Developing A Software Estimating Information System

By

Boniface C. Nwugwo

December 2000
Abstract

Effective software project estimation is one of the most challenging and important activities in software project management. Proper project planning and control is not possible without a sound and reliable estimate for size and effort. The software industry as a whole, does not estimate projects well and does not use estimates appropriately, despite the existence of more than a dozen software estimating models and techniques. Software engineering practitioners continue to show apathy in the usage of software estimating tools, causing growing concerns in the software-engineering community. My organization is not exempt from this malady, and that is why we have embarked on a concerted effort to build an estimating tool that practitioners can use to estimate software size and effort consistently and more accurately. This paper examines why software estimating is important and provides an insight into the efforts that went into developing an information management system for estimating and tracking actual software project size in a medium-sized, software development center of excellence. Using the organization’s historical data as a benchmark, the system was designed to provide an ability to calculate software size and effort estimates given the type of project and its complexity as parameters. It is envisaged that the database would continue to grow, providing more historical data for more accurate estimation.
# TABLE OF CONTENTS

Abstract

Introduction

Definition of Terms

History of Software Cost Estimation

The Software Estimation Process

WDC Estimation Information Management System

## Abstract

Introduction

Definition of Terms

History of Software Cost Estimation

The Software Estimation Process

WDC Estimation Information Management System

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
Introduction

Lord Kelvin has been quoted to have said in his popular lectures and addresses of 1889 that, “when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind” (Pressman, 1987, p.89). For the past decade, the software engineering community has begun to take Lord Kelvin’s words seriously. Today, software plays an important role in our lives. We want products that affect our lives to have good quality attributes. In order to determine quality of software we must have some metrics to measure quality. If we can measure the quality attributes of software, we can also have more precise, predictable and repeatable control over the software development processes. And with only a small understanding of the desired software system, estimations of costs begin.

Some years ago, software costs represented a small percentage of the overall cost of a computer-based system. As a result, even a sizable error in estimates of software cost had relatively little impact. However, in today’s business environment, software is now the most expensive element in many computer-based systems. Therefore, a large cost-estimation error can make the difference between profit and loss. Software estimation is one of the fundamental factors in the successful management of software development projects. If an organization is not able to estimate how "much" software to build, then it has little hope of correctly estimating when it will be ready, or how much it will cost. The risk is that the organization will under-bid or over-bid, and not deliver on time.

If we want to estimate cost and effort of a software project, we need to know the size of the software. One of the first software metrics to measure the size of the software, is the SLOC (Source Lines of Code). The SLOC measure is used extensively in the COCOMO and COCOMO II cost estimation models. Another size metric is Function Points (FP) which reflects the user's view of a system's functionality and gives size as functionality. A unit (the function point) expresses the amount of information processing that an application offers the user. The unit is separate from the way in which the information processing is carried out technically.
This paper discusses the history of software size estimation, and the estimation process comparing and contrasting the leading estimation models (COCOMO II and SLIM). In addition, a description of a size-estimating information system (SizeEST) for a medium-sized software development organization is presented. Because software cost estimating is a very touchy subject for most people, it’s hard to always agree on what project size for example, means. For that reason, a definition of some of the terms that will be used in this paper is provided below.
Definition of Terms

Effort

Effort is the time required for accomplishing all software engineering activities (analysis, design, code and test) to deliver a project for end-user availability. It’s often measured in person-months, but could also be expressed in units of hours, days, weeks, or years. Effort estimation is one of the most common techniques for costing any engineering development project. For that reason, many effort estimation models have been developed in the past couple of decades (Albrecht & J.E. Gaffney, 1983; Bailey & Basili, 1981; Basili & Freburger, 1981; Boehm, 1981; Conte, Dunsmore, & Shen, 1986; Kemerer, 1987; Matson, Barrett, & Mellichamp, 1994, to name a few; Putnam, 1978; Walston & Felix, 1977), to name a few. In most cases, effort estimation models consist of two phases, an estimation of the software size and estimation of the effort and duration of the project. Because many factors besides software size influence the effort and time required to develop software, most effort estimation models adjust for a number of productivity factors (Kitchenham, 1992).

Function Points (FP)

Other than source lines of code (SLOC), the only other major software size measure is function points. Function points (FP) is defined to mean those pieces of code that perform some specific activity related to inputs, inquiries, outputs, master files, and external system interfaces (FAA, 1999). Function points are derived using an empirical relationship based on countable measures of software’s information domain and subjective assessments of software complexity. Originally designed for business information systems applications, “the function point measure may not be relevant to control-oriented or embedded applications in the engineered products and systems domain” (Pressman, 1987, p.92). Models based on function points focus more on the sizing stage of software project planning while others such as COCOMO focus more on the productivity stage of the software project development (Heemstra, 1992).
Software Size

Software size has been defined as a measure that provides an indication of the amount of work to be done and the amount of resources needed to do the work (Druyun, 1994). It refers to a quantifiable outcome of the software project. Since other physical objects are easily measurable, it might be assumed that measuring the size of software products should be straightforward. However, in practice, size measurement can be difficult.

Simple measures of software size are often rejected because they do not provide adequate information. It might therefore, be prudent to define size in terms of multiple attributes. For example, if human size is measured by weight, then it can be determined how many people can safely ride in an elevator. But what if we would also like to know whether the passengers will bump their heads on the elevator door? Then we could measure human size in terms of weight and height, which would then allow us to determine the number of people that can safely ride in an elevator without bumping their heads on the elevator door.

Similarly, software size can be described with four attributes: length, functionality, complexity, and reuse. Size is primarily a cost factor in most models and can be measured using SLOC or function points (FP). However, the most commonly used measure of source code program length is the number of lines of code (Fenton & Pfleeger, 1997). For most people, SLOC is the most useful size measure. The reason being that it is precise, machine countable, environment specific, and can reflect the mix of reused, modified, and new code. The principal disadvantage is that SLOC is hard to visualize early in a project.

It is a well-known fact that software size and development time are directly related. Thus, if one knew the size of a planned program, and given historical productivity data, if the right methods are used, a pretty good estimate of the time to develop the program can be made. This of course, assumes that an appropriate size measure was used.
Source Lines of Code (SLOC)

Source lines of code (SLOC) is defined to mean all executable source code statements including deliverable job control language (JCL) statements, data declarations, data typing statements, equivalence statements, and input/output format statements (FAA, 1999). SLOC can be commented (CSLOC) or non-commented (NCSLOC). Any statement that upon its removal, does not cause the program not to compile, (e.g. comments, blank lines, and non-delivered programmer debug statements), is not included in the SLOC.

The most general approach to size measurement is to count the number of text lines in a source program. Typically, this means ignoring blank lines and lines with only comments, and counting all other text lines. The advantage to this approach is that it is simple and easy to automate. The disadvantage is that the SLOC approach is sensitive to formatting. Those programmers who write very open code will get more SLOC for the same program than would their peers who used more condensed formats. As a consequence, establishing coding and counting standards to cover program formatting is crucial to obtain consistent size measure.
History of Software Cost Estimation

Chronicle of Software Size Estimation

Software parametric cost models were virtually unknown until the 1960s, when RCA PRICE Systems released their proprietary PRICE Software (PRICE-S) model for general use on a time-sharing service. In the 1960s, while at RCA, Frank Freiman developed the concept of parametric estimating, which led to the development of the PRICE model for hardware. The PRICE model was the first generally available computerized estimating tool and was later extended to handle software in the 1970s. In 1977, Frank Freiman and Dr. Robert Park developed PRICE-S. It was the first commercially available detailed parametric software cost model to be extensively marketed and used (Brundick, 1995). In 1987, the model was modified and re-validated for modern software development practices.

Unfortunately, the early models of the 1970s have some shortcomings. One of the shortcomings is that the independent variables are often "result measures" such as the size in lines of code. Such values can be measured easily, but only at the end of the project. It is very difficult to predict the values of such variables when you truly need them, before the start of the project. This resulted in massive reluctance by people to use the models despite that they were based on statistical analyses of actual result data. Another complaint of such models is that they assume that software will be developed using the same process as was used previously (Pressman, 1993). Today, this assumption is becoming increasingly unrealistic.

At the end of the 1970s, (Albrecht & J.E. Gaffney, 1983) developed function point analysis (FPA) to estimate the size and development effort for management information systems. Later in the 1970's, two authors endeavored to define models based on theoretical grounds. (Putnam, 1978) based his Software Lifecycle Model (SLIM) on the Norden-Rayleigh curve and empirical results from fifty United States Army projects. Although still in use, this model is not generally believed to be especially accurate by authors such as S. Conte (Conte et al., 1986). The other author, (Halstead, 1977) defined software size in terms of the number of operators and operands that appear in a program and proposed relations to estimate the
developing time and effort. To obtain this size information before the start of a project was nearly impossible because a good understanding of the detailed design is not available until later. Subsequent work by (Conte et al., 1986) has shown that Halstead's relations are based on limited data, and Halstead's model is no longer used for estimation purposes. However, (Coleman, Ash, Lowther, & Oman, 1994) have recently reported some success in using it to predict the "maintainability" of software.

Following the progress of the 1970s and the introduction of personal computers (PCs) in the 1980s, many models were programmed. Several firms began to sell computerized estimating tools. Following the publication of the COCOMO equations in 1981, several tools that implemented COCOMO appeared during the latter half of the 1980s. After the U.S. Department of Defense (DoD) introduced the Ada programming language in 1983 to reduce the costs of developing large systems, (Boehm & Royce, 1988) defined a revised model called Ada COCOMO. This model also addressed the fact that systems were being built incrementally in an effort to handle the inevitable changes in requirements.

Later, (Tausworthe, 1981) extended the work of Boehm, Herd, Putnam, Walston and Felix, and Wolverton to develop a cost model for NASA's Jet Propulsion Laboratory. Tausworthe's model was further extended by Donald Reifer to produce the PC-based SOFTCOST-R model, which is now sold by Resource Calculations Inc. By mid 1980s, (Jensen, 1984) extended the work of Putnam by eliminating some of the undesirable behavior of Putnam's SLIM. The SLIM equation has development effort proportional to size (in source lines of code) cubed divided by development time to the fourth power. (Jensen, 1984) asserted that development effort is proportional to the square of the size divided by the square of the development time.

In the 1990s there was evidence of renewed attention on developing improved cost models. The Software Engineering Institute (SEI) started a small initiative on software estimation improvement in 1994. Of more interest is an effort by (Boehm, Clark, Horowitz, Madachy et al., 1995) at the University of Southern California, to revise and extend the COCOMO model, which has served the industry well for more than fifteen years. The new version of COCOMO called COCOMO II, which has been released, addresses
Developing a software estimating information system

the various types of development problems that plagued the early models. The COCOMO II model will be presented briefly later in this paper.

**Some of the Commercially Available Software Estimation Tools**

It has been said that good software metrics are a good thing. Trying to manage a project without good metrics is like trying to lose weight without ever getting on a scale. Sure you may do well, or you may do poorly, but there is no way for you to really know. In 1979, a report by the Government Accounting Office (GAO) concluded that the software engineering industry had a poor track record regarding predictions. The GAO report further concluded that only two percent of software contracted for was useable exactly as delivered (Brundick, 1995). Twenty different studies also had come to the same conclusion. Perhaps that was what motivated people many years ago to develop models and tools for software cost estimation. Some of the early commercial tools for estimating cost include PRICE-S, REVIC, SASET, SEER-SEM, SLIM, SoftCost-R and SoftCost-Ada.

Although PRICE-S model is proprietary, it can be leased for yearly use on IBM or compatible PC, and operates within Microsoft Windows. It is also available for use on a UNIX workstation. The model is applicable to all types of software projects, and considers all DoD-STD-2167A development phases.

Another one of these early models is the Revised Intermediate COCOMO (REVIC) developed by Ray Kile and the U.S. Air Force Cost Analysis Agency. It is a copyrighted program that runs under DOS on an IBM PC or compatible computer. The model predicts the development costs for software development from requirements analysis through completion of the software acceptance testing and maintenance costs for fifteen years. REVIC software cost model, based on the Intermediate COCOMO, is used for predicting software development life-cycle costs (Cost & Schedule), maintenance life-cycle costs, and staffing levels. REVIC uses the Intermediate COCOMO set of equations for calculating the effort (man-power in staff-months and staff-hours) and schedule (elapsed time in calendar months) to complete software development
Another model developed in the 1980s is a sophisticated software cost-estimating model called Software Architecture Sizing and Estimating Tool (SASET). Martin Marietta Astronautics built the model during 1986-1990 under contract with the Naval Center for Cost Analysis under the auspices of the Office of Naval research. The model combines database information from over 500 successfully completed software projects with "expert" factors to determine software development and maintenance costs, schedules, and sizing numbers used by engineers, estimators, and top-level management (Brundick, 1995).

System Evaluation and Estimation of Resources Software Estimation Model (SEER-SEM) is another one of the commercially available tools for software cost estimation. Developed and marketed by Galorath Associates, Inc. in Los Angeles California, it is part of a family of software and hardware cost, schedule and risk estimation tools. SEER-SEM is a software development and maintenance analysis Computer Aided Software Engineering (CASE) tool that estimates software development and maintenance cost, effort, schedule, staffing, reliability and risk. It includes a knowledge base of algorithms to aid the analyst in producing concept level estimates (Grehan, 1994). The primary purpose of this tool is cost estimating. Sponsored by the Aeronautical Systems Center (ASC)/FME, Air Force Materiel Command (AFMC), SEER-SEM is currently used throughout the Department of Defense.

SLIM for Windows 3.1, developed by Quantitative Software Management, Inc. (QSM) and based on the book by (Putnam & Meyers, 1992). The SLIM tool is based on QSM's Software Equation, which was derived from the Rayleigh-Norden model and has been validated over a fifteen year time period with thousands of real, completed projects (Brundick, 1995).

SLIM-Estimate for Windows is the Windows-based version of the popular SLIM estimation and planning tool, which has been used since 1978 by many top Fortune 500 and technology companies, government, and defense organizations worldwide (QSM, 1999). SLIM is applicable to all types and sizes
Developing a software estimating information system 10

of projects. It computes schedule, effort, cost, and staffing for all software development phases and reliability for the main construction phase. Because the software equation effectively models design intensive processes and is not methodology dependent, SLIM works well with waterfall, spiral, incremental, and prototyping development methodologies. It works with all languages, and function points as well as other sizing metrics. It is specifically designed to address the concerns of senior management, such as:

1. What options are available if the schedule is accelerated by four months to meet a tight market window?
2. How many people must be added to get two months schedule compression and how much will it cost?
3. When will the defect rate be low enough so I can ship a reliable product and have satisfied customers?
4. If the requirements grow or substantially change, what will be the impact on schedule, cost, and reliability?
5. How can I quantify the value of my process improvement program?

The SoftCost-R model was developed by Dr. Don Reifer based on the work of Dr. Robert Tausworthe of the NASA Jet Propulsion Laboratory. Resource Calculations, Inc. of Englewood, Colorado now markets SoftCost. It contains a database of over 1500 data processing, scientific and real-time programs. SoftCost-R is applicable to all types of programs and considers all phases of the software development cycle. The model is available for lease on IBM PC's. A separate model, SoftCost-Ada is available to model Ada language and other object-oriented environments. SoftCost-Ada has been developed to match the new object-oriented and reuse paradigm, which are emerging not only in Ada, but also in C++ and other object-oriented techniques. It contains a database of over 150 completed projects, primarily Ada.
Major Software Estimating Techniques and Models

COCOMO 81

The COstructive COst MOdel (COCOMO) originally published by Dr. Barry Boehm in 1981 under the simple name COCOMO (Boehm, 1981), is the most widely used software estimation model in the world for effort and cost estimation. The COCOMO 81 model predicts the effort and duration of a project based on inputs relating to the size of the resulting systems and a number of "cost drives" that affect productivity (Hong, 1998). COCOMO 81 model is defined as a hierarchy of models as follows: the Basic model, the Intermediate model, and the Advanced model.

Basic COCOMO

The basic COCOMO model is a static single-valued model that computes software development effort and cost as a function of program size expressed in estimated lines of code (LOC) (Pressman, 1993). The basic COCOMO equations take the form:

\[ E = a_b (KLOC)^{b_b} \]
\[ D = c_b E^{d_b} \]

Where \( E \) is the effort (in person-months) required, \( D \) is the development time (duration) in chronological months, and \( KLOC \) is the estimated number of thousand lines of code delivered for the project. The coefficients, \( a_b \) and \( c_b \) and the exponents \( b_b \) and \( d_b \) are given in the Table-1 below and are dependent on one of three project development types which (Boehm, 1981) called Organic, Semi-detached, and Embedded modes of development.

<table>
<thead>
<tr>
<th>Software Development Mode</th>
<th>( a_b )</th>
<th>( b_b )</th>
<th>( c_b )</th>
<th>( d_b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>2.4</td>
<td>1.05</td>
<td>2.5</td>
<td>0.38</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
<td>2.5</td>
<td>0.35</td>
</tr>
<tr>
<td>Embedded</td>
<td>3.6</td>
<td>1.20</td>
<td>2.5</td>
<td>0.32</td>
</tr>
</tbody>
</table>

In the organic mode, relatively small software teams develop simple software projects in a highly familiar, in-house environment, with a set of less than rigid requirements. In this type of environment, most of the people connected with a project have had extensive working experience with related systems within the organization. They tend to have a thorough understanding of how the system under development would contribute to the organization’s objectives. Usually, the size range of the products developed by such groups is less than fifty thousand delivered source instructions (KDSI). Only a few organic-mode projects develop products with more than 50 KDSI (NASA, 1997).

The semi-detached mode of software development represents an intermediate (in terms of size and complexity) software project in which teams with mixed experience levels must meet a blend of rigid and less than rigid requirements. Generally, the size range of products developed in a semi-detached mode of development extends up to 300 KDSI (NASA, 1997).

The embedded mode software development is a software project that must be developed within tight constraints. The software product must operate within (i.e., embedded in) a tightly coupled hardware, software, regulations, and operational constraints. An example of such a project includes a flight control software for aircraft or an electronic transfer system (NASA, 1997).

**Intermediate COCOMO**

The intermediate COCOMO model incorporates all characteristics of the intermediate model with a feature to assess the cost driver’s impact on each phase of the software-engineering life cycle. It calculates effort as a function of program size and a set of cost drivers. Those cost drivers include product assessments which are subjective, personnel, hardware, and other project attributes (Pressman, 1993). The Intermediate COCOMO model computes effort as a function of program size and a set of cost drivers (Pressman, 1997). The Intermediate COCOMO equation takes the form:

\[ E = a_b (KLOC)^b \times EAF \]
The factors $a$ and $b$ for the Intermediate COCOMO model are shown in Table-2. The effort adjustment factor (EAF) is calculated using 15 cost drivers. The cost drivers are grouped into four categories: product, computer, personnel, and project. Each cost driver is rated on a six-point ordinal scale ranging from low to high importance. Based on the rating, an effort multiplier is determined using a table describing the cost multipliers (Boehm, 1981). The product of all effort multipliers is the EAF.

### Table-2: Effort Parameters for Intermediate COCOMO model

<table>
<thead>
<tr>
<th>Software Development Mode</th>
<th>$a_b$</th>
<th>$b_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>3.2</td>
<td>1.05</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
</tr>
<tr>
<td>Embedded</td>
<td>2.8</td>
<td>1.20</td>
</tr>
</tbody>
</table>


### Advanced COCOMO

The Advanced COCOMO model incorporates all characteristics of the intermediate model, plus an assessment of the cost driver's impact on each step of the software engineering process (Pressman, 1993). The Advanced COCOMO model computes effort as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle. It applies the Intermediate model at the component level, and then a phase-based approach is used to consolidate the estimate (Fenton & Pfleeger, 1997). The 4 phases used in the advanced COCOMO model are requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT). Each cost driver is broken down by phase as in the example shown in Table-3 (Boehm, 1981).

### Table-3: Analyst Capability Effort multiplier for Advanced COCOMO Model

<table>
<thead>
<tr>
<th>Cost Driver</th>
<th>Rating</th>
<th>RPD</th>
<th>DD</th>
<th>CUT</th>
<th>IT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACAP</td>
<td>Very Low</td>
<td>1.80</td>
<td>1.35</td>
<td>1.35</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td>Nominal</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>0.75</td>
<td>0.90</td>
<td>0.90</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>Very High</td>
<td>0.55</td>
<td>0.75</td>
<td>0.75</td>
<td>0.70</td>
</tr>
</tbody>
</table>
Estimates made for each module are combined into subsystems and eventually an overall project estimate. Using the detailed cost drivers, an estimate is determined for each phase of the lifecycle.

Advantages of COCOMO

Although obsolete now, the COCOMO 81 model has a number of advantages. These advantages are:

1) The COCOMO cost estimation model is used by thousands of software project managers, and is based on a study of hundreds of software projects.

2) Unlike other cost estimation models, COCOMO is an open model, so all of the details are published, including:
   a) The underlying cost estimation equations
   b) Every assumption made in the model (e.g. "the project will enjoy good management")
   c) Every definition (e.g. the precise definition of the Product Design phase of a project)
   d) The costs included in an estimate are explicitly stated (e.g. project managers are included, secretaries aren't)

3) COCOMO is transparent, you can see how it works unlike SLIM and other models.

4) Cost drivers are particularly helpful to the estimator to understand the impact of different factors that affect project costs.

Drawbacks of COCOMO

The COCOMO model is not without problems, as a matter of fact, it has several drawbacks, which was the main reason for the development of COCOMO II. Some of the drawbacks are:

1) It is hard to accurately estimate KDSI early on in the project, when most effort estimates are required.

2) KDSI, actually, is not a size measure it is a length measure.

3) It is extremely vulnerable to misclassification of the development mode.

4) Success depends largely on tuning the model to the needs of the organization, using historical data, which is not always available.
COCOMO II

In a 1994 survey by the Software Engineering Institute, software cost estimation users expressed the need for a cost estimation model that takes into account the environmental issues raised by the software development technologies of the nineties and the 2000s (Park, Goethert, & Webb, 1994). These needs listed here in priority order, were:

1) for support of project planning and scheduling,
2) project staffing, estimates-to-complete,
3) project preparation,
4) replanning and rescheduling,
5) project tracking, contract negotiation,
6) proposal evaluation,
7) resource leveling,
8) concept exploration,
9) design evaluation, and
10) bid/no-bid decisions.

For these reasons, COCOMO II was developed to provide more up-to-date support than its COCOMO 81 and Ada COCOMO predecessors did (Boehm, Clark, Horowitz, Westland et al., 1995).

The first implementation of COCOMO II was released to the general public in mid-1997. Since its release, COCOMO II has seen improvements every year. The 1997 COCOMO II was calibrated to 83 data points (historical software development projects), using a 10% weighted-average approach, blending empirical data with expert opinion to calibrate the model parameters. Using those 83 data points, the 1997 release demonstrated an accuracy of within 20% of actuals, 46% of the time for effort and within 20% of actuals, 48% of the time for a non incremental development schedule (Boehm et al., 2000).

The 1998 release of the model was calibrated to 161 data points, using a Bayesian statistical approach, blending empirical data with expert opinion to calibrate the model. Using those 161 data points, the 1998 release demonstrated an accuracy of within 30% of actuals 75% of the time, (and within 30% of the actuals 80% of the time after stratification by organization) for effort. It also showed within 30% of actuals
Developing a software estimating information system

72% of the time (within 30% of the actuals 81% of the time after stratification by organization) for a non incremental development schedule (Boehm et al., 2000).

While each release of the COCOMO II tool has seen improvements in its user friendliness, the 1998, 1999, and 2000 model calibrations are the same. This means that no new data points have been added to the database used to calibrate the 1999 and 2000 releases of the tool beyond those that appeared in the 1998 calibration database (Boehm et al., 2000).

COCOMO II has been tuned to modern software life cycles. The original COCOMO 81 model has been very successful, but it doesn't apply to newer software development practices as well as it does to traditional practices. COCOMO II targets the software projects of the 1990s and 2000s, and will continue to evolve over the next few years.

Putnam Software Life Cycle Model (SLIM)

The estimation model otherwise known as the Software Life Cycle Management (SLIM) was developed by Larry Putnam of Quantitative Software Measurement in the 1970s (Putnam & Meyers, 1992). It is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project (Pressman, 1987). The SLIM model is based on the analysis Putnam conducted on the software life cycle using the Raleigh distribution of project personnel level versus time. It supports most of the popular size estimating methods including ballpark techniques, source instructions, function points, etc. It makes use the Rayleigh-Norden Curve to estimate project effort, schedule and defect rate. The Rayleigh-Norden Curve may be used to derive a software equation that relates the number of delivered lines of code to effort and development time (Putnam & Meyers, 1992).

SLIM uses linear programming, statistical simulation, program evaluation and review techniques to derive a software cost estimate. The software equation used for development is:

\[ K = \left( \frac{\text{Size}}{(C_k \cdot t^{4/3})} \right)^3 \]
Where $K$ is the total life-cycle effort expended (in person-years), $\text{Size}$ is the lines of code, $t$ is development time (in years). $C_k$ is the technology constant, combining the effect of using tools, languages, methodology, QA procedures, etc. The values of the technology constant can vary from as little as 2000 for a poor software development environment to 11,000 for an “excellent” environment. The constant $C_k$ can be derived for local conditions using historical data from past projects, and it is highly recommended. However, in the absence of historical data, Putnam’s recommended figures for different types of projects are:

- Real-Time Embedded = 1500
- Batch Development = 4894
- Supported and Organized = 10040

In a recent study, (Pengelly, 1995) indicated that SLIM does not perform accurately on small projects. However, (Londeix, 1987) reports that SLIM is suitable for software developments that meet the following criteria (1) Software size is greater than 5000 lines, (2) Effort is greater than 1.5 person-years, and (3) Development time is over 6 months.

Advantages of SLIM

Like the COCOMO model, SLIM also has some advantages which include:

1) SLIM is not widely used but there is a SLIM tool.
2) Uses linear programming to consider development constraints on both cost and effort.
3) SLIM has fewer parameters needed to generate an estimate over COCOMO and COCOMO II

Drawbacks of SLIM

The disadvantages of the SLIM model include:

1) Estimates are extremely sensitive to the technology factor
2) Not suitable for small projects
3) In order to use the SLIM model the software size must be identified in advance.
4) SLIM may not be affordable for small businesses. Based on a 1995 figure, a single user license costs $16,500 and $30,000 for the multi-user license. In addition, there is an annual maintenance fee (Giles & Barney, 1995).
Putnam Beta (PERT Analysis) Method

The Putnam Beta method is the PERT method combined with statistical analysis. The formula for an estimate is the basic PERT method $t(e) = (a + 4m + b)/6$ where $a =$ optimistic, $m =$ most likely, $b =$ pessimistic, which generates a beta curve around the expected time. Estimates can be derived by using the PERT analysis to calculate the weighted average of the best case (the shortest or most optimistic estimate), the worst case (the longest or most pessimistic estimate), and the most likely (expected) values.

By calculating the standard deviation (some call it “sigma”) of whatever we are estimating (size, effort or duration), we can determine the probabilities that allow us to state with some confidence that the estimated value can reasonably be achieved without running past the planned pessimistic value. One standard deviation for example, is equivalent to 68.3% probability of achieving the estimate, two standard deviations is equivalent to 95.5% probability of achieving the estimate, and three standard deviations equals 99.7% probability of achieving the estimate.

Delphi Approach

The Delphi technique is a group forecasting approach, generally used for future events such as technological developments. Originally developed at the Rand Corporation in the late 1940s as a way of making predictions about future events, the name comes from the divinations of the Greek oracle of antiquity, located on the southern flank of Mt. Parnassos at Delphi (Boehm, Abts, & Chulani, 1997). In the recent years, the Delphi technique has been used as a means of utilizing estimates from a group of informed individuals and feedback summaries of these estimates for additional estimates by these informed individuals until a reasonable consensus emerges. Participants are asked to make some assessment regarding an issue, individually in a preliminary round, without consulting the other participants in the exercise. The first round results are then collected, tabulated, and then returned to each participant for a second round, during which the participants are again asked to make an assessment regarding the same issue, but this time with knowledge of what the other participants did in the first round. The second round usually results in a narrowing of the range in assessments by the group, pointing to some reasonable middle ground regarding
the issue of concern. The original Delphi technique avoided group discussion, the Wideband Delphi technique accommodated group discussion between assessment rounds (Boehm, 1981).

**Work Breakdown Analysis**

Long a standard of engineering practice in the development of both hardware and software, the work breakdown structure (WBS) is a way of organizing project elements into a hierarchy that simplifies the tasks of budget estimation and control. It helps determine just exactly what costs are being estimated. Moreover, if probabilities are assigned to the costs associated with each individual element of the hierarchy, an overall expected value can be determined from the bottom up for total project development cost (Baird, 1989).

The WBS is iteratively developed. As more information about the software products becomes available, the WBS is updated. The first attempts at building the WBS focuses on the products to be delivered, grouped under the Computer Software Configuration Items (CSCIs) to be developed. As plans are completed and the software engineering and test environments are identified, the WBS is updated.

WBS-based techniques are good for planning and control. A software WBS actually consists of two hierarchies, one representing the software product itself, and the other representing the activities needed to build that product (Boehm, 1981). The product hierarchy as shown in Figure-1 describes the fundamental structure of the software, showing how the various software components fit into the overall system. The activity hierarchy as shown in Figure-2 indicates the activities that may be associated with a given software component.

![Figure-1 - WBS Product hierarchy](image-url)
Aside from helping with estimation, the other major use of the WBS is cost accounting and reporting. Each element of the WBS can be assigned its own budget and cost control number, allowing staff to report the amount of time they have spent working on any given project task or component, information that can then be summarized for management budget control purposes.
The Software Estimation Process

The process of developing a cost estimate for software is not terribly different from the overall process of estimating any other element of cost. There are, however, aspects of the process that are peculiar to software estimating. Some of the unique aspects of software estimating are driven by the nature of software as a product. Other problems are created by the nature of the estimating methodologies (Brundick, 1995).

But why is it so difficult to estimate the cost of software development, one may ask? The answer lies in the nature of the product itself. Software development effort itself is often plagued with problems that are also directly or indirectly responsible for the difficulties encountered in estimating that effort. For example, one of the first steps in estimating is to understand and define the system to be estimated. However, software is intangible, invisible, and intractable. It is inherently more difficult to understand and estimate a product or process that cannot be seen and touched. That characteristic of software is what gives people the false sense that the user can alter software anytime to accommodate new requests. As software can grow and change as it is written without any visible, physical change to the system, users think it’s not a big deal making changes to the original specifications. Even when the hardware is inadequately designed, or when hardware fails to perform as expected, the tendency is to “fix” the problem through changes to the software. Such changes may and often occur late in the development process, and sometimes results in unanticipated software growth. It is such “scope creep” and unanticipated changes that give software a bad name of being “late” and over budget.
Therefore, one of the most challenging tasks in project management is how to accurately estimate needed resources and required schedules for software development projects. The software estimation process includes estimating the size of the software product to be produced, determining which functions the software product must perform, estimating the effort required, developing preliminary project schedules, and finally, estimating overall cost of the project. Size and number of functions performed are considered major productivity (“complexity”) factors during the software development process. Effort is divided into labor categories and multiplied by labor rates to determine overall costs. Therefore, software estimation is sometimes referred to as software cost estimation. Since software cost has risen to be a major cost in any software-based project, the process of estimating software activities should be approached as a major task and therefore should be well planned, reviewed and continually updated. The basic steps required to accomplish software estimation are described in the following paragraphs.
Understand What You Are Estimating

Unless one understands the scope of work that needs to be done, it is impossible to generate any sort of meaningful estimate. For that reason, identifying the tasks using a statement of work (SOW) helps in clarifying the scope of a project. A statement of work might include such things like a top-level description of the software to be developed, imposed technical and/or management standards, project constraints (cost, schedule, staffing, target machine, etc.), and any external dependencies.

In addition, creating a high-level conceptual design helps in understanding the product. The design is subdivided into components (subsystems and/or modules) until each component can be reasonably estimated. This whole exercise should be documented in a Work Breakdown Structure.

Estimate Size

The first step to an effective estimate is an accurate estimate of the size of the software to be built. The source(s) of information regarding the scope of the project should, wherever possible, start with formal descriptions of the requirements - for example, a customer’s requirement specification or request for proposal, a system specification, a software requirement specification. If re-estimating a project in later phases of the project’s lifecycle, design documents can be used to provide additional detail. However, the lack of a formal scope specification should not be a hindrance to do an initial project estimate. A verbal description or a whiteboard outline is sometimes enough to start. In any case, communicating the level of risk and uncertainty in an estimate to all concerned and re-estimating the project as soon as more scope information is determined is a must.

Using historical data coupled with an automated tool can make life very easy. Where historical data is available, with the object types and their numbers of methods determine where the new object falls in size relative to the objects in the historical database. It is a good idea to stick with average object sizes unless there is reason to believe that the new object will be much larger or smaller than normal. This helps to avoid making consistent overestimates or underestimates.
Estimate Effort

An effort estimate is required for subsequent stages of the estimation process. First estimating size, and then determining effort using historical productivity data can derive the effort estimate. If historical productivity data is unavailable, effort can be estimated directly by using one of the techniques discussed above. If historical data exists, relating effort to software size by using the historical data and/or an automated estimating tool to calculate effort from size estimates. If historical data is not available, use at least 2 estimating techniques to estimate effort directly. Compare all the estimates generated. If there is a significant discrepancy, iterate until the source of the discrepancy is understood and resolved.

Estimate Schedule

The next important step in estimating a software development project is to determine the project schedule from the effort estimate. This generally involves estimating the number of people who will work on the project, what they will work on (the Work Breakdown Structure), when they will start working on the project and when they will finish ("staffing profile"). Once this information is available, you need to lay it out into a calendar schedule (tools such as Microsoft Project can be used to do this). Again, historical data from the organization’s past projects or industry data models can be used to predict the number of people needed for a project of a given size and how work can be broken down into a schedule.

Estimate Cost

There are many factors to consider when estimating the total cost of a project. These include labor, hardware and software purchases or rentals, travel for meeting or testing purposes, telecommunications (e.g., long-distance phone calls, video-conferences, dedicated lines for testing, etc.), training courses, office space, and so on. Exactly how one estimates total project cost will depend on how an organization allocates costs. Some costs may not be allocated to individual projects and may be taken care of by adding an overhead value to labor rates ($ per hour). Often, a software development project manager will only estimate the labor cost and identify any additional project costs not considered "overhead" by the organization.
The simplest labor cost can be obtained by multiplying the project’s effort estimate (in hours) by a general labor rate ($ per hour). A more accurate labor cost would result from using a specific labor rate for each staff position (e.g., Technical, QA, Project Management, Documentation, Support, etc.). The estimator would have to determine what percentage of total project effort should be allocated to each position. Again, historical data or industry data models can help.

Follow up

To improve our ability to estimate software projects we must be able to learn from the past. As human memory is fallible, we must keep written records. The following information need to be documented so as to provide a basis for future estimates – the initial estimate, estimating methods used, and any assumptions made with regard to how the estimate was derived. Ideally, there would be a readily available, easy-to-use corporate repository for this data. Such a database may not exist, however, nothing prevents project leads and project managers from keeping their own records.

Progress on the project needs to be reviewed frequently and estimates updated to reflect the effort remaining. This implies tracking and documenting actual effort spent on each task, the actual size of each module and any lessons learned as to why the original estimate was inaccurate.
WDC Estimation Information Management System

The cost of developing computer software consumes an increasing portion of many organizations' budgets. As this trend continues, the capability to estimate the size, effort and duration required to develop a candidate software product becomes increasingly important. SizeEST is an automated software size estimation tool, which fulfills this need. Assimilating SizeEST to any small to medium-sized software development organization's particular environment can yield significant reduction in the risk of cost overruns and failed projects.

SizeEST accepts a description of a software product to be developed and computes estimates of the software size, effort required to produce it, and the calendar schedule required, as a function of the defined set of development life-cycle phases. The program as shown in Figure-5, is menu-driven and mouse sensitive with context-sensitive help that makes it possible for a new user to easily operate the program and to learn the fundamentals of cost estimation without having prior training or separate documentation. The implementation of these functions makes SizeEST unique from the varied commercial cost estimating tools.

When WDC decided to go through the Software Engineering Institute’s Capability Maturity Model (CMM) for software level 2 assessment, it was understood by the organization that software estimation is one area that a gap existed. That gap needed to be closed if there was any hope of passing the SEI CMM-Based Appraisal for Internal Process Improvement (CBA-IPI) assessment. All members of the leadership/management team for the center (comprising of the WDC Development Manager, QA & Test Manager, Operations Manager, and the Software Engineering Manager) were familiar enough with software cost estimation to realize that it represented the biggest obstacle for the organization to be CMM level 2 certified. They quickly defined the goals for the estimation tool. By applying the system approach to this process, what was done in developing this system can be summarized as follows.
Developing a software estimating information system

Figure 4 - SizeEST main user interface
Figure-5 - SizeEST program user interface
The Planning Phase

In planning our strategy to overcome the issues we faced with project estimation, we basically went through the following systems approach steps as described by (McLeod, 1998).

Recognizing the problem

The need for a better estimating process as opposed to the Delphi method we were using was identified during a mini-assessment of the organization. For me, it became evident that project estimating was taking too long and was not consistent. Having recognized that there was a problem, the next step was to actually define the problem as it relates to the organization.

Defining the problem

Once I realized that a problem existed, there was a need to fully understand the problem well enough to pursue a solution. Rather than attempting to gather all the information, I sought to identify where the problem existed and its cause. For this purpose, I sought the assistance of our systems engineer, and the entire leadership team.

Setting the system objectives

After discussing the goal of creating a simple database application that can provide us historical development data for software project estimation (project size, effort, duration, and cost), the WDC Development Manager basically signed off on the idea and gave me the authority to create the application program. With the assistance of the systems engineer and the leadership team, we developed a list of objectives that the system must meet in order to satisfy the users. The initial task was to define the various project categories (very small, small, medium, large and very large-sized project), and what we meant by project complexities. After that, the next task was to benchmark our data by collecting the actual size, effort, cost, project categories and complexities of all the projects we completed in the past as a source of our historical data.
Identifying system constraints

The new system of course, was not free from constraints. In addition to the constraints imposed by the environment, such as the need of project managers to obtain estimation reports, other constraints were imposed by management. Such constraints included the requirement to use existing software and to have the system up and running in less than two months.

Conducting and preparing a feasibility study proposal

Apart from investigation into the pros and cons of using the only database engine we had at our disposal, Microsoft Access, there was no feasibility study conducted. So the only report that resulted from this exercise was a one-page summary of the benefits and drawbacks of using Microsoft Access database.

Approval of the study

Recognizing the budget constraint imposed on the organization by the parent company, the organization’s development manager did not spend much time in approving Microsoft Access as the database engine to use since it was not going to cost extra money. With that stamp of approval, I went to work.

The Analysis Phase

Announcing the system study

To secure the cooperation of the employees, I had to involve them in the process. So I devised a rollout program to get their buy in. First, a meeting was called where our intentions were revealed to everyone, and subsequent meetings were held to get their input and explain what would be needed to make this a success. The benefits of the new system was discussed and rationale for choosing the size estimation method (the Putnam Beta 3-point estimating technique) we chose. The rollout program continued and included demonstration of the actual system after it was implemented and training on how to use it.
Organizing the project team

Since I chose to do this project by myself so as to use it for my course work, there was no project team per se. Other than the coordination of the rollout plan, getting people’s buy-in, getting the project leaders involved in defining what we meant by small-sized, medium-sized, or large-sized project, the project complexities, and training, it was a one-person project.

Defining information needs

In order to learn more about the user’s information needs, I engaged in a couple of information-gathering activities, which included personal interviews and meetings with the project leaders and project managers. The documentation of the existing technique used by the organization was reviewed and documentation for the proposed new system was presented. This exercise made it possible for me to understand and record the information needs of the users.

Defining the system performance criteria

With the information needs defined, it was now possible to specify exactly what the system should accomplish, that is, the system’s performance criteria. The identified performance criteria were debated and adopted after the project leaders reached a consensus on how to achieve them.

Preparing and approving the design proposal

Having understood the needs and performance criteria of the users, it became easier to propose an design strategy. The design proposal was provided in the form of a prototype to the manager with the opportunity to make a second go/no go decision. The prototype made it easier for the manager to give it her final blessing, and the design began.
The Design Phase

Armed with the understanding of the existing system and the requirements for the new system, I began the arduous task of determining the processes and data required by the new system. The activities are summarized in the following steps below.

Preparing the detailed system design

I took a top-down design approach, which is characteristic of structured design whereby the design proceeds from a system to a subsystem level. Using data flow diagrams (DFD) and starting with the context diagram, the system was decomposed into lower subsystems. A data dictionary, which described the contents of the database, was generated from this exercise.

Identifying and evaluating alternative system configurations

Although this step is very important in design, a lot of time was not spent on identifying alternative system configurations for this system. The reason was because the system was already constrained to use existing equipment and to keep the cost of developing very minimal. In short, there were no alternative configurations for the new system.

Preparing the implementation proposal

An implementation proposal outlining the implementation work to be done, the expected benefits, and the costs was drafted and submitted to the manager for approval. This proposal included the system objectives, constraints, and system design and installation schedule. In addition, a cover letter of transmittal accompanied the proposal to make it formal.

Approval/Disapproval of the system implementation

The decision to proceed with the implementation phase was important since implementation would greatly increase the number of participants. As long as the expected benefits from the system exceeded the costs, the implementation would be approved. The benefits indeed exceeded the costs and the green light was given to continue the implementation.
The Implementation Phase

According to (McLeod, 1998), the implementation phase is the acquisition and integration of the physical and conceptual resources to produce a working system. This is the time to turn the design into a meaningful system that the user can use to solve his/her problem. The work that went into implementing SizeEST can be summarized in the following steps.

Planning of the implementation

At this point, the managers and project leaders had a good understanding of the work needed to implement the system design. This knowledge was used to develop a very detailed implementation plan.

Announcing of the implementation

Just like the announcement of the initial system study, the actual implementation project was announced to the employees partly to inform them of the decision to implement the new system, and in part to ask for their cooperation. Employees were kept up-to-date throughout the implementation phase, and I must say that that helped in selling the new system when it finally became operational.

Obtaining resources

This is one of those steps that I didn’t have to do much but utilize the resources that we already have. All the hardware and software resources needed for this project were already in existence. Other than preparing more detailed documentation such as structured English and program flowcharts, the coding was performed and the programs tested. The end product was of course a software library of application programs.

Preparing the database

As stated before, the decision was to use Microsoft Access database as the database management system (DBMS). Existing data was gathered, and reformatted to conform to the new system design. All the data were prepared and entered into the database by employing the help of some employees who have worked with the existing system.
Educating the participants and users

It was expected that the new system would be used by at least all project leaders and project managers. That meant that not only did they have to be trained on how to use the system, they had to be educated on the parametric equations that were used in the calculations. It was important that the participants trust the data information that the system generates and most of that trust would come from the type and amount of training they are given. With that in mind, these participants were involved all the way throughout the development stage of the new system. And I must say that they have embraced the new system well.

Preparing the cutover proposal

The process of halting use of the old system and starting use of the new one is called cutover (McLeod, 1998). When it was clear that all of the development work was drawing to a close, I recommended to the manager through a memo, for cutover to commence. Further, it was recommended that the new system be used in parallel with the existing process.

Approval/Disapproval of the cutover proposal

Since it is a good practice for an organization to use more than one estimating technique for estimating, the recommendation to run the new system in parallel with the existing practice was a welcome one. The manager and the project leaders reviewed the project status and approved the recommendation. The two methods for estimating (the automated SizeEST program) and the existing Delphi technique would be used together to complement each other.

Cutting over to the new system

As mentioned before, the cutover approach used was parallel execution. Both the old system and the new one will be fully utilized until such a time that the organization feels confident that the new system is consistently within plus or minus ten percent accuracy. When that happens, the old approach will be eliminated entirely, but meanwhile, both systems will run in parallel. This cutover also signaled the end of the development portion of the system life cycle and the beginning of the system use phase.
The Use Phase

The use phase involves the actual usage of the system, auditing of the system to ensure that it’s meeting its objectives, maintenance of the system (tweaking or fine-tuning), and eventually re-engineering the system.

Using the new system

Users are allowed to use the system now to do their estimates. The bottom-line is that the new system meets the objectives that were identified in the planning phase. Although it’s still too early to tell, so far the system is doing what it was supposed to do. The ultimate verdict will come after the projects whose estimates were determined using the new system have completed.

Auditing the system

After the new system has had a chance to settle down, a formal audit will be conducted to determine how well it is satisfying its objectives and performance criteria. This audit will be conducted twice a year and the results will be shared with the manager, project leaders, and all users.

Maintaining the system

It was agreed up-front that while this system is in use by the organization, modifications and enhancements will be made so that the system continues to provide the needed support. The system maintenance will be performed basically for three reasons – to correct any errors found in the program or design, to keep the system current, and to improve the system.

Re-engineering the system

Everything that has a beginning will have an end. Eventually, when the system is no longer usable or if it becomes obsolete, it would most likely be re-engineered. When that happens, there would be a proposal that must include descriptions of the inherent weaknesses in the system, statistics concerning the cost of maintenance, and so on. At that time, management and the project leaders would decide either to re-engineer or not to re-engineer the system.
Summary and Conclusion

This paper has discussed the history of software size estimation, and the estimation process comparing and contrasting two of the leading estimation models (COCOMO and SLIM). Some basic software estimating tools were examined including a description of a size-estimating information system (SizeEST) for a medium-sized software development organization. As discussed in the paper, the software estimating organization must be able to customize the software-estimating tool to its own software development environment. This requires collecting historical data from past-completed projects to accurately provide the inputs that the software tool requires.

My software development organization recognized the importance of software size estimation that it embarked on developing its own size estimation tool that took into account its own unique development environment and practices. The organization has designated the project leaders to be thoroughly trained in the use of the tools. This group of individuals is responsible for software estimating activities for their respective projects. It is a recommended practice to use two or more software estimating tools and my organization has resolved to use the new tool in conjunction with the Delphi method that it has always used.

When the software estimation process is done correctly, the benefits realized far outweigh the cost of doing the estimating. Major benefits that can be achieved include lowering the cost of doing business, increasing the probability of winning new contracts, increasing and broadening the skill-level of key staff members, acquiring a deeper knowledge of the proposed project prior to beginning the software development effort, and understanding, refining, and applying the proper software life cycle mode (Brundick, 1995). My organization understands this now and is poised to take advantage of its newfound good practice. The hope is that as these software estimating components are enhanced, refined, and continually applied, the software estimating process, associated tools, and resulting products attain higher levels of quality and ultimately benefit all.
References


