the report on my reference number 25 or [25]. Product metrics are measurement of the software product at any stage of development including the requirements and the complete system and they include measure of complexity, size etc. The Process metrics are measurement of the process in building the software product, and they include measures of development time, methodology or staff experience etc.

Below is a list of measurements and metrics popularly used in software engineering in general [25] but some of them in my view, and as seen in the report “Test Execution Effort and Capacity Estimation” by Eduardo Aranha and Paulo Borbal [2], can be useful in analysing our research area of software test effort measurement and although I list all the metrics that I find below, I shall be writing short description or details only about those I consider relevant to test effort evaluation at this stage which is my research focus[25] [31]. And therefore I shall classify them as “measurement and metrics that should be used in software testing” The measurement and metrics popularly used in software engineering are the following:

- Quality,
- Size ( SLOC, function points, feature points, etc),
- Complexity,
- Requirements,
- Effort,
- Productivity,
- Cost and schedule,
- Scrap and rework, and
- Support.
4.1) Measurements and Metrics that should be used in Software Testing [25]

Of the above stated measurements/metrics, the following in my view should be used in testing:

- Size: SLOC, function points, feature points
- Complexity
- Requirements
- Effort
- Cost
- Schedule

Of the above, you will find the following four measurements and metrics more relevant and regularly used in this thesis work especially in the step by step performance of test effort in chapter 5:

- Effort metric calculated here for this thesis in man-hour (mh)
- Requirements
- Cost
- Complexity: as execution points (in this thesis)
- Productivity: as execution points per minute (in this thesis productivity is discussed in chapter 5).

This is because in the next chapter, these are the only metrics that will appear as I describe how to perform test effort evaluation. And in the next chapter five (5) which is my last chapter, is about the research area question number (c): “Describe, step-by-step, how to perform a test effort evaluation in an industrial software development organization.”, we shall see why and how these metrics are used in test effort evaluation.

Below I have written short notes on these metrics.

Size [25]

As mentioned above, the size measurement is a product metric and it is a very important metrics that can directly affect the software cost as well as test effort. The metric for measuring size is SLOC, function points and feature points. The software size has been dealt with above. See the sections 3.1 and 3.2 under software size above.

Requirements [25]

Software requirements is in my view an important topic to be discussed at this stage because information from the software requirement specification document forms the main criteria or guidelines under which we must work on a particular job as software engineers. In this thesis, I am going to be suggesting shortly, a step by step description on how to perform test effort evaluation and when we begin that, we should have a set of requirements to analyse and actually
know the number of these requirements so that we can multiply them by a factor when solving complexity issues from the software size as we will see shortly. This calculation of effort was given in the report titled: “Test Execution Effort and Capacity Estimation by Eduardo Aranha and Paulo Borba” [2]

There are two kinds of requirements according to the report: “Software Estimation, Measurement, and Metrics GSAM version 3. Chapter 13” [25], these are the user-specified (or explicit) requirements and the derived (or implicit) requirements necessary to translate requirements into code. Converting the explicit requirements to implicit ones can mean a lot of work to be done before the design stage but are needed (technically) instead of mere documented words from or given by the customer which may be ambiguous. But each implicit requirement should be traced back to the explicit requirement to be used in test planning. And during testing, missing requirements must be detected and reported but of course fishing out problems or incompleteness is the reason we carry out testing.

**Complexity [25]**

Software complexity is a very relevant metric to our research area of software test effort and later in the preceding sections complexity calculation is done in practical in the test effort estimation section. Complexity is a measure of size in software development and the earlier we compute the software complexity in the development cycle, the better we can manage the software process [31].

According to my research, complexity measurement is directly related to the software design and the code and this is why there is a link or relationship between design complexity and design errors [25] the more complex a system the higher the probability of errors and the more the number of errors. There is also a link between code complexity and latent defects [25]. And by studying factors or properties that contribute to complexity, high-risk applications can be identified and where necessary revised or tested more thoroughly.

As mentioned above size is a major contributor to complexity other contributors are:

The number of interfaces among the modules (fan-in and fan-out)

Fan-in: How many modules are invoking a particular application [25].

Fan-out: How many modules are invoked by a given application [25].

Another factor is the structure of the module meaning the number of paths within that module.

Estimating the complexity can help us determine the amount of test needed to be done in other to adequately cover the interfaces or calls (the design) and the statements or branches (coded logic).

Complexity is calculated in this thesis, in execution points. The more the complexity of a test case, the higher should be the execution points [44]. And this is one of the areas that the traditional method of software development effort estimation does not cover. In the traditional method, complexity is factored into the function points analysis as seen in page 15 “function points” under section 3.2. But in our proposed framework, complexity is the main approach to determining size and therefore effort. This is the main contribution of this research work [2].

**Effort [25] [2] [26] [27] [28]**

Effort has been discussed in many parts of this thesis especially on chapters two (2) and three (3) where I give effort formula as a function of the software size and the cost factors. Our main
goal in this thesis is to find effort. For our example, later in the next session, effort will be deduced from the size and complexity using the tasks and requirements in the software to be tested as already stated in the previous sessions and as suggested in the report by Eduardo Aranha and Paulo Borba. [2]

However, and in general, effort is calculated by many estimation models, the best known being COCOMO by Barry Boehm [25] [26] [27] [28]. But these models are not suited for testing purpose that is why we will not be using COCOMO. These terms and models have already been discussed above in the chapter 3 “Traditional software development effort estimation methods/metrics”

Cost and Schedule

As mentioned before size is directly related to cost and schedule [25] and poor schedule estimates can lead to delay in delivery and subsequently cost both in testing and in overall software development. As stated in the report “software estimation, measurement and metrics” other operational environment and qualitative issues (the software’s characteristics) are issues of complexity which also affects cost and schedule [25] and are also used to calculate effort and cost by many estimation models including in this report, as we will see in the next chapter.
Chapter 5  
A Step by Step Approach to  
Test Effort Evaluation

There are no specific methods for calculating test effort similar to what you have in software development [37], my proposed approach should not get into the same level of details as in other software development effort estimation models like COCOMO. However, a generic approach for performing test effort evaluation means that it will be a “one size fits all” if you like, or as broad and common approach as it can be and this is why I break down this process into steps as shown in the figs 5.1 and 5.2 charts below.

5.1) A look into “efforts” to evaluate

As part of my contribution to the research work, going through parts (a) to (d) below, the reader can see a more detailed analyses of the steps or tasks involved in the estimation process. More practical application of these steps to real life example(s) follows further down after these sections.

| Evaluate Effort of the “Test Preparation phase”  
| • Calculate Effort for the test planning, strategy, staffing etc.  
| Evaluate Effort of the “Test Development and Test Execution phases”  
| Calculation of Effort of the following activities:  
| • Specific test case plan and design, structured steps to follow, test to be run etc 47.  
| • Decomposition of Requirements[37]  
| • Develop Test Cases [37,19]  
| • Execute Test Cases according to Specification[2]  

Fig 5.1 Diagram showing Efforts to be analysed and their activities: Here, I have combined the test development and execution into one phase.
c) Further information of the steps

**STEP 1: Test Preparation phase**

- Read test specification document
- Planning (including strategy, test scope, technology to utilise etc [47].)
- Staffing (if applicable)[32]
- Test environment acquisition and configuration[32]
- ....
Considerations:

- The test environment acquisition/configuration efforts are subjective, based upon nature of project, as the test environment of a web-based project may be different from that of a mobile application [2] [7].
- The same above could also be true of the planning and staffing effort, but also this depends on number of staffs handling or working on this step and of course the staff/team's competence and dexterity.
- Effort is calculated in man-hour (for this research) and should reflect the total time the engineer(s) or team completes this process or step.

STEP 2: Test development and Execution phases

- Analysis and decomposition of requirements [37]
- Test Categorisation
- Writing test scripts
- Executing Test scripts (including complexity and execution points evaluation)
- ...

Considerations:

- The test development phase time can vary according to the size of the software to be tested and risk factors, the more safety critical systems require more elaborate test scripts.
- Writing test scripts that will cover all the requirements can among other factors depend upon the test team experience on similar projects and on their dexterity and so many other factors in my view.
- I presume that the execution time is a performance issue either due to the machine capability or the method used in the software to be tested (e.g. the use of inefficient method in the software can affect its execution time negatively).
- Test case execution time can vary depending upon machine speed and load (since testing in an environment where the system is loaded with several simultaneous users or transactions can affect performance).

STEP 3: Test Productivity

- Execute several test cases and record time spent in executing them
- Use this time to evaluate test productivity and test execution effort[44]

Considerations:

- The test execution effort depends on several factors apart from simply the complexity of the test case.
d) Steps Flow

Fig 5.3: A generic test effort evaluation process suggestion [44].
According to my research there is no one simple formula or method for computing effort, I do believe that the best approach would be to employ a variety of methods from the ones we have touched earlier in this report. In the dummy example below, effort is computed from different parameters from the total time spent on different activities such as test preparation, to functionality decomposition and even complexity.

5.2) A Dummy Test Effort Evaluation Project

Suppose we have a project to be tested and effort calculated with result displayed after testing is complete. Let’s call this project “Harness”. Project Harness could be any web based application and the same effort principle could be adopted in other projects of different application. For our research purpose, the effort should be quantified in man hours or (Mh). So the concept of man hours (Mh) should be adopted and used for the remaining parts of this thesis. Using the step by step approach proposed above, for this project, we then begin the series of steps taken to acquire the effort figures starting with the test preparation phase as below:

**Project**

Project name: Harness

Project type: Web application

Project Description: To carry out test effort estimation for Harness using our generic approach.

**Solution**

**Step 1: Test Preparation**

1. Management and/or test specialists select staff members who will form members of the test team. This event is called staffing (as stated above). Time utilized for this event is recorded, and for the purpose of this research let’s say five man hours or 5 Mh.

2. Test team(s) attend first meeting and carry out detailed deliberations of the project at hand and look at its explicit requirements. Other activities of this event include looking at previous similar projects, appointing roles, test environment analyses, etc. I call these activities “planning” and it is expected (I believe) to last more than one meeting and should take hours or days but should be computed in hours, again for our research purpose. A test plan is drafted and submitted to management at the end of this event. Time is recorded say 50 Mh.

3. The test environment, tools etc is set up and time recorded say 30 Mh.

**Step 2a: Test Development**

1. Break the functionalities into requirements maturity levels i.e. based upon how much requirements decomposition has occurred [40]. We can have up to 5 levels (1-5) with level 5
representing the greatest detail possible of a functionality and level 1, the least. Record the time spent. This functionality decomposition can enable us to work in a top-down fashion. The table below is a way this could be done.

<table>
<thead>
<tr>
<th>Level</th>
<th>Explanation</th>
<th>Examples</th>
</tr>
</thead>
</table>
| 1     | At level 1, major parts or features of component or system are specified (from the test specification document). In other words, at level 1, all the major functionalities or features of the system are known and listed but details and constraints that could arise when testing these functionalities are yet unknown. As we work from level 1 to 5, greater details can be revealed. | Examples of major features in a web-based system:  
  - The system should handle Flights booking online  
  - Users should be able to make payments online.  
  - The system must be secure  
  - Shall have Email capability  
  - Users should be able to carry out file operations  
  - Etc.                                                                                                                                                                                                                      |
| 2     | At level 2, the major tasks in each feature or functionality of the application to be tested are allocated to the tester. Actual testing is now ready to begin.                                                                                                                                                                                                 | Examples of these tasks:  
  - Book-Flight  
  - Un-book-Flight  
  - Pay-By –Credit-Card  
  - Pay-By-Invoice  
  - Upload-Files  
  - Download-Files  
  - Send Email  
  - Receive Email                                                                                                                                                                                                                                                                     |
| 3     | A feature is selected and the interface established, the tester can begin to navigate the interface of this feature. This operation is observed for stability. For instance for the task “Book-Flight”:                                                                                                                                                                 | - The tester books a new flight by selecting “book flight”  
  - This operation should be done in the main window  
  - The tester populates fields with origin, destination of travel, to/return trip, time and date, number of travellers,                                                                                                                                                                     |
| 4 | At this level, information from levels 1 to 3 is already known. |
|   | The error/exceptions handling are encountered |
|   | The nature of data needed is known |
| **Examples of errors:** |
|   | “No destination selected” |
|   | “Enter a valid address” |
|   | “Invalid delimiter entered” |
|   | “The use of the back button is not allowed at this stage” |
|   | “Please wait while information is being processed” |

| 5 | Information of levels 1 to 4 are known |
|   | In this level constraints are known e.g: |
|   | “Only one payment information page can be active at one time” |
|   | “Only one flight booking entry page is allowed” |
|   | The input/test data is fully ready. |
| **Test input is fully identified** |
| **The total of the above examples represent level 5 information.** |

**Fig 5.4: Functionality Decomposition (Top-Down approach)**

The goal of fig 5.4 above is to sufficiently decompose a giving requirement to testable level so that the requirement can be validated. The level at which you identify a capability can have effect on its accuracy i.e. for instance at level 1, levels 2 to 5 information is yet unknown and may be
speculative at this point which is why estimates at higher level such as level 1 are more likely to produce inaccurate figures than at lower level like level 4 [40].

The next step now is to determine the average test script development time based on requirement maturity levels and scheduled aggressiveness factor (the factor used to determine amount of time to be allocated for a testable script based on requirements) [40], if you want to proceed on this method to derive the amount of time allocated to developing manual test script on a per-requirement basis [40]. But unfortunately I was unable to find more detailed information about this as at the time of writing this report.

The total time used for the above functionality decomposition will give an estimated time spent for the test development phase. The next phase is the test execution where complexity level plays a major role in determining test effort.

**Step 2b: Test Execution**

As suggested before, the execution complexity of test cases can be used to ascertain what effort is required in executing them and then can be useful for planning purposes [44]. This step presents the measuring of execution complexity of functional test cases based on a controlled natural language structure or the CNL [44].

The CNL is a kind of specification language inspired by the *test case specification* as defined in chapter 1 as “a document or language specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item” [1]. The CNL can be said to be a language because it is usually written in a natural language. But in the case of the test case specification, redundancy and ambiguity problems often arise because of the lack of writing standards and this can affect clarity, not only in test execution but also in its complexity estimation [44]. The above problems can be avoided when this language is controlled. “A controlled natural language (CNL) is a subset of a natural language with restricted grammar and lexicon, in order to have sentences written in a more concise and standard way” [44].

In the language, a test step or sentence is identified by the structure: a verb and its argument. The verb represents the action to be performed while the argument is more info about this action.

As an example, in our airline booking web application above, the phrase “launch file operation” has the verb launch being the action to be performed and the argument “file operation” indicates what operation or application to be launched. The list of lexicon and grammar for web application may be slightly different from other application such as the mobile application however they do follow the same pattern such that our web application will be a good example of how step 2b can be performed in general.

**Execution Complexity**

Now we look into the execution complexity of a test case and this involves three main aspects:

1. Evaluate complexity level of the test step based on the functional and non-functional characteristics it exercises [44].
2. Express the execution complexity level in execution points which is a quantitative measure of execution complexity [44].

3. Sum up all the execution points of each test step to get the execution complexity of the test case [44]. This is shown in the diagram 5.5 below.

Before I proceed, all the method I am employing for this test execution phase including the complexity evaluation is derived from the article titled “measuring test execution complexity” by Eduardo Aranha and Paulo Borba [44].

Now we evaluate the execution complexity for each test case by analysing each test step defined in the specification which in other words means analysing the functional and non-functional characteristics of the tested system being exercised by the test step [44].
And using intuitive and expert judgement, the above diagram will be supported by another diagram (fig 5.6) to show how to get execution complexity from some of the functional characteristics of the system to be tested.

<table>
<thead>
<tr>
<th>Functional Characteristics</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td># of navigations</td>
<td>&lt;5</td>
<td>5-15</td>
<td>&gt;15</td>
</tr>
<tr>
<td>Sum of the size of each input</td>
<td>&lt;20</td>
<td>20-40</td>
<td>&gt;40</td>
</tr>
<tr>
<td># of items to verify</td>
<td>&lt;3</td>
<td>3-8</td>
<td>&gt;8</td>
</tr>
</tbody>
</table>

Fig 5.6: Rules for defining complexity levels from functional characteristics [44].

In the above diagram, for a particular characteristic, there is a score of low, average or high that indicates the complexity level of the test step.

Again using intuitive and expert judgement, we repeat this rule for the non-functional characteristics shown here in fig 5.7 below. In that diagram we can see that the test step of opening or saving of small files has an average complexity associated in terms of the system “performance” than the opening or saving of very large files since you have to scroll, attach files and other file operations. Similarly the playing of short clips has a shorter waiting time than that of a large clip. Similarly all test steps should have complexity levels associated with them.

<table>
<thead>
<tr>
<th>Non-Functional Characteristics</th>
<th>Low Complexity</th>
<th>Average Complexity</th>
<th>High Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERFORMANCE</td>
<td>No effect</td>
<td>Opening and saving of small picture files, playing short audio clips...</td>
<td>Playing long audio or video clips, making large graphic designs....</td>
</tr>
<tr>
<td>USE OF NETWORK</td>
<td>No effect</td>
<td>Sending/receiving small data...</td>
<td>Making VOIP or video calls, upload/download (sending and receiving) of large multimedia data ...</td>
</tr>
</tbody>
</table>

Fig 5.7: Rules for defining complexity levels for non-functional characteristics [44].
Quantitative Rating of Complexity Levels

After the evaluation of complexity levels for all characteristics of the test steps, we rate them into quantitative values called execution points which is a generic term used to rate test execution complexity and this value can then be used to compare the complexity of different test cases as shown in the diagrams below [44].

We begin by assigning weights using intuitive knowledge and expert judgment. These weights reflect the impact of each characteristic on the test execution complexity. Now, the weight to assign for each test step depends upon the nature of the project under testing and the experience from previous similar projects and the experience gained from the project at hand can be handy when testing other future projects as well [44].

We will later compare the actual values (i.e. the values from actual testing) of complexity with the assigned arbitrary rating and use the later to calculate the total execution points as we will see in the diagrams 5.8 to 5.11 below. Table 5.8 describes the arbitrary values or weights assigned to each complexity level of functional characteristics. For instance if we have a test step with functional characteristic “# of navigations” with an average level of complexity, this contributes with the value of 10 execution points out of all execution complexity of functional characteristics [44].

<table>
<thead>
<tr>
<th>Functional Characteristics</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Navigations</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Sum of the size of each input</td>
<td>35</td>
<td>80</td>
<td>120</td>
</tr>
<tr>
<td># of items to verify</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>

Fig 5.8 Weights assigned to complexity levels of functional characteristic [44].

<table>
<thead>
<tr>
<th>Non-Functional Characteristics</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Performance</td>
<td>0</td>
<td>50</td>
<td>150</td>
</tr>
<tr>
<td>Use Of Network</td>
<td>0</td>
<td>80</td>
<td>200</td>
</tr>
</tbody>
</table>

Fig 5.9 Weights assigned to complexity levels of non-functional characteristic [44].
Similarly for the non-functional characteristics, if we have a test step that uses the network with a high complexity level, this will impact the total execution complexity with 200 execution points. We continue by rating the complexity level for all functional and non-functional characteristics into execution points as shown in the table 5.10 below. The execution complexity is then determined by the total execution points rating that each test step contributes [44].

<table>
<thead>
<tr>
<th>Functional /non-functional Characteristic</th>
<th>Actual Value</th>
<th>Complexity Level</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Navigations</td>
<td>3</td>
<td>Low</td>
<td>5</td>
</tr>
<tr>
<td>Sum of size of each input</td>
<td>60</td>
<td>Average</td>
<td>80</td>
</tr>
<tr>
<td># of items to verify</td>
<td>2</td>
<td>Low</td>
<td>10</td>
</tr>
<tr>
<td>System Performance</td>
<td>No Influence</td>
<td>Low</td>
<td>0</td>
</tr>
<tr>
<td>Use of Network</td>
<td>Sending a small message</td>
<td>Average</td>
<td>80</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>175</td>
</tr>
</tbody>
</table>

Fig 5.10 Summary of execution complexity and ratings of execution points of test steps [44].

<table>
<thead>
<tr>
<th>Test Step</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch “book-flight” application</td>
<td>15</td>
</tr>
<tr>
<td>Populate fields and search flights</td>
<td>20</td>
</tr>
<tr>
<td>Select flights</td>
<td>10</td>
</tr>
<tr>
<td>Add personal information</td>
<td>20</td>
</tr>
<tr>
<td>Add payment information</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Cancel Booking</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>150</td>
</tr>
</tbody>
</table>

Fig 5.11 The execution complexity of a test case [44].
**Test Execution complexity, productivity and execution effort Summary**

Using the method above we can now add up all execution points to get the execution complexity of the test case or project at hand. In table 5.11 we can see that using our earlier sample project of a flight booking website, the execution complexity of this sample test case is 150 execution points. With the rules of ranges used in the complexity evaluation (fig 5.6 & fig 5.7) small changes in the test specification should **not** affect or change the execution complexity of a test case and the more complex test cases should have more number of execution points than the less complex ones meaning that the execution points of a test case is a reference about how complex the test case is [44].

The execution complexity is very useful if you want to compare two test cases for instance a test case rated with execution points of 1000 is 25% and 50% more complex than those rated with 750 and 500 execution points respectively hence should take more time to be executed than they.

**Test productivity**

“Productivity is a measure of output from a production process, per unit of input. For example, labor productivity is typically measured as a ratio of output per labor-hour, an input.” [8]

Productivity is especially useful in revealing the required effort in executing test cases because testers could execute several test cases comparing the proportionality between the amounts of time it takes to execute a test case in relation to its execution points [44]. This information can then be used to compute the test productivity, meaning that for instance, if after executing several test cases, and we found out that it took 1 minute to execute a test case of say, 90 execution points; this means that the test productivity in this case is 90 execution points per minute. And therefore a new test case with 135 execution points can then be said to require 1.5 minutes to be executed i.e. 135/90. This is the meaning of test productivity [44].

However, the test execution effort estimation is more complex than this. Environmental factors, team experience, tools utilised, and other numerous factors must also be considered therefore this research is only a suggestion of a possible way to estimate and by no means exhaustive.
Chapter 6

Conclusion and future work

In this thesis report the approach to the subject of test effort evaluation has been two fold which are evaluating the effort of the test preparation and test development/execution phases respectively. At the end of the second step (i.e. evaluating the effort of the test development/execution phase), the test team will be able to deduce the execution points of the test case which is a quantitative measure of its execution complexity, hence a measure of the required effort to execute the test case. After testing is done, the actual effort required should be comparable to the estimated effort. Two techniques have been suggested for this second step both of them in chapter 5. The first is that of requirement decomposition and the other is about evaluating the test execution complexity as mentioned above. Both methods suggest ways of carrying out testing of test cases in such a pattern that the effort can be estimated before testing. To estimate effort after testing is done, the organisation must still calculate the execution complexity as proposed in this report, to be able to see how large and complex the test case is, in order to allocate the appropriate resources for the project.

Using a dummy project “Harness” which is a case near a real life application, I tried to show how they can be applied using real life terms.

The term requirements decomposition means that requirements are broken down and decomposed into maturity levels (which signify how much decomposition of requirements has occurred). The maturity levels are 1 to 5 with level 5 representing the greatest detail possible of functionality and level 1 representing the least.

This method enables us to work in a top down fashion for instance we saw that in our flight booking website example, the requirement: “The system should handle flights booking online” was at level 1. At this level we only have high level definition of requirements and no level of details are known but when we move downwards we begin to see at level 2, functions like “Book-Flight”, “Unbook-Flight” from the breakdown of the above high level definition and also functions like “Pay-By-Invoice”, “Pay-By-Credit-Card” from the breakdown of the requirement: “users should be able to make payments online” of level 1.

Working further down to level 5 the most detailed level and the greatest decomposition of all the functionality we could see the exceptions and error handling mechanism and constraints more clearly and at this stage the test input is ready and we can realise that estimates based upon this level is the most accurate since that all parts of the system is peer into than estimates based upon say at level 3 which is the stage prior to revealing of these errors and exceptions.

According to the article I read about this method, the next stage would have been to determine the average test script development time based on those requirement maturity levels and another factor called “the scheduled aggressiveness factor” which is the factor used to determine the amount of time to be allocated for a testable script based on requirements [40]. Continuing on this line we should be able to derive the amount of time allocated to developing manual test script on a per-requirement basis [40]. But unfortunately I was unable to find neither the
materials nor more detailed and sufficient information about this as at the time of writing this report.

After the requirements decomposition, we then measure the execution complexity of test cases based upon a controlled natural language or CNL. The CNL is a kind of specification language that has the structure of verb and argument. So a sentence like “launch the message application” is in CNL format with the verb being “launch” and the argument is the remaining part of the sentence. Based on this language we can evaluate the complexity level of a test step, execution points, execution complexity, test productivity and possibly the test execution effort.

The complexity level of a test step is a measure of the functional and non-functional characteristics it exercises. For instance the non-functional characteristic of “performance” takes an average toll on complexity level of the system when playing short audio clips but a high complexity level when playing long audio clips.

In the same vein, if the “# of navigation” is less than 5 this is a low complexity level but 5-15 and >15 are average and high complexity levels respectively exhibited by that functional characteristic. This execution complexity is then further expressed into execution points, which is a quantitative measurement of the complexity level of each test step. Sum up all the execution points and you get the execution complexity of the entire test case.

The rules used in getting or rating these complexity levels are by intuition and expert judgement and using the rule we can see that in our example test case of a flight booking system, summing up all the ratings of test steps we derived a figure of 150 execution points in chapter 5. With this rule, small changes to the test case specification should not seriously affect this execution complexity figure; this cushioning effect is inbuilt into the rules of ranges for defining these execution points. Since the execution points is a measure of how complex the test case is, therefore more complex test cases should have larger figures than less complex or smaller ones.

The execution complexity is very useful if you want to compare two test cases for instance a test case rated with execution points of 1000 is 25% more complex than those rated with 750, and also 50% more complex than those rated with 500 execution points respectively hence should take more time to be executed. This is important because it can reveal the required effort in executing test cases because testers could execute several test cases comparing the amounts of time it takes to execute a test case in relation to its execution points. This information can then be used to compute the test productivity and testers can estimate a productivity of say 85 execution points per minute therefore a new test case with 212.5 execution points can then be said to require 2.5 minutes to be executed [44]. However, further research needs to be done on the subject of test effort evaluation to get a more complete solution because the test execution effort estimation is more complex process than we have stated.
For future work, if there is a way that issues such as environmental factors, team experience, tools utilised, and other variables could be analysed, graded and factored into the overall effort calculations, it can give a more accurate framework. This can be very difficult to achieve because, from experience gained from this thesis, I have seen that these factors vary from person to person and from project to project. This is true irrespective of whether we are evaluating test effort before or after testing, because all these variables are always part of testing. This research only calculated effort mainly based upon hypothetical requirements breakdown and complexity calculations, and did not take the above factors into consideration in the calculation. Therefore, more work can be done by the would-be researcher, to look into these variables as a future work.

Also as future work, the steps discussed in this report such as test preparation phase, could be better standardised by recording in a tabular form, all variables such as, testers experience level: (low, average or high). And testers’ qualification level (expert, intermediary or beginner), etc, can be graded and recorded. Even up to the execution phase; progress can be monitored such as machine performance, major test steps and test levels covered, etc. In this tabular modification, the number of execution points can also be entered in the appropriate field. This can lead to a more organised framework and to a more standard way to record information, helpful for test effort evaluation, especially after testing is done.
REFERENCES


[8] Wikipedia online dictionary


[27]  http://www.bth.se/fou/cuppsats.nsf/all/1ce4270c92a2ca97c12575470045b588/$file
/DV2403_Master_Thesis_Sohaib_Shahid_Bajwa.pdf  (Last visited 2011-08-11)

[28]  ranger.uta.edu/~khalili/Somerville%20Chapter%2026.ppt  (Last visited 2010-04-10)


Using%20Use%20Case%20Points.pdf  (Last visited 2011-08-11)


(Last visited 2011-08-11)