A New Estimation Methodology for Reusable Component-Based Software Development Projects

A DISSERTATION PRESENTED TO

THE DEPARTMENT OF INFORMATION SYSTEMS

UNIVERSITY OF CAPE TOWN

IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE

MASTER OF COMMERCE DEGREE

IN

INFORMATION SYSTEMS

by

C COURT

23rd July 1999

University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Preface

This report is not confidential. My great thanks go to the following people. Mr Joe van Niekerk, who co-authored the Dev Monitor software estimation tool described in this dissertation. Prof. Paul Licker PhD, my dissertation advisor, for his continuous support and constructive criticism. Mr Malcolm Salida, who spurred me on and assisted with the proofreading of the dissertation.

Very special thanks go to my wife Debbie, who provided support when it was much needed and showed great patience with all the late nights.

Certain commercial products have been mentioned within this dissertation. The trademark for Clarion is owned by TopSpeed Corporation and Delphi is owned by Inprise Corporation.

I certify that this report is my own work and all references are accurately reported.

C COURT
email: court@mweb.co.za

University of Cape Town

Contents

University of Cape Town

University of Cape Town

Abbreviations

Provided below is a list of abbreviations used within the dissertation and their meaning.

4GL – 4th Generation Language

CAPE – Computer Aided Parametric Estimation

CASE - Computer Aided Software Engineering

CMM – Capability Maturity Model

COCOMO – COnstructive COst Model

DSI – Delivered Source Instructions

EAF – Effort Adjustment Factor

FPA – Function Point Analysis

GUI – Graphical User Interface

ICASE – Integrated Computer Aided Software Engineering

IFPUG – International Function Point User Group

KDSI – Thousands of Delivered Source Instructions

KSLOC – Thousands of Lines Of Code

MM – Man Months

LOC – Lines Of Code

OO – Object Oriented

SEI – Software Engineering Institute

SLIM – Software LIfecycle Model

SLOC – Source Lines Of Code

TDI – Total Degree of Influence

UFPC – Unadjusted Function Point Count

University of Cape Town

List of Figures

University of Cape Town

List of Tables

University of Cape Town

"There is no such thing as a late software project...

It was the estimate that was wrong in the first place."

Dr Peter Finklestein, 1997

University of Cape Town

Abstract

Title: A New Estimation Methodology for Reusable Component-Based Software Development Projects

Author: Cliff Court

Dissertation Advisor: Professor Paul S Licker PhD

Estimating the duration of software development projects is a difficult task. There are many factors that can derail software projects. However, estimation forms the fundamental part of planning and costing any project and is therefore very necessary.

While several formal estimation methodologies exist, they all exhibit weaknesses in one form or another. The most established methodologies are based on early software development methods and it is questionable as to whether they can still address more modern development methods such as reusable component-based programming. Some researchers believe not and have proposed new methodologies that attempt to achieve this.

Thus what is needed is a methodology that takes into account modern component-based development practices and, as a result, provides acceptable accuracy for the software organisation. This dissertation attempts to uniquely satisfy both of these requirements.

A new methodology called "ReCom" is proposed. In order to motivate this methodology, two approaches were combined. Firstly, a set of characteristics required for a successful

University of Cape Town

estimation methodology was established by reviewing the work of 15 researchers. Secondly, 6 different existing estimation methodologies were reviewed, three of which were well-established older methodologies and three more modern methodologies. Weaknesses in each were highlighted, particularly in regard to how they address reusable component-based development projects.

The ReCom methodology combines the success characteristics required of an estimation methodology with a process that overcomes most of the weaknesses found in the other methodologies. ReCom also provides a unique ability to address the individual programming speed of each developer. In this way, ReCom appears unique among the estimation methodologies researched.

Due to time constraints, only limited empirical testing of ReCom was possible. However, useful observations were made. In particular, difficulties in implementation were highlighted and early indications of developer programming speed differences were evident. Further, it was evident that ReCom should be enhanced to better encompass non-programming activities such as compiling and testing. Currently the basic ReCom model does not adequately address these activities.

There is significant scope for further research regarding ReCom and its implementation.

University of Cape Town

Chapter 1: Introduction

Since the 1960's, attempts have been made to quantify the amount of effort and cost that a particular software project will consume. A number of estimating methods have been developed and are in regular use by software developers. Yet researchers still report projects being overdue and over budget on an on-going basis (Keremer, 1987; van Genuchten, 1991; Lederer and Prasad, 1995). While there are many reasons for late software delivery, inappropriate estimation methodologies could make it difficult for any developer to deliver a finished system within a given timeframe.

Of course, any estimation effort amounts to predicting the future, a difficult task in any situation. But software estimation is not quite as random as say, predicting a winning number at a roulette wheel. Software projects are often similar. Unlike the past when all software had to be written "from scratch", recent development tools make it easier to access pre-built software components that are easier to implement and customise.

Therefore, by recording our historical efforts in implementing each component when undertaking projects that make use of these new development tools, estimating software project effort and cost should be possible to an acceptable degree of accuracy.

For the purposes of this dissertation, the accuracy of an estimate for a given project is defined as a formula made up of the absolute value of the estimated software development time and the actual software development time, expressed as a percentage of the actual development time, subtracted from 100%. An estimate is considered completely accurate if this equals 0%, thereby making the equation equal to 100%. An estimate is considered acceptably accurate if

University of Cape Town

the accuracy is greater than a particular percentage value determined by the assessing person e.g. an assessor may decide that an accuracy above 90% is acceptable.

This may be shown in the following formula:-

$$\text{Accuracy} = 100\% - \frac{|\text{ESDT} - \text{ASDT}|}{\text{ASDT}} * 100$$

where ESDT = Estimated Software Development Time
ASDT= Actual Software Development Time

An estimate is considered to be acceptably accurate if:-

$$\text{Accuracy} > x\%$$

where $x\%$ is a percentage determined by the person assessing the accuracy.

Yet, although it may seem reasonable to expect a high degree of accuracy for software estimates, academic and research papers continue to highlight weaknesses in popular existing estimation methodologies (Jones, 1998; Kitchenham, 1997; Abran and Robillard, 1996). For example, one issue is that the majority of published estimation models were developed over fifteen years ago (Boehm, 1981; Albrecht, 1979). Software has advanced considerably since that time and it is questionable whether such estimation methods – even given that they have been updated from the original form (Boehm et al, 1998)– cater well for today's software development methods.

Predominant among the latest software development technologies is the ability to reuse ready-made software components. These are pre-written chunks of software source code that have been generalised such that they provide many of the common functions of a given

University of Cape Town

application requirement. For example, an email component might contain the functions to compose and send messages, receive messages, and send file attachments. Instead of programming this functionality every time it is required in a project, it is easier and less time-consuming to reuse these components. Typically the components are customisable so that they can be modified from the general case to fit the specific need of the project. This concept is revolutionising the development of software. In particular, when used effectively, components can save a tremendous amount of effort when compared with the older method of rewriting the common functions in new source code for every project. However, the majority of popular estimation methodologies either do not address reusable components at all or provide only minimal support for such components. (Morgan, 1997).

The aim of this dissertation is to present a new estimation methodology that will specifically address development projects that make use of reusable components. There will be three parts to this. The first part will investigate academic opinion as to what characteristics should be used to create a more accurate estimating methodology. Thereafter follows a review of existing estimation methodologies and their suitability for estimating software projects that make use of reusable components. Finally, a new methodology – Reusable Component-Based Software Estimation – will be introduced. Developed as a consequence of two earlier papers by the author, (Court, 1997a and 1997b), the methodology will meet the majority of accepted characteristics for successful estimation and will specifically address the reusable component technologies found in today's development tools.

As an important byproduct of this dissertation, a software tool was developed to automate the proposed methodology This proved useful for testing the applicability of the methodology

University of Cape Town

and provided an important checklist characteristic for successful estimation, the automation of the process.

The structure of the dissertation is as follows. After this first introductory chapter, the second chapter provides a formal discussion of the research problem and proposes a theoretical model for describing and evaluating estimation methodologies. The third chapter will discuss the generally accepted characteristics of a rigorous estimation methodology. The fourth chapter will discuss what constitutes components and component-based development. The fifth chapter provides a comprehensive overview of popular estimation methods and how they fit the theoretical model. In the sixth chapter these estimation methods will be critically reviewed to highlight the weaknesses of their applicability to component-based software projects and of their ability to satisfy the characteristics of a rigorous estimation methodology. In the seventh chapter, the proposed new Reusable Component-Based Software Estimation methodology will be introduced with supporting references from the previous chapters. Chapter eight will provide physical applicability of the methodology through the use of the automated tool known as "Dev Monitor". The dissertation will conclude with a chapter on initial test results using the methodology with suggestions for future research to provide validation and improvement.

University of Cape Town

Chapter 2: Statement of the Dissertation Problem

Estimating the effort and cost of a software system represents the first activity in a software project management process (Pressman, 1992). The reasons for this can be easily realised when one looks at the amount spent on software projects annually. In 1985, this figure was reported at US$140 billion worldwide with the 1990 figure predicted at that time to be over US$250 billion (Boehm and Papaccio, 1988). More recently, figures of between US$300 billion and US$ 600 billion have been suggested just to address the issue relating to software systems affected by the Year 2000 considerations (Cassell et al, 1997). Considering the magnitude of these numbers, it is clear that a good understanding of what a future system will cost is very important.

Lederer and Prasad (1995) suggest that both over and underestimating projects can have a negative impact on the organisation. In the case of underestimating, such estimates may convince management to proceed with projects that are delivered late and therefore fail to deliver the required payoff. In the case of overestimation, management may be swayed not to proceed with beneficial projects.

The Software Engineering Institute (SEI) at Carnegie Mellon University has developed a well-accepted model for gauging the capability of a software development team (van Solingen et al, 1998; Hall and Fenton, 1997). The primary model created by the SEI is known as the Capability Maturity Model (CMM) and provides for five levels of capability. The lowest level is termed the "Initial Level" (Level 1) and is characterised as ad-hoc or chaotic in nature. Clearly this is not an ideal level for organisations to operate within. In order to make the step up to the next level, known as the "Repeatable Level", the SEI states that policies for managing a software project and the procedures to implement it must be

University of Cape Town

established. A key procedure required within this still low level of the CMM is a carefully developed estimation process (Paulk et al, 1993, p L2-11).

Others have also pointed out the importance of a suitable estimation process. The National Research Council for Canada reported that software estimation was an important requirement for three key reasons (Vigder and Kark, 1994, p12):

- Planning and budgeting - whereby management could make strategic decisions based on estimates
- Project management - whereby estimates are used to monitor and control implementations and determine the relative success thereof.
- Communication between team members - This assumes that a by-product of the estimation process is a breakdown of the effort required. This breakdown can then be used as a basis for team members to better understand their roles and targets.

The above reasons appear both reasonable and logical and it is therefore accepted that estimation is an important management process. However, it appears that many software estimates are not very accurate. van Genuchten (1991) reported on the results of three separate research projects which showed that on average 60% (Jenkins et al, 1984), 80% (Phan et al, 1988) and 50% (van Genuchten, 1991) respectively, were delivered later than the originally estimated date of completion. A review of four different algorithmic methods by Keremer (1987) showed inaccuracy rates of between 85% and 772%, with in excess of 80% of the projects being completed *earlier* than predicted by the estimations.

University of Cape Town

This is not altogether unexpected considering that many factors influence the estimation process. Among these are project complexity, political issues, management assessment and risk management. In the popular estimation methodologies, project complexity is factored into the process (Boehm, 1981; Albrecht, 1979). Issues that affect complexity include the number of data elements within a project, transactions rates and data size. Organisational politics (manifesting itself as opposing views regarding project decisions) have also been shown to affect the estimation outcome (Lederer and Prasad, 1991). They report that depending on which player (user, manager or developer) performs the estimate, pressures from the other players can influence the outcome of the result.

Given the complexities of software application development, the issues highlighted above must be factored into any estimation methodology in order to formulate acceptably accurate results. A model is shown below by which various estimation methodologies may be abstracted and compared.

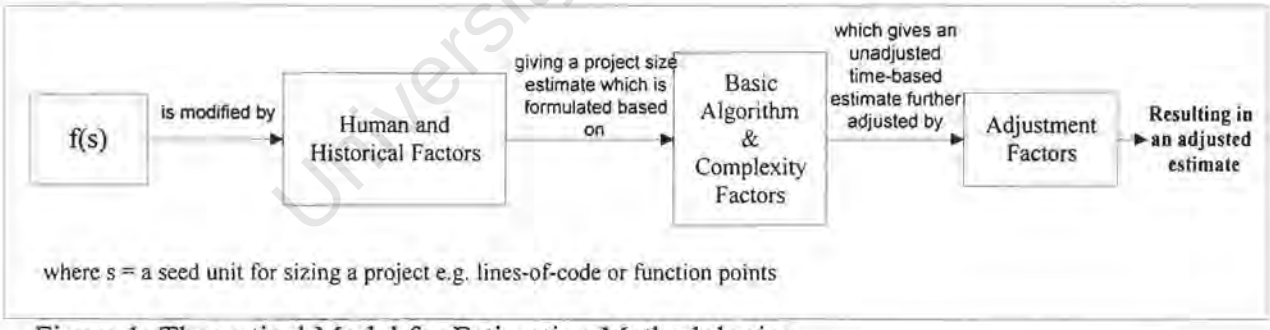

Figure 1: Theoretical Model for Estimation Methodologies

The model assumes that all estimates begin with a seed value, which provides the basis for the estimation methodology. This is typically a sizing unit, such a function points or lines-of-source-code. Two key characteristics, human factors and historical records influence the determination of this basic seed value. The human factors represent the subjective nature of most estimation methodologies while historical records often form the basis for making

University of Cape Town

estimation decisions through analogy with previous systems. The next part of the model has the influenced seed value being transformed into an initial time estimate through calculations based on factors such as complexity characteristics. Applying a set of adjustment criteria to the initial estimate produces the final estimate.

Using the above model as a basis for comparing and evaluating estimation methodologies, it is useful to review various issues that contribute to the difficulty in producing accurate estimates.

Putman and Myers (1997) suggest that the software estimation problem has been 'solved' but software is still not delivered on time because of management-related factors. They claim that management often do not correctly assess either the feasibility or risk profile of a project. In other words, it will always be difficult to estimate a project cost if the project has little chance of success in the first place. This is an example of a complicating factor that can affect the accuracy of an estimate, but which can not be easily factored into any methodology. The most one can hope for in these cases, is that management are aware of the issues and try to minimise them.

Känsälä (1997) points to profiling the risk of projects and suggests that traditional cost estimating methods assume that projects will not encounter any exceptional problems. As a result, he suggests that one should always perform a quantitative risk analysis of the project to achieve a more realistic estimation result. Many varied events can cause delays in software projects e.g. hard disk failure or the loss of a key programmer. In order to accommodate risk as highlighted above, an estimation methodology should cater for risk in some form.

University of Cape Town

Consistency on the part of developers plays a further complicating role in the estimation process. Schach (1990) reported on research undertaken by Sackman et al (1968), where they found differences of up to 28 to 1 between the performances of similarly experienced software developers. Since the accuracy of estimates is measured by how closely they match the actual time to develop an application and since programmer performance directly affects the actual development time, this research indicates that individual programmer performance could significantly affect software estimate accuracy. This prompted Schach to comment:-

> "On the basis of these results, it is clear that we cannot hope to estimate software cost or completion time with any degree of accuracy" (Schach, 1990, p82)

Schach's pessimistic view of software estimation was written at a time before reusable software components became popular. By tracking the reuse of software components by each individual developer, the proposed new estimation methodology can largely address the concerns of Schach.

Another area where human-related issues can affect the final delivery date of a project is interruptions. van Solingen et al (1998) found that interruptions of any kind can increase the project times by 15%-20%. Since van Solingen et al were able to determine a percentage range for how much of a project's development time is affected by interruptions, it is reasonable to expect estimation methodologies to be able to accommodate this issue.

Thus we can deduce that while estimation is a vital management requirement for software projects, it is fraught with difficulty for many reasons. Nevertheless several estimation

University of Cape Town

techniques are available that have been developed since the 1960's (Stutzke, 1997). The various techniques are based on seed counts of various parameters such as program source lines-of code (Boehm, 1981) and function-points (Albrecht, 1979), as well parametric models based on scientific algorithm (Putnam, 1978).

However, these estimation methods were all developed during the late 1970's and early 1980's when more recent development techniques such as Graphical-User-Interfaces (GUIs) and object-orientation were not yet available. Of particular importance to estimation, the reuse of software through object-orientation or ready-made software components is critical. This is because such reuse can significantly reduce the effort, and therefore time, required to complete a given software project.

Software 'chunks' that have been previously written for the purposes of reuse are often termed "objects" or "components". As such reusable software is often called object-oriented or component-based software. In an ideal case, all new projects can be constructed by connecting these objects or components together, much like pre-fabricated housing materials that are cemented together to construct a house in a much shorter time than building each brick one at a time.

For those software projects that are well suited to this modular form of construction, the estimation methods developed in excess of 15 years previously have limited value (Jones, 1998; Morgan, 1997). These older methods had no way of accommodating reusable components and always assumed that all software was developed from the start each time. A number of researchers have questioned these relatively old methods and their applicability to today's programming environments (Pfleeger et al, 1997; Marple, 1997) and, in some cases,

University of Cape Town

the developers of these original models are significantly overhauling their models to try and address the latest development methods. One such example is the lines-of-code model known as COCOMO (COnstructive COst MOdel), developed by Boehm (1981), and which he is updating now as the COCOMO II model. One of the significant new characteristics of this new model is specific support for component-based development methods. Further, the International Function Point User Group (IFPUG) official counting rules manual has been updated in recent times to partially address the issue of reusability of software objects and components.

To demonstrate how inappropriate these old estimation methods are for reusable component-based development projects, consider the following. A project has ten tables of data that can be browsed by a user, where each table has similar but slightly different data. Using the source lines of code methods such as COCOMO (Boehm, 1981) and SLIM (Putham, 1979), one must count the number of source lines of code for each of the ten tables. But when developing these programs with reusable components, a developer uses only one standard table component and only changes the data definitions for the table each time. In other words, the lines-of-code estimate will work out approximately ten times higher than it should be. This naturally has significant implications for estimates using the old methods with these new development techniques.

For the purposes of this dissertation, the term "object" is taken to be synonymous with "component" and the word "component" will be used for the remainder of the dissertation. This decision has been taken because the dissertation views all reusable elements of programming similarly and the term "object" is normally associated only with software source code routines or screen elements. The dissertation will also consider elements such as

University of Cape Town

data elements and business rules and as such, the term "components" was chosen as it encompasses a broader definition.

Given the above background, the statement of the dissertation problem can be summarised as:-

> *A repeatable estimation methodology is required that will a) consider the use of modern tools with their propensity for reusable software components and b) provide an acceptable level of accuracy for the software organisation.*

In order to address the dissertation problem, a survey of academic papers summarising consensus as to which characteristics are required in order to achieve an acceptable level of estimation accuracy will follow. Thereafter, it will be important to review a background to components and component-based development as well as the history of estimation methodologies. From this it will be possible to extract the most useful elements to use in a proposed new methodology that supports reusable components.

A new estimation methodology will be proposed taking into account the collective wisdom attained from the above. As the methodology is defined, the motivation for its various processes will be explained by referring back to the above background information.

In order to demonstrate the applicability of the new methodology, a software tool has been developed by the author and a research assistant. The working and physical implementation of this tool will be discussed in detail. By making use of the tool to estimate projects, one can

University of Cape Town

test the accuracy of the methodology by comparing the actual time taken for development with the estimated time reflected by the tool.

As this is a "mini-dissertation", limited testing of the methodology has occurred to date. However, the results of the testing that has occurred will be presented. Finally, potential weaknesses and future areas of improvement will also be discussed.

University of Cape Town

Chapter 3: Requirements for a Successful Estimation Methodology

This chapter will propose a set of characteristics required of an estimation methodology in order to produce successful software estimates. The characteristics have been determined by reviewing a wide range of literature on software estimation and software metrics. The characteristics can be broadly divided into two categories - those practices that will increase the likelihood of maintaining project timings close to the estimate (positive characteristics) and those that reduce the likelihood of maintaining project timings close to an estimate (negative characteristics). It therefore follows that one should promote positive characteristics and attempt to reduce the effects of the negative characteristics in order to raise the accuracy of estimates.

A set of five characteristics is proposed for a successful estimation methodology as shown below in Table 1. The five have been selected after a review of 15 research papers wherein these particular characteristics were repeatedly deemed important by the researchers. Supporting discussion for each characteristic follows hereafter.

1	The methodology should specify that a metrics program be established that collects data relating to the efforts of the software development organisation. Such metrics should be stored in a database and easily accessible. The data within the database should be used to form the basis for future estimates.
2	Where possible, collection of the estimation metrics data must be automated and should not be a burden for the team member. However, the person should be aware that data collection is occurring and well-informed as to how and what the data will be used for. Collection of metrics data should occur continuously during the development project.
3	The methodology should be well documented and reduce the human influence on the outcome of the estimate by both providing a series of steps that are independent of human subjectivity but still allow for human-related factors such as interrupts or individual productivity rates. This will allow estimate results to be easily repeatable by different estimators.
4	The methodology must be flexible enough to take into account risks that might delay projects and further, be able to adjust easily should these events occur.
5	The methodology should be supported by all players including management and instituted as a mission-critical part of the day-to-day culture of the software organisation.

Table 1: Characteristics required for a successful estimation methodology

University of Cape Town

1) A metrics program should be established that collects data relating to the efforts of the software development organisation

Putnam and Myers (1997) suggest that one of the most important criteria for successful estimation is a repeatable software development process. Putnam and Myers refer to a study by the Software Engineering Institute, where it was found that two-thirds of organisations studied had no such repeatable software process in place and suggest that therefore these organisations would have difficulty calculating accurate estimates for future projects.

As a result, the report further suggests that a historical record of previous efforts should be kept and that metrics from previous projects are vital for planning the next one. Specifically they say "To estimate the time and cost of next time, you must know and be able to repeat what you did last time." (Putnam and Myers 1997, p 106)

The metrics suggested as important to measure and record are:-

- The amount of function built or modified, typically in source-lines-of-code or function points
- Development time – the amount of time in months in the main build stage
- The effort, which equates to the cost of the project and can be calculated as:- Person months x labour rate
- Process productivity, which is the rate at which work is accomplished
- Defect rate, which is a quality measure, typically of number of defects per unit measure such as function point.

Lederer and Prasad (1992), Park et al (1994), Vigder and Kark (1994) and Pressman (1992) also support the use of such metrics and their retention in databases for later use. In particular, Pressman states that "Software metrics are used as a basis from which estimates are made" (Pressman, 1992, p 43).

University of Cape Town

Having reviewed other estimation methodologies, it is clear that none of them explicitly require the software organisation to collect and retain project metrics for future estimates. In some methodologies, this is implied. For example, a lines-of-code-based methodology such as SLIM or COCOMO would expect an estimator to judge the number of lines-of-code required for a new project based on their experience of previous projects. However, this is not a formal requirement of the methodology.

2) Where possible, collection of the estimation metrics data must be automated and should not be a burden for the team member.

Hall and Fenton (1997) suggest that the data collection for the metrics program should be automated to minimise extra work for the developers. Without this automated data collection, they suggest developers may resist such data collection efforts. Hall and Fenton are also careful to note that the developers must be fully aware of such data collection efforts and ideally, should take part in the development of such metric collection programs.

This is supported by Offen and Jeffrey (1997) who suggest that specific tools for automated metrics collection be implemented. They also suggest that the data collection is integrated into the regular software development process. The rules of the International Function Point User Group (IFPUG, 1996) also subscribe to this, stating that such measurements should form part of the culture of the organisation and should be integrated into the development process.

University of Cape Town

3) The methodology should be well documented and reduce the human influence on the outcome of the estimate such that estimate results are easily repeatable by different estimators.

One of the reasons that both COCOMO and Function Point Analysis are popular methodologies is that they are both well-documented. COCOMO is available in the public domain and is comprehensively documented in the book - *Software Engineering Economics* (Boehm, 1981). The Function Point Analysis methodology is available to members of the International Function Point User Group (IFPUG), which has published a *Counting Practices Manual* (IFPUG, 1994). This manual provides a formal set of methods for performing function point counts.

Park et al (1994) at the Software Engineering Institute suggest that organisations should develop a set of policies for estimation standards. This is echoed by Vigder and Kark (1994), who propose that organisations should formalise when and how estimates are performed through the provision of policy documents.

However, even if the methodology is well-documented, it has been shown above that estimates can be influenced by human factors such as subjectivity. This would reduce the repeatability of most of the existing methodologies i.e. one estimator is unlikely to get the same result as another due to subjective parts of the methodology. A simple example of this would be a lines-of-code based methodology such as COCOMO. If one estimator decided a project would have 100 000 lines of code, while another decided it would have 200 000 lines of code, the difference in result is readily apparent.

University of Cape Town

van Solingen et al (1998, p 103) highlight the impact of human factors by quoting Fenton and Pfleeger as saying:-

> "For many years, sociologists have studied personal attributes, both as individuals and teams, and their effect on productivity and products. These characteristics include age level and type of education, intelligence, gender, marital status, type of remuneration and more. Although we do not yet measure these aspects of software developers and their work habits, it is clear that when researchers find them relevant to other professions, they are likely to be relevant to ours."

In the review of estimation methodology weaknesses in Chapter 6, we will see that many aspects of existing methodologies are vulnerable to human subjectivity.

Most directly relating to the estimation process, Lederer and Prasad (1991) reported a political factor relating to software estimation. According to their research, depending on which 'player' within an organisation performs the estimate, the final result is more or less likely to be padded or reduced in size. The authors indicated that there were six such players:

1) User management – executives responsible for the application area of the business.
2) User representatives – the liaison between the application area and the IS department
3) IS management – responsible for the output from the IS department
4) Estimators – planners within the IS department, but who do not do development
5) Developers – those that perform the development process
6) Estimator-developer – those that perform both the estimate and the development of the project.

University of Cape Town

Of these players, the authors found that estimator-developers padded their estimates more than all others, while user representatives and user management shrink estimates more significantly than any of the IS representatives. This is not surprising as it is likely that developers would want to give themselves as much time as possible in an estimate to ensure that they deliver within the estimate timeframe. Alternatively, the user representatives would want to minimise estimates, both to reduce the system cost and to receive the system as soon as possible. The authors also found that IS management were the most politically-orientated group in the estimation process, encouraging higher estimates when they wanted projects delivered on time, and they shrank estimates when they wanted to undertake more projects.

The researchers suggest that if management wishes to reduce padding of estimates, they should separate the estimation process from the developers. However, in a later paper by the same authors (Lederer and Prasad, 1992), it was recommended as a guideline that the initial estimation task should be assigned to the final developers. This may potentially promote padding of estimates but will result in the final responsibility for delivery of the software being left to the developer - because the blame for a late delivery cannot then be easily placed on someone else.

University of Cape Town

What this research does highlight is that estimates can be manipulated to suit individual objectives. Ideally then, an estimation methodology should endeavor to minimise such practices wherever possible. This should be achieved by removing as much subjectivity as possible from the methodology implementation steps.

While the above highlights the impact of human influence on the estimate, one should also account for the human influence on the actual time spent on the project development. van

Solingen et al (1998) report that software projects can be significantly delayed by interruptions suffered by the programmer and that such interruptions can increase the delivery time of projects by as much as 20%. The researchers describe an interrupt as any distraction that causes a developer to stop performing their planned activity and respond to the interrupt's initiator. Examples of this include unsolicited phone calls or even a cup of coffee offered by a colleague. It is suggested that an interrupt-awareness program (where the programmers are informed about the potential delays caused by interruptions) should be implemented. By raising the awareness of how interrupts cause delays in software delivery, developers will be encouraged to limit the number and effects of such interrupts within the software development process.

Additionally, Boehm and Papaccio (1988) report that one should work to get the best from one's developers through careful selection, motivation and management of staff. In this way, productivity can be maximised – allowing actual development times to better match estimates.

These issues highlight the impact of human influence on estimation efforts. While an estimator does not have control over matters such as motivation or management of staff, the methodology should reduce the subjectivity required from the estimator as well as take into account real-life issues such as interrupts that affect the developers. By doing this and providing a well-documented implementation guide, there would be an increased likelihood of different estimators agreeing on estimates given the same project specifications.

University of Cape Town

4) The methodology must be flexible enough to take into account risks that might delay projects and further, be able to adjust easily should these events occur.

Känsälä (1997) developed a set of 164 potential risks that are likely to delay software projects and therefore raise the chances that the estimate for the project is overrun. Examples of these risks include volatility of requirements (requirements creep), availability of key staff, interfaces to other systems and unnecessary features. Känsälä warns that existing cost estimates are based on the assumption that no exceptional problems will occur during the project. As a result, Känsälä suggests that estimation methodologies should take into account risk in some form to provide more accurate results.

In certain cases, the risk of project delay, and therefore difference between actual and estimated project delivery can be reduced through careful project management. Requirements creep, the practice of adding functionality over and above the original specification, is considered a practice that should be minimised as it generally extends project delivery times beyond the initial estimate. (Jones, 1998; Prasad and Lederer, 1992; Vigder and Kark, 1994). Thus, wherever possible, no new functionality should be added to a project without significant justification as well as a re-estimate of the newly functional project.

Further management practices that can lead to improved correlation between actual development times and estimated times include writing less programming code, avoiding rework and developing an integrated project support environment (Boehm and Papaccio, 1988). The first two items mentioned can be considered negative characteristics because one should be minimising them in order to improve programmer productivity and thereby improve the chances of having an estimate and actual software development time correspond.

University of Cape Town

These are particularly important characteristics when considering more modern software development methods which, through the use of reusable components, allow the developer to reduce the number of code lines they write and, automatically avoid rework by reusing these components effectively.

No methodology can directly prevent project-delaying events such as requirements creep from occurring. However, a methodology should make it easy to adjust estimates if such events do occur and take into account the risk of such occurrences in the initial estimation process.

5) The methodology should be supported by all players including management and instituted as a mission-critical part of the day-to-day culture of the software organisation.

Lederer and Prasad (1992) suggest that management should carefully study and approve estimates. Their research showed that estimates specifically approved by management were more accurate than those that were not approved by management.

Park et al (1994) found in a study of 249 government, military and industry computer managers that while it was important for estimation and related metrics collection to be an ongoing management goal, these activities should also become a routine activity within the software organisation.

Vigder and Kark (1994) of the National Research Council of Canada concur with both of these issues and propose that clear and proper project management procedures should be

University of Cape Town

implemented. These procedures should allow managers to be aware of the state of projects including costs, completion level and estimates of effort required to complete the project.

These issues are also addressed by the IFPUG (1996), who recommend the following guidelines as part of 11 characteristics required for an effective measurement program for estimation purposes:-

- Measurements should be integrated into the development process as opposed to being seen as a separate process.
- Management must be seen to be supporting the metrics effort.
- Measurement should form part of the culture of the organisation.

Finally, Hall and Fenton (1997) suggest that where developers are involved in the setting up of such programs, they are less likely to criticise the results.

Thus a successful methodology should form an integral part of the development process, supported by both management and developers, as a requirement of the methodology.

University of Cape Town

Issues

A key aspect of this review of characteristics required for a successful estimation methodology is that data relating to the development effort must be collected, ideally automatically. This is potentially a sensitive issue as developers may feel that a "Big Brother" approach is being adopted i.e. they are being watched for the purposes of evaluating their efforts.

While the researchers mentioned in this review have indicated strongly that automated data collection is a significant requirement, several have indicated the importance of developer co-operation.

George (1996) reported on five case studies where computer-based monitoring was implemented. He found that, while in some cases monitoring was used in a punitive manner, the implementation was malleable. George suggests that management have a key role in designing monitoring systems that are effective but not viewed as too onerous. In doing this, George reports that monitoring schemes may not only be tolerated but also approved of by workers.

Fenton and Hall (1997) suggest that the collection process must be completely transparent and that all participants should be made fully aware of what and how data is being collected. They further suggest that developers should be involved in the design of the metrics program to ensure buy-in.

Offen and Jeffery (1997) also subscribe to the idea that everyone involved in the data collection process should know what is being measured. Also they say that developers should be adequately trained in software measurement and that constructive feedback is provided to those being measured.

The crux of this issue is whether the results of such data collection should be used for evaluating developer performance. IFPUG (1996) are specific in their answer to this. They state, "One should measure processes, not people" (IFPUG, 1996, p.3-2).

University of Cape Town

This is supported by Offen and Jeffery (1997), who argue that the data collection program should not be used to assess individuals.

Lederer and Prasad (1992) however, propose exactly the opposite. In their list of nine management guidelines for improving cost estimates, they specifically suggest that such data should be used to evaluate project personnel, perhaps providing pay increases for those who have produced within a certain range of a project estimate. It is possible, however, that a number of developers would object to being evaluated in this way.

Thus we have a requirement for successful estimation methodologies that must be carefully presented to developers in such a way as to gain acceptance and not be viewed as a potential demotivator.

University of Cape Town

Chapter Conclusion

We have summarised the most important characteristics required for a successful estimation methodology. The following chapter will provide an overview of software components and component-based development. Thereafter, we will be in a position to review and evaluate a number of existing estimation methodologies in regard to their compliance with these characteristics and their suitability to component-based software development.

Chapter 4: Component Based Development Background

After a review of academic literature, there does not appear to be a generally accepted definition for component-based development (Kozaczynski and Booch, 1998). However, in order to propose an estimation methodology for such development projects, it is important that this form of software development is defined adequately. In the previous chapter, three different methodologies made reference to such development.

For the purposes of this dissertation, it is important that a specific definition is provided for component-based development. In order to achieve this, we must first define the word "component" in the context of software development.

University of Cape Town

What is a Component?

During the 20th International Conference on Software Engineering, delegates attempted to find a generally acceptable definition of what constitutes a software component (Brown and Wallnau, 1998). While delegates could not agree on a single definition, the following three definitions were proposed by the group:-

- A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realisation of a set of interfaces.
- A component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime.

- A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third party composition.

Pancake (1995) describes components as units of functionality that can be adopted and used by other programmers both inside and outside the developer's organisation. Further, since components are both modular and independent, the are well suited for reuse. However at the same time, they can also be modified through an easy set of rules.

From the above, it is clear that no one definition describes components in a way generally accepted by all. It is also clear that these definitions refer specifically to the functional nature of components used in the software development process and do not take into account the data elements of a given system. The importance of data as a component type of its own is reflected in several estimation methodologies such as COCOMO II and GUI-Point analysis where the number of data tables is specified as one of the elements counted within the methodology implementation. While the functional description of components is very important, a description of the role of data manipulated by software, as happens in all software systems, should also form part of the definition.

Therefore a dual description of components is proposed for the purposes of this dissertation as follows:-

Functional Component Description

A functional software component is an independent pre-written module that has specifically documented functions and interfaces. Such a module is available as a

University of Cape Town

reusable building block for multiple software projects. The functions contained within the component may be utilised by linking the module to the main body of the software system and making calls to the starting points of each function as required. The behavior of the component may be modified through the setting of documented properties within the module. Various entry points are made available to the software developer to either disable, override or enhance each of the available functions within the component.

Data Component Description

A data component is one of the items within a software system that describes the information manipulated by the system, methods for traversing the information, or relationships within the information used by the system. Specifically such items include files (tables), fields (columns), sort keys and indexes, views, relationships and any other data structures.

University of Cape Town

With components defined as indicated above, a component-based development definition can now be sought. Three existing estimation methodologies, which will be reviewed in detail in the next chapter, make reference to development using software components.

Morgan (1997) based her COSTMO-4GL estimation methodology on 4GL components. Specifically, Morgan writes that 4GL applications can be characterised by the "structural components" of forms, reports, tables and modules. For the purposes of this dissertation, using this as a definition for component-based development was thought to be limited by this small number of component types and that this should be broadened to cater for non-4GL components as well.

Marple (1997) based the presentation layer tier of his GUI-Point estimation methodology on a count of graphical components. These included items such as buttons, pick lists, and widgets (preprogrammed graphical objects with a specific use). However for the other two tiers – business rules and database tiers – Marple does not refer to components, rather using the algorithm section of Jones' (1998) Feature Point Analysis for the business rules tier and Albrecht's (1979) logical files and external interfaces count for the database tier. Since this is clearly a partial use of components and only for the graphical interface, this was not deemed a useful approach of component-based development.

The COCOMO II model (Boehm, 1998) proposes a methodology specifically for applications "composed from interoperable components". Similar to Morgan's COSTMO-4GL counting practice, the Application Composition model within COCOMO II calls for the count of screens, reports and 3GL components. These can be compared to the forms, reports and 4GL modules within the COSTMO-4GL model. The COCOMO II Application Composition model's method of counting Object Points has been derived almost solely from research conducted by Banker et al (1991) where Integrated Computer Aided Software Environmental (ICASE) tools were investigated.

Banker et al (1991) reported that ICASE development environments differ from 3GL environments in two ways:-

- "Structurally, since systems can reuse the designs and functionality of existing systems through reusable software modules and routines; and,

University of Cape Town

- Functionally, since the tools that support CASE software development are quite different from those used in traditional development and actually change the process itself." Banker et al (1991, p 129)

From this basis, Object Point Analysis was proposed whereby a count of ICASE software objects is conducted and where such objects reside in a central object repository. The balance of the methodology proposed by Banker et al (1991) now forms part of the Application Composition model used in COCOMO II.

While Banker et al introduced the concept of breaking applications down into a group of reusable objects, it was specifically restricted to the ICASE development environment. For example, there is no mention of objects as defined by object-oriented development (Booch, 1986).

Therefore, as in the case with COSTMO-4GL, the Application Composition model described within COCOMO II does not satisfactorily encompass a complete definition of component-based development.

Smith et al (1998) describe component-based software development as applications comprised of interacting, independent components. These components may be commercially available software modules (termed "off-the shelf"), internally developed reusable components or newly developed software artifacts. This represents a good general definition of such development because the number and types of components are not limited. However this definition does raise a challenge with regard to estimation. The definition combines all elements that might be termed "components". This means that both custom (one-off)

University of Cape Town

modules especially written for an application and reusable modules need to be combined into an estimation model. As a result, an estimation methodology will be required that will take into account any number and type of software components.

With the definition of functional and data components provided earlier in this chapter, we are able to describe both the functions of a system as well as the data modified by the system. Extending this further, we can now define component-based development in an extended version of Smith et al's definition as : -

> Software applications developed with interacting independent components. These components may be commercially available components, internally developed reusable components, newly-developed software artifacts and all data components used within the application.

University of Cape Town

An Example of a Component-Based Development System

It is instructive to provide an example of such components as used in a real-life system. In order to show this, a commercially available component-based development tool, Clarion 5, has been selected. A simple invoicing program is presented and is comprised of several built-in components, termed "templates" by the Clarion documentation.

These standard functional components include the following:-

- Frame – a menu structure for selecting and calling different functions
- Browse – a screen that provides viewing of data in a scrolling list format

- Form – a screen that provides update (insert, modify, delete) functions for data records
- Report – a process that allows one to preview and print a set of data formatted as required

The standard data components are defined and subsequently accessed through a central "Data Dictionary". These include:-

- Files – a collection of data fields grouped as a logical unit
- Fields – data variables and their attributes such as validation rules and display settings (e.g. font, point size)
- Keys – a set of separate ordering tables to allow fast access to data in pre-defined sequences
- Relationships – a set of rules governing how fields interact with each other across data collections

The functional components are customised to the needs of the invoicing application by setting their data properties and where necessary, modifying their functional behavior through the embedding of custom-written software at specified entry points within the component framework.

Thus using only these components, it is possible to create a simple invoicing application. Using a set of screens captured from the Clarion 5 development environment, it is possible to demonstrate the concepts employed.

University of Cape Town

The application is assumed to have at least the following characteristics:

1) The system will retain a list of customers, invoice headers, invoice details and product details.
2) Each new invoice entered will require a valid customer ID and invoice number. A customer may have multiple invoices
3) An invoice may have multiple invoice detail items.
4) Each invoice detail item must include a description contained within the products list.

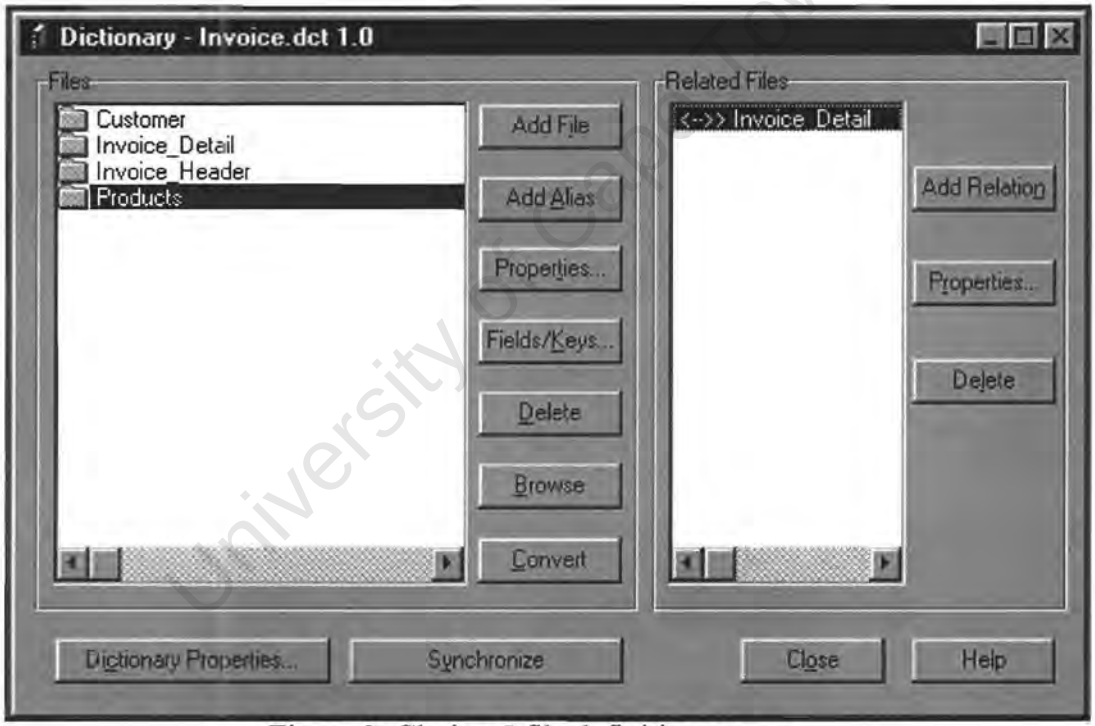

Figure 2: Clarion 5 file definition screen

University of Cape Town

In Figure 2 above, the files required for the invoicing system are shown with the 'Products' file component highlighted. These are the basic file data components and are stored within the data dictionary. Within each file component is contained a number of fields and sort keys as shown in Figures 3 and 4 below.

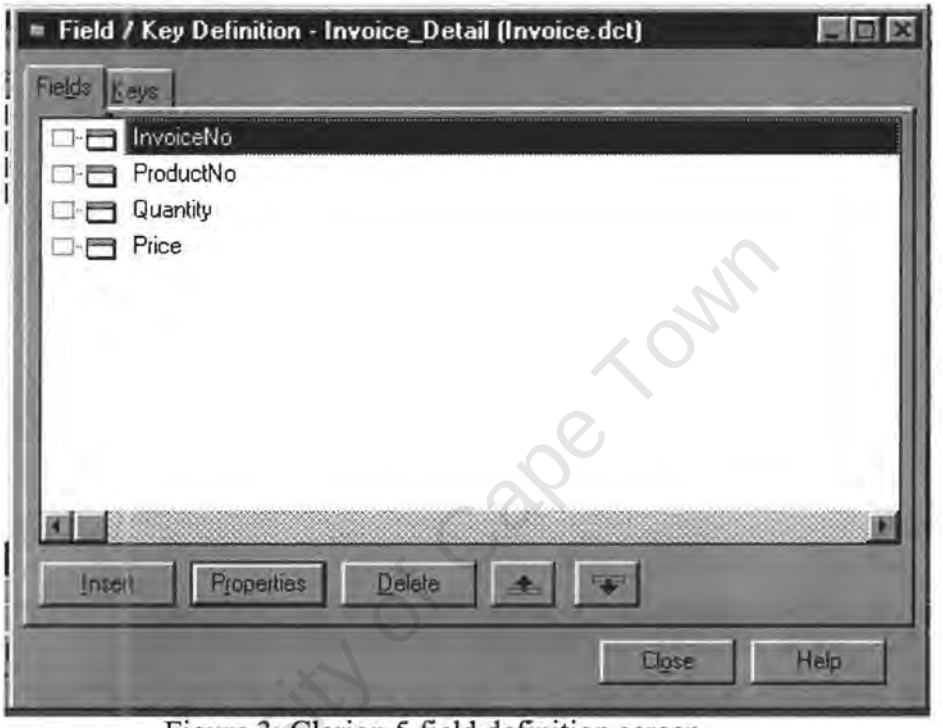

Figure 3: Clarion 5 field definition screen

University of Cape Town

In Figure 3 above, the field data components within the 'Invoice_Detail' file component can be seen with the 'InvoiceNo' field component highlighted.

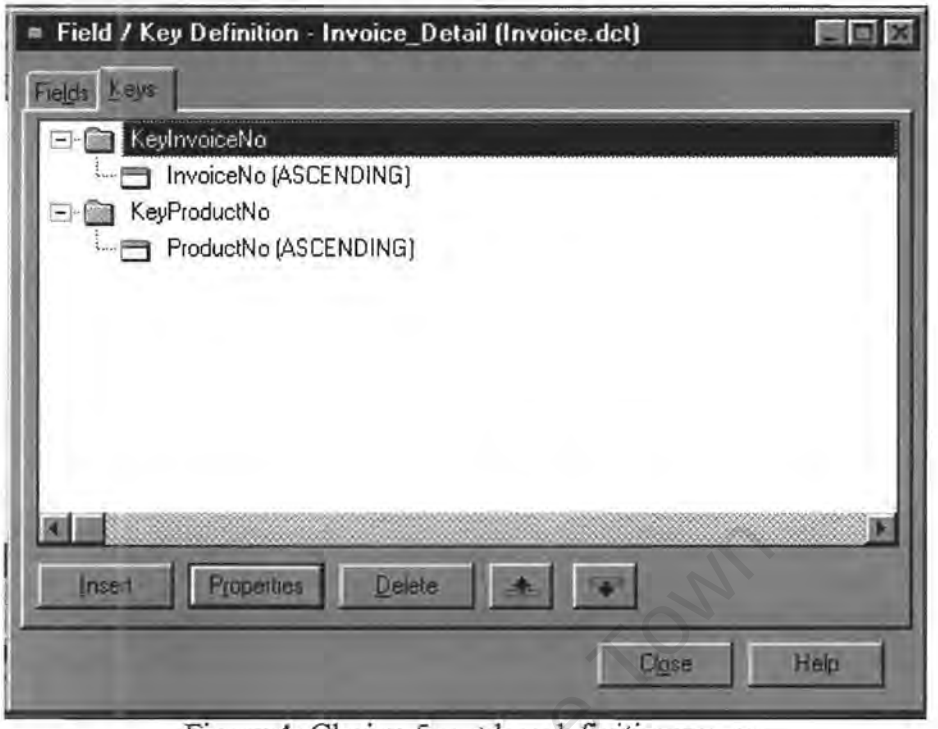

Figure 4: Clarion 5 sort key definition screen

In Figure 4 above, the key data components within the 'Invoice_Detail' file component can be seen with the 'KeyInvoiceNo' key component highlighted.

Finally, within the data dictionary, the relationship data components are defined. These dictate the rules as how data in one file may be affected by a data change in another. A relationship definition screen example is provided in Figure 5. This screen provides an example of a single relationship data component.

University of Cape Town

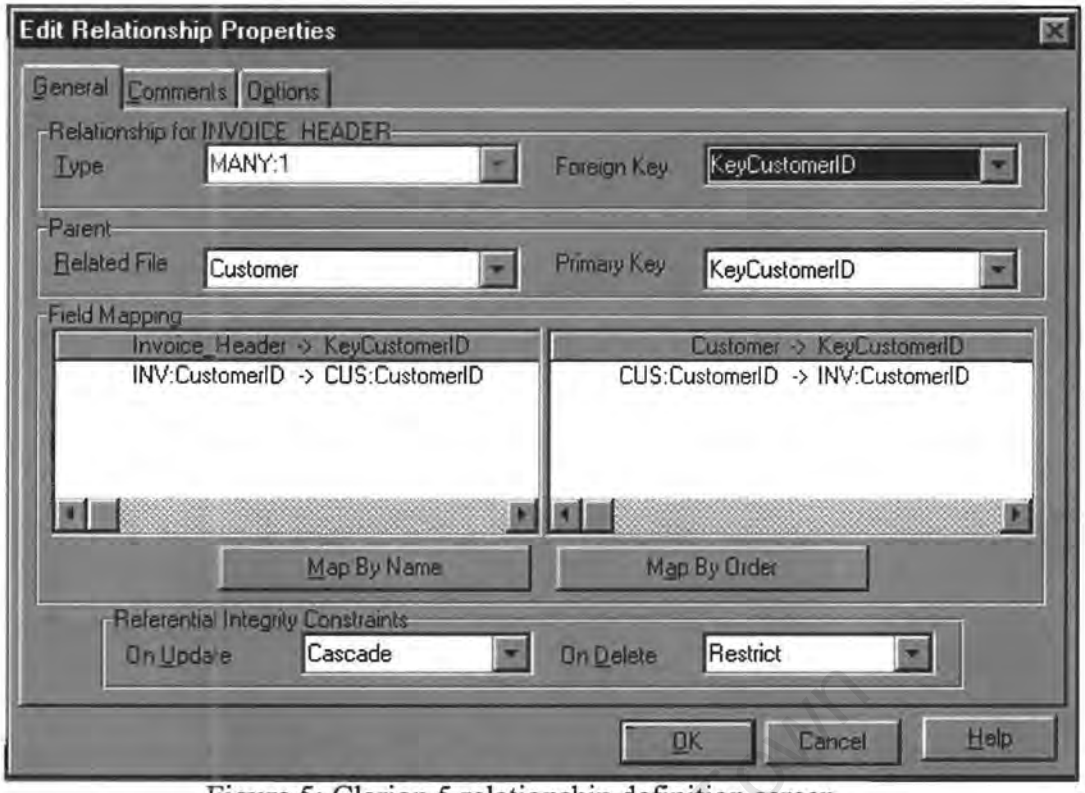

Figure 5: Clarion 5 relationship definition screen

With the data components defined, the application is then created by combining a set of built-in functional components that come with the Clarion development toolset. These include the components called 'Frame', Browse, 'Form' and 'Report' – see Figure 6.

Figure 6: Clarion 5 application procedure tree screen showing components

Figure 6 provides an example of a fully developed invoicing system, which is comprised of a "Frame" component, seven "Browse" components, five "Form" components and three "Report" components. These are all the functional components of the system. Each "Browse" procedure (e.g. "BrowseCustomer" or "BrowseProducts") represents a unique instance of the general "Browse" functional component supplied with the Clarion 5 toolset.

As per the definition of functional components, these Clarion 5 components can be modified to fit the particular requirement of the application. An example of a screen to effect such modifications is shown below in Figure 7.

Figure 7: Clarion 5 component modification screen

From Figure 7, we see that the "Browse" component "BrowseCustomer" may be customised in several ways. The data components manipulated by this functional component can be declared by pressing the "Files" or "Data" buttons. The screen seen by the end-user is painted by selecting the "Window" button. Further functional components may be called by pressing the "Procedures" or "Extensions" buttons. Each component used within the system has a

similar set of property screens, each specific to its functionality. While the above is specific to Clarion 5, any component-based development environment will provide similar component-customisation facilities.

Programming Language Considerations

It should be noticeable that throughout the invoicing application example provided above, there has been no mention or view of source language statements. If one were to use non-component-based tools, each part of the application, except for the visual screens, would have been shown as lines of language code. In a component-based system such as Clarion 5, the lines of programming language have been pre-written by the developers of the product. The programmer creating the system merely customises this pre-written code by modifying the behavior of the component as seen above.

This does not mean, however, that all language programming is removed. In the case of Clarion 5, there are several "Embed" points built into each component that allow a developer to drop a chunk of custom programming code into a specific area of the component, thereby modifying its behavior further. The "Embed" button can be seen on the Figure 7 screen. If one presses this button, the screen shown in Figure 8 is seen. This represents all the entry points to the "BrowseCustomer" instance of the built-in "Browse" component where the basic component functionality can be overridden removed or enhanced.

University of Cape Town

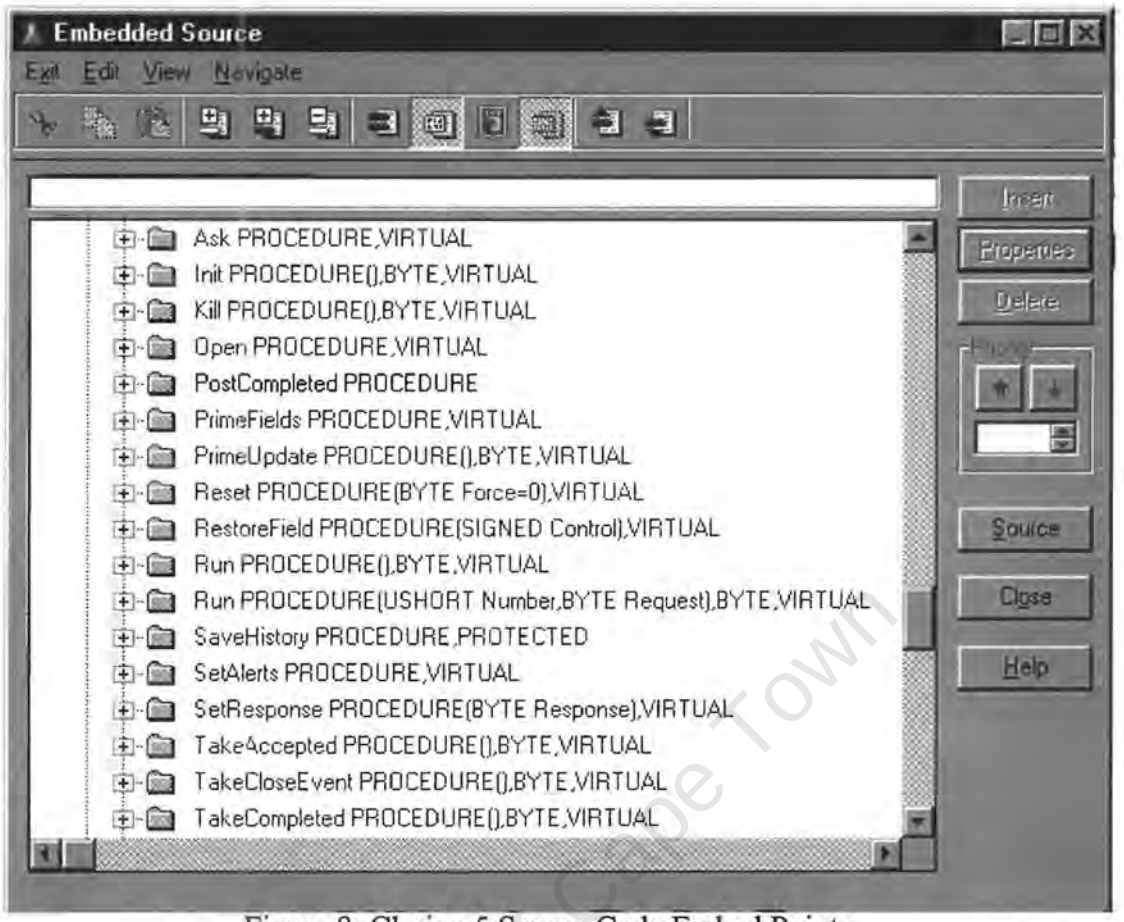

Figure 8: Clarion 5 Source Code Embed Points

Within Figure 8, we can see an Embed Point called "PrimeFields". This is an example of an entry point into the component where a developer can initialise the variables used within the component to any required values. The "Open" Embed Point would allow a developer to enter language statements at the point immediately after the component's visual screen had been displayed.

Thus while even pre-written components may require a certain amount of unique language statements in order to customise an application to the requirement, this is substantially less than the non-component-based alternative, which requires by far the majority of the application to be coded with custom language statements and modules.

University of Cape Town

This difference between the development methods (component vs non-component-based development) epitomises the promise of component-based to reduce the time and cost of software development. It also represents an important differentiator for estimation efforts. This will be discussed further in the next chapter.

Component Reusability

The discussion so far nas addressed components in general. The thrust of modern software development revolves specifically around *reusable* software components. The intention of this is to reduce software development time and therefore the cost by using pre-built and pre-tested components.

A review of the literature has provided little in the way of a definition of what constitutes a reusable component versus a non-reusable component. Within an object-oriented context, Pancake (1995) reports that objects must go through a specific "generalisation" process in order to become reusable. The key aspect of such a generalisation process is the specification of the component's interfaces i.e. how external modules will interact with the component and which data components will be affected.

Thus a reusable component can be defined as a component (data or functional) that has been specifically generalised for use in more than one software project.

University of Cape Town

Chapter Conclusion

In order to address the issue of software component-based estimation, it has been necessary to define components and component-based development. After reviewing a number of definitions from the literature, it seems that no single definition for software components exists and further, that data components were not accounted for within these definitions. Therefore new definitions for components, both functional and data were provided as well as an extended definition of component-based development. Reusable components have specifically been defined as a subset of the general component definition.

A key characteristic of component-based development has also been highlighted. This is that such development significantly reduces the amount of effort expended on writing custom lines of language code. A review of the literature did not provide any research figures as to how much effort had been saved through the use of components. However, the company that develops the Clarion 5 toolset has suggested that, by using components, the amount of effort to create applications can be reduced by 90% when compared to developing without using components i.e. using only language source statements to create an application.

With these definitions, we now proceed to review several existing estimation methodologies, followed by their suitability to component-based estimation and their compliance with important estimation methodology characteristics.

University of Cape Town

Chapter 5: Overview of Existing Estimation Models

According to Stutzke (1997), the formal study of software estimation did not occur until the 1960's. During the 1960's, Frank Freiman developed the concept of parametric estimation, which in turn led to the development of the "PRICE" model for hardware. The PRICE model was extended in the 1970's to handle software. A company was formed (as with several other estimation models) to offer estimation services based on the PRICE models. The company, "PRICE Systems" now offers several cost estimation models for deriving cost and schedule estimates for electronic, mechanical, aerospace and software projects. All PRICE models are based on Computer Aided Parametric Estimating (CAPE) methods. These make use of characteristics that can be readily quantified, such as weight and size, to estimate variables that are difficult to quantify, such as cost and production schedules.

During the 1970's, the requirement to provide better estimates grew in importance and more research was conducted accordingly. During the late 1970's, Lawrence H Putnam used theoretical methods to develop a model called "Software Lifecycle Model" (SLIM). The SLIM model depends on Source Lines of Code (SLOC) to gauge a project's general size and then makes use of a commonly used statistical distribution curve, called the Rayleigh curve, to produce its effort estimates (Kemerer, 1987). Parr (1980) later adapted this curve used by Putnam to develop a model better suited to smaller applications and where the team is partially in place at the start of the project.

Also at the end of the 1970's, Allan Albrecht developed Function Point Analysis for estimating the size and effort of software projects. This method has become particularly

University of Cape Town

popular with a large following through its International Function Point User Group (IFPUG) (Jones, 1998).

In 1977, Halstead produced a theoretical method of sizing software projects termed "Software Science". Under this model, he defined the size as a function of the "operators" and "operands" used in the program (Albrecht and Gaffney, 1983). He then proposed a set of relationships in order to estimate the development time and effort. Halstead's model applied mainly to small programs or functions and one is required to combine the results across all functions in a larger system in order to gain an overall result.

In 1981, Barry Boehm published his "COCOMO" model based on his analysis of 63 software development projects. The COCOMO model has become one of the most important forms of a lines-of-code (LOC) estimation model due to the fact that it is comprehensively documented in the book "Software Engineering Economics" (Boehm, 1981). Boehm's work was extended during 1983 with the US Department of Defence's use of the ADA language. This update became known as "ADA COCOMO".

Several researchers have extended these works and have then gone onto create commercial offerings, resulting in limited research papers on their specific findings. Robert Tausworthe further extended the work of Boehm and several others (Putnam, Wolverton etc.) during his work at the NASA Jet Propulsion Laboratory (Stutzke, 1997). Donald Reifer further extended this into a commercial product called "SOFTCOST-R". A further example of this is work conducted by Randall Jensen, where he extended the SLIM model of Putnam. This updated model is currently sold under the name "Software Estimation Model" as part of the "SEER" toolset.

University of Cape Town

Updated or enhanced versions of existing methodologies include the "COCOMO II" model from Boehm. This research is currently being conducted at the University of Southern California. Jensen continues to enhance his models and has developed a model termed "SAGE" to specifically handle the effects of management on software project cost and schedule (Stutzke, 1997). Jones with Albrecht enhanced the original Function Point Analysis model to create the Feature-Point model in 1986 (Jones, 1998).

More recently, new estimation methodologies have been proposed to try and address newer development methods. These include GUI-Point Analysis (Marple, 1997), COSTMO-4GL (Morgan, 1997) and an updated COCOMO II from Boehm (1998).

Other techniques that can be used to determine software project size, cost, effort and duration include (McLeod and Smith, 1993):

- Wideband Delphi technique - Whereby a group of experts are polled for their opinions in an iterative process in order to reach general consensus.
- Work Breakdown Structures - The project is broken down into measurable blocks of work effort to complete each task, rather than by program function of number of source lines. The benefits of this approach include the reduction of subjectivity and ensuring that all major activities are satisfied.
- Analogy and Experience - Under this method, projects are characterised in such a way that 'experts' can search for analogous or similar situations for which effort and cost is known (Sheppard and Schofield, 1996).
- Iterative (Maintenance) models - This method employs a continuous estimation process where each iteration improves the accuracy of the estimation. Abdel-Hamid (1993)

University of Cape Town

provides an argument for three types of maintenance model, the Adaptive for new models, Corrective for those in which errors have been discovered and Perfective, for those projects nearing completion and requiring removal of inefficiencies.

Using the theoretical estimation model proposed in the previous chapter, we will now review a selection of the most popular estimation methodologies (Jones, 1998) as a well as a group of more modern methodologies. The popular methodologies will include Boehm's COCOMO, Albrecht's Function Point Analysis and Putnam's SLIM. The more modern methodologies will include Feature-Point Analysis (Jones, 1998), GUI-Point Analysis (Marple, 1997), COSTMO-4GL (Morgan, 1997) and the COCOMO II model.

In chapter 6 we will question the usefulness of each methodology with respect to component-based development projects.

University of Cape Town

The COCOMO Model

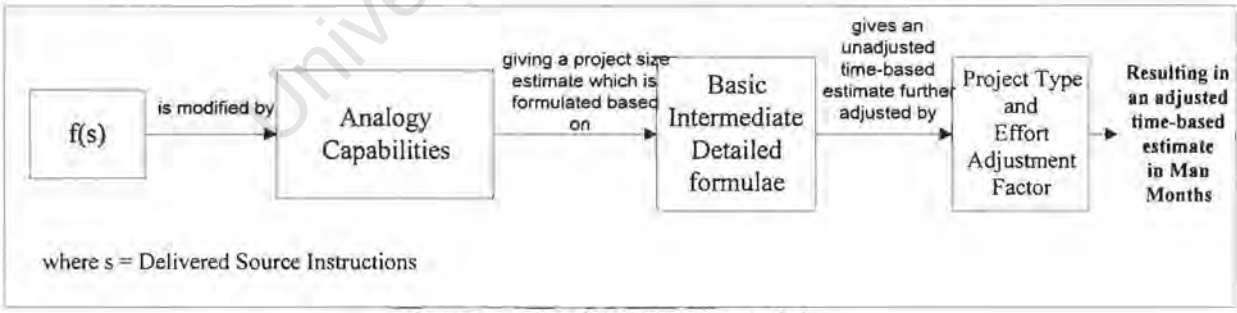

Figure 9: The COCOMO model

The COCOMO model is based primarily on a measure known as "Delivered Source Instructions" (DSI). This represents the sizing seed value for the input of the model. It is important to define the meaning of DSI. COCOMO defines DSI as:-

- "*Delivered*. This term is generally meant to exclude non-delivered support software such as test drivers. However, if these are developed with the same care as delivered software, with their own reviews, test plans, documentation, etc., then they should be counted." (Boehm, 1981, p58)

- "*Source Instructions*. This term includes all program instructions created by project personnel and processed into machine code by some combination of pre-processors, compilers and assemblers. It excludes comment cards and unmodified utility software. It includes job-control language, format statements and data declarations. Instructions are defined as lines of code or card images." (Boehm, 1981, p59)

The determination of the count of DSI is left to the estimator. This leaves significant room for subjective opinion. Ultimately the sizing seed value is as good as the estimator's ability to compare the forthcoming project size with previous efforts. Historical records detailing previous DSI counts would be useful in assisting estimators, but much is still left as to how well the estimator can find analogy between the historic records and the new system.

The COCOMO model provides three levels of estimation: Basic, Intermediate and Detailed. These correspond to the complexity characteristics of the model. Each level provides increasingly more adjustment factors in order to provide a more accurate estimation.

The first product of the estimation activity provides a measure of the basic effort using a particular equation defined by Boehm. Apart from the three levels of estimation, Boehm also provides for three different modes of development. These are Organic, Semi-detached and Embedded systems where Organic systems are considered those small-to-medium sized projects developed with in-house skills and familiar software, Embedded systems are defined as ambitious and tightly constrained, and Semi-detached is considered to be between the first two types. (Boehm, 1981, p74). The effort equations for the Basic model are shown in Table 2.

University of Cape Town

Development Mode	Basic Effort Equation
Organic	$MM = 2.4 \times (KDSI)^{1.05}$
Semi-detached	$MM = 2.4 \times (KDSI)^{1.12}$
Embedded	$MM = 2.4 \times (KDSI)^{1.20}$

Table 2: COCOMO Basic model Effort Equations by system type

These development mode equations provide one of the adjustment factors for the COCOMO model i.e. depending on what complexity project is being undertaken, the effort equations are adjusted accordingly.

The result of these effort equations provides a measure of man-months (MM) - defined as 152 hours of working time by Boehm.

Both the Intermediate and Detailed models introduce additionally the Effort Adjustment Factor (EAF) and slightly modified coefficients as shown below in Table 3 for the Intermediate model.

University of Cape Town

Development Mode	Intermediate Effort Equation
Organic	$MM = EAF \times 3.2 \times (KDSI)^{1.05}$
Semi-detached	$MM = EAF \times 3.0 \times (MM)^{1.12}$
Embedded	$MM = EAF \times 2.8 \times (MM)^{1.20}$

Table 3: COCOMO Intermediate model Effort Equations by system type

The EAF is calculated by determining the product of 15 separate cost driver multipliers (Boehm, 1981, p118). Examples of these cost drivers are database size, product complexity and programming language experience. The estimation process used for the Intermediate and Detailed models allows for the system to be split into modules, which can then be attributed to their specific set of cost drivers, thereby improving the accuracy of the estimation.

The Detailed model defines specific phases for each system and assigns different cost driver multipliers to each phase. The phases defined for the Detailed model are Requirements, Product Design, Detailed Design, Code and Unit Test, Integrate and Test, and Maintenance.

The COCOMO model, and others like it, were developed based on statistical evaluations of analysed projects during the 1970's (Stutzke, 1997). Boehm's COCOMO model was developed based on his analysis of 63 software development projects (Boehm, 1981). The adjustment factors used in COCOMO were derived by using correlation techniques, which were then incorporated into the model using regression techniques (Stutzke, 1997).

Function Point Analysis Model

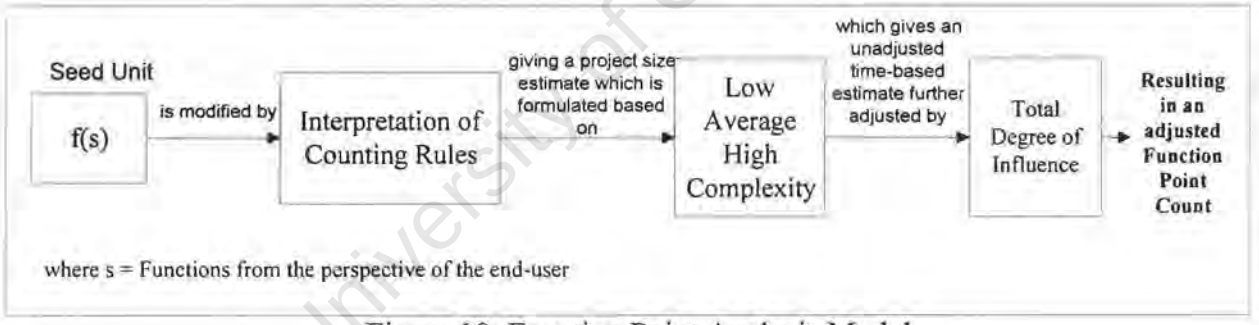

Figure 10: Function Point Analysis Model

In October 1979, AJ Albrecht published a paper titled *"Measuring Application Development Productivity"* which proposed a method of estimating the size of a software development project regardless of the software language or environment. Albrecht's seed unit of measure is the function point and this method of software project sizing has become well accepted in industry (IFPUG, 1994; Jones, 1998; Heller, 1996).

University of Cape Town

Function points are intended to be units of an application that are either provided or received by a user such that the user can identify them. These are categorised into five separate units of function which are defined by the IFPUG (International Function Point User Group) "Function Point Counting Practices Manual" as:-

- Internal Logical Files (ILF) - This represents the group of data that is used and maintained by the application.
- External Interface Files (EIF) - These functions represent a store of data used by the application but for which the user is not responsible for their maintenance.
- External Inputs (EI) - This represents data used by the application but that is provided from an external source, rather than being maintained internally as in the case of the ILF.
- External Outputs (EO) - An EO is a product produced by the application and provided to the user in various formats. These outputs are derived using the ILF and EI data described above.
- External Inquiries (EQ) - These functions allow for the user to request specific outputs from the application data.

University of Cape Town

In the initial stage of estimating the software project size, one is required to take a view of the application from the perspective of a user and break it down into a number of functions in each of the above five function types. This represents the sizing seed value for the theoretical model shown in Figure 10. The initial sizing function count and categorisation is influenced by the estimator's view of what a user would consider a function as well as what type of function it represents. The IFPUG Counting Practices Manual provides a well-documented set of rules by which an estimator is meant to perform such counts, but there are gaps in the

counting rules. A specific example of this is how the analysis caters for reusability of components. It does not. The only reference to reusability occurs in the adjustment factors and then, only if one is *creating* a new component to be reused in the future (IFPUG, 1994, pp 7-12).

Once this initial step has been completed, the "Functional Complexity" for each value is determined. A set of functional complexity factors has been defined by the IFPUG and each set is based on a combination of data groupings, termed "File Types Referenced" and "Data Element Types". From this set of factors, one determines the relative complexity of the particular function category. The measure of complexity is characterised as Low, Average or High. As an example, the set of factors for External Inputs is shown below in Table 4 (IFPUG, 1994).

	1 to 4 DET	5 to 15 DET	16 or more DET
0 to 1 FTR	Low	Low	Average
2 FTR	Low	Average	High
3 or more FTR	Average	High	High

where FTR = File Types Referenced, DET = Data Element Types

Table 4: External Input Complexity Matrix (IFPUG, 1994)

This "Functional Complexity" calculation represents the complexity segment of the theoretical model in Figure 10.

Once the relative complexity has been determined for each category of function, a multiplier factor is then applied to each category function count, which are then summed to provide an

University of Cape Town

"Unadjusted Function Point Count". The multipliers for each category are provided below in Table 5 (IFPUG, 1994).

Function Type	Functional Complexity	Multiplier
Internal	Low	x 7
Logical	Average	x 10
Files	High	x 15
External	Low	x 5
Interface	Average	x 7
Files	High	x 10
External	Low	x 3
Inputs	Average	x 4
	High	x 6
External	Low	x 4
Outputs	Average	x 5
	High	x 7
External	Low	x 3
Inquiries	Average	x 4
	High	x 6

Table 5: Unadjusted Function Point Multiplier

University of Cape Town

The adjustment factor represented in the theoretical model is termed the "Processing Complexity" of the application. In order to determine this, fourteen system factors are provided. One rates each factor on a scale of 1 (no influence) to 5 (extremely strong influence). Once the ratings have been determined for each of the factors, they are summed to provide a "Total Degree of Influence" (TDI). The fourteen factors are :-

1) Data communications

2) On-line update

3) Distributed processing

4) Complex processing

5) Performance

6) Usable in other applications

7) Heavily-Used Configuration

8) Installation ease

9) Transaction Rates

10) Operational ease

11) On-line Data Entry

12) Multiple sites

13) Design for end-user efficiency

14) Facilitate change

These factors may adjust the unaffected function point count by up to 35% either positively or negatively (Low and Jeffery, 1990). However, like the determination of the number of source lines of code, the determination of the these influencing factors allows for subjectivity on the part of the estimator.

The final stage in sizing a software project using this method is to apply the figures to the following equation :-

$$\text{Total Function Point Count} = \text{UFPC} \times ((\text{TDI} \times 0.01) + 0.65)$$

where UFPC = Unadjusted Function Point Count
TDI = Total Degree of Influence.

Software Lifecycle Model (SLIM)

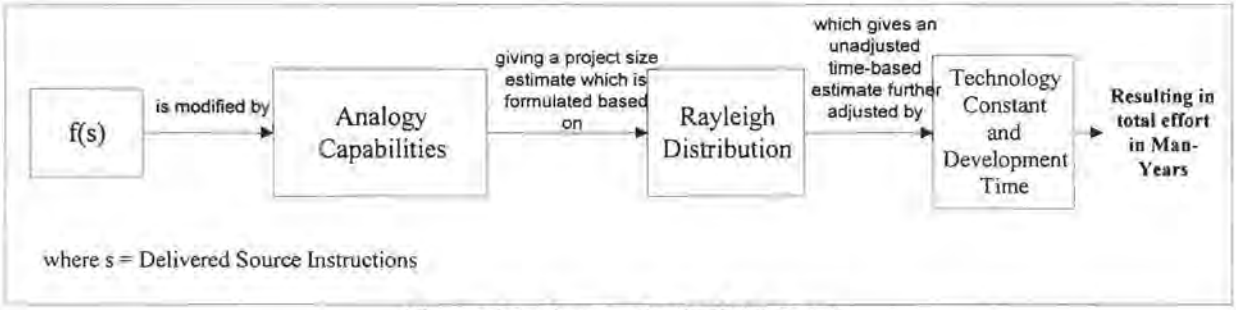

Figure 11: Putnam's SLIM Model

University of Cape Town

The SLIM estimation methodology is based on the Rayleigh statistical distribution model. The methodology calculates the total or "life-cycle" effort in man-years for a given project. As a result, SLIM is often used for determining staffing levels rather than determining the development time of a project as one might do with COCOMO (Marple, 1997).

Like COCOMO, SLIM makes use of "Delivered Source Instructions" as its sizing seed value. This also relies on the ability of the estimator to accurately estimate the number of such source instructions. Once the number of DSI has been determined, the estimator applies the following formula, termed the "Software Equation" by Putnam, to determine the life-cycle effort in man-years:-

$$K = \frac{S_s}{C_k t_d^4}$$

Where K= life-cycle effort in man years
S_s = number of delivered source instructions
C_k = the Technology Constant
t_d = development time in years

SLIM differs significantly from both COCOMO and Function Point Analysis in that one requires both the DSI count and the expected development time for the project in order to ascertain the effort in man-years. This allows an estimator to select a set of development time periods and then determine the corresponding set of effort results. Theoretically, this allows a software organisation to specify a delivery date for a project as long they then apply the resultant man-power effort calculated from the Software Equation.

Researchers (Boehm 1981; Marple 1997) have questioned the relationship between the development effort K and development time t_d. This is because, according to this relationship, one can halve the man-year effort required for a project simply by increasing the

University of Cape Town

development time by 19%. This highly linear rule would seem to be questionable given that the complexity of software projects increase significantly with size and duration.

In terms of the theoretical model, once the DSI count has been established, one uses the Software Equation to determine the effort. This equation represents the Rayleigh statistical distribution which can then be adjusted according to two parameters, the Technology Constant C_k and the development time t_d. The Technology Constant is determined by considering several factors relating to the characteristics of the project. These include hardware constraints, use of modern programming practices and programmer experience. C_k ranges in value from 2000 for a poor software development environment to 11000 for an excellent development environment (Pressman, 1992). The determination of the Technology Constant relies on a subjective assessment of the development environment characteristics and therefore also raises questions about the usefulness of the SLIM methodology.

University of Cape Town

Feature Point Analysis Model

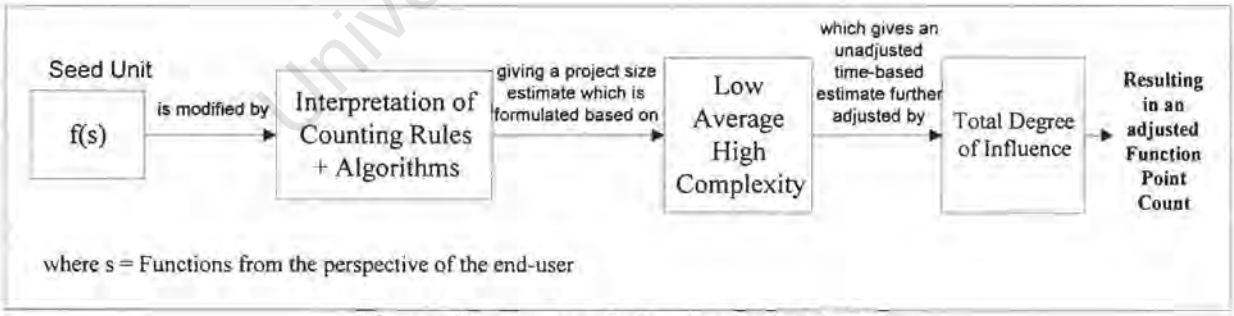

Figure 12: Feature Point Analysis Model

Feature Point Analysis is a modification of the Function Point Analysis methodology. Developed by Jones and the Albrecht in 1986, Feature Points were proposed to encompass software projects that had a high number of algorithms. While Function Point Analysis was developed for Management Information Systems (MIS), it was found to be consistently

inaccurate for embedded, real-time and system projects (Jones, 1998). In order to address this, Feature Point Analysis adds a sixth parameter to the functional units of Function Point Analysis. This new function unit is the count of algorithms within the system. Further, Feature Point Analysis modifies the complexity multiplier of the Data Files such that the results will be very similar to Function Point Analysis for all projects except those with a high number of algorithms.

Functional Unit	Function Point Analysis Average Multiplier	Feature Point Analysis Average Multiplier
Inputs	x4	x4
Outputs	x5	x5
Inquiries	x4	x4
Data Files	x10	x7
Interfaces	x7	x7
Algorithms		x3

Table 6: Comparisor of Function Point and Feature Point average complexity multipliers

As a result, the theoretical model for Feature Point Analysis is almost identical to the Function Point Analysis model except for the additional functional unit and different complexity multipliers. Because algorithms have been added as a functional unit, the term "algorithm" needs to be defined as it does not appear in the IFPUG counting manual.

Jones (1998) defines countable algorithms as follows:-

- The algorithm must deal with a solvable problem.
- The algorithm must deal with a bounded problem.
- The algorithm must deal with a definite problem.
- The algorithm must be finite and have an end.
- The algorithm must be precise and have no ambiguity.
- The algorithm must have an input or starting value.

University of Cape Town

- The algorithm must have output or produce a result.
- The algorithm must be implementable, in that each step must be capable of execution on a computer.
- The algorithm can include or call upon subordinate algorithms.
- The algorithm must be capable of representation via the standard structured programming concepts of sequence, if-then-else, do-while, CASE, etc.

Feature Points represent a good example of how estimation methodologists often use a base methodology and effectively calibrate it to better suit a particular type of development project. However, Jones (1998) is careful to say that Feature Points are still experimental at this time. Further, this calibrated model will generally suffer the same fundamental weaknesses of the base model such as interpretation of the function point counting rules. Marple (1997) has also suggested that Feature Points do not adequately cater for graphical-user-interface (GUI) applications that are so common today.

The methodologies presented thus far were developed between 1979 and 1986. Component-based development methods were not available during this time and this raises the question as to whether these methodologies can cater for this type of development (Jones, 1998; Morgan, 1997). However, more recent methodologies have been proposed and we now review them in the context of our theoretical estimation model below.

University of Cape Town

GUI-Point Analysis Model

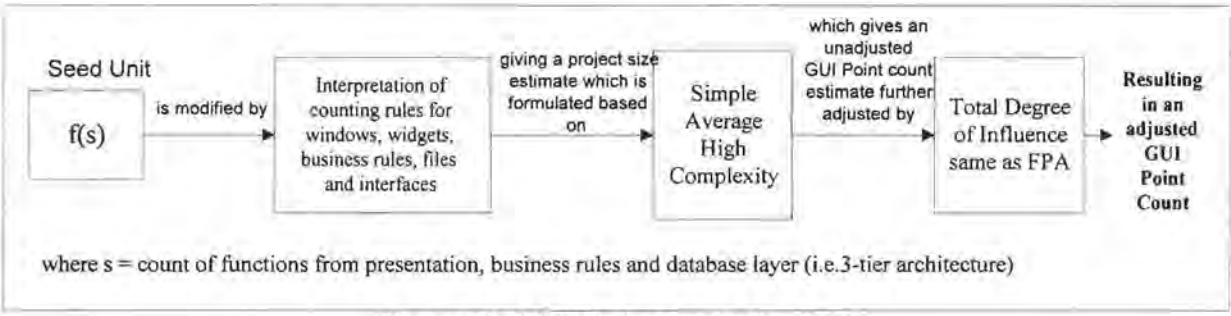

Figure 13: GUI-Point Analysis model

GUI-Point analysis proposed by Marple (1997) makes an attempt to create a more accurate estimation model for 3-tier architecture systems. The 3-tier architecture logically partitions an application into presentation, business rules and database tiers. These tiers can reside on physically different computers or processors and thereby have their performance optimised through dedicating computing resources to their specific function. Marple indicates that 3-tier systems have been common and that due to the split between tiers, a different estimation methodology approach is required.

He thus proposes a modified function-point analysis (FPA) methodology. Instead of the standard five functional types used in FPA, Marple introduces three new functional types for the presentation tier of a 3-tier system and one new functional type for the business rule tier. Marple retains the logical files and external interfaces counts used in FPA to cater for the database tier. The functional complexity table developed by Marple is shown in Table 7:-

Complexity	Simple	Average	Complex
Number of Windows	x 1	x 1	x 1
Number of Static Widgets	x 0.5	x 0.5	x 0.5
Number of Dynamic Widgets	x 1	x 2	x 4
Number of Business Rules	x 1	x 3	x 5
Number of Logical Files	x 5	x 7	x 9
Number of External Interfaces	x 5	x 7	x 10

Table 7: GUI-Point Functional Complexity multiplier table

University of Cape Town

These multiplication factors are used to determine the unadjusted GUI-Point count. From this value, the same adjustment factors are used as for FPA to generate the adjusted GUI-Point count. In order to determine these multipliers, Marple took a set of six 3-tier application and counted the number of lines of code to implement each type of function. He then averaged the result for each complexity and finally used a conversion table developed by Capers Jones (1996) to convert the source line count into function points. This conversion method from source lines to function points and vice-versa is termed "Backfiring" by Jones. From this it was possible to determine the multiplier values.

Marple suggests that 3-tier systems are popular in object-oriented development. While this may be the case, the model only partially addresses reusability, that of GUI components, which Marple terms "widgets'. The use of backfiring to determine the multipliers is also questionable as the usefulness of the backfiring conversion table from Jones is a highly debatable issue. This issue will be discussed further in the chapter 6 when the various methodologies are evaluated for component-based development methods.

University of Cape Town

COSTMO-4GL Model

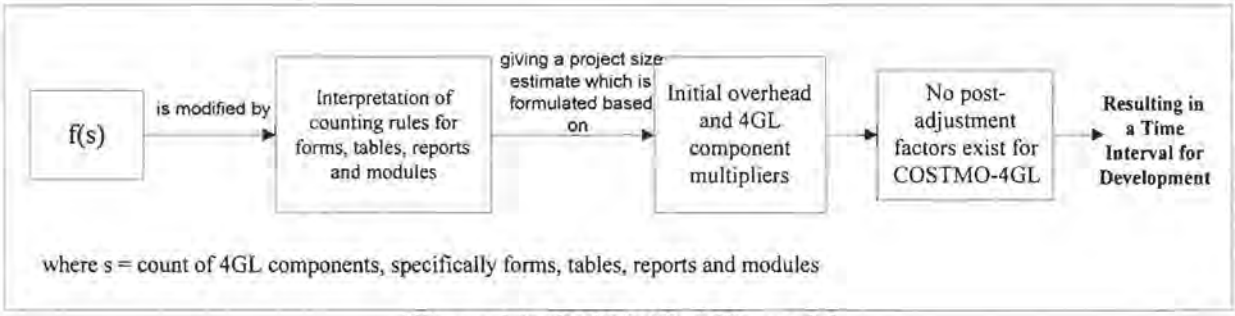

Figure 14: COSTMO-4GL model

Proposed by Morgan (1997), the COSTMO-4GL estimation methodology is one of first to specifically address component-based development methods. Using fourth generation

languages (4GLs) as the basis for her estimation methodology, Morgan proposed that all 4GL applications are comprised of four basic building blocks – forms, tables, reports and modules. The selection of these four items was based on earlier research by Verner (1991). Using a simple count of these items, Morgan proposed that an estimate for development time can be determined by using the following equation:-

$$LOE_p = a + b_1*Forms + b_2*Reports + b_3*Tables + b_4*Modules$$

Where LOE_p = number of person days to complete the project
a = the y intercept
b_1 = number of person days to develop one form
b_2 = number of person days to develop one report
b_3 = number of person days to develop one table
b_4 = number of person days to develop one module

For the purposes of evaluation it is important to describe Morgan's definitions (1997, p139) of each of these 4GL building blocks.

- Forms are objects created by the developer for interaction or navigation by the user. The 4GL provides templates for the user to use. These templates can be used as menus or for data entry and query screens or other forms required by the application
- Reports are objects that a developer uses to retrieve data from tables, which are then formatted and presented to the user. The 4GL tool provides an easy-to-use interface and ready-made products which allow the developer to create tailored reports.
- Tables are objects used to store data. The 4GL tool provides utilities to create the tables automatically from design models.
- Modules represent that portion of a software development application that cannot otherwise be delivered except to be created. These may include algorithms,

University of Cape Town

transaction handling or processes. These modules may be constructed using a 3GL such as C.

Morgan's methodology is significantly different to the others represented by the general theoretical model because she does not first develop a size estimate and then apply adjustment factors to determine a final development time. Instead, Morgan proposes that the *y* intercept, the constant component of the equation, will include any such influences on the result. She specifically refers to application complexity and programmer experience as example of such influences.

Although Morgan has made an effort to address the needs of estimating component-based application development, the limited influence of adjustment factors and the limited number of component types used in the model suggest a limited usefulness for this model. These limitations will be discussed further in Chapter 6.

University of Cape Town

COCOMO II Model

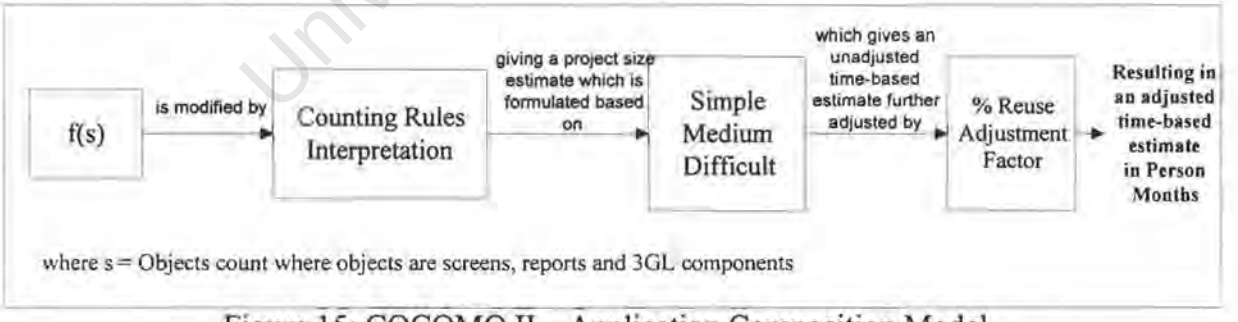

Figure 15: COCOMO II – Application Composition Model

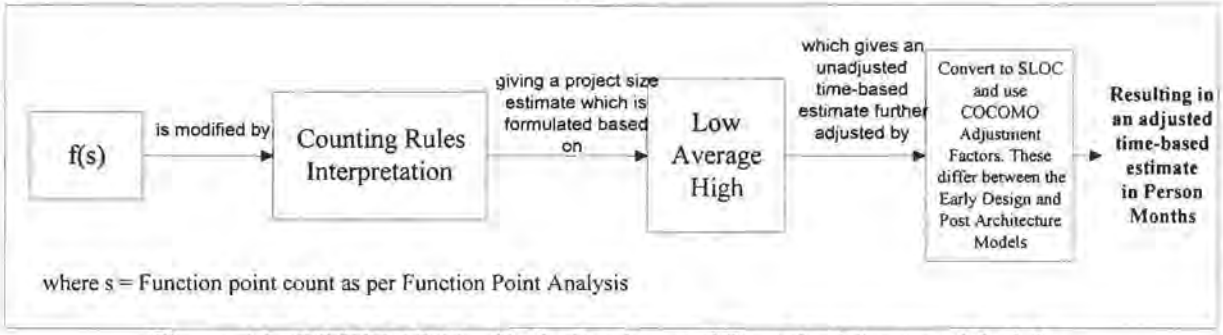

Figure 16: COCOMO II – Early Design and Post Architecture Models

The COCOMO II model is currently being developed by Barry Boehm (1998) at the University of Southern California as an update to Boehm's 1981 COCOMO model. In this updated effort, Boehm recognises that not all software projects can be categorised and therefore estimated in the same way. Therefore the COCOMO II model proposes three different estimation models. They are the Application Composition, Early Design and Post Architecture models. These are presented within the estimation theoretical model diagrams in Figure 15 and 16 above. The Early Design and Post Architecture models share a diagram as they only differ in their adjustment factors.

The Application Composition model addresses those applications that can be composed from interoperable components. The COCOMO II model definition manual (1998) suggests that examples of these component-based systems include GUI-builders, database or object managers, middleware for distributed or transaction processing as well as domain-specific components for financial, medical and other applications. The model uses an object count as its sizing seed. COCOMO II uses research by Banker et al (1991) who proposed an Object Point count for CASE system components. Within this context, objects are defined as screens, reports and 3GL modules.

University of Cape Town

Once a count for each of these three categories of objects have been performed, a complexity rating is determined, similar in nature to the function points model, but based on a count of data tables and views based on both the client and server computers. A complexity rating of simple, medium or difficult is determined and a multiplier is assigned for each of the three object types as shown in Table 8.

Object Type	Simple Complexity	Medium Complexity	Difficult Complexity
Screen	x1	x2	x3
Report	x2	x5	x8
3GL Component			x10

Table 8: Complexity multiplier table for Application Composition model

The basic object point count is given as:

$$\text{Object Count} = \text{Screens} * m_1 + \text{Reports} * m_2 + \text{3GL Components} * m_3$$

where m_1 = screen complexity multiplier
m_2 = report complexity multiplier
m_3 = 3GL component complexity multiplier

The final estimate in Person-Months is determined applying adjustment factors for reuse and developer experience as shown below:-

$$PM = \frac{(\text{Object Count}) * (100 - \%\text{Reuse})}{100 * PROD}$$

where %Reuse = the reuse you expect to be achieved in this project
PROD = the developers experience and capability and tool capability

Both the Early Design model and the Post Architecture model make use of Unadjusted Function Point Count (UFPC) as their sizing seed. The COCOMO II model then suggests that Function Point Analysis' use of adjustment factors is inadequate and proposes that those adjustment factors proposed in the original COCOMO would be more useful. As such, in both models, the UFP are converted to source lines of code using Jones conversion tables.

University of Cape Town

From there the two models make use of different lines of code-based adjustment factors to determine the estimate. In the case of the Early Design model, only 7 factors are used. This is because the Early Design model is proposed for those projects where little is known about the project details, but early estimates are required to make a decision about its feasibility.

The Post Architecture model uses 17 adjustment factors. This model is proposed for projects where a software life-cycle has been developed. It is to be used in the development and maintenance of software products in the application generators, system integration and infrastructure sectors.

The COCOMO II model is important in several aspects. It concedes that earlier models, including its predecessor COCOMO, are no longer valid for modern development methods and makes an attempt to encompass a variety of existing estimation concepts. It also recognises that one model does not suit all software development projects. Importantly, in the context of this dissertation, it specifically recognises that component-based development requires its own estimation methodology with its own sizing seed unit type.

Since Function Points are determined from the user's perspective, it is appropriate that they should be used for the Early Design model. However COCOMO II's use of Jones' conversion tables, as in the case of Marple's GUI-Point methodology remains questionable and potentially downgrades the value of the Early Design and Post Architecture models.

University of Cape Town

Chapter Conclusion

In this chapter, a number of estimation methodologies have been reviewed in the context of the theoretical estimation model. These have included both popular older models and more recent models. It is clear from the more recent estimation models that researchers have seen the need to address new software development methods. In particular, component-based development is targeted as an area requiring a new estimation methodology. Both Boehm and Morgan directly address this, while it is partially addressed by Marple with his count of screen widgets and business rules.

Although these attempts have been made to address component-based development, there are weaknesses evident in the methodologies. These weaknesses will be described in detail in the following chapter.

University of Cape Town

Chapter 6: Suitability of Existing Estimation Methodologies to Component-Based Software Development and Successful Estimation Characteristics

In Chapter 5, we described six estimation methodologies. Although, in the case of the older methodologies, several researchers have analysed weaknesses (Matson et al, 1994; Finnie et al, 1997; Kitchenham, 1997; Pfleeger et al, 1997), for the purposes of this chapter we will specifically review their suitability for component-based development estimation and their compliance with the five characteristics required of a successful estimation methodology described in Chapter 3.

Rather than review each methodology separately, they will be grouped together based on their type of initial sizing seed count and a more general evalution of the group will follow. By doing this we are able to evaluate all the methodologies within three groups.

University of Cape Town

Characteristic	Methodologies Containing Such
Use of Source Lines of Code (SLOC) count	COCOMO, SLIM
Use of Function Points (FP) counts	FPA, Feature Points, GUI-Point, parts of COCOMO II
Use of Components/Objects counts	COCOMO II Application Composition Model, GUI-Point, COSTMO-4GL

Table 9: Groupings of methodologies based on type of initial sizing seed count

Use of Source Lines of Code (SLOC)

SLIM and the original COCOMO methodology as well as the Early Design and Post Architecture models within COCOMO II are based on lines-of-code. In the case of component-based development, it is very important to note that Boehm, the originator of the COCOMO methodology, has seen fit to develop a methodology specifically for component-based development within the later COCOMO II model. This methodology does not use

source lines of code at all and represents a strong argument that lines-of-code is not a suitable characteristic on which to base such estimates.

Further, from the review of component-based development in Chapter 4, we saw that this type of development specifically aims to reduce the quantity of custom-written code lines in a given application. With the emphasis on reusing components, and therefore significantly reducing the lines of code within those, it is also obvious that a count of source lines is not relevant to component-base development. This is supported by Morgan (1997) who reported that although both 4GL and object-oriented software development require source code to be written, it is an insignificant portion of total development.

In Table 10 below, SLOC-based methodologies are compared with the 5 estimation methodology success characteristics to assess their compliance.

Characteristic	Methodology Requirements	Compliance
1	Establish metrics program and history database	Not a specific requirement for any of the methodologies
2	Automate data collection	Not a specific requirement for any of the methodologies
3	Well-documented and not susceptible to human subjectivity. Should be easily repeatable by different estimators.	SLOC-based estimates call for a highly subjective initial count of lines required for a given project. Typically the SLOC-based methodologies are well-documented
4	The methodology must be flexible enough to take into account risks that might delay projects and further, be able to adjust easily should these events occur.	SLOC-based estimates have no specific method for taking risk into account. They will adjust with certain delaying events such as requirements creep but are similarly prone to subjective factors when selecting how many new lines of code are needed for the new features
5	Must be instituted as a mission-critical part of the culture of the organisation	Not a specific requirement for any of the methodologies

Table 10: Compliance of SLOC-based estimation methodologies with success factors

University of Cape Town

From the above, it is clear that any estimation methodology based on source lines of code is weakened by the subjectivity factor of the seed unit count and completely unsuitable for component-based development projects. It is therefore unlikely to produce adequate estimate accuracy for such projects.

Use of Function Points (FP)

Estimation methodologies based on a function point count are the most popular. Apart from the original methodology developed by Albrecht, the Feature Point, GUI Point and two models from within the COCOMO II methodology make use of such counts as well.

Kemerer (1987) and Low and Jeffery (1990) found function points to be a more accurate measure of software size than source lines of code. However, Matson (1994) was critical of function points because of the variability of the results they produced. Although Matson suggested that this variability could be reduced by using a single or small group of estimators, he ultimately reported that the adjustment factors used within the function point analysis methodology (and used by Feature Points and GUI-Points as well) were the major reason for such variability.

A further criticism of function points is that different estimators will produce different counts for the function points (Kitchenham, 1997). Research of this issue has show differences in counts of between 12 and 30%. Kemerer and Porter (1992) found specific counting interpretations that could cause such differences. These included counting backup files as Logical Internal Files, whether or not menus were counted among the External Interface Files and whether help screens were counted.

University of Cape Town

Function point counts represent a sizing measure that is technology independent. As such, function point counts cannot account for component-based development benefits such as high reuse of components. In order to get an estimate of time from this sizing method, some form of conversion is required. This conversion is dependent on the language or toolset used. As a result, conversion tables are typically used to convert function points either directly to time measures or to lines of source code depending on which language is used. Thus, the conversion factor must account for component-based development benefits such as high reuse of components.

A common conversion is from function points to lines-of-code. Albrecht and Gaffney (1983) found that there was a strong correlation between function point count and eventual source lines of code. Similarly, Kemerer (1987) also found that function points can be used as a predictor of lines of code. As a result, organisations have developed conversion tables based on historical counts of lines of code and function points for different programming languages. In particular, the conversion of function points to a set number of code lines has been termed "Backfiring" by Caper Jones (1995), who has suggested that the conversion is bi-directional.

Jones has developed a table of over 400 different development languages, which he has made freely available over the Internet (Jones, 1996). These language-specific conversion rates are regularly used by organisations. In fact both the GUI-Point methodology and the COCOMO II models of Early Design and Post Architecture make specific reference to Jones' table.

However, while researcher's during the 1980's found a correlation between function points and lines of code, this correlation must be questionable given modern tools such as component-based development toolsets. Taking two different examples, Clarion 5 from

University of Cape Town

TopSpeed Corporation and Delphi from Inprise Corporation (formally Borland), both products allow the developer to choose whether they will use component-based, object-oriented or procedural development methods – or a combination of these.

However, Jones' conversion table for these products offers only one conversion rate (average source statements per function point) for each product. In the case of Clarion 5, this rate is 58 and for Delphi, it is 29. For the purposes of estimation, this single rating of tools with multiple development mode capabilities must be questioned. It explicitly implies that the same number of lines of code will be created by a given development tool regardless of how one develops. It also then follows that an estimate for a particular project will also always be the same when a lines-of-code methodology is used to create the final result, regardless of the development methodology. This seems highly unlikely and leads one to question the usefulness of function-point to lines of code conversion or backfiring.

Added to this, Jones' specifically states that:

> "Source code inspection for common applications was done. Then the volume of code for the application in a measured language was hypothesized. ACTOR, CLARION, and TRUE BASIC are examples of languages that were inspected and their levels hypothesized by subjective means." (Jones, 1996)

And further during a discussion of how languages were researched:

> "Research was done by reading descriptions and genealogies of languages and making an educated guess as to their levels" (Jones, 1996)

Thus with an admission that such evaluations of the various languages contained guesswork and subjectivity as well as the failure to account for different development methodologies within a single tool, any estimation methodology based on these conversion tables must be called into question.

University of Cape Town

Since these conversions themselves have such problems, it is clear that estimates based on function points are not suitable for estimating component-based development projects. Both GUI-Points and COCOMO II make use of such conversion factors indicating a specific weakness in these models.

Further, function point productivity has been reported to range between one function point per programmer month to 64 (Morgan, 1997 reporting on Jones, 1995). As such, Morgan (1997) determined that the function points model, using a productivity estimate of function points per programmer month, cannot be accurately extended to estimating the cost of developing 4GL applications or system comprised of structural components.

The 14 factor Total Degree of Influence (TDI) used within the Function Point model has also been criticised. Symons (1988) argued that the use of 14 factors was limited and unlikely to be satisfactory in all cases. Symons also argued that the 0-5 range that one can assign to these factors was too simple and also unlikely to be valid for all cases. Kitchenham (1997) supported this view and also reported that there is no evidence that the TDI improves the estimates produced with function points.

Probably the greatest limitation of function point estimation with regard to component-based development is its limited support for reuse. The only formal support for reusability is within the 14-factors of the Total Degree of Influence, where reusability represents one of the factors. However, even this provides very little assistance because:

University of Cape Town

- The factor description in the IFPUG counting rules manual specifies that one rates the factor based on whether the software developed within this project will be used in other applications. i.e. It does not cater for previously-written software that is reused in the given application.
- As discussed earlier in this section, the TDI has been questioned in entirety as to how much use it provides in the whole estimation process.

Hotle (1996) suggests that both empirical and practical evidence indicate that function point analysis is well suited to OO development. Hotle reports that when using object-oriented development methods, one should stick to function point counts as the best method of estimating projects. However this suggestion is based on his premise that using function points will normalise (do not need to account for) the levels of inheritance of a particular object. This means one assumes the "black box" functionality of the entire object for estimating purposes rather than the functionality of individual methods within an object. However, Hotle fails to note that function points do not take into account the reusability factor of these objects. As such, one must question his assertion that function point analysis is well-suited to object-oriented or component-based development.

In Table 11 below, function point-based methodologies are compared with the 5 estimation methodology success characteristics to assess their compliance.

University of Cape Town

Characteristic	Methodology Requirements	Compliance
1	Establish metrics program and history database	Not a specific requirement for any of the methodologies
2	Automate data collection	Not a specific requirement for any of the methodologies
3	Well-documented and not susceptible to human subjectivity. Should be easily repeatable by different estimators.	Both the interpretation of counting rules and the setting values to compute the TDI provide for significant subjectivity within these methodologies
4	The methodology must be flexible enough to take into account risks that might delay projects and further, be able to adjust easily should these events occur.	The methodology results will adjust with delaying events such as requirements creep but are similarly prone to subjective factors for new functions. Project risk is not accounted for.
5	Must be instituted as a mission-critical part of the culture of the software organisation	Not a specific requirement for any of the methodologies

Table 11: Compliance of function point-based estimation methodologies with success factors

Certain modifications to the Function Point Analysis model have been suggested to improve its shortcomings such as Symons (1988) "Mark II" approach and Jones' (1998) Feature Points. However, no specific modifications have been proposed to significantly account for the reuse of components within a component-based development project. From this and shortcomings provided within Table 11, it seems clear that Function Point Analysis and other methodologies based thereon are not suitable for component-based development estimates and also do not satisfy the majority of the estimate success characteristics. This would include Feature Point, GUI-Point estimation as well as the Early Design and Post Architecture models within COCOMO II.

.

University of Cape Town

Use of Components/Objects

Two of the methodologies reviewed, COSTMO-4GL and the Application Composition model within COCOMO II, propose a count of objects or components as their sizing seed. In the case of the GUI-Point model, the presentation layer part of the estimate also requires a count of components, although these are specifically screen components.

Since these models are describing applications comprised of components, it seems logical that a count of such should form the sizing basis. However, in all three cases, the definition of the components is restricted to certain named types of component.

In the case of COSTMO-4GL, the only components allowed are forms, reports, tables and modules. In this model, any component not falling within the first three categories is simply "lumped" within the "modules" category. A similar situation occurs within the Application Composition model of COCOMO II. In this case the model allows for only three categories: screens, reports and 3GL modules. The GUI-Point methodology restricts its category of components only to screen types. Just as Symons (1988) argued that the 14 factors within the Total Degree of Influence of the Function Point methodology was too limiting, in all the above cases, it appears too limiting to restrict the categories of components to a certain set.

In Table 12 below, methodologies based on counts of either objects or components are compared with the 5 estimation methodology success characteristics to assess their compliance.

University of Cape Town

Characteristic	Methodology Requirements	Compliance
1	Establish metrics program and history database	Not a specific requirement for any of the methodologies
2	Automate data collection	Not a specific requirement for any of the methodologies
3	Well-documented and not susceptible to human subjectivity. Should be easily repeatable by different estimators.	Subjectivity of the seed component count exists but is lower than SLOC or function point counts. GUI-Points Analysis still retains the subjectivity within the TDI computation.
4	The methodology must be flexible enough to take into account risks that might delay projects and further, be able to adjust easily should these events occur.	The methodology results will adjust for events such as requirements creep but risk is not accounted for.
5	Must be instituted as a mission-critical part of the culture of the software organisation	Not a specific requirement for any of the methodologies

Table 12: Compliance of object/component-based methodologies with success factors

Thus, it seems appropriate that any estimation methodology developed for component-based development should use a count of components as a sizing measure. However, in order to make such a methodology more general, the number and categories of such components should not be restricted and should ideally encompass all data and functional components.

While these methodologies are more applicable to component-based development projects due to their basic seed count, they are still weak as far as the estimation success factors are concerned.

Chapter Conclusion

It has been shown that only the methodologies based on an initial seed count of components would be suitable for a component-based project estimate. However, all of the methodologies

University of Cape Town

that allow for this limit the type of components that may be used. This restricts their applicability to component-based projects, which may need to take other types of component into account. Further, none of the methodologies satisfy the 5 characteristics required for a successful estimation methodology.

As a result, the next chapter will propose a new estimation methodology that largely satisfies both component-based development projects and the success characteristics.

University of Cape Town

Chapter 7: Proposed New Reusable Component-Based Estimation Methodology - ReCom

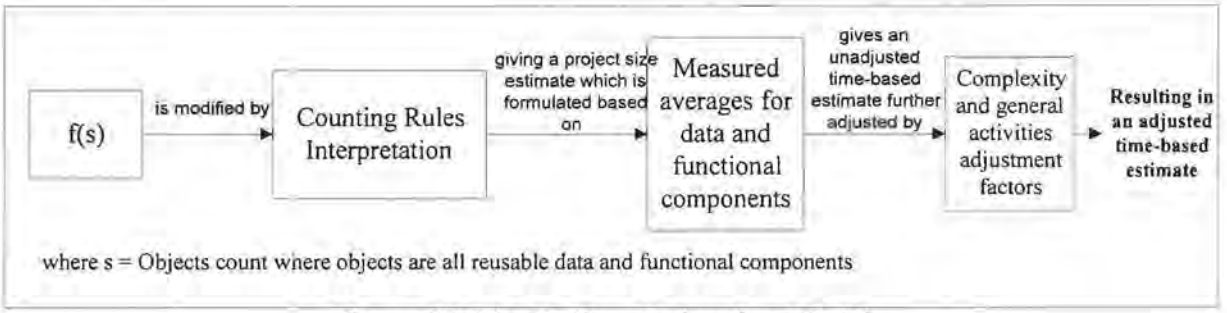

Figure 17: The ReCom Estimation Model

Figure 17 provides a diagrammatic view of a new estimation methodology that provides specifically for component-based software projects. The methodology is termed "ReCom" or **Re**usable **Com**ponent estimation methodology and its definition follows below.

The seed unit for this methodology is based on a count of all components, both data and functional. This follows the Application Composition model within COCOMO II, GUI-Point and COSTMO-4GL methodologies in this regard. What is unique to this model is that the seed components counted are not restricted as in the other models. Thus while the COCOMO II model restricts its types of components to three, COSTMO-4GL to four and GUI-Point to only screen objects, ReCom allows for a more general case. A ReCom component is defined as any component, data or functional, that is supported by the development toolset and can be reused.

This does leave room for a subjective approach as to what constitutes a reusable component. However, unlike lines of code, the components used by such development toolsets are, by their nature, more easily discernable. Therefore while there is a lack of clarity within lines-of-

University of Cape Town

code counting practices as to whether lines of test code or comments should be included, components operate at a higher level of abstraction, thereby reducing such ambiguity.

The component counting process is similar to that of function points. However there is a fundamental difference between the two in that Function Point Analysis bases its seed count on a user-perspective of what constitutes a function, thereby distancing itself from the technology used for development. Alternatively, ReCom's counting process is specific to the technology used. This is a reasonable approach as the entire basis for the methodology is the component-based mechanism.

ReCom assumes that because components used in this form of development are reused on an ongoing basis, it is possible to measure the implementation times for each type of component. As one measures these implementation times, an average time can be calculated that forms the basis of estimates for future implementations of the same component. This concept is taken directly from a number of researchers who advise that future estimates must be based on earlier efforts as discussed in Chapter 3. Importantly, the averages measured must be kept for each individual developer.

University of Cape Town

The general equation for the ReCom is:-

$$\sum_{x=1}^{n} K_{xc} * T_{xc}$$

where n = number of unique component types
K_{xc} = quantity of each type of component
T_{xc} = average time to implement the particular component for a given developer
c = component implementation difficulty (simple, medium, complex)

As a practical example of this methodology, assume that a toolset contains a reusable reporting component i.e. a component that is used to create printable reports from the system data. Every time a developer uses the reporting component, the methodology requires that the elapsed time taken to implement such a component over the entire development period be measured and stored in a database such that an average time of implementation for the reporting component can be calculated. When one needs to estimate the time required to implement reports in a new project, one multiplies the number of required reports with the average time to implement the report component taken from the database. This will yield the estimated time to create all reports for the system. A detailed example follows later in this chapter.

An important aspect of ReCom is that the averages taken should be kept individually for each developer. In this way, each developer's particular style and speed of development will be taken into account. Hihn and Habib-agahi (1991) were critical of this aspect of lines-of-code estimation methodologies, which do not account for individual programmer experience, capability and coding style. Thus while SLOC and all other methodologies group all developer efforts as equal, the ReCom provides a profile for each developer assigned to a project. This naturally raises questions about performing such estimates for a newly employed developer. This and other such issues relating to ReCom will be discussed later in the chapter. However in the case where such average measurements exist, the estimator is able to apply each particular developers' implementation times to the new estimate. This is likely to yield a more representative result than using a "one size fits all" formula as found in most of the existing methodologies.

University of Cape Town

This basic form of the methodology resolves at least two of the weaknesses found in the existing methodologies reviewed:

- The methodology uses components as its sizing seed rather than lines of code. As components are comprised of different numbers of lines of code themselves, this has the effect of sizing the application at a higher level of abstraction. This makes the counting effort far more simple. This will remove many of the subjective counting rule problems found in the other methodologies.
- The average implementation time has been selected as the key value used because, as more and more data is collected, the average will adjust to cater for all manner of issues that affect software development. These include interruptions, requirements creep and rework or corrections. As such the methodology does not have a composite adjustment factor such as COCOMO or Function Points Analysis, which have been reported to cause significant variation in results between different estimators.

University of Cape Town

However, while the use of average component implementation times does provide benefits when compared with existing methodologies, it will be obvious to any developer that while some component implementations are straight forward, others can be complex and highly time-consuming. Therefore ReCom allows for three average implementation times per component based on the relative difficulty of the implementation. These three are termed the 'Simple', 'Medium' and 'Complex' implementation averages.

These three levels of difficulty represent a function of the time taken to complete the implementation of a given component. Further, these levels refer only to functional

components and not to data components. This is because there is no logical argument for breaking data components into levels of difficulty. Elaborating further, while one might argue that a data table is a compound structure comprised of several other data elements such as columns and indexes, the methodology takes this into account by counting all data components, whether they form part of a larger compound object or not. Thus we count all tables as well as the fields and keys within. By adopting this approach, the methodology removes the subjectivity of how compound a data component might be. Instead we are left with a simple count of such objects.

Because the methodology is general enough for all component-based development systems, it does not prescribe at what points a particular component's implementation is deemed simple, medium or complex. Rather, the estimator is required to set these points according to a reasonable assessment of past projects. As an example, it may be decided that any report, created with the reporting component that took less than one hour to implement over the course of the project is deemed simple. Any report that took between one hour and four hours is deemed medium and any report that took more than four hours to implement is considered complex.

As a result three averages must be kept for each functional component, while each data component only requires one average to be kept. Therefore, in the general case of the methodology, the estimator must not only perform a count of components, but must also categorise the functional components into their levels of implementation complexity. This, of course, does introduce an element of subjectivity to the methodology. However, this is less prone to variation than lines-of-code estimates, decisions on function counting rules and the relative weights of the adjustment factors found within existing methodologies. For example,

University of Cape Town

ReCom's subjective elements are limited to what constitutes an object and selecting one of only three levels of implementation difficulty as compared to function points and COCOMO with their counting rule interpretations and multiple adjustment factors. Unfortunately the relative levels of subjectivity found within the various methodologies is beyond the scope of this dissertation although this could form the basis for ongoing research in the future.

Although this methodology is being proposed for the first time in this dissertation, it is expected that reasonable initial breakpoints for each level will be established by estimators. Thereafter, as projects are monitored, logical breakpoint timings will emerge and these can then be applied in future estimates, adjusting the averages where necessary.

Automation

It is a specific requirement of the methodology that the measurement and calculation of these averages be automated. There are several reasons for this requirement.

- Within the review of characteristics for successful estimation methodologies, the automated collection of metrics was a key requirement
- By automating the measurement process, the onus of record keeping is removed from the developer.
- The results can be expected to be more accurate and less vulnerable to human error.

University of Cape Town

The automation requirement does raise problems where no such automatic utilities exist. Implementation of the methodology therefore requires that the software organisation either purchase such automated data collection tools or develop their own.

Initial Considerations

When a new component or implementation level of a component is required to be estimated and no average for this yet exists, the estimator should choose a reasonable maximum implementation time for simple and medium components. These times then become the breakpoints between the simple/medium and medium/complex implementation levels. In the case of a new component it is recommended that the estimator then use the following values for each component implementation level as an initial starting point:

- Simple initial level - half of the simple/medium breakpoint time
- Medium initial level - the mid-point time between the simple/medium and medium/complex breakpoints
- Complex initial level - the break point between the medium and complex levels plus the simple/medium break point time.

As an example of this, we earlier suggested a reporting component that had a simple/medium breakpoint time of one hour and a medium/complex breakpoint time of four hours. As such, if the component had never been used before such that no measured averages exist, one would use half-an-hour for any simple reports, two-and-a-half hours for any medium level reports and five hours for any complex reports.

University of Cape Town

An Example of ReCom

In Chapter 4, where component-based development was discussed, an example invoicing application was shown developed with Clarion 5. Using this same example, the ReCom will be applied to show its implementation more clearly.

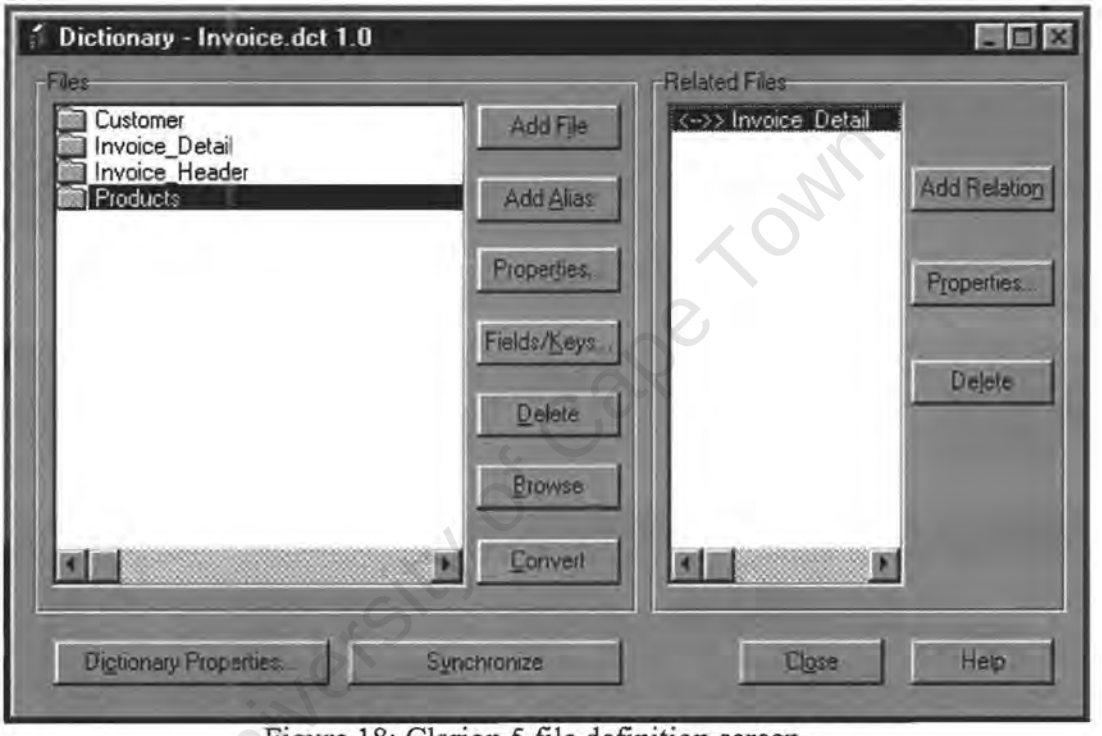

Figure 18: Clarion 5 file definition screen

University of Cape Town

From the central data dictionary of Clarion 5, we can simply count the main data components. These are:-

Number of files: 4
Number of keys: 7
Number of data fields: 14
Number of relationships: 3

For the purposes of this example, we will assume that the average implementation of data components, i.e. the time required to create the initial declaration and subsequent modifications during the project, is ten minutes each.

Figure 19: Clarion 5 application procedure tree screen showing components

Using the above component 'tree', one can see:

Number of Frame components: 1
Number of Browse components: 7
Number of Form components: 5
Number of Report components: 3

For the purposes of this example, we will assume that the following implementation difficulty levels exist.

Component	No. of Simple	No. of Medium	No. of Complex
Frame	1		
Browse	4	2	1
Form	3	2	
Report	1	1	1

Table 13: ReCom Component Count and Implementation Difficulty Rating Table

University of Cape Town

In this example, the following average implementation-level times are assumed for each component in minutes (mins).

Component	Simple Average	Medium Average	Complex Average
Frame	10 mins	35 mins	52 mins
Browse	15 mins	80 mins	220 mins
Form	12 mins	75 mins	340 mins
Report	30 mins	140 mins	240 mins

Table 14: Example of average component implementation times for example

Using the above information, we can now determine the time estimate for the example.

Data Components	**Quantity**	**Average Time**					**Totals**
Files	4	10 mins					40 mins
Keys	7	10 mins					70 mins
Field	14	10 mins					140 mins
Relationships	3	10 mins					30 mins
Functional Components	**Simple Quantity**	**Simple Average**	**Medium Quantity**	**Medium Average**	**Complex Quantity**	**Complex Average**	
Frame	1	10 mins					10 mins
Browse	4	15 mins	2	80 mins	1	220 mins	440 mins
Form	3	12 mins	2	75 mins			186 mins
Report	1	30 mins	1	140 mins	1	240 mins	410 mins
						Project Total	**1326 mins**

Table 15: Full estimate using ReCom for example application

Thus the development time for this example is 1 326 minutes. Assuming a developer day of eight hours, this translates to 2.76 developer days.

Note that the methodology provides an estimate of development time and not of scheduled time. Therefore although this estimate indicates 2.76 develop days, it may take longer to actual complete if the developer is also undertaking other tasks.

University of Cape Town

ReCom and Successful Estimation Methodology Characteristics

A review of the successful characteristics required for a successful estimation methodology (see Table 16) indicates that ReCom possesses the majority of the characteristics and at the same time, has few of the weaknesses found in existing methodologies.

1	The methodology should specify that a metrics program be established that **collects data** relating to the efforts of the software development organisation. Such metrics should be stored in a database and easily accessible. The data within the database should be used to form the basis for future estimates.
2	Where possible, collection of the estimation metrics data must be **automated** and should not be a burden for the team member. However, the person should be aware that data collection is occurring and well-informed as to how and what the data will be used for. Collection of metrics data should occur continuously during the entire development project.
3	The methodology should be **well documented** and reduce the human influence on the outcome of the estimate by both providing a series of steps that **are independent of human subjectivity** but still allow for human-related factors such as interrupts or individual productivity rates. This will allow estimates results to be **easily repeatable** by different estimators..
4	The methodology must be **flexible** enough to take into account risks that might delay projects and further, be able to **adjust easily** should these events occur.
5	The methodology should be **supported by all players** including management and **instituted as a mission-critical part of the day-to-day culture** of the software organisation

Table 16: Characteristics required for a successful estimation methodology

Characteristics 1 and 2 form the basis of ReCom where the implementation times for each component are tracked continuously and stored in a database. The resulting averages are then used for all subsequent estimates. All such data collection must be automated as a requirement of the methodology.

Characteristic 3 and 4 are largely addressed within ReCom. Human subjectivity is reduced to what constitutes a component (which is expected to be minimal) and more significantly, the level of implementation complexity of each component. However, although this subjectivity

University of Cape Town

exists, there is less of it compared with the counting of both lines of code and function points, and particularly the adjustment factors found on the existing methodologies. As such ReCom is less variable in terms of subjectivity and much more likely to produce similar estimates from different estimators on the same project. ReCom clearly does account for human factors such as interruptions and errors through the averaging process.

ReCom directly addresses the issue of risk within estimates. By averaging the implementation times of components, as more and more component implementations are monitored, delays within the projects will form part of the average. As such, when estimating for future projects, the risks of such delays will be factored into the estimate through the averages. While it may be argued that certain projects have higher risk in certain areas than others, any attempt to address this within the methodology would simply serve to introduce subjectivity back into the estimation process i.e. one would have to guess the risk level. Where delays such as requirements creep occur, ReCom will adjust by allowing the estimator to either add additional components to the estimate or modify the implementation difficulty of existing components within the estimate.

For ReCom to be successful, the data collection process must form an integral part of the development process as specified by the fifth characteristic. This can only occur if supported by both management and developers alike. Since the automation of data collection is a key element of the ReCom methodology, it can only be implemented if management and developers allow such automation to occur within the software organisation. Thus Characteristic 5 is also largely addressed by ReCom.

University of Cape Town

ReCom and Previous Methodologies' Weaknesses

ReCom is based on a seed count of components thereby removing the problems of guessing lines of code, and largely reducing the subjective problems associated with function point counting rules. However, unlike COSTMO-4GL, GUI-Points and COCOMO II's Application Composition model, ReCom does not restrict the type of component that can be counted. This generalises the model far more than the other methodologies.

ReCom also largely removes the human influences that cause variability in methodologies as the results are based on accurately measured historical records. At the same time, while the other methodologies use highly subjective values to produce adjustment factors, ReCom takes into account most project factors through its averaging results.

Finally, ReCom has no requirement for backfiring or any other form of conversion to reach its estimate values.

Potential Issues with ReCom

ReCom is a very young and largely untested estimation methodology to date. Although it is based on sound academic research and real-world development characteristics, it will still require refinement before it constitutes a trusted estimation methodology. Even in this early stage, there are certain assumptions that have been made that may be queried. These issues are discussed below:-

University of Cape Town

- ReCom requires an adequate amount of component monitoring before representative averages for implementation times can be used in future estimates. This dictates that an organisation wishing to apply ReCom must start collecting component implementation times as soon as possible if they wish to start using ReCom.

- Allied to this issue is just how much data must be collected to generate representative averages? This is a large unknown. However, an organisation may start using data collected from just one project in order to start using ReCom, but then expect estimates to improve with each attempt thereafter.

- Implementation times required by ReCom are specific to a given developer. This presents a potential problem where a new developer must take over a project from another. If implementation times for the new developer are available, ReCom makes it easy to re-estimate the remaining project time for the new developer. This can be done by performing the ReCom on the remaining components still to be implemented in the project. However, in the case where no implementation times exist for the new developer, a similar problem exists as discussed in the previous issue.

- ReCom currently only takes into account the component implementation difficulty factor for implementation time. It does not take into account other factors such as project type or developer experience. For example, COCOMO has three different project types: Organic, Semi-detached and Embedded systems. Although ReCom is specifically aimed at component-based development systems,

University of Cape Town

it is likely that the implementation time averages will vary between new systems, maintenance of existing systems, additions to existing systems and rewrites of systems. An addition to the database used by ReCom could easily address this by keeping different average component implementation times depending on the type of project. The downside of this would be the corresponding delay in collecting sufficient data for each type of project.

Further, differences in times are also likely to occur depending on the experience of the developer. An experienced developer may implement components significantly faster than a less experienced developer. By categorising the experience of developers, projects can be staffed via ReCom such that less experienced, and therefore normally cheaper, developers may be used on projects if time permits. As above, the ReCom database could be adjusted to address developer experience.

- The quality of automatically captured data must be carefully monitored. A developer may start working on the implementation of a specific component and then go on lunch, leaving the automated data collection system thinking that the developer is still busy with the component. Such a scenario will skew the ReCom averages. Other examples of this problem relate to development activities that are not performed on the computer itself and therefore make automated timings difficult. For example, a project meeting that determines which components to use cannot easily be timed automatically and if left out could have an affect on the averages.

University of Cape Town

This issue highlights the fact that although ReCom requires the software organisation to automate data collection to reduce the likelihood of error, the system is not perfect and inaccuracies are likely. At some point, the management of the organisation must take overall responsibility for the methodology implementation and make every effort to ensure that data capture is as accurate and representative as possible.

- ReCom assumes that the majority of the application being developed will be composed of components. As a result ReCom has virtually no use for non-component-based development projects where alternative methodologies are recommended instead.

- Since the seed unit used by ReCom is a count of reusable components, the methodology can only be used once enough detail about the projects exists such that the number and type of components are known. This requires a fairly advanced specification. It is this issue (the timing of the estimate in the project lifecycle) that has prompted Boehm to include a specific estimation model for early stage estimates – the Early Stage Model. In this model, Boehm suggests that the use of function points provides the most accurate sizing measure prior to detailed project specifications becoming available.

It is assumed that where a component is not used, unique or custom-written source code is developed instead. ReCom is general enough that even this form of development can be taken into account. By assuming that such custom code itself is a reusable component, we can measure it and reuse it in future estimates. Most custom developed source code resides in

University of Cape Town

some form of module. As such, ReCom considers these modules as a reusable component termed "Source Code". Like any component, the Source Code component may require a small amount of effort to create i.e. a small amount of time, or may require a great deal of effort. Thus we can apply the Simple, Medium and Complex types to each custom-written module and by monitoring these, one can determine average implementation times for Source Code components just like other components.

University of Cape Town

Chapter 8: A Physical Implementation of ReCom

A methodology has little use if it represented only in theory. ReCom was researched and proposed such that it would provide a useful real-life estimation methodology for developers who use component-based development tools. Therefore as an important part of the dissertation effort, a physical implementation of ReCom has been created in order to present the practical viability of the methodology. Since automation of the data collection process is key to ReCom, this practical implementation is crucial to showing its applicability.

Thus, using the Clarion 5 toolset as the target, a new utility add-on tool has been developed by the author and co-developer, Mr Joe van Niekerk, to effect ReCom for Clarion 5 developers. The utility tool is called "Dev Monitor". It comprises two main products.

- Dev Monitor – this is the data collection utility that runs in the background of the Microsoft Windows operating system and tracks which component is being used within the Clarion 5 environment. It assumes that a component is in use until another is selected. Thus as soon as one component is selected, a timer starts. Once another component is selected, the previous timer is stopped and the time, along with the component name and type are written away to a log file.
- Dev Stats – this product imports the log files created by Dev Monitor and stores the results into a database. The Dev Stats product then allows the user to see the entire time logged for a particular project, the average implementation time per component in a given project and the average implementation times for all components over all projects logged to date. Dev Stats also allows the user to create a new estimate. The estimator chooses the components and their expected implementation difficulty to be

University of Cape Town

used in the new project. Dev Stats then automatically applies the relevant implementation average times to the estimate. Once all components have been entered, an estimate of the project is provided by Dev Stats as a total at the bottom of the estimating screen.

The Dev Monitor utility has been extended and enhanced as the ReCom methodology has progressed. The following sequence of screens and discussion provides an overview of Dev Monitor and a practical implementation of ReCom. Please note that the data displayed within these sample screens is for illustration purposes and does not constitute accurately logged data.

Once the utility tool has been included into a project being developed within the Clarion 5 development environment, it will keep track of all components that are worked upon. This includes functional components, termed "templates" in Clarion 5, as well as data components. The data components can be declared and modified in both Clarion 5's Data Modeller or Data Dictionary tools. The tracking utility, Dev Monitor, will track times in all these areas. In Figures 20 and 21 below, two development screens from Clarion 5 are shown with the Monitor in operation. One screen shows data components being monitored within the Data Modeller and the other shows a functional component being monitored from within the main development environment.

University of Cape Town

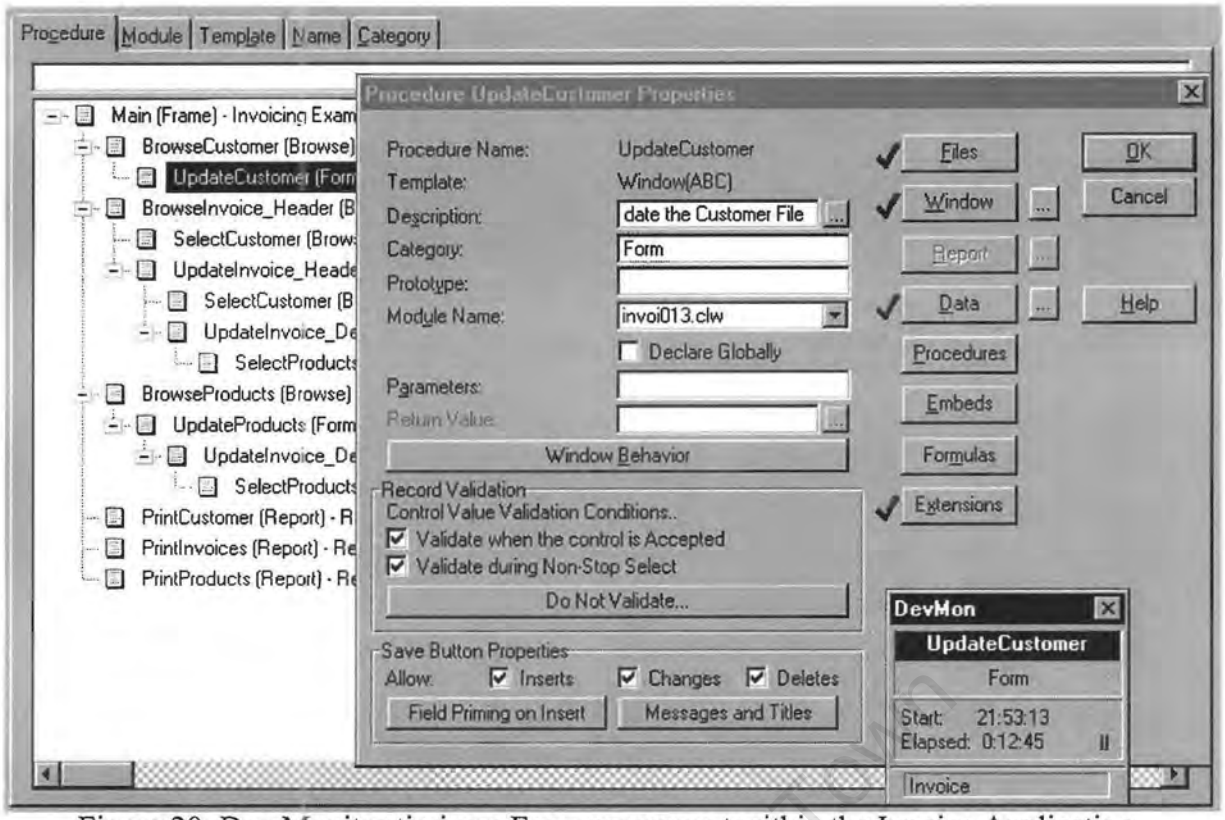

Figure 20: Dev Monitor timing a Form component within the Invoice Application

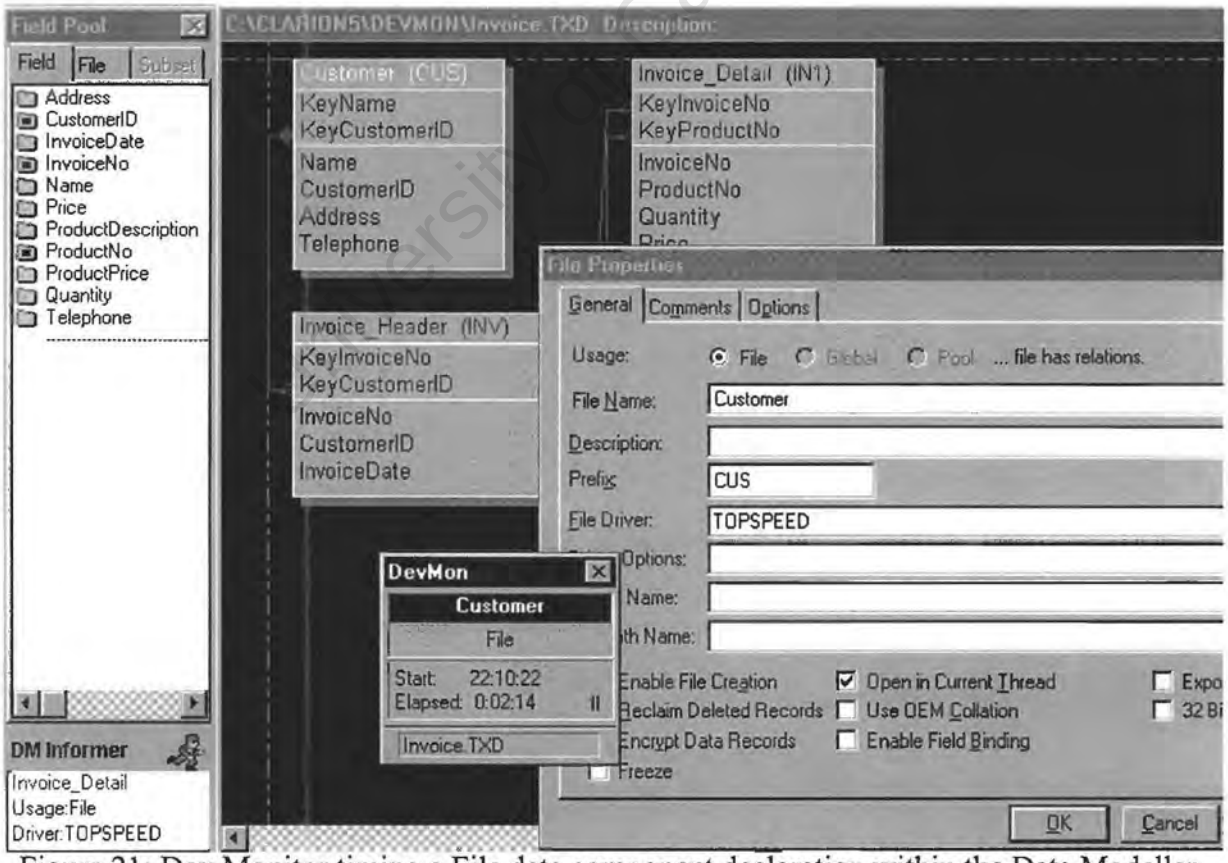

Figure 21: Dev Monitor timing a File data component declaration within the Data Modeller

From the above diagrams it is clear how Dev Monitor times the components used within the Clarion 5 environment. There are some additional issues that are also addressed by the Monitor:

1) The Monitor is visible to developer. As discussed in the chapter devoted to requirements for a successful estimation methodology, it was found that the developers must be aware both that they are being monitored and also why. The developer is always aware that Dev Monitor is in operation since it is visible on the development screen. It should be noted that the developer has the ability to hide the Monitor if they do not want to see it. Even in this hidden condition, the Monitor is available in the 'taskbar' when using Windows 95, Windows 98 or Windows NT.
2) While the Monitor tracks component implementation times, a user-defined activity may be entered to track other activities not performed on the computer. An example of this, such as a project meeting, is shown in Figure 22 below.

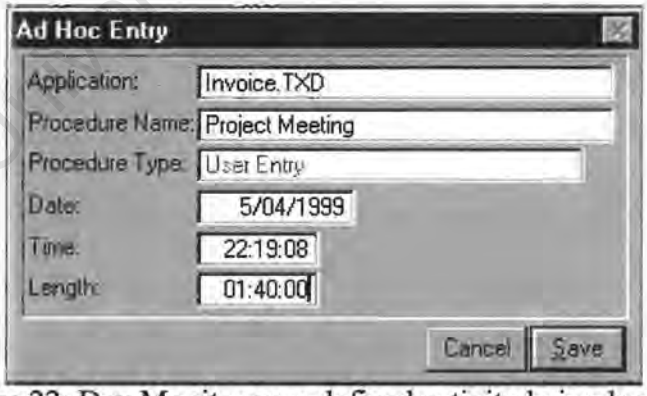

Figure 22: Dev Monitor user-defined activity being logged

3) If a developer chooses to stop working, the logging by the Monitor can be suspended. Further, if circumstances dictate, a developer may choose to ignore a particular component and it will not be logged. In most circumstances all activity

University of Cape Town

should be logged to provide the most representative averages for component implementation, but this feature was nevertheless made available. In the case where a developer leaves the Monitor logging and, for example, goes out to lunch, a maximum time without activity may be set, after which the Monitor suspends logging. All these aspects along with other setup related activities are available via the screen shown in Figure 23 below.

Figure 23: Dev Monitor setup and control menu

The data logged by the Monitor is then imported into the Dev Stats utility. Dev Stats stores the data into a database and then provides screens and reports that are useful to the developer. These include reports of the actual project time logged to date, average component implementation times both by project and overall for all projects and finally, provides a ReCom estimation screen. This estimation screen allows an estimator to implement ReCom using the averages calculated within the Dev Stats utility. Examples of these various screens are provided in Figures 24 through Figure 28.

University of Cape Town

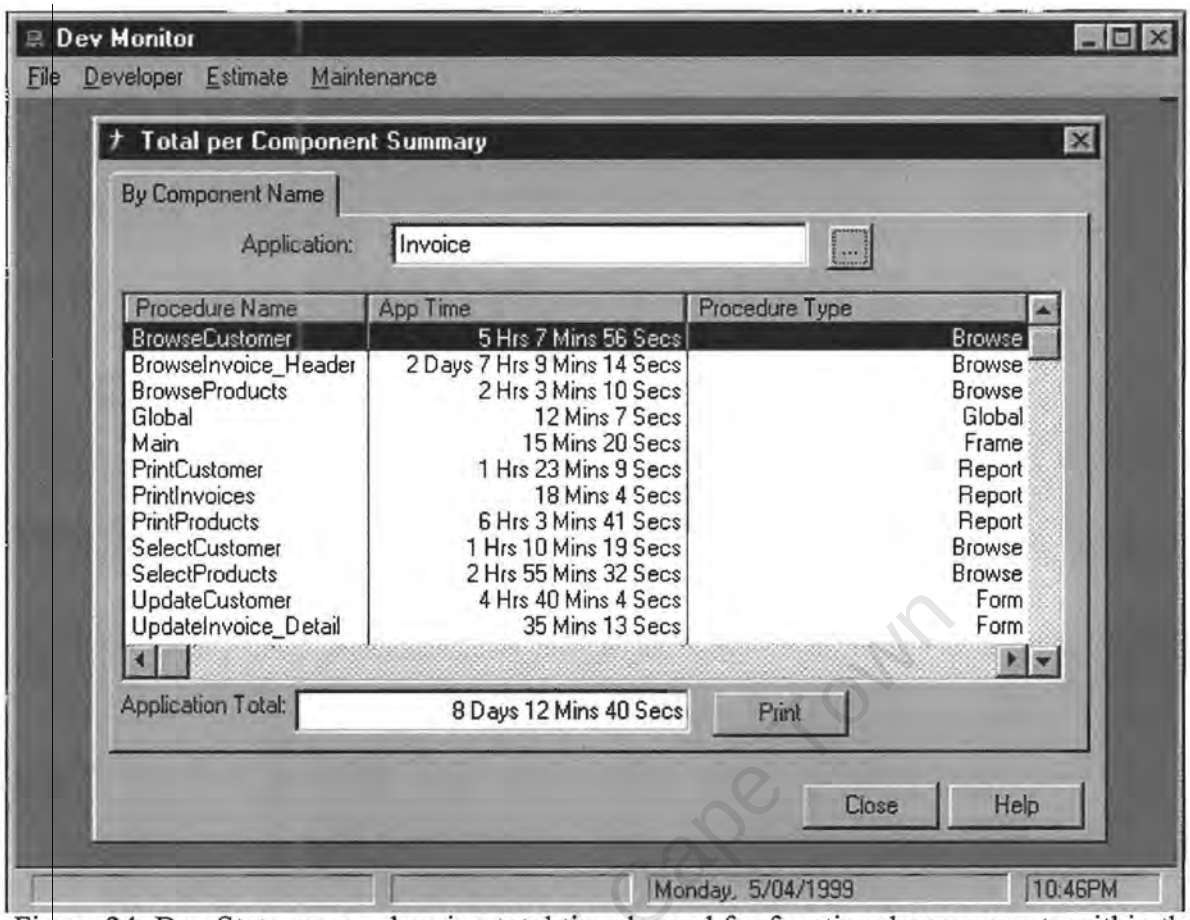

Figure 24: Dev Stats screen showing total time logged for functional components within the Invoice application

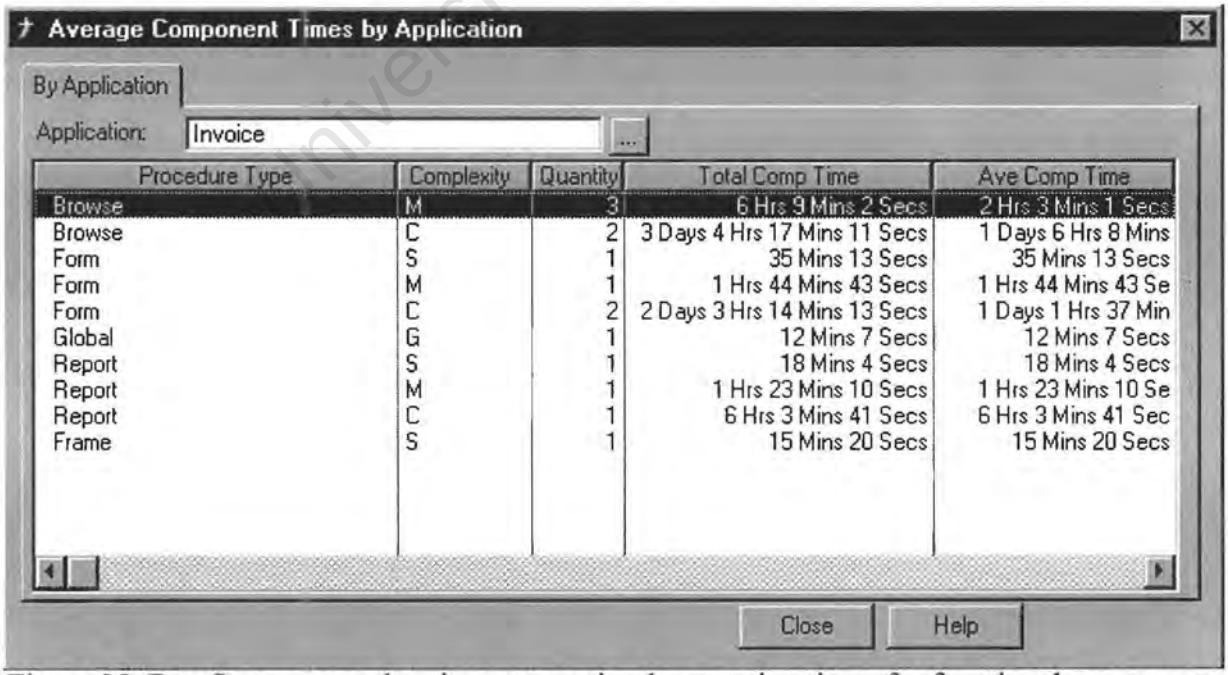

Figure 25: Dev Stats screen showing average implementation times for functional component only within the Invoice application

University of Cape Town

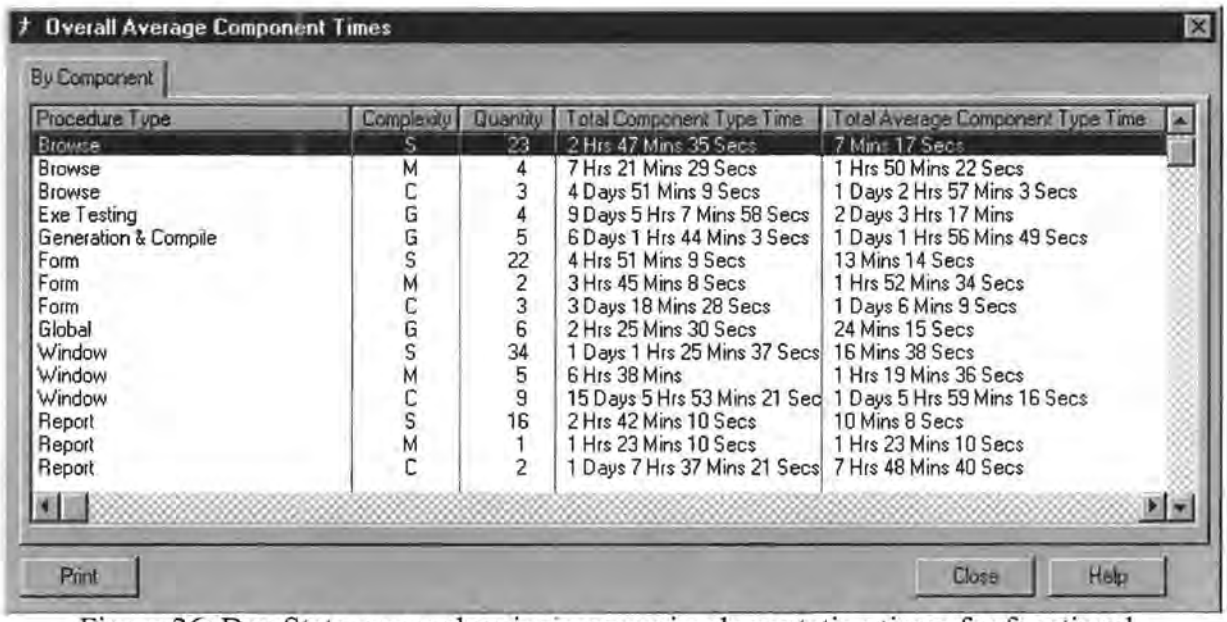

Figure 26: Dev Stats screen showing average implementation times for functional components for all applications logged to date.

In both Figures 24 and 25, it can be seen that a complexity column exists. This indicates whether a component has a simple, medium or complex implementation difficulty. However, a fourth complexity type, termed "Global" and marked as a "G" exists as well. This is provided for those components that do not logically have different implementation categories. For example, in Figure 26, a component termed "Generation and Compile" exists. This is the activity of generating source code and compiling it into an executable application file. It is not feasible to have a three-category implementation of this activity as it would be impossible to characterise the generate-and-compile activity as simple, medium or complex difficulty. Thus the average effort required to perform the activity, is deemed to only have one category, that of Global. Global components are specifically listed within the Dev Stats setup so that they can be logged as such. Note that this also applies to all data components. Again it is not feasible to determine whether, for example, a file definition, is simple, medium or complex. If the file consists of multiple items, thereby making more time-consuming to implement, these will be taken into account within the other data component counts such as fields, keys, etc.

University of Cape Town

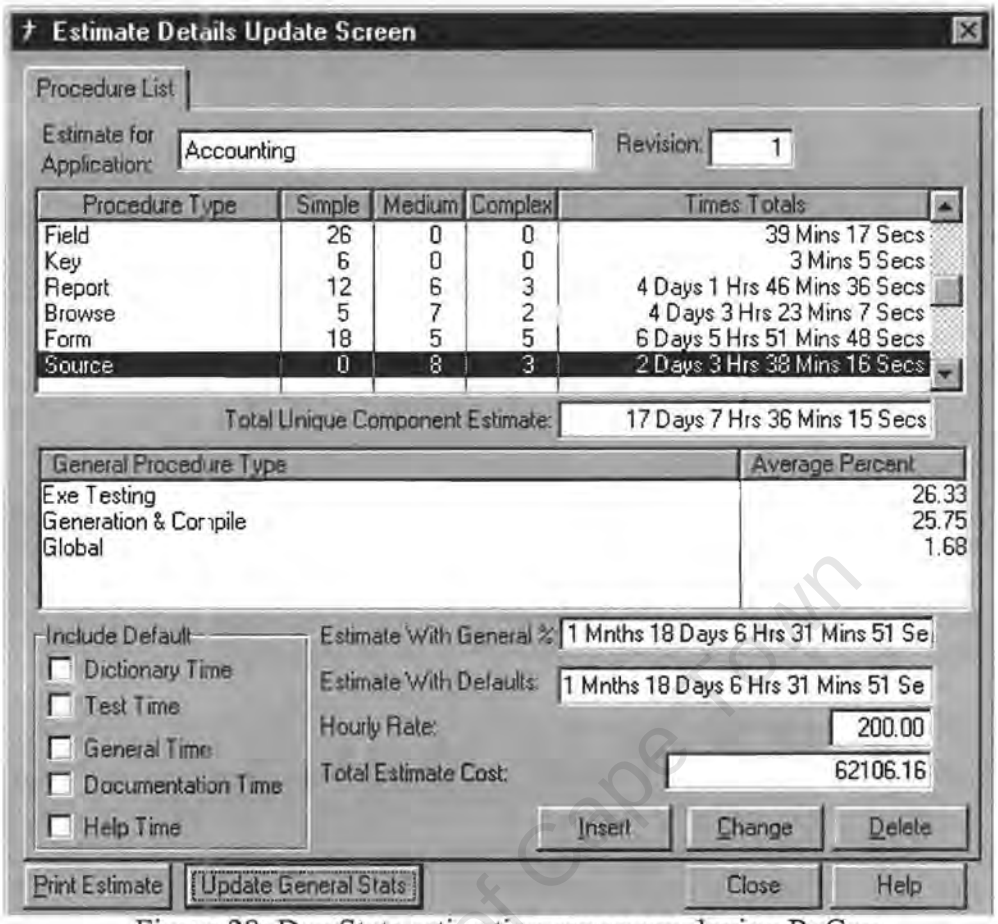

Figure 28: Dev Stats estimation screen employing ReCom

Figure 28 shows an estimate developed for an example project called "Accounting". Dev Stats allows for multiple revisions of estimates for a given project. This allows an organisation to re-estimate during the project or as requirements change. A list of components, both data and functional is then displayed. These have been selected from the existing set of components in the Dev Stats program that have been imported into the past and now reside within the database. The estimator indicates how many of each component are required for the new project and, where applicable, provides the implementation difficulty for each component. Since data components, shown as "Field" and "Key" in Figure 28, do not have a specific category, they are shown only as simple components. The remaining components are functional components and have a count of simple, medium or complex implementation difficulty. By entering these component counts and implementation

University of Cape Town

As discussed within the ReCom definition in Chapter 7, the decision as to what constitutes a simple, medium or complex implementation of a particular component is left to the software organisation. Dev Stats provides an interface for the estimator to adjust the breakpoints between these difficulty categories but defaults the simple/medium breakpoint at 1 hour and the medium/complex breakpoint at 3 hours for any new component. The estimator can then change these breakpoints for each individual component as required. Figure 27 shows this screen within Dev Stats.

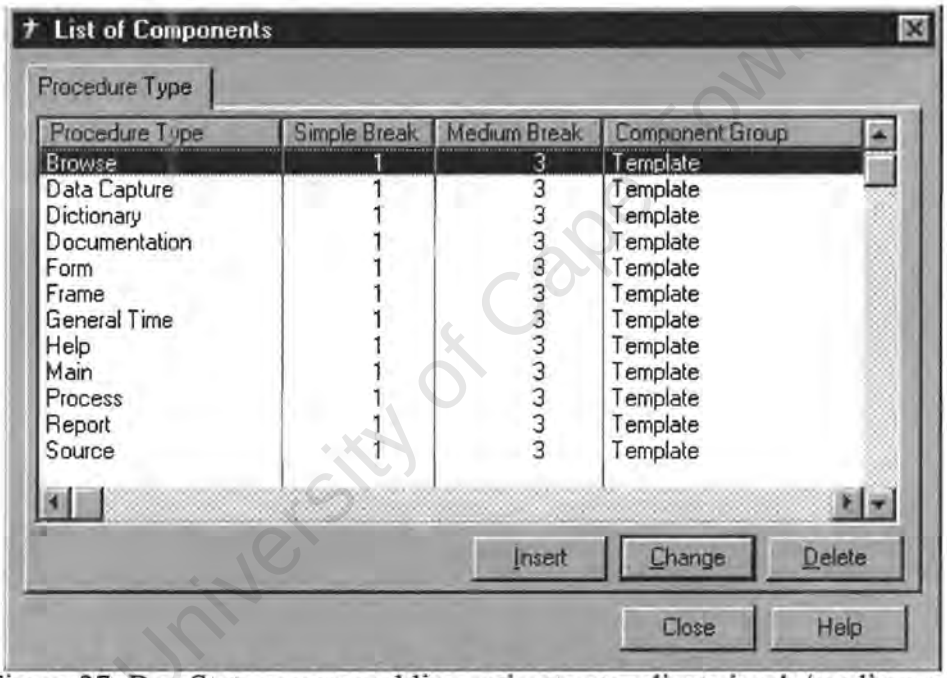

Figure 27: Dev Stats screen enabling estimator to adjust simple/medium and medium/complex breakpoints for each component

Finally, the Dev Stats utility allows the software organisation to use this database of information to create new estimates using ReCom. This is performed in a single screen shown below in Figure 28.

difficulties, Dev Stats automatically calculates the implementation averages for each component (and their difficulty) and provides a total for each component in the right hand column. This reflects how ReCom removes the subjectivity from the estimator as the methodology makes use of factual data rather than analogy or someone else's empirical data. The total for all entered components is displayed under the list of components.

Just as ReCom is a young methodology, the functionality of Dev Stats is also evolving. As such, Dev Stats also provides some additional estimation information. Displayed in the second list within Figure 28, Dev Stats takes the average times to perform global activities such as generate-and-compile and testing of the application and applies them as a percentage of the overall system. For example, if through logged information, Dev Stats determined that, on average, testing comprised 10% of total development effort, then it is assumed the functional and data component implementation effort makes up the remaining 90%. The overall estimate is therefore adjusted accordingly. This goes beyond the basic methodology employed by ReCom and this dissertation but forms part of on-going research.

Finally, Dev Stats allows the estimator to enter a sample cost per hour for development and a total monetary cost is provided for the estimator. This is illustrative for the estimator who may have to provide a development cost to a potential customer.

Chapter Conclusion

In this chapter, a utility tool developed to physically implement ReCom has been described. This tool was used within three development projects. The resulting information has been detailed in the next chapter together with observations that can be drawn about implementing ReCom.

University of Cape Town

Chapter 9: Early ReCom Empirical Results

The ReCom estimation methodology has been created to provide more accurate estimates specifically for reusable component-based development projects as compared with other existing methodologies. ReCom's process flow is designed to satisfy generally-accepted characteristics for successful estimation methodologies as well as reduce or remove weaknesses evident in other methodologies.

However, at this early stage of the methodology development a large-scale research effort will be required to test its usefulness. In fact, since ReCom requires a significant sample of data for each developer just to determine that single developer's component implementation averages, such a rigorous study is expected to take significantly longer than those of other methodologies.

However, a small group of development projects was selected to test the feasibility of *implementing* the ReCom methodology. Useful experience and early data has been gained through this effort and is reported within this chapter.

University of Cape Town

Research Method

The research effort set out to apply the automated data collection part of the ReCom methodology in order to start collecting implementation averages for each sample developer. These averages would be collected on an on-going basis and applied to later estimates in order to determine the accuracy of the ReCom methodology.

Although the research effort has been started, it is currently only possible to report on the implementation issues discovered and early conclusions that can be drawn from the data collected to date.

All projects made use of the Clarion 5 development toolset as the data collection tool developed, "Dev Monitor", has been used for the research process. Three Clarion 5 developers were approached to apply the automated data collection tool "Dev Monitor" to their projects. Two of the developers have more than five years of Clarion experience, while the third has one-and-a-half years of Clarion experience. All three developers are experienced in component-based development methods.

Each of the three developers was working on a different project. The first project was the re-development of a merchant banking system from an MS-DOS platform into the equivalent system on a Microsoft Windows platform. The second project was the new development of a pallet tracking system for the citrus fruit industry. The third project entailed the maintenance (small enhancements and bug fixes) of a jewelry manufacturer's invoice and ordering system. By using different developers, one of the early conclusions that could be determined was whether there is a significant difference in average implementation times of commonly used components between developers.

University of Cape Town

Implementation Difficulties

While the automation of data collection required for ReCom is a requirement of the methodology, it represented several implementation challenges in its own right. Firstly neither Clarion 5 nor other tool vendors provide such data collection utilities. And secondly, the developers were required to be mindful of the logging process to ensure as accurate logging as possible.

Because of the issues mentioned above, several practical difficulties were encountered when attempting to begin the ReCom data collection process. These are detailed below.

As a result of no automated logging utility forming part of the Clarion 5 toolset, the Dev Monitor utility was developed for the purpose. However, as Dev Monitor is an add-on utility, there were several development activities within the Clarion 5 development environment where it was not possible to automate the logging of elapsed time during the research period.

For example, the times spent within the Clarion 5 Data Dictionary and entity-relationship Data Modeller could not be logged. This limitation is serious in the context of ReCom because it meant that actual times spent to implement data components could not be logged automatically. However, since then Dev Monitor has been enhanced to log this data component information but it is not contained within the current research data. Another problem is that the logging software cannot automatically detect when a developer is either not working on a component or finished working with a component e.g. when on lunch, or exiting the development environment.

University of Cape Town

This has both positive and negative implications for the logging process. In the case where a developer stops working on a component due to an interruption such as a personal phone call, the logging system should log this time as part of the averages used by ReCom in order to take into account the realities of development life as experienced by developers. This will satisfy the interruption concerns voiced by van Solingen et al (1998). However, if the developer leaves the system logging time during a lunch break, the logging will erroneously inflate the average implementation time for that component. Efforts to limit this potential problem are being made within Dev monitor by halting the logging process after a set time e.g. 20 minutes, and asking the developer whether work on a particular component is still taking place. Dev Monitor also allows logging to be suspended if the developer chooses to stop work on a component for a period but does not wish to leave the environment. And further, Dev Monitor will allow the developer to discard the last logged component time-period if it is not representative of the actual time spent on the component implementation e.g. if the system was left logging overnight, giving a completely false time-period for a component.

From these early observations, two important conclusions can be drawn regarding the automation of the data collection process. Firstly, ReCom's usefulness will be limited to those components that can be logged. Secondly, even with automated logging, the data cannot be considered absolutely accurate. However the reason for using average implementation times for components is that the effect of such inaccuracies can be minimised as the data sample grows. Also, while the responsibility of developers to accurately track their development efforts has been largely reduced, it still requires some effort on their part to ensure that the logging process is as representative as possible.

University of Cape Town

Results To-Date

Table 17 shows the basic details for each of the project researched. For each project, the developer, project description, component count, logged time and elapsed time have been provided.

Developer	Project	Data Components	Functional Components	Total Logged Times	Total Elapsed Times
A	Pallet tracking	612	78	33.50 days	39 days
B	Banking	443	88	14.75 days	20 days
C	Jewelry orders	299	62	3.25 days	17 days

Table 17: Component and time details for sample projects monitored

The data component count for each project includes all files, fields, sort keys and file relationships. The functional component count includes all components that the Dev Monitor tool logged during the monitoring process. The Total Logged Times are made up of all times logged during development of the functional components. As mentioned earlier, at the time of collecting this information, the Dev Monitor tool was unable to log the time spent on creating and maintaining the data components. The Total Logged Times days are based on an 8-hour working day. The Total Elapsed Times represent the number of days that logging took place. It excludes weekends and public holidays.

Two observations can be made regarding this first set of data.

- The percentage of logged time to elapsed times for the pallet tracking, banking and jewelry orders systems were 85.9%, 73.75 and 19.1% respectively. Clearly the jewelry orders system logged time stands out as being a small percentage of the elapsed time. This can be explained by the fact the jewelry system project was a maintenance project. As such, the developer was only required to either fix

University of Cape Town

reported bugs or make minor enhancements to the existing system. Although a large number of functional components were worked with, limited time was spent on each in this project. This is reflected in the small amount of time actually logged compared with the number of days on which the development actually took place.

As the logging process only automatically monitors actual time physically spent working with the components, it is not possible to ascertain whether the developer spent the rest of the time on non-programming work e.g. analysis or documentation of the problem, or on non-related work on another project altogether. It is reasonable to deduce that averages for maintenance projects be kept separately from new projects although a larger data sample is needed to verify this.

- A simple count of functional components worked upon during the logging period is not indicative of the total time actually logged for the project. This is especially clear for the maintenance project but can also be seen for the two new projects, which have an 11% difference in their functional component count but a 56% difference in their logged times. This observation does not lessen the usefulness of ReCom as the methodology does not rely on a simple component count, but rather on a count of each individual component type used in a project.

Table 18 shows the results for each component type used by the three projects and their respective average implementation times. It can be seen that several of the components are common to all three projects e.g. the Browse and Report components. For the sample

University of Cape Town

projects, all component implementation difficulties were deemed Simple if the implementation time was less than one hour. If the time was greater than one hour but less than three hours, the component implementation time was deemed average. Any times above three hours were deemed complex.

Table 18 also provides times for "General' components. These are activities associated with development but that cannot be logically broken up into Simple, Medium or Complex implementation times. These include the times taken to generate and compile programs, test the program functionality and declare functionality global to the entire program. Collection of this data is not required for ReCom as defined in this dissertation but may form part of an extended version of the methodology as further research is conducted. Useful observations can be made from this data and are provided below in Figure 29.

University of Cape Town

Component Name	Implementation Difficulty	Project A Sample Quantity	Project A Average time	Project B Sample Quantity	Project B Average time	Project C Sample Quantity	Project C Average time
Browse	Simple	13	8m 35s	22	5m 52s	10	14m 17s
	Medium			4	1h 50m 1s		
	Complex	1	4h 33m 23s	1	6h 57m 38s		
Form	Simple	11	15m 51s	18	6m 31s	4	21m 29s
	Medium	1	2h 8s			1	1h 8m 36s
	Complex	1	5h 3m 49s	4	1d 2h 32m		
Report	Simple	8	13m 4s	3	4m 54s	21	6m 52s
	Medium						
	Complex	1	1d 1h 33s				
Window	Simple	15	23m 30s	16	4m 31s	3	2m 7s
	Medium	3	1h 21m 43s	2	1h 30m 13s		
	Complex	8	1d 5h 56m				
Source	Simple	6	10m 28s	8	6m 26s	13	11m 49s
	Medium	1	1h 13m 45s			1	1h 8m 36s
	Complex						
Generation and compilation	General		4d 6h 55m		7h 12m 44s		4h 26m 20s
EXE Testing	General		7d 4h 37m		4d 6h 53m		1d 45m 14s
Global	General		1h 54m 38s		1h 34m 24s		15m 24s

Table 18: Summary of average component implementation times for three sample projects

Key: d=days, h=hours, m=minutes, s=seconds

University of Cape Town

Even from the small amount of sample data, it is clear that the different projects have different average implementation times for the same category of component. For example, the average time to implement a simple Browse component is 8 minutes 35 seconds for project A, 5 minutes 52 seconds for project B and 14 minutes 17 seconds for project C.

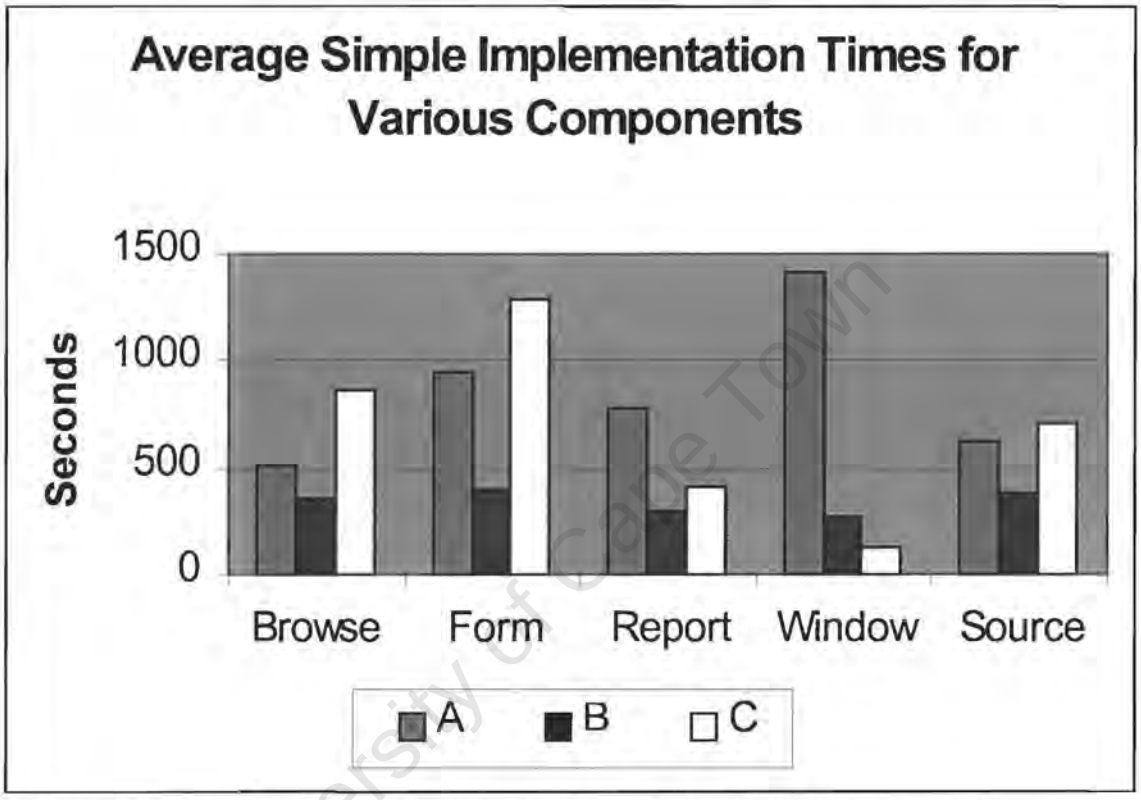

Figure 29: A comparison of average simple implementation times for various components from the three sample projects

University of Cape Town

Figure 29 provides a graphical comparison of the average simple implementation times for the most reused components in the sample. The average simple implementation times were used as they represented the largest individual set of samples for each component.

It is difficult to draw a meaningful conclusion from the data for Project C given that the project was a maintenance project, while A and B were new projects. Project C was staffed with the least experienced developer. However, there is consistency in the results between Projects A and B. Specifically, the average component implementation times for A were

always greater than those of Project B. Since the experience of both Project A and B developers is similar, this consistent difference in implementation times could be attributed to one of two things. It could either indicate proof that different developers do indeed develop at differing rates or that the projects were sufficiently different so as to co-incidentally provide such results within the small sample set. This can only be finally determined with a much larger sample set.

ReCom does not specify the times at which the Simple-to-Medium and Medium-to-Complex breakpoints should occur but leaves that to the software organisation to determine. Although the implementation times for simple components was set at one hour for all components in the sample projects, the data for all components showed simple averages to be well under the one hour mark. In fact, the average simple implementation time for all the components was 10 minutes and 25 seconds. If, as larger sets of data were collected, this average was maintained, the software organisation could use this data to reduce the Simple-to-Medium breakpoint from one hour to something lower. Similarly, longer-term data could be used to find a suitable (to the software organisation) breakpoint time between Medium and Complex component implementation times. Finally, it is expected that these breakpoints will differ for each component type e.g. Browse, Form etc.

As mentioned above, although the initial definition of ReCom does not call for the collection of times for general development activities such as compiling or testing, this was collected during the course of the sample projects. By summing the times expended on the code-generating, compiling, testing and global declaration activities, the general activity time percentage of the total logged time could be determined. The percentage of total time logged for these general activities were 37.75% for Project A, 40.34% for Project B and 52.36% for

University of Cape Town

Project C. These percentages represent a significant proportion of each project. As a result, it would make sense to expand the ReCom methodology to include such activities in a future revision.

Chapter Conclusion

Although the sample data set collected was small, some useful observations could be made and improvements to the methodology implemented. Specifically, the automation of data collection requires careful implementation. Although the collection of the data is automatic, it should not be assumed that the data is 100% accurate and that the larger the data set logged, the more likely the averaging process will smooth inaccuracies. One of the most obvious conclusions that can be drawn from this effort is that the tool vendors need to assist the software organisations by including such data logging utilities as a standard part of their offerings. This would negate the need for third-party efforts, which are more likely to fall short of a full logging facility.

The data provided an indication that developers with similar experience and projects do differ in their implementation times for the same type of component. This, however, requires a much larger sample set to validate. The data also indicated that with a larger sample, the breakpoints between the average Simple, Medium and Complex component implementation times will become easier to specify by a software organisation.

Finally, time taken to perform tasks such as compiling and testing was shown to represent a significant portion of a project time. As such the ReCom methodology should be enhanced to take such activities into account as well.

University of Cape Town

Chapter 10: Conclusion and Further Research Possibilities

The ReCom estimation methodology has been created to provide accurate estimates for those software projects that make use of reusable data and functional components. In order to create a rigorous estimation methodology that would satisfy these needs, this dissertation has considered two significant factors.

Firstly, the characteristics required of a successful estimation methodology as determined by a consensus of academic research were detailed. The ReCom methodology has largely satisfied these characteristics, and certainly has done so more than the other methodologies reviewed within the dissertation. Of greatest significance, ReCom removes most of the subjectivity from the estimation process. Subjectivity, being the most predominant weakness among the other methodologies hindering a repeatable process by multiple estimators on the same project.

Secondly, ReCom directly addresses the issue of reusable components. While three of the reviewed methodologies do account for components within their estimation processes, they are all limited in more than one way. ReCom expands the definition of components to include both data and functional components and then further address the unique development styles found from one developer to another. No other methodology provides this personalisation in a non-subjective manner.

Within Chapter 2 of this dissertation a theoretical model for estimating methodologies was presented. This was used to describe the existing methodologies as well the ReCom methodology. By making use of this model, it was possible to highlight the problems that

University of Cape Town

affect estimates such as the choice of seed unit with the associated subjectivity of the seed count, applicability of the methodologies' algorithms to component-based development and the various adjustment factor determinations.

In the case of component-based development projects, the ReCom methodology provides a set of estimating processes that addresses these issues better than the others reviewed. ReCom shares the seed unit concept of components/objects with the Application Composition model of COCOMO II and COSTMO-4GL. However, ReCom takes the additional step of expanding the definition of components to include all functional and data components. This improves upon the limited number of components/objects used in the other two methodologies for the seed count. This is expected to make the counting of components easier and more representative of the whole project for the estimator.

Both the algorithms and the adjustment factors shown within the theoretical model allow a large amount of variability in the final estimate. In the case of existing methodologies, the algorithms are typically based on a set of empirically-derived formulae that often have limited similarity to the type of projects undertaken today. In the case of the adjustment factors, tremendous scope for subjectivity is 'built-in' to the methodologies virtually assuring variable results between estimators. ReCom largely resolves both of these issues. The estimation algorithm is based on the unique data of each individual developer. Further, since ReCom has been specifically created for projects that reuse the same components, such data is far more representative of projects using this technology. ReCom has also removed a major source of variability by dispensing with adjustment factors. Instead, since ReCom makes use of 'hard' historical data, variability is significantly reduced.

University of Cape Town

However, while ReCom may have been well motivated within this dissertation, it has yet to be validated with a rigorous empirical research project. Due to time constraints for this dissertation and the requirement of ReCom to have historical data in order to effect the methodology, only limited data was available for discussion.

What has been observed is that ReCom's requirement for automated logging of component implementation times has encountered difficulties, which need to be addressed before ReCom can be considered seriously. These difficulties can be largely addressed if tool vendors provided logging tools within their offerings. Further, by also logging non-programming tasks such as generating, compiling, testing and others, it can be seen that the non-programming tasks consume a significant proportion of total development time and that ReCom will have to be enhanced to adequately take this in account.

Thus the opportunity for on-going research and enhancement of ReCom certainly exists and would contribute to the on-going quest to find more accurate and repeatable estimation methodologies. Among the opportunities that present themselves are:-

- A detailed and long-term empirical research project that would monitor a number of developers using a particular component-based development tool. The component implementation times should be tracked and used to perform estimates for future projects undertaken by each developer. From this, the usefulness of ReCom can be determined.
- ReCom can be enhanced to address the significant non-programming activities within development projects. The Dev Monitor program has already started to address this issue but further research is required.

University of Cape Town

- By tracking each individual developer's average component implementation times, a study of the amount of differentiation between developers' programming times can be ascertained. If it were found that no major differences are present, ReCom could be more simply implemented as not as many samples would be required to gain the necessary averages.
- It is possible that the Function Point model could be improved by integrating aspects of ReCom. Function Point Analysis remains a very useful sizing metric but has a significant weakness when attempting to turn the function count into a time estimate when reusable components are used. With the reusable component-related data provided by ReCom, it would useful to study whether this data could be used in the latter stages of the Function Point methodology to improve time estimates.

ReCom now joins the other three modern methodologies COSTMO-4GL, GUI-Points and COCOMO II as a method of addressing software estimation for component-based development software projects. It represents a serious attempt to extract the best parts of those others methodologies and produce a more useful contribution to estimators.

University of Cape Town

Chapter 11: References

Abdel-Hamid TK: "Adapting, Correcting and Perfecting Software Estimates: A Maintenance Metaphor", *Computer*, March 1993, pp 20-29

Abran A, Robillard PN: "Function Points Analysis: An Empirical Study of Its Measurement Processes", *IEEE Transactions on Software Engineering*, Vol 22 No 12, December 1996, pp 895-909

Albrecht AJ: "Measuring Application Development Productivity", *Proceedings of the Joint IBM/SHARE/GUIDE Application Development Symposium*, October 1979

Albrecht AJ, Gaffney JE Jnr: "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Transactions on Software Engineering*, Vol SE-9 No 6, November 1983, pp 639-648

Banker RD, Kauffman RJ, Kumar R: "An Empirical Test of Object-based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment", *Journal of Management Information Systems*, Winter 1991-92, Vol 8 No 3, pp 127-150

Behrens CA: "Measuring the Productivity of Computer Systems Development Activities with Function Points", *IEEE Transactions on Software Engineering*, Vol SE-9 No 6, November 1983, pp 648-652

Berry RH, Schoenborn RM: "Estimating Requirements for a Large Software Engineering Project", *ACM Conference proceedings on TRI-Ada '92*, pages 375-383

Boehm BW: "Software Engineering Economics", *Prentice Hall*, Englewood Cliffs NJ, 1981

Boehm BW, Papaccio PN: "Understanding and Controlling Software Costs", *IEEE Transactions on Software Engineering*, Vol 14 No 10, October 1988, pp 1462-1477

Boehm BW: " COCOMO II Model Definition Manual", *University of Southern California*, Version 1.4, 1998

Booch G: "Object-Oriented Development", *IEEE Transactions on Software Engineering*, Vol SE-12 No 2, February 1986, pp 211-221

Brown AW, Wallnau KC: "The Currect State of CBSE", *IEEE Software*, September/October 1998, pp 37-46

Brown D: "Simple Steps to Successful AD Metrics Programs", *Gartner Group*, 4th December 1996

Cassell, J, Schick K, Hall B, Phelps J: "Management Edge: Year 2000 - Top View.", *Gartner Group*, Stamford, CT,1997.

University of Cape Town

Court C a): "An Overview of Software Project Cost Estimation Models", Unpublished Masters technical report, University of Cape Town, 18th September 1997

Court C b): "Management and Tool Choice Issues Affecting Software Project Estimation Methods", Unpublished Masters technical report, University of Cape Town, 1st December 1997

Finnie GR, Wittig GE, Desharnais JM: "Reassessing Function Points", *Australian Journal of Information Systems*, Vol 4 No 2, May 1997, pp 39-45

Furey S: "Why We Should Use Function Points", *IEEE Software*, March/April 1997, pp 28-30

George JF: "Computer-Based Monitoring: Common Perceptions and Empirical Results", *MIS Quarterly*, December 1996, pp 459-480

Hall T, Fenton N: "Implementing Effective Software Metrics Programs", *IEEE Software*, March/April 1997, pp 55-64

Heller R : "An Introduction to Function Point Analysis", *Process Perspectives*, No. 4 - Fall 1996, <http://www.processtrat.com/pp_4.htm>, Last Accessed 1/9/97

Hihn J, Habib-agahi H: "Cost Estimation of Software Intensive Projects: A Survey of Current Practices", *Proceedings of the 13th International Conference on Software Engineering. Austin, TX, May 1991,* IEEE Computer Society Press, pp 276-287

Hotle M : "Function Points in an OO Environment", *Gartner Group*, Stamford, CT, 30th July 1996

IFPUG : "Function Point Counting Practices Manual", *International Function Point Users Group*, Release 4.0, January 1994

IFPUG: "Guidelines to Software Measurement", *International Function Point Users Group*, Release 1.1, Septemeber 1996

Jeffery DR, Berry M: "A Framework for Evaluation and Prediction of Metrics Program Success", *Proceedings 1st International Software Metrics Symposium*, IEEE Computer Society Press, Los Alamitos, Californis, 1993, pp 28-39

Jeffery DR, Low GC, Barnes M: "A Comparison of Function Point Counting Techniques", *IEEE Transactions on Software Engineering*, Vol 19 No 5, May 1993, pp 529-532

Jenkins AM, Naumann JD, Wetherbe JC: "Empirical investigation of system development practices and results", *Inform. Manage.*, Vol 7, pp 73-82, 1984

Jones C: "Estimating Software Costs", *McGraw-Hill*, New York, 1998

Jones C: "Backfiring: Converting lines of code to function points", *Computer*, November 1995, pp 87-89

University of Cape Town

Jones C: "Programming Languages Table", WWW page, *SPR*, Release 8.2, March 1996, <http://www.spr.com/library/0langtbl.htm>, Last Accessed 10/9/98

Känsälä K: " Integrating Risk Assessment with Cost Estimation", *IEEE Software*, May/June 1997, pp 61-66

Kemerer CF: "An Empirical Validation of Software Cost Estimation Models", *Communications of the ACM*, Vol 30 No 5, May 1987, pp 416-429

Kemerer CF, Porter BS: "Improving the Reliability of Function Point Measurement: An Empirical Study", *IEEE Transactions on Software Engineering*, Vol 18 No 11, November 1992, pp 1011-1024

Kitchenham B: "The Problem with Function Points", *IEEE Software*, March/April 1997, pp 29-31

Klepper R, Beck D: "Third and Fourth Generation Language Productivity Differences", *Communications of the ACM*, Vol 38 No 9, September 1995, pp 69-79

Kozaczynski W, Booch G: "Component-Based Software Engineering", *IEEE Software*, September/October 1998, pp 35-46

Laranjeira LA: "Software Size Estimation of Object-Orientated Systems", *IEEE Transactions on Software Engineering*, Vol 16 No 5, 5th May 1990, pp 510-522

Lederer AL, Prasad J: "The Validation of a Political Model of Information Systems Development Cost Estimating", *ACM Conference proceedings from SIGCPR 91*, pages 164-173

Lederer AL, Prasad J: "Nine Management Guidelines for Better Cost Estimating", *Communications of the ACM*, Vol 35 No 2, February 1992, pp 51-59

Lederer AL, Prasad J: "Perceptual Congruence and Information Systems Cost Estimating", *ACM Conference proceedings from SIGCPR 95*, pp 50-59

Lederer AL, Prasad J: "A Causal Model for Software Cost Estimating Error", *IEEE Transactions on Software Engineering*, Vol 24 No 2, February 1998, pp 137-147

Low GC, Jeffery DR: "Function Points in the Estimation and Evaluation of the Software Process", *IEEE Transactions on Software Engineering*, Vol 16 No 1, January 1990, pp 64-71

Matson JE, Barrett BE, Mellichamp JM: "Software Development Cost Estimation Using Function Points", *IEEE Transactions on Software Engineering*, Vol 20 No 4, April 1994, pp 275-287

McLeod G, Smith D: "Managing IT Projects", *Inspired Press*, Cape Town, July 1993, pp 7.1 -7.33

Marple II OM: "GUI-Point Analysis – A New Model for Software Size Estimation" *Florida State University*, Doctoral Thesis, 1997

University of Cape Town

Morgan JN: "A Cost Estimation Model for Fourth Generation Language (4GL) Software Development Environments", *George Mason University*, Doctoral Thesis, 1997

Mukhopadhyay T, Vicinanza SS, Prietula MJ: "Examining the Feasibility of a Case-Based Reasoning Model for Sofware Effort Estimation", *MIS Quarterly*, June 1992, pp 155-171

Offen RJ, Jeffery R :"Establishing Software Measurement Programs", *IEEE Software*, March/April 1997, pp 44-48

Pancake CM: "The Promise and the Cost of Object Technology: A Five-Year Forecast", *Communications of the ACM*, Vol 38 No 10, October 1995, pp 33-49

Park RE, Goethert WB, Webb JT: "Software Cost and Schedule Estimating: A Process Improvement Initiative", *Software Engineering Institute*, Special Report CMU/SEI-94-SR-3, Carnegie Mellon University, Pittsburgh

Parr FN: "An Alternative to the Rayleigh Curve Model for Software Development Effort", *IEEE Transactions on Software Engineering*, Vol SE-6 No 3, May 1980, pp 291-296

Paulk MC, Weber CV, Garcia SM, Chrissis MB, Bush M: "Key Practices of the Capability Maturity Model, Version 1.1", *Software Engineering Institute*, Technical Report CMU/SEI-93-TR-025, Carnegie Mellon University, Pittsburgh

Pfleeger SL, Jeffery R, Curtis W, Kitchenham B: "Status Report on Software Measurement", *IEEE Software*, March/April 1997, pp 33-43

Phan D: "Information Systems Project management: an integrated resource planning perspective model", Ph.D thesis, Dept of Management Information Systems, University of Arizona, Tucson, 1990

Pillai K, Sukumaran Nair VS: "A Model for Software Development Effort and Cost Estimation", *IEEE Transactions on Software Engineering*, Vol 23 No 8, August 1997, pp 485-497

Pressman R: "Software Engineering: A Practitioner's Approach", *McGraw-Hill*, New York, 1992

Putnam LH: "A General Empirical Solution to the Macro Software Sizing and Estimation Problem", *IEEE Transactions on Software Engineering*, Vol SE-4 No 4, April 1978, pp 345-361

Putnam LH, Myer W: "How Solved Is the Cost Estimation Problem", *IEEE Software*, November/December 1997, pp 105-107

Sackman H, Erikson WJ and Grant EE: "Exploratory experimental studies comparing on-line and off-line programming performance," *Communication of the ACM*, Vol. 11, #1, 1968, pp. 3-11.

Schach SR: "Software Engineering", *Aksen Associates Inc*, 1990

University of Cape Town

Shepperd M, Schofield C, Kitchenham B: "Effort Estimation Using Analogy", *Proceedings of ICSE18*, Berlin 1996

Smith RK, Parrish A, Hale J: " Cost Estimation for Component Based Software Development", *Proceedings of the 36th annual ACM conference on Southeast conference*, 1998, Pages 323 - 325

Stricker C: "Evaluating Effort Prediction Systems", Published in "Software Quality Assurance and Metrics : A Worldwide Perspective", *International Thompson Computer Press*, 1995, pp 281-294

Stutzke RD: "Software Estimating Technology", Unpublished paper, *SAIC*, <http://www.saic.com/Publications/techtrends/stutzke.html>, Last Accessed 1/9/97

Symons CR: "Function Point Analysis: Difficulties and Improvements", *IEEE Transactions on Software Engineering*, Vol 14 No 1, January 1988, pp 2-11

van Genuchten: "Why is Software Late? An Empirical Study of Reasons for Delay in Software Development", *IEEE Transactions on Software Engineering*, Vol 17 No 6, June 1991, pp 582-590

van Solingen R, Berghout E, van Latum F: "Interrupts: Just a Minute Never Is", *IEEE Software*, September/October 1998, pp 97-103

Verner J, Tate G: "Approaches to Measuring Size of Application Products with CASE Tools", *Information and Software Technology*, Vol 33, No 9, November 1991, pp 622-628

Vigder MR, Kark AW: "Software Cost Estimation and Control", *National Research Council Canada*, February 1994

University of Cape Town