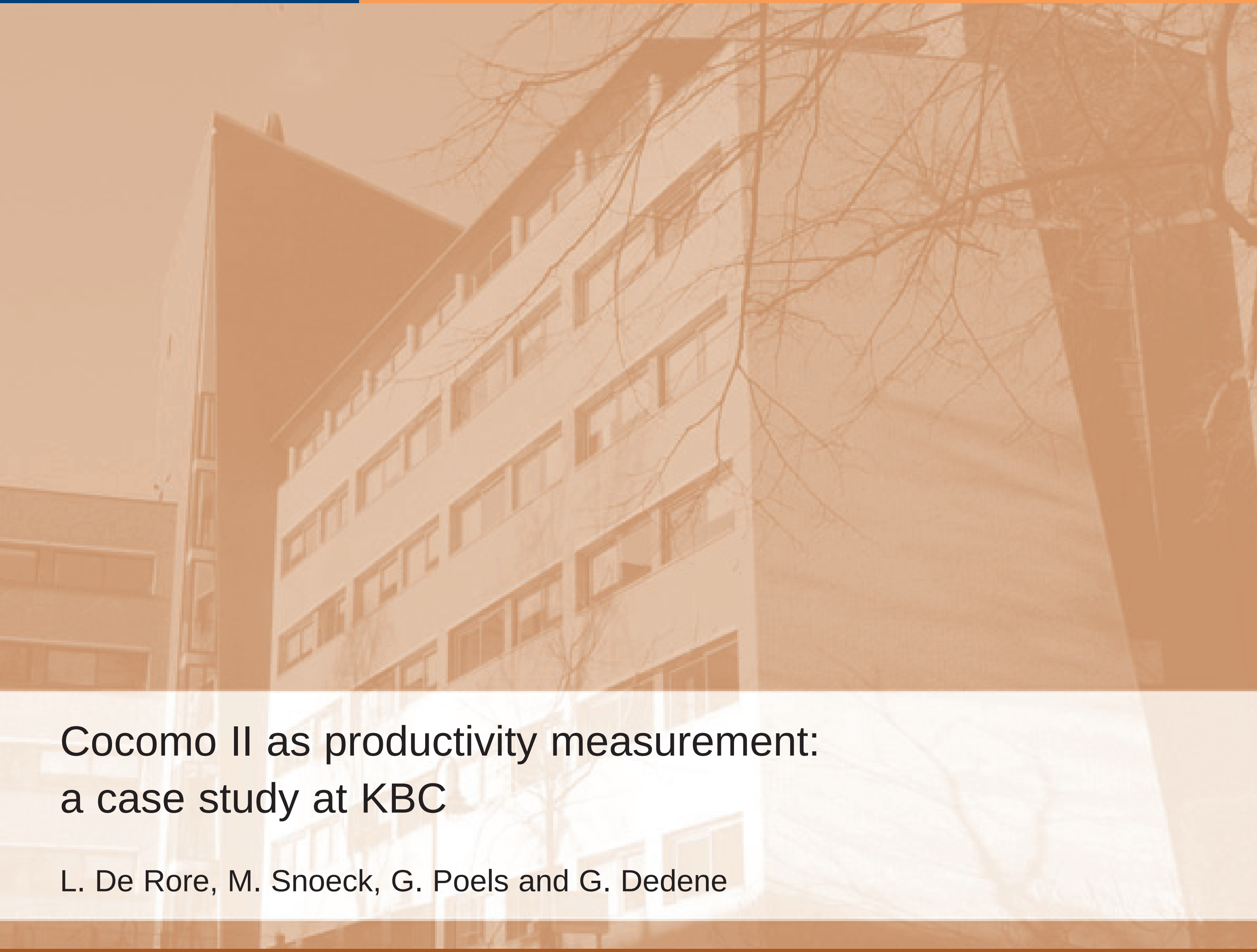
KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculty of Business and Economics



Cocomo II as productivity measurement: a case study at KBC

L. De Rore, M. Snoeck, G. Poels and G. Dedene

DEPARTMENT OF DECISION SCIENCES AND INFORMATION MANAGEMENT (KBI)

KBI 0829

Cocomo II as productivity measurement: a case study at KBC

Lotte De Rore[1], Monique Snoeck[1], Geert Poels[2], and Guido Dedene[1,3]

[1]Katholieke Universiteit Leuven, Department of Decision Sciences & Information Management, Naamsestraat 69, B-3000 Leuven (Belgium),
{lotte.derore;monique.snoeck;guido.dedene}@econ.kuleuven.be
[2]Universiteit Gent , Faculty of Economic and Applied Economic Sciences, Hoveniersberg 4, 9000 Gent (Belgium) ,
geert.poels@ugent.be
[3]Universiteit van Amsterdam, Amsterdam Business School, Information Management, Roetersstraat 11, 1018 WB Amsterdam (The Netherlands)

Abstract

Software productivity is generally measured as the ratio of size over effort, whereby several techniques exist to measure the size. In this paper, we propose the innovative approach to use an estimation model as productivity measurement. This approach is applied in a case-study at the ICT-department of a bank and insurance company. The estimation model, in this case COCOMO II, is used as the norm to judge about productivity of application development projects. This research report describes on the one hand the set-up process of the measurement environment and on the other hand the measurement results. To gain insight in the measurement data, we developed a report which makes it possible to identify productivity improvement areas in the development process of the case-study company.

Contents

1 Introduction

1.1 Measurements in Software Engineering

Measurement is an every day life activity. You can think of temperature, the price for goods, the distance between two cities or the size of our clothes. Measurements help us to understand our world and they allow us to compare it [18].

> "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules." ([18], p5)

Measurements can be used to deliver information about attributes of entities. An entity is an object (such as a person or a good) or an event (such as a journey) in the real world. An attribute is a property of such an entity. In fact, we don't (and can't) measure the entities but we measure the attributes of the entities. By performing measurements, we can make concepts more visible and therefore more understandable and controllable [18]. Sometimes, we think attributes are unmeasurable. "What is not measurable, make measurable" Galileo Galilei said, and indeed, we should try to measure the unmeasurable, in order to improve our understanding of particular entities and attributes. Especially in software engineering, lots of things are still seen as unmeasurable [18]. It is difficult to quantify what good software is or what a successful project is. Therefore, we need measurements in software engineering to assess the status of projects, products or processes. As Fenton [18] explains, measurements will help us to *understand* what is going on in the projects, help us to *control* what is happening and it will encourage us to *improve* our processes and products. One of these software metrics, namely software productivity, is the topic of this research report.

1.2 Software Productivity

Productivity can be defined as the rate of output per unit of input. In a classical manufacturing environment, this measure is rather straightforward: you can measure the effect of using labor, materials or equipment. The output can be measured as a number of products you deliver. In software engineering too, it could be useful to compare the output to the input. However, how can we define input and output in software engineering? Input is the amount of effort we spend on the project to deliver the software. But for the amount of output, there is no straightforward measurement. The problem is that there is no definition for what exactly is being produced with a software project [22]. One could see it as a physical amount of product, i.e. an amount of lines of code that are produced. However, one could also see the software product as a delivery of functionality or even express it in terms of the quality attributes it meets. All these measurements try to quantify the *size* of the delivered good, namely the software product. Therefore, we express the software productivity as one of size over effort.

$$\text{Productivity} = \frac{\text{Size}}{\text{Effort}}$$

1.3 Overview

Although software productivity is the main topic, first we will focus in Section 2 on several size measurements in software engineering: namely lines of code and functional size measurements. Section 3 describes on the one hand how these size measurements can be used to express productivity based on historical data. And on the other hand, we introduce our innovative approach to use an estimation model as productivity measurement. We illustrate this approach with a case study in the ICT-department of one of the major bank and insurance companies from Belgium, KBC. This is introduced in Section 4. A brief overview of the estimation model of our choice, namely the Cocomo II-model, is given in Section 5. Next, Section 6 describes the set-up of the measurement environment including the decisions we took and their consequences. Section 7 describes the first measurement results. Further, we developed a new report to gain insight in the influence of the effort multipliers in this company. The report and its application on our data is described in Section 8. Finally, Section 9 gives some reports for the future and Section 10 gives some conclusions about our approach to use an estimation model for productivity measurements and further research with respect to our results.

2 Software Size Measurement

Productivity can be measured as amount of product divided by the effort needed. Effort can be measured as working hours spent to the development of software. To measure the amount of product is quite more complex; therefore we use size as a measurement of product that is delivered. Historically, lines of code (LOC) has been the first measurement for software size. But because of the many paradoxes that it yields [22], more user oriented size measurements have been developed. What a project delivers to the user is not an amount of lines of code, but rather an amount of functionality. Therefore, another approach is to define software size as the amount of delivered functionality. In this section, firstly lines of code and the problems with this measurement will be investigated. Secondly, two function point based measurements, namely IFPUG [3] [6] and Cosmic FFP [1] [5], will be discussed.

2.1 Lines of Code

When measuring a software product, the most convenient thing is to measure what is delivered and to express this as the amount of lines of code that are delivered. Although it sounds as an easy way to express the size of software, this measurement yields some problems and paradoxes.

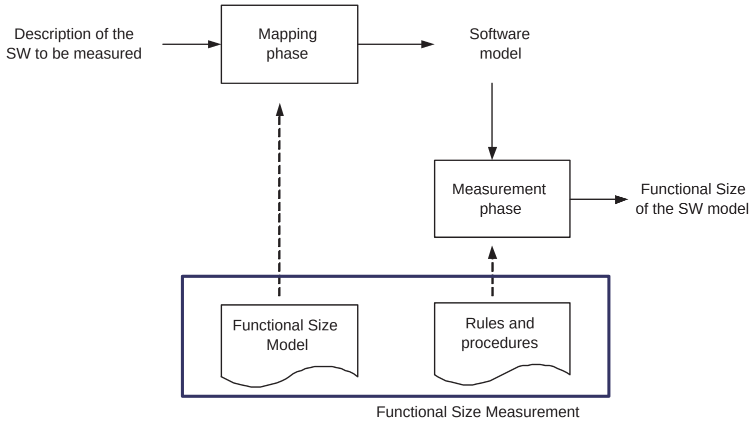
The first problem is that there is no universally agreed-to definition for exactly what a line of code is [22]. Some counts only include executable lines of code, while others also include data definitions and even comment lines. With this difference in count, a range of as much as 5 to 1 can be obtained between the most diffuse counting method and the most compact one [22]. Not only on the program level, but also on the project level there is a problem with the definition about which lines of code to count. It happens more and more that projects don't start from scratch, but rather add functions to existing systems. The question is then whether only the new lines of code should be included in the count or also the reused and modified ones.

Next, the measurement of size by counting the lines of code is very implementation dependent. Dependent on the kind of programming language used, the size of the software product will be different although the delivered product is the same. A high level language will need less lines of code to perform a same amount of functionality. In order to solve this problem, tables with conversion factors from one language to another are created [22] [15].

Not only has the programming language an influence on the amount of lines of code, but also the programming style. By using LOC as a size measurement, a programmer who writes a lot of code in an unstructured way will appear to be more productive than a programmer who writes his code after some thought in a well-structured way, despite the fact that the latter will have qualitatively better code.

Additionally, it is difficult to take into account the complexity inherent to the development. Lines of code that include a lot of complexity and calculations will be accounted for the same amount as regular lines of code. And most software engineering activities do not directly involve code.

As a result, although lines of code seems the right size measurement for software products, it doesn't always give a correct representation of what actually is delivered.



Description of the SW to be measured
Mapping phase
Software model
Measurement phase
Functional Size of the SW model
Functional Size Model
Rules and procedures
Functional Size Measurement

Figure 1: Functional Size Measurement

2.2 Functional Size Measurement

With the *Functional size measurement* (FSM), ISO/IEC 14143-1 [7], the focus is no longer on measuring how the software is implemented, but rather on measuring size in terms of functions required and delivered to the user, the functional size of software. The size of software is derived by quantifying the functional user requirements (FUR). This measurement approach is developed to give an answer to the search for a measurement independent of language, tools, techniques or technology used to develop the software. Size will be measured as the functions delivered to the user and will be derived in terms understood by the user. As can be seen in Figure 1, a functional size measurement method consists of two phases: a mapping phase and a measurement phase. In the mapping phase, the functional user requirements are transformed into a model that can be measured in the measurement phase. In the measurement phase, the measurement rules of the FSM method are applied to derive a size of the software, the *functional size*. In the following, two functional size measurements will be discussed: IFPUG function points [3] [6] and the Cosmic Full Function Points [1] [5].

2.2.1 IFPUG Function Points Analysis (FPA)

Introduction Function Points method was first developed by Allan J. Albrecht in the mid 1970's and was an attempt to overcome difficulties associated with lines of code as a measure of software size and to assist in developing a mechanism to predict effort associated with software development [12]. The method was first published in 1979 and in 1984 Albrecht refined the method. In 1986, the International Function Point User Group (IFPUG)

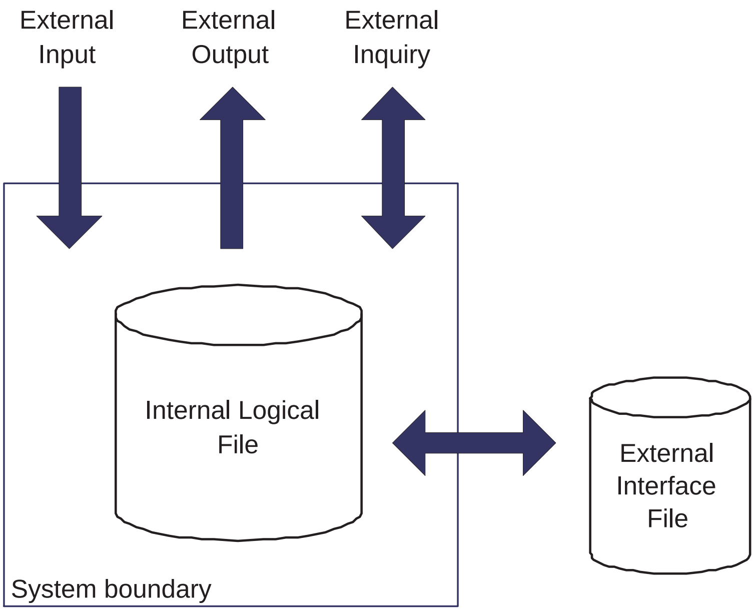


Figure 2: IFPUG Function Points

was created. They are responsible for promoting and encouraging the use of function points. Besides organizing conferences, they also published several versions of the Function Point Counting Practices Manual [28] and they offer professional certificates for practitioners of FPA. The ISO/IEC20926 defines the measurement rules for the IFPUG 4.1 method [6]. IFPUG supports also the data collection for the ISBSG database [4] that can be used for benchmarking. Several affiliate organizations of the IFPUG exist in Italy, France, Germany, Australia, India and many other countries.

The Method The functional size model of FPA defines software as a collection of elementary processes. There are two basic function types: data and transactional. Transactional functions represent the functionality provided to the user to process data. Transactional function types are: external inputs, external outputs and external inquiries. Data functions represent the functionality provided to the user to meet internal or external data requirements, the data can be maintained by the application in question (internal logical file) or can be maintained by another application (external interface file).

An *external input* (EI) is an elementary process in which data crosses the boundary from outside to inside; for example data coming from a data input screen. An *external output* (EO) is an elementary process in which

	Single	Average	Complex
External Input	3	4	6
External Output	4	5	7
External Inquiry	3	4	6
Internal File	7	10	15
External File	5	7	10

Table 1: Rating for elementary processes [28][6]

derived data passes across the boundary from inside to outside; for example a report. An *external inquiry* (EQ) is an elementary process with both input and output components that results in data retrieval from one or more internal logical files and external interface files. The input process does not update or maintain any internal logical file or external interface file; and the output side does not contain derived data. A screen full of customer address information would be an example of an EQ. An *internal logical file* (ILF) is a user identifiable group of logically related data that resides entirely within the application boundary and is maintained through external inputs. An *external interface file* (EIF) is a user identifiable group of logically related data that is used for reference purposes only. The data resides entirely outside the application boundary. [28] [6]

Each elementary process is given a rating - simple, average or complex - depending on the number of record element types, file type referenced and data element types involved in the process. A *record element type* (RET) is a user recognizable subgroup of data elements within an ILF or an EIF. A *file type referenced* (FTR) is a file type referenced by a transaction, each FTR is an ILF or an EIF. And a *data element type* (DET) is a unique user recognizable, non-recursive (non-repetitive) field. [28] [6]

In order to compute the number of unadjusted function points for a software project, a number of function points are assigned to each elementary process depending on the weight that is given - simple, average or complex. For example, a simple external input process will be assigned 3 function points, while a complex external output process will be assigned 7 function points. Table 1 lists all the scores given to each elementary process. The total amount of unadjusted function points is the aggregation of all the unadjusted function points for each elementary process.

A *value adjustment factor* (VAF) can be calculated to measure the contribution to the overall size of some general system characteristics including technical and quality factors. The fourteen characteristics are rated on a scale from 0 (no influence) to 5 (strong influence throughout) to determine

F1	Data communications
F2	Distributed data processing
F3	Performance
F4	Heavily used configuration
F5	Transaction rate
F6	On-line data entry
F7	End-user efficiency
F8	On-line update
F9	Complex processing
F10	Reusability
F11	Installation ease
F12	Operational ease
F13	Multiple sites
F14	Facilitate change

Table 2: General Systems Characteristics

the degree of influence. Table 2 lists all the characteristics. The influence factor, that needs to be multiplied with the unadjusted function points to calculate the total number of function points, can be determined with following formula:

$$\text{Influence factor} = 0.65 + 0.01 * \sum_{i=1}^{14} F_i$$

As each characteristic has a range from 0 to 5, the influence factor reduces the unadjusted function points at most with a factor of 0,65 and increases it at most with a factor 1,35.

Evaluation By using function points, one avoids the problems with the paradoxes of lines of code. However, the IFPUG-method is still considered to be mathematically flawed. It classifies user functions on an ordinal scale (simple, average, complex) and then subsequently uses operations that are (theoretically) not allowed on an ordinal scale [10] [11] [35] [25].

Performing a measurement with the IFPUG-method is not straight forward. Hence, a measurer will need training and an automation of the measurement will be very difficult. However in recent research, especially with respect to object-oriented development, automatization of function point counts are proposed [9] [8].

The IFPUG-method will not be applicable to all kinds of software applications. The method only takes into account the data movements that happen across the boundary of the system. Therefore, for example, an application with very complex data calculations inside the boundary will not be rewarded by this kind of measurement.

Software development has changed considerably since the introduction of this method. At that time, most developments occurred on a single platform, namely the mainframe, while now most of time, one works with distributed systems. The general system characteristic 'distributed data processing' tries to quantify the difference. However, as one factor can only induce a difference of maximum 5%, the IFPUG method is not able to quantify sufficiently the difference between a single platform and distributed software.

Additionally, projects and applications are no longer just complex, but also very complex and very very complex. However, with the IFPUG 4.1 method, functions can only be classified on the scale simple to complex. This means that the more complex functions do not get a higher rating than the complex functions. Variations in the FPA method are proposed to take into account the complexity [27].

2.2.2 Cosmic Full Function Points

Introduction In 1999, the Common Software Measurement International Consortium (Cosmic) published a new method of functional size measurement, Cosmic FFP (ISO/IEC 19761 [5]). This method was equally applicable to MIS/business software, to real-time and infrastructure software and to hybrids of these [2] [5]. As such, a new method was developed to address the critique that FPA is not universally applicable to all types of software.

The method The software to be measured by the Cosmic-FFP method [1] [5] is fed by input, it produces useful output to the users and manipulates pieces of information designated as data groups which consist of data attributes. In the mapping phase, first a hierarchical set of layers are identified. In each of these layers, the functional processes are identified. A *functional process* is an elementary component of a set of Functional User Requirements, comprising a unique, cohesive and independently executable set of data movements. For each functional process, the component data movements are identified. The Cosmic-FFP model distinguishes four types of data movements: entry, exit, read and write. *Entries* move data attributes from a single data group from the user across the boundary to the inside of the functional process; *exits* move data attributes from a single data group from inside the functional process across the boundary to the user; *reads* and *writes* retrieve and move data attributes from a single data group from and

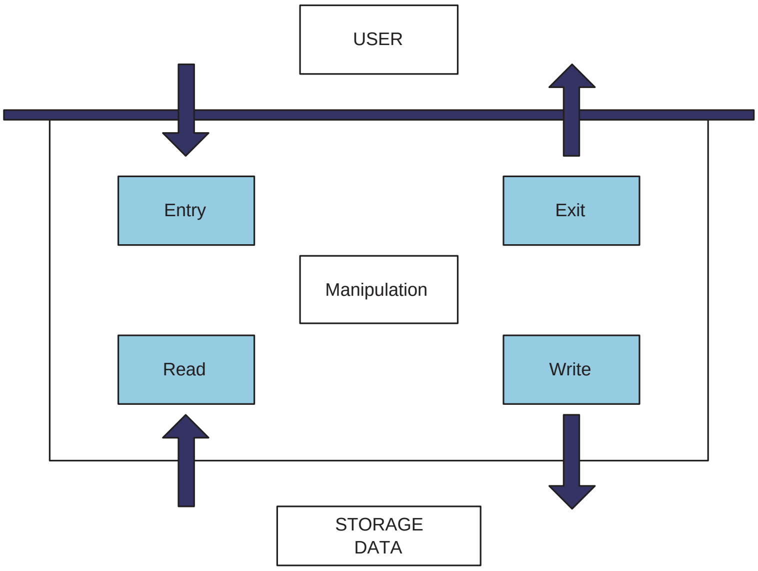


Figure 3: Cosmic FFP sub-processes

to persistent storage. [5] A graphical representation of these data movements can be seen in Figure 3.

In the Cosmic FFP model, each identified data movement is assigned a single unit of measure, namely 1 Cfsu (Cosmic functional size unit). The size of each identified layer corresponds to the aggregation of all data movements recognized in the layer. Finally, the total size of the software being measured corresponds to the addition of the size of each identified layer.

Evaluation The Cosmic FFP method is mathematically more correct that the FPA as they perform no transformation to an ordinal scale as in the IFPUG method. As the software product being measured is divided into several layers that are counted separately, the Cosmic FFP can be used to count multi-layered architectures [32]. Another consequence of the layering in the method is that the measurements are performed from different viewpoints: not only from the end users point of view, but for each layer a 'user' is identified.

This model only takes into account the data movements; there is no separate counting for the files such as the ILF and EIF in the IFPUG method. Also the data manipulations inside a layer are not counted. In addition, there is no adjustment factor to take into account some general characteristics of the software product developed.

The measurement method uses the functional user requirements as a starting point. However, as the measurement is based on the sub-processes or data movements in the software product, very detailed information and documentation needs to be available to perform the size count.

3 Software Productivity

3.1 Software productivity as size over effort

In order to assess productivity on projects, one should be able to compare the measured productivity rate with some normative productivity rate. This normative rate could be based on available data, for example an industry benchmark. The International Software Benchmarking Standards Group (ISBSG) [4] keeps a repository of project data. The historic data available in that database could help to suggest a workload on the basis of the measured function points. There is a large amount of data (about 3000 projects [34]) available of projects that used function points to measure the project output. As the Cosmic Full Function Points method is rather recent, there is few historical data available in the ISBSG database [34]. Given the limited number of Cosmic FFP data, the normative transformation from Cosmic FFP to effort is less well-grounded.

3.2 Estimation model as productivity measurement

Instead of taking historical data as productivity norm, one can also use an estimation model as productivity measurement. An estimation model can be used to set project budgets and schedules or to perform trade off analysis. It estimates the workload required for a project with certain characteristics. By comparing the estimation of the effort needed with the actual effort spend, one has an indication about how productive one is compared to the estimation model. When a project spends more time than prescribed by the model, the project will be judged as less productive compared to when the project spends less time than prescribed by the model. In other words, the estimation of the model can be used as the norm for the productivity measurement and projects are benchmarked against this norm.

Much research has been done about the calibration of estimation models (see e.g. [13, 30, 21, 24]). For parametric models, once information about the own projects is obtained, it can be interesting to calibrate the model to the specific company situation [38, 23, 19, 29]. This implies that in subsequent measurements, one will benchmark projects no longer with the model-norm but with a company-specific norm. Hence, one should only make changes to the model when one wants another frame of reference.

Remark that function points and cosmic full function points are no estimation models. They are size measurement models and these size measurements can be used to make an effort prediction [37]. An example of an estimation model is the Cocomo II-model [15].

4 The KBC-case

4.1 Goal of the company

With their program 'Expo 2005', the ICT-department of the KBC bank and insurance company not only had the ambition to reduce the ICT-costs between 2001 and 2005 with 30%, but also to improve their ICT services and to lift up their ICT performance to a level both quantitatively and qualitatively in conformity with the market. Part of ICT activities are the development of new applications. The company rightly wonders to what extent it delivers enough value with the development of new applications given the invested time and resources. In other words, it seeks an answer to the question: What is the productivity of ICT-development in our company?

The project described here, tries to find an answer to that question. By embedding one or more techniques into the development process, the company wants to measure the productivity of its projects on a continuous basis and to compare itself with other similar companies. An additional goal of developing productivity measurement techniques is to pinpoint the different parts of the development process where improvement is possible. Hence, with the help of these techniques, the company should be able to adjust its development process with respect to efficiency on a continuous basis.

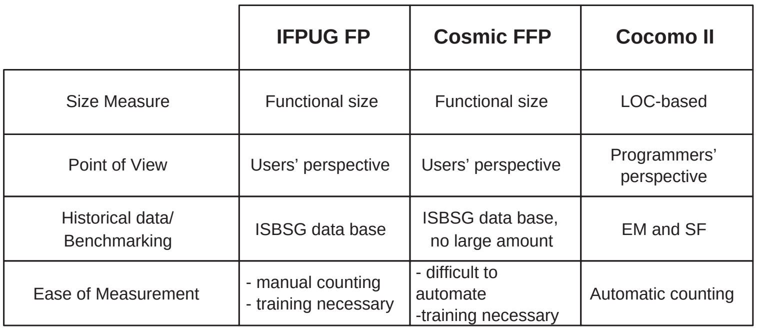
4.2 Choice of measurement method

The company's goal with this project was to set up an environment for assessing and measuring the performance of its software development departments. It was not their intention to analyse projects on an individual basis, but rather to look for trends in the whole ICT-development area. For this project, two function point based measurement methods were considered: IFPUG [3] and Cosmic FFP [1]. In addition we also considered the use of Cocomo II [15]. Although this model works both with function points and lines of code as size measurement, we mainly considered it as a LOC-based model.

For the choice of the appropriate measurement method several conditions and requests of the company had to be taken into account. First of all, the company wanted a *flying start*, meaning that years of measurements before they could gain any profit out of it was out of question. Also, the time and effort required to collect the necessary data should be kept to a minimum and not create overhead for project managers. As a result, techniques which offer the opportunity to automate the measurement process would be preferred. Finally, preference would be given to a technique allowing benchmarking with other companies. On the other hand, although the techniques under consideration entail productivity measurements on a project basis, it was not the intention to evaluate each project separately. Neither was it the intention to use the method as a tool to estimate the duration of a project. As a result of this, measurement can be done at project completion time (when more information is available) and accuracy of the measurement needs to be evaluated on a portfolio basis rather than on a per-project basis.

The methods under consideration are either function point based (IFPUG FP and Cosmic FFP) or based on lines of code (COCOMO II). In the first case, the size of software is measured in terms of units of functionality delivered to the user (function points) and subsequently a translation is made from function points to effort. The best possible sources for counting function points are user functionality as written down in software requirements or user manuals. The company has criteria formulating which projects have to document their requirements in a repository-based case-tool. As a result, not all projects document their user requirements in a repository-based case-tool. A major advantage of IFPUG over COSMIC is the availability of historical data about the mapping of function points into effort [4] as this allows benchmarking the own productivity against companies with similar development environments. A small scale project in which we attempted to count IFPUG function points by means of an *automatic* counting of screens and database tables (compared to a manual counting) was not conclusive [36]. As a result of this experiment, we concluded that in this company there are no artefacts that are systematically available and that can be used for automatic counting of function points per project. Using a function point based method would hence require manual counting. Because of the need of a 'flying start', this would take too long. A manual counting would also induce a small but nevertheless real additional cost per project that management is not prepared to pay for.

The Cosmic FFP method is rather recent [1]. It has the advantage of offering a more simple way of counting units of functionality which could be easier to automate than the IFPUG counting rules. It was therefore still a candidate to consider. On the other hand, there is not a lot of historical data available allowing the transformation of Cosmic FPP into effort estimations



	IFPUG FP	Cosmic FFP	Cocomo II
Size Measure	Functional size	Functional size	LOC-based
Point of View	Users' perspective	Users' perspective	Programmers' perspective
Historical data/ Benchmarking	ISBSG data base	ISBSG data base, no large amount	EM and SF
Ease of Measurement	- manual counting - training necessary	- difficult to automate -training necessary	Automatic counting

Figure 4: Comparison of different models

in different types of environments. Because of expected difficulties to link the measured function points to the expected workload of the project, the lack of benchmarking opportunities and remaining difficulties for automated counting, we decided not to use this method either.

The third method under consideration was the Cocomo II-method. Cocomo II uses lines of codes as size measure, and, as pointed out by Jones [22] there are many productivity paradoxes with lines of code. These paradoxes are the most important reason to reject LOC as size measurement and to use function points instead, as these were established to resolve these paradoxes. However, if one succeeds to set up an environment that rules out the famous paradoxes, then the Cocomo II-model can be considered as theoretically and mathematically more correct than the FP model [10] [11] [35] [25]. For Cocomo II, the project size is seen from the point of view of the implementation. This contrasts with the methods described before where project size is seen from the user's point of view. Because in this particular company the productivity measurements are to be used from the software developer's perspective, this former point of view is more interesting than the user's point of view.

A last point of consideration is the ease of measurement. The numbers of lines of code can be counted automatically. This last element was the deciding factor to choose for the Cocomo II-method. The main negative point with this method was the paradoxes with lines of code. Setting up an environment such as to rule out these paradoxes has been kept in mind in the further development of the project in order to avoid wrong conclusions being drawn from the results.

Before we describe the set-up of the measurement environment, the Cocomo II-model is briefly introduced in the next section.

5 Constructive Cost Model (Cocomo II)

In 1981, B. Boehm published Cocomo, the constructive cost model, a model to give an estimate of the number of man-months it will take to develop a software product. This first model [14], referred to as Cocomo81, has been developed based on expert judgement and a database of 63 completed software projects. However, software development has changed considerably since this model was introduced: projects follow a spiral or evolutionary development model instead of the waterfall process model Cocomo81 assumes, the complexity of software projects has increased and more and more projects use commercial off-the-shelf (COTS) components. As an answer to these evolutions, the constructive cost model was revised to the new version: Cocomo II [15].

Cocomo II consists of two models: the *Early Design model* and the *Post-Architecture model*. The *Early Design model* is a high-level model and can be used in the architectural design stage to explore architectural alternatives or incremental development strategies. This model is closest to the original Cocomo. The *Post-Architecture model* on the other hand is a more detailed model that can be used for the actual development stage and maintenance stage. It is the most detailed version of Cocomo II. Both the Early Design model and the Post-Architecture model use the same formula to estimate the amount of effort required to develop a software project. Besides these two models, also the *Application Composition model* is described by B. Boehm [15]. The *Application Composition model* can be used as sizing metric for applications composition; and the estimation is based on the number of screens, reports and 3GL components. In the remainder of this section we will focus only on the Post-Architecture model.

5.1 The Model

Formula The Cocomo II-model uses a size measurement and a number of cost drivers (scale factors and effort multipliers) to estimate the amount of effort required to develop a software project. The estimated effort is expressed as person-months (PM) and can be retrieved with the following formula:

$$\mathrm{PM} = \mathrm{A} \cdot \mathrm{Size}^{\mathrm{E}} \cdot \prod_{i=1}^{n} \mathrm{EM}_i$$

where

$$\mathrm{E} = \mathrm{B} + 0.01 \cdot \sum_{j=1}^{5} \mathrm{SF}_j$$

Precedentedness	Is the project similar to several previously developed projects?
Development flexibility	Is there any flexibility with respect to the requirements?
Architecture/Risk resolution	Is there a lot of attention for architecture? Are risks been taken into account?
Team cohesion	Are there problems to synchronize the different stake holders?
Process maturity	What is the CMM level of the development team?

Table 3: Scale factors [15]

In this formula, A and B are constant factors. The values for these two parameters were obtained by calibration of the 161 projects in the Cocomo II-database [15] and are initially equal to 2.94 and 0.91 respectively. In the exponent of the formula, one finds 5 *scale factors* (SF) that account for the economies or diseconomies of scale encountered for software projects of different sizes. When the exponent is smaller than 1, one will have an economy of scale. This means that when the size of the project doubles, the effect on the effort will be less than doubled. However, when the exponent is larger than 1, the project shows a diseconomy in scale and doubling the size will cause a more than doubling in the effort. The exponent consists of 5 scale factors: precedentedness, development flexibility, architecture/risk resolution, team cohesion and process maturity. Table 3 gives a description of each of these factors. Scale factors are defined on the level of the project. Each scale factor has a range of rating levels from *very low* to *extra high*. Each rating level has a weight. This weight is initially determined using the 161 projects in the Cocomo II-database. Initially, these projects were used to determine the values using multiple regression [17]. In later stadium, Bayesian analysis [16] is used to calibrate the initial weights of the parameters.

The *effort multipliers* (EM) are project characteristics that have a linear effect on the effort of a project. The post-architecture model defines 17 effort multipliers. Similar with the scale factors, effort multipliers have several rating levels, each with a weight that is initially determined using the projects in the Cocomo II-database. Each effort multiplier, with exception of the required development schedule, can be rated for an individual module. One can divide the effort multipliers into several classes: product factors (required software reliability, database size, product complexity, developed for reusability and documentation match to life-cycle needs), platform factors (execution time constraint, main storage constraint and platform volatility), personnel factors (analyst capability, programmer capability, personnel continuity, applications experience, platform experience and language and tool experience) and project factors (use of software tools and multisite development). Table 4 gives a description of each of these effort multipliers.

Size Measurement COCOMO II is a LOC-based method. As we have seen in the paradoxes that exist with lines of code, some guidelines for counting the size are necessary to have a good model estimation. COCOMO II only uses size data that influences the effort; this includes new code as well as code that is copied and modified. The size is expressed in thousands of source lines of code (kSLOC). In order to define what a line of code is, the model uses the definition check list for a logical source statement as defined by the Software Engineering Institute (SEI) [31]. One can also use unadjusted function points as a size measure, but then needs to translate these to kSLOC to import the size in the formula [33].

As stated before, the size measurement in the COCOMO II-formula includes all data that influences the effort of the project. However, reused or modified code can not be counted as much as new written code. Therefore, a formula is used to make these different counts equivalent in order to aggregate them into one size measurement for the project or module of the project. The equivalent kSLOC can be computed with the next formula:

$$\text{Equivalent kSLOC} = \text{Adapted kSLOC} \cdot (1 - \frac{\text{AT}}{100}) \cdot \text{AAM}$$

Where

$$\text{AAF} = (0.4 \cdot \text{DM}) + (0.3 \cdot \text{CM}) + (0.3 \cdot \text{IM})$$

$$\text{AAM} = \begin{cases} \frac{\text{AA}+\text{AAF}\cdot(1+(0.02\cdot\text{SU}\cdot\text{UNFM}))}{100} & \text{for AAF} \leq 50 \\ \frac{\text{AA}+\text{AAF}+(\text{SU}\cdot\text{UNFM})}{100} & \text{for AAF} > 50 \end{cases}$$

In these formulas, AT represents the amount of automatically translated code, DM represents the percentage of design that is modified, CM represents the percentage of code that is modified and IM represents the percentage of integration effort needed to integrate the adapted or reused code. One can see that the amount of effort to modify existing software is not only a function of the amount of modification (AAF), but depends also on the understandability of the existing software (SU) and the programmer's relative unfamiliarity with the software (UNFM). AA represents the effort that is needed to decide whether a reused module is appropriate to be used in the application.

Required software reliability	How large is the effect of a software failure?
Database size	How many test data is needed?
Product complexity	How complex is the product with respect to control operations, computational operations device-dependent operations, data management operations and user interface management operations?
Developed for reusability	Are the components developed so that they can be reused?
Documentation match to life-cycle needs	How many of the life cycles are documented?
Execution time constraint	Are there any constraints with respect to the execution time?
Main storage constraint	Are there any constraints with respect to the storage space?
Platform volatility	Are there major changes and how frequently are they on the platform?
Analyst capability	What is the capability of the analysts?
Programmer capability	What is the capability of the programmers?
Personnel continuity	What is the project's annual personnel turnover?
Applications experience	What is the experience of the project team with this type of application?
Platform experience	What is the experience of the project team with the platform?
Language and tool experience	What is the experience of the project team with the used languages and tools?
Use of software tools	Is there any software tool used to develop the product?
Multisite development	What is the site collocation and which communication support is there available?
Required development schedule	What is the schedule constraint imposed on the project team?

Table 4: Effort multipliers in Post-Architecture Cocomo II Model [15]

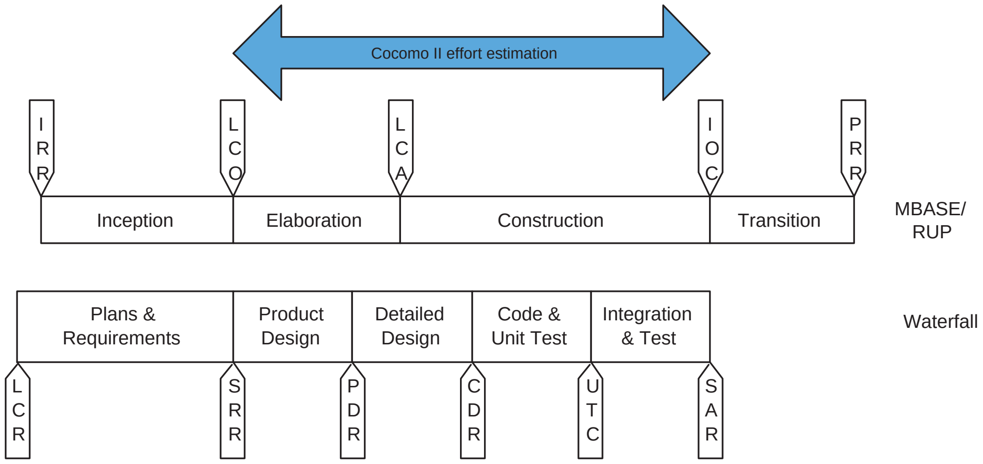


Figure 5: Cocomo II effort estimation

From this formula, one can see that software that is reused without any modification will still count for some equivalent SLOC. This is due to the effort needed to decide whether the module is appropriate to reuse (AA) and the effort needed to integrate the reused software in the overall product (IM).

With these guidelines to measure the size of the developed product, one can estimate the effort required to develop a software project. For both projects using a waterfall model as well as projects using a spiral development process, Boehm [15] gives a description about the phases that are included in the effort estimation with the Cocomo II-model. This can be seen in Figure 5. For a project developed with the waterfall model, the effort included in the estimation begins when the software requirements review (SRR) is completed and ends when the software acceptance review (SAR) is completed. When a spiral development process is used, the estimated effort begins with the life-cycle objectives review (LCO) and ends with the initial operational capability (IOC). This means that the requirements phase at the beginning of a project as well as the maintenance phase at the end is not included in the effort estimation.

5.2 Evaluation

The Cocomo II-model is a rather simple model. There is no training needed to perform the measurement. Once a good procedure is set up, the count of LOC is trivial. Then, you just have to determine the rating of the cost drivers and the formula can be used to obtain the estimated effort. However, the difficulty is in determining the correct and truthful values for these cost drivers. Although the model gives an explanation about each cost driver, there is still some subjectivity possible when determining the values for each cost driver. An incorrect assessment of one of the cost drivers can have a significant influence on the estimation [20].

The main disadvantage of the Cocomo II-model is the use of lines of code as a size measurement. As we have seen before, there are some paradoxes with lines of code [22]. Although the model only includes source lines of code and gives a description about what to include, there are still some of the paradoxes present, e.g. resulting from the use of different program languages or the different programming styles. When implementing this model, one should be aware of these paradoxes and keep them in mind when interpreting the results.

With respect to the use of the model as a productivity measurement, one can use the formula and the values of the cost drivers to benchmark themselves against other companies, more in particular against the 161 projects used to calibrate the model. As such, the model is seen as the norm for what a productive project should be. Nevertheless, as with most parametric models [38] [23], it can be interesting to calibrate the model with own projects. In doing this, you will loose benchmark possibilities with other companies or projects, but you will receive a model better adapted to the own environment. Consequently the new frame of reference will be the own projects in stead of the Cocomo II-norm. However, we need to mention that a lot of data is needed to perform a full calibration. The amount of data should be large relative to the number of model parameters. In this case, the model consists, besides the two parameters (A and B), of 17 effort multipliers and 5 scale factors. As a rule of thumb, 5 data points are needed for every parameter that needs to be estimated. This means, when we perform multiple regression on the log(effort) [17], there is a data set of at least 120 data points needed. Nevertheless, a first calibration of the constant factors A and B in the model, can be performed with less data (5 data points for A and 10 data points for factor A and B). This first calibration can already provide a much better fit to the own environment.

Although the model can give a frame of reference to define the productivity of a project, the main strength of the Cocomo II-model is the extensive list of project characteristics (the scale factors and effort multipliers). These provide a list of project characteristics that have an influence on software development, but they also quantify the amount of influence. As such, these cost drivers indicate the points of special interest where improvement in the productivity is possible.

6 Set-up of the measurement environment

As in a practical environment it is not always possible to faithfully follow the theory, this section describes the problems we encountered and the decisions we made during the different steps of the set-up process of the measurement environment.

6.1 Critical factors

Having opted for Cocomo II after the initial analysis, this section describes a number of critical factors we investigated because they are determinant to conclude if a measurement with the Cocomo II-model is possible in this particular company. The Cocomo II-formula uses lines of code and a number of scale factors and effort multipliers to estimate the effort required to develop a piece of software.

In order to make a correct assessment of the productivity, namely a correct comparison between the actual workload of the project and the outcome of Cocomo II, it is important to count exactly the same things as the Cocomo II-model prescribes. There are three major points to consider, namely, a correct time registration, a correct count of the lines of code and a correct match between the time registration and the lines of code.

6.1.1 Time registration

For the time registration it is important that all and only the workload is counted that is also included in the Cocomo II-model. Before we can do that, we have to see whether the life-cycle stages of a project in the company match with the life-cycle stages from Cocomo II. In the company, each development project goes through two major phases: *work preparation* (WP) and *work execution* (WE). According to Cocomo II, the *plans and requirements*-phase has not to be counted as part of the development effort. However, the WP-phase in the company is rather extensive and seems to include more than plans and requirements only. We therefore need to consider including part of the WP in the workload. An additional problem is that one *work preparation* can lead to several *work executions*. So there is no one to one matching between WP and WE. The difference in time spent to WP for the different projects ranges from 8% to 14% of the total workload (compared to [15] where plans and requirements amounts 2% - 15% (with an average of 7%) of the total project effort). The deliverables after WP are relatively uniform for all the projects. So, given that each project works in the same way and approximately in the same time, WP can not be a differentiating factor with regard to the productivity of WE. So we decided to not include the workload of WP in the total workload, knowing that although this will not distort internal project evaluation, this might yield a too positive picture when benchmarking against standard Cocomo II-results (since less work is included).

Another point of attention is the fact that the company works with *interface team activities*. These are activities that are necessary for a project, but that are offered in subcontracting for implementation to another team. The work executed in these interface teams has to be seen as a part of the project, because it is a request of the business. The fact that a question is partially or completely executed by an interface team rather than within the project team is a consequence of the way development teams are organised in the company. So not only the work performed by the project team itself, but also the work performed by the interface teams has to be captured in the computation of the workload. It is interesting to question whether using multiple teams has an influence on the productivity. This will be dealt with in a specific report.

Apart from identifying the tasks to include in the time registration, we also need data on the number of work hours spent for each task. The company works with a Project Diagnose System (PDS), as an internal ICT macro planning tool. Each team member has to register his/her hours performed for a project in PDS. Each project is attached to a PDS-record identified by a unique P-number. Reliable time registration means that there should be a correct time registration on the correct PDS-record by all employees. Since correct registration has been a company policy for many years, we can assume that data extracted from PDS is reliable enough for productivity measurement on a portfolio basis.

6.1.2 Counting lines of code

The lines of code that are necessary for the measurement are a second critical point. For the details of the implementation of the automated count, we refer to Section 6.3. Here we discuss the correct count in order to match with the project's time registration. The PDS records the effort spent on a project until the moment of delivery. Hence, for a correct match between size and effort, the lines of code have to be counted as they are at the moment of project delivery. In our case, the company had not yet implemented a version management system. Although all the versions of the software of the past five years are stored because of audit requirements, no automated procedures exist to restore the code as it was on a particular moment. As a result, the only code-base is the version in the run-time environment and it is not possible to reconstruct history. This means that the only way a lines of code count can happen correctly is for new projects at the moment of a unit of change (UOC), this is the moment that the changes are set in production environment. If the count happens at a later moment, changes to the code base can already have been made by other projects, so the count will be incorrect. Because the company does not have version management system, the situation at the moment of UOC, can not be restored at a later point in time.

In addition to establishing the correct moment of counting, we also need an inventory of all the modules that were created or modified during the project. In case of update of existing code, the before and after situation is needed. As explained in the next section, the inventory is accomplished by the P-number. Finally, dealing correctly with reuse of code was an important issue to resolve some of the known paradoxes of productivity measurements with lines of code. The details of how we dealt with reuse of code are explained in a separate further section.

6.1.3 Matching effort and size

Finally, a correct count of the lines of code and a correct registration of the workload is not enough: we also need a correct match between these lines of code and the time registration. In the case of our bank and insurance company, this matching was established with the PDS-record of a project. In that way, for each project, the time registration and also the lines of code can be collected. Important here is that there is a match between the two. No lines of code from teams that do not register time on the PDS-record should be counted. Similarly, no time should be included from teams that did not deliver lines of code or no time of tasks should be counted of which the lines of code are not included in the count. So there needs to be a correct matching between the time registration on a PDS-record and the inventory of modules. In our case, it was not always possible to allocate the right modules to the right project. As the counting was to be implemented for future projects a minor change was made to allow connecting the modules to a PDS-record. From now on, for each delivered module, a P-number has to be added that identifies the PDS-record on which the development time for that module has been/will be registered.

6.2 Scale factors and Effort multipliers

For each project, the cost drivers (scale factors and effort multipliers) need to be rated between *very low* and *extra high.* In order to do this as objectively

PMAT	Nominal	CMM level 2
PVOL	Low	Major change every 12 months minor change every month
ACAP	High	75th percentile
PCAP	High	75th percentile
PCON	Very High	3%/year turnover
SITE	High	same city or metropolitan area wide-band electronic communication

Table 5: Fixed values for the cost drivers

as possible, the tables with the description for each rating, given in [15], were transformed into multiple-choice questions. The possible answers were adjusted to the terminology used within the company and examples were added to give more explanation. By means of pilot runs, the clarity of the questions was checked. The persons questioned did not know the influence of their answer on the calculated effort as we left out the resulting values of the effort multiplier or scale factor.

Some answers are impossible for the kind of projects that appear in the company. For example, the most extreme value for the RELY effort multiplier, i.e. risk to human life, will never occur within the projects at KBC. To avoid people choosing these impossible answers, they were left out of the questionnaire. Also for the complexity effort multiplier, some answers were left out.

Additionally, there are also some cost drivers which will have the same value for all the measured projects. Therefore, these are not included in the questionnaire and are given a fixed answer. These answers are determined by the company. An overview of these factors is given in Table 5.

The company works with different platforms: mainframe-based development is used for the headquarters and the business logic, while for the distribution channels (branches and regional offices) a client server architecture is used. The platforms work with different programming languages and also the kind of projects are different. Therefore, a slightly different questionnaire with respect to terminology, explaining examples and impossible answers, was produced for the different platforms.

As can be seen in [15], some cost drivers are subdivided in multiple criteria. For example, product complexity is subdivided in control operations, computational operations, device-dependent operations, data management operations and user interface management operations. Each criterion forms a question, but also a summary question is included and serves as a control question. When the summary answer diverges too much from the average sub-answers, the projects need to be evaluated for correctness.

For each cost driver, a normative rating, namely the rating with the highest probability of frequency within the company, has been determined. Projects that differ too strongly from this norm are evaluated for correctness.

The questionnaire is to be sent to the project leader via e-mail after the completion of the project. There is still the remark that determining the cost drivers with a questionnaire is still subjective. However, with the actions taken (in-house terminology, explanatory examples, control questions, normative answers etc.), we reduced the influence as much as possible.

6.3 Lines of code

In order to have a correct baseline to compare with to calculate the productivity, the input for the Cocomo II-formula needs to be as correct as possible. Therefore, as far as possible, the guidelines to count the lines of code described in the book of Cocomo II [15] are followed, but with the condition that the count should happen automatically. The company works on different platforms. Mainframe-based development is used for the headquarters. For the distribution channels (branches and regional offices) a client server architecture is used. The business logic is mainframe-based, whereas the client side is developed on pc-based platforms. Both the server-side and the client-side use multiple program languages. Each of the different platforms has a different tool used to register the modules that have to be counted. For mainframe Changeman is used, for client platforms the tool Clearcase. As explained before, one of the major concerns was to rule out the paradoxes inherent to using LOC as size measurement. Jones [22] identifies a lot of paradoxes. In the following sections, we explain a number of choices that we made such as to rule out these categories of paradoxes. As a general remark, one should notice that since the goal of the project is not the measure productivity at the level of the individual project, but rather at the level of application portfolios, an imperfect resolution can be sufficient, as long as the paradoxes are resolved to a large extent.

6.3.1 Resolving paradoxes resulting from the use of different program languages

We have decided to only count the program languages that represent the majority of lines of code on the platforms. The languages that will be counted are: APS, VA/G, native COBOL, Sygel, JAVA and JSP. The other languages take less than 1% of the portfolio and are left out. A line of code written in one program language has not the same value as a line written in another program language. According to [22] one of the paradoxes with lines of code is that high-level languages will be penalized when their counts are compared to the counts of third generation languages. In order to make the counts of the different languages comparable and to be able to summarize them to one count of lines of code per project, conversion factors have been derived.

Within the training center of the company, several modules have been written in the different program languages in order to compare them with each other. That way, we obtain correct conversion factors specifically applicable for projects written by programmers in this company. For COBOL environments (mainframe), the conversion factors are:

$$1 \text{ line COBOL} = 2.1 \text{ lines APS} = 9.9 \text{ lines VA/G}$$

For the JAVA environment (client platforms), there is also a difference between handmade code and code generated by SYGEL. A code generator will generate more code than written by a programmer. According to experience experts, SygelJava is about 10% more voluminous than handmade Java and SygelJSP is about 50% more voluminous than handmade JSP. Reduction factors have to be applied to the generated code.

$$\frac{\text{SYGELJAVA}}{1.1} = \text{JAVA}$$

$$\frac{\text{SYGELJSP}}{1.5} = \text{JSP}$$

The counts on mainframe and the counts on the client platforms are kept separate because they are not comparable. One of the reasons being that there is a difference in the way they are retrieved. A second reason, as seen before, is that there is also a difference in the way the effort multipliers are determined.

6.3.2 Resolving paradoxes resulting from new versus modified lines of code

According to Cocomo II, a modified module has to be counted differently than a new one. Not all the lines of code of a modified module have to be counted, but only the modified lines. In our case, it was not possible to retrace the modified lines of code after delivery of a project. However, it turned out to be possible to count a modified line of code as a new line of code and a deleted line of code. Even then, there is still a difference between the count of modified modules on mainframe and the count of modified modules on the client platforms.

In the case of projects on mainframe, modules that are modified by a project are compared to the old modules that can be found via Changeman in the production environment. For client platform projects however this is more difficult. There are only two releases per year for client platforms, so mostly, in between those two releases, more than one project makes changes to one module (class in this case). There are two possibilities to handle this situation. Either all the changes are assigned to the project that makes the most changes or a correct match is made between each change and the project responsible for it. Although it was a lot of work to make the correct match, the second option was preferred. In the first option it could be that a project that makes a lot of small changes to a component gets all the changes charged and the project that makes one big change gets no changes charged. The first way of counting would distort the measurements too much. So, each time a project makes a change to a module, it has to add his P-number to that change. When counting lines of code, all the different versions of the module that appear between two releases have to be compared with the previous version and the P-number will allow to attribute the change to the correct project.

6.3.3 Resolving paradoxes resulting from multiple deliveries of the same piece of code

Many projects in the company do not deliver their project totally at once, but have staged deliveries. Management wants to stimulate the delivery for a project in a single stage. When a module has been delivered in multiple stages with each time some modifications, more modified lines of code will be counted in the case the lines of code are counted at each delivery than in the case where they are only counted at the last delivery. So in order to stimulate one delivery, only the last project delivery should be counted. As such, projects with multiple deliveries will be penalized.

Unfortunately, the information in PDS about which of the deliveries is to be considered as the last one is not always correct. The date of the last delivery is almost always correct, but it happens frequently that the last delivery shifts in time and that the corresponding date in PDS is corrected after that date has passed. More in particular: the correct date is frequently set after the moment we count the lines. As a result there is no other possibility than counting the lines of code after each project delivery and to summarize the counts after the last delivery.

6.3.4 Resolving paradoxes resulting from different programming styles

Although some of the paradoxes with lines of code have already been countered by using conversion factors and reduction factors for generated code, there are still a number of important paradoxes to address. A programmer who writes a lot of code in an unstructured way will appear to be more productive than a programmer who writes his code after some thought in a well-structured way, despite the fact that the latter will have qualitatively better code. In the case of our company, the danger for this kind of paradox is limited because most programmers receive the same in-house training and will therefore have a rather uniform programming style. Uniformity of programming style is also stimulated by having senior programmers review the code of junior programmers under their supervision.

Another problem is dead code. A program with a lot of dead code will gain lines of code for a same effort and so appear to be more productive. In our case, existing company policies require the project leader to ensure that no dead code is added to or left behind in a program. Also, care will be taken to communicate clearly that the measurements do not have the intention to evaluate results on a per project basis. In this way the desire to be more productive by creating more lines of code than necessary should decrease.

Finally, as described in the next paragraph, the reuse of code can also lead to a paradox. Reuse has to be stimulated, but a project that does not reuse anything and makes everything from scratch will have more lines of code and appear to be more productive. This paradox was the most difficult to deal with adequately and is explained separately in the next section.

6.3.5 Resolving paradoxes resulting from reuse of code versus new code

Reuse is an important but rather difficult issue to implement. Whatever way we measure, we had to ensure that reuse of code is rewarded and not penalized because there are less new lines of code delivered.

From an estimation point of view (e.g. Cocomo II-model), modules that are reused can not be counted like normal (written) code, but on the other hand it would be wrong not to count them. After all, time is spent to decide whether the module will be used and also to integrate the module into the program. From a productivity measurement point of view, modules that are reused should be counted completely, as you deliver the same output (lines of code), for less input (effort). However, using the Cocomo II-model as a productivity measurement, we define productivity as a comparison between the estimated workload needed for the project compared to the actual effort, rather than comparing the output with the input. Therefore, we estimate the effort following the Cocomo II way of calculating effort as close as possible. However, one could still object that a project which does not reuse while it should, will turn out to be as productive (more LOC and more effort) as a project that does reuse (less LOC and less effort, but same ratio) although the total effort is less in the latter. Yet at KBC, there is a coding practice that when a module with the needed functionality is available, you *have to* reuse this, rewrite these modules is just not done within this company. Therefore, to take into account reuse in our productivity measurement, we follow the Cocomo II-guidelines to count reuse.

In [15] a formula is described to transform the lines of code of a reused module into equivalent lines of code that can be added to the counted new lines of code. For the reuse without making any changes to the module, not all the criteria of the formula are applicable and the formula reduces to:

$$\text{Equivalent kSLOC} = \text{Reused kSLOC} \cdot \frac{(\text{AA} + 0.3 \cdot \text{IM})}{100}$$

Because in the company, mostly the same components have to be reused, each time the same kind of effort is performed to integrate the modules. We therefore decided to use standard values for each of the criteria in the formula. The assessment and assimilation increment (AA) is given the value 2, which means that basic module search and documentation are performed in order to determine whether the reused module is appropriate to the application. The percent of integration required for adapted software (IM) is set to 10, meaning that only 10% of the effort is needed to integrate the adapted software into the program and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size. That way, reused code will count for 5% of the number of lines that are reused.

Two possible ways for counting the lines of code of the reused modules were considered. Either a database can be set up with all the reusable components and their number of lines of code or either each time a module is reused, the lines of code are recounted at the moment that lines of code for the reusing project are counted. At first sight, the best solution seemed to use a database or file with the reused modules and their line of code count. However, it is possible that in the same UOC where you reuse a module, another project modifies that same module. By relying on the database an old version of the module would be taken into account, whereas you reuse the new version. So the wrong amount of lines of code would be assigned to the reused module, unless, for all the modified modules, the lines of code are recounted before the UOC. However, it is not clear how these modified reused modules can be identified. Neither can the modules that are reused for the first time be identified. So, the only option is to recount the reused modules each time they are reused.

Because recounting all the reused modules each time they are reused within an UOC can lead to a large CPU usage, we looked for another option. A *size* measurement for each program is given by the library system wherein all the programs are stored. This measurement diverges from the Cocomo II-standard: it gives the total amount of lines of source code, blank lines and comment lines included. If we apply a constant reduction factor per program language to take these blank and comment lines into account, this coarse measurement can be used for the count of reused lines of code. After all, reused code counts only for 5% of the number of lines that are reused. And even with this slightly different way of counting, the trends in the application of reuse within the company, what is most interesting for management, can still be deduced from these results.

Finally, we have to consider that reuse is recursive: a reused module can in its turn reuse other modules. In our measurements, reuse is only counted 1 level deep. If module A reuses module B and module B reuses module C, then only module A is counted as 'normal code' and module B as 'reused code'. Module C is not counted because starting from module A no effort has been performed to decide whether or not to reuse module C. That effort has been performed the moment that module B was created and should not be counted as part of the effort of creating module A.

For client platform projects that are written in Java and JSP, reuse can be detected by the *import*-statement. The problem is that with an import statement the whole class is reused while probably only a part of that class is needed. To take care of that, reuse for open systems projects is only counted for 3% in stead of 5%. Another problem are the wildcards. With an *import ** statement a lot of classes can be imported for reuse, while only some of them are actually needed. Within the company, there is an explicit directive not to use such wildcards to call functional classes. They still can use wildcards to call system technical classes, but following the Cocomo II-guidelines, these are not included in the code count. So we can assume that wildcards will not be distorting our results.

7 Measurement Results

7.1 Initial Calibration of the Model

The COCOMO II-model has been created by using expert judgement and statistical models on a dataset of 161 projects to determine the initial values of the parameters in the model [16]. As with the most parametric models, to improve the accuracy, the model should be calibrated to the own environment [23] [38]. This is important when one wants to use the model as an estimation model. In order to have a good reference model for the productivity measurements, we will include the data retrieved from the own projects to adjust the parameters to the company. As mentioned before, the company has no full version management. Hence, the idea of having an initial first calibration with the projects delivered in the year before the model was introduced in the company was not possible. The time registration and the values for the scale factors and effort multipliers for past projects could be found, but as the code base could not be reconstructed due to the absence of a versioning environment, no measure for the lines of code could be found.

As a consequence, the project had to start with the default COCOMO II values, and calibration will be done gradually as more project information is collected. After the first 3 pilot projects it was clear that the COCOMO II-model overestimates the effort in the case of this company. For each of the first three projects, the workload estimated by COCOMO II was 2.5 or 3 times higher than the actual performed workload. On that basis, a first calibration was performed by setting the constant factor A of the model [15] to 1 in stead of 2.94.

7.2 Measurement results

After one year of measuring, a first analysis is performed on the retrieved data base. The data base consists of 22 projects. All these projects were developed on mainframe. We only measured new application or added functionality. No migration projects or conversions due to technical reasons were measured.

The development environment was relative stable during the measurement period and is actually already stable for some years. However, we have to say that the administrative discipline lacks sometimes, which means that in some cases the recorded actual effort differs slightly from the real actual effort. This introduces some uncertainty in the measurement results. However, as long as the numbers we obtain are never used as absolute results, but only as an indication of areas that need further investigation, this is not a problem.

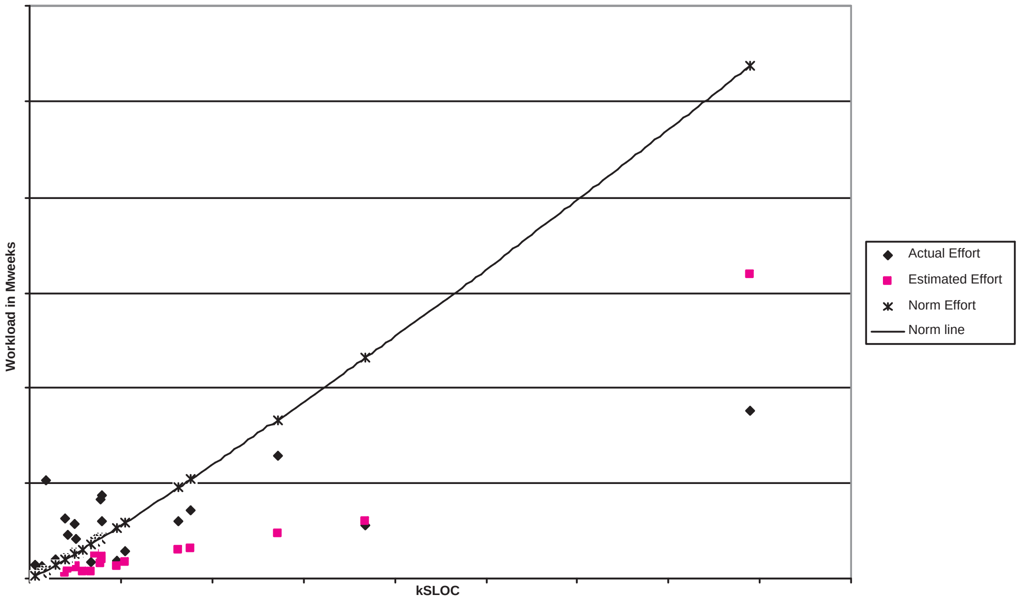


Figure 6: Workload versus LOC, no calibration (A=1;B=0.91)

	Cocomo II	Calibrated A and B
Pred(.20)	23%	32 %
Pred(.30)	32%	50 %

Table 6: Prediction Accuracy of the 22 Projects

The size of the 22 projects is in a range from 1.2 kSLOC to 158 kSLOC. The effort ranges from 24 person weeks to 351 person weeks. The accuracy of the predictions by the Cocomo II-model is shown in Table 6. Only 5 of the 22 projects are predicted within 20% of the actual effort and 7 of the 22 projects within 30% of the actual effort. Figure 6 shows a graph where the produced lines of code are plotted against the work effort. In the report we see the actual effort as well as the estimated effort according to the Cocomo II-model (with the initial calibrated $A = 1$ and $B = 0.91$). The norm line indicates the effort needed for the project when all cost drivers are set to the norm value (i.e. the nominal rating or the rating indicated as the norm within the company). As we can see in Figure 6, most projects need more effort than predicted by the Cocomo II-model. Can we conclude from this report that KBC is not productive compared to Cocomo II?

First of all, we performed an initial calibration by reducing the constant parameter A from 2.94 to 1. It is possible the pilot projects used for this

	Cocomo II	Calibrated A and B
MMRE for projects estimated within 30%	17%	10 %
MMRE for projects estimated not within 30%	100%	79 %

Table 7: MMRE before and after calibration

initial calibration were not a good sample for the typical project in this company. Consequently, more projects are productive compared to the initial Cocomo II-model than we can conclude from our results. Secondly, we made some assumptions during the set-up of the measurement environment. These assumptions will not influence the productivity between projects, but they can have an influence on the benchmark with Cocomo II. Therefore, we decided to use our measured project data to perform a new calibration and as such create a new reference to compare our projects with. A reference basis that is adjusted to the own company and projects.

We used the 22 projects to calibrate the two constant parameters in the Cocomo II-formula (using the Cocomo II-tool provided with [15]). The calibrated parameters are 15.16 for the constant factor A and 0.33 for the constant factor B in the exponent. With this new calibration, we have a better fit between the effort according to the Cocomo II-model and the actual effort (see Figure 7). According to Table 6 7 of the 22 projects are estimated within 20% and 11 of the 22 projects are estimated within 30% of the actual effort. As can be seen in Table 7, the mean magnitude of relative error (MMRE) indicates a significant improvement in the estimation for both the projects that are estimated well (within 30% of the actual) as those that are estimated poorly (not within 30%). For the calibrated effort, the average error is around 10% for 50% of the projects (these within 30% of the actual effort) and 79% for the 50% projects not within 30% of the actual effort.

As can be seen from the norm line in Figure 7, the constant factor B equal to 0.33 induces a huge economy of scale within this company. The exponent in the Cocomo II-formula can be at most about 0.65. Although there is no consensus about whether in software development there is an economy or diseconomy of scale [26], we can state that a large economy of scale is unusual. If these values are indeed applicable in the whole IT department at KBC, this would mean adding up all the projects into one large project would help to reduce the effort. This sounds not very plausible. Several things might cause this extreme economy of scale.

A first reason are projects included in the calibration. We only have 22 projects, this means each project can have a considerable influence on the calculation of the paramters. When we look at Figure 6, we see a possible

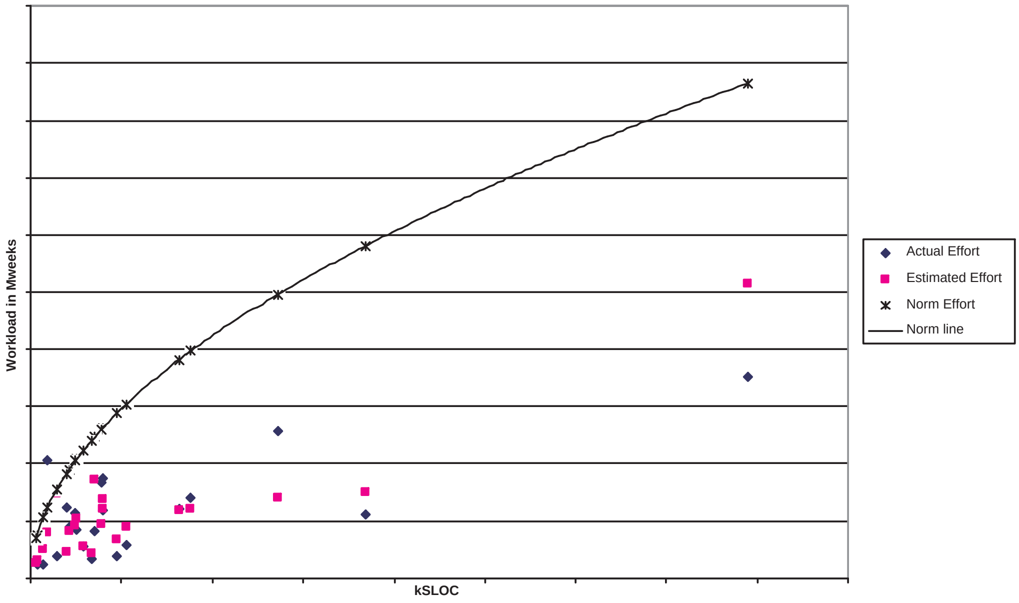


Figure 7: Workload versus LOC, calibrated A and B (A=15.16;B=0.33)

	A	B
Cocomo II	2.94	0.91
calibrated with 22 projects	15.16	0.33
calibrated with 21 projects	14.06	0.36

Table 8: Calibrated parameters

outlier. One project is quite larger than the rest and, in contrast to the other projects, seems productive compared to Cocomo II. To check this hypothesis, we performed a new calibration without the possible outlier. The results can be seen in Table 8. The exponential parameter will still induce a large economy of scale.

Due to the captation process in our measurement environment, not all the projects delivered during our measurement period were included in the measurements. Firstly, projects with multiple deliveries, for which the first delivery happened before the measurement environment was in place, could not be measured. Indeed, due to the fact that there is no full version management system, the lines of code of these first deliveries could not be restored or measured. Hence, these projects are not be included. Secondly, we made the assumption that we only count the most frequently used program languages. This means that projects with a too large portion of other program language code, could not be included in the measurements. Thirdly, as the measurements on mainframe and on client based projects are separate, the projects with a component on both platforms are not entirely included, but are divided into two projects: a mainframe project and a client based project. We might conclude that mostly the more complex projects were not captured in our analysis, which might have distort our calibration.

Another explanation can be found in the nature of bank and insurance applications. These applications mainly have a repetitive nature due to the transactions. Modeling n transactions will not need n times so much effort as modeling one transaction.

Notice that the 22 projects are all mainframe projects. Hence, we have no results yet from the client-based projects. Because the company works with different measurements of the lines of code for mainframe and client based projects and because the questionnaire for the project characteristics is also slightly different for the two environments, it is better to perform a separate calibration for projects on mainframe on the one hand and for client based projects on the other hand. Therefore, we will not use this calibration for the client-based projects, but perform an own calibration on these projects once we have collected measurement data about these projects.

8 Deducing Improvement Areas

8.1 Cost drivers: a source of information

One of the major strengths of the Cocomo II-model [15] is the extensive list of cost drivers (effort multipliers and scale factors). These cost drivers capture the characteristics of software development that affect the effort to complete the project. Hence, these factors are a very useful source of information in addition to the relative productivity of a project computed as the comparison between the actual effort and the calculated effort according to Cocomo II, to see whether your project is productive according to the Cocomo II-model. By looking at the ratings of the cost drivers for each project (no influence, positive, negative influence) the cost drivers can indicate which characteristics of your project had an influence on the estimated effort. As such, they are a management instrument that gives an indication which parameters need attention within the company in order to improve the productivity. In the remainder, we only focus on the effort multipliers and no longer on the scale factors.

As explained before, the actual value of the effect of a cost driver was determined by using the 161 projects in the Cocomo II-database. Since very

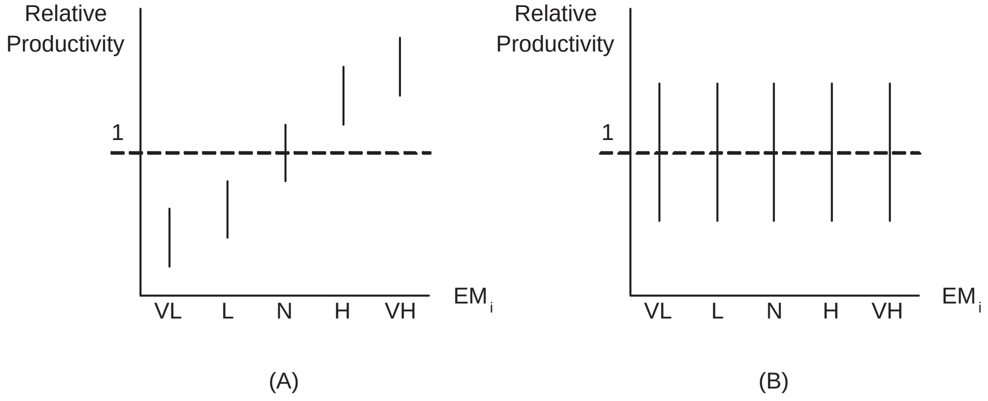


Figure 8: Initial Report

little information is available about these 161 projects, we can assume that chances are real that they are different from the own projects. Therefore, it is advisable to calibrate the model to determine whether the values are correct for the own environment. In the next section we describe which reports can be produced to identify the real effect of an effort multiplier in the project data.

8.2 Report for Effort Multiplier

8.2.1 Initial idea

The initial idea was to construct a graph for each effort multiplier in which the relative productivity (the Cocomo II-estimated effort divided by the actual effort) was plotted against the rating of the effort multiplier. The goal with this report was to identify for which ratings of the effort multiplier most projects were productive (relative productivity ¿ 1) and for which ratings most projects where unproductive (relative productivity ¡ 1). As such, one could obtain suggestions for which ratings of the effort multipliers some attention was needed in order to improve the productivity or in order to obtain a change in the rating of the effort multiplier. An example of the kind of report we expected to obtain is shown in Figure 8(A). In this report, we set out the relative productivity of projects against the rating for a particular effort multiplier EM_i.

The position of the different projects with the same rating for EM_i is presented with a line and the length of this line indicates the estimation error or the variance of the estimation. From the report shown in Figure 8(A) we see that projects with a rating of *very low* for EM_i have a lower relative productivity rating than the projects who rated EM_i as (e.g.) *low.* Hence, one can conclude it is advisable to focus the attention on projects with a lower rating and develop improvement action for these projects as they have a lower relative productivity. Remark that in the case of a perfect estimation model, all the projects would be on the dashed line (relative productivity = 1).

However, after we produced the reports with the first measurement results of our case study, we didn't feel this report provided enough information. Most reports looked more like Figure 8(B): for each rating of an effort multiplier, we had productive projects as well as non productive projects. What we really want to see in a report is whether the effort multipliers identified in the COCOMO II-model are indeed factors that influence the effort of the projects in the company. And if an effort multiplier indeed has an influence on the effort, is this influence comparable with the effect we measure in the data we retrieved from the projects we measured. We produced a new kind of report which we present in the next section.

8.2.2 New Report: Influence of Effort Multiplier

Assume we focus on the effort multiplier EM_i. What we want to see is whether this effort multiplier has indeed an influence on the effort of the projects in this particular company. When our estimated effort is equal to our actual effort, we get the following equation:

$$Effort = A \cdot Size^E \cdot \prod_{j=1 j\neq i}^{n} EM_j \cdot EM_i$$

In order to investigate the influence of EM_i, we calculate the *normalized effort with respect to EM_i*: this is the effort divided by all the other effort multipliers. As such we obtain an effort where the influences of all the other effort multipliers are neutralized. As an example, consider a project with two effort multipliers EM_1 and EM_2, with respectively values 1,10 and 1,20. The total Effort is calculated as $A \cdot Size^E \cdot 1,10 \cdot 1,20 = A \cdot Size^E \cdot 1,32$. If we want to investigate the effect of EM_1 only, we need to divide the calculated Effort by EM_2 to eliminate its effect: $NormalizedEffort = (A \cdot Size^E \cdot 1,10 \cdot 1,20)/1,20$. Remark that we implicitly assume here that the influence of the other effort multipliers is estimated correctly by the COCOMO II-formula. So in general we can state that:

$$NormalizedEffort_{EM_i} = \frac{Effort}{\prod_{j=1 j\neq i}^{n} EM_j} = A \cdot EM_i \cdot Size^E$$

As we can see from this equation, there is a linear relationship between the normalized effort and a size measure (namely: $Size^E$); the quantification of this linear relationship is dependent of the value of the effort multiplier. Consequently, we have expressed the influence of the effort multiplier EM_i as a linear relationship between a size measure and an effort measure. In the report, the $NormalizedEffort_{EM_i}$ is plotted against $Size^E$. A regression line with intercept equal to zero can be determined through the different data points with the same rating for the effort multiplier under consideration. The equation for this regression line will be

$$NormalizedEffort_{EM_i} = R \cdot Size^E$$

The slope of the regression line, denoted with R, is directly proportional with the value for the effort multiplier. Hence, we found a way to quantify the influence the effort multiplier has on the projects included in the analysis.

8.2.3 Interpretation of the report

The value of an effort multiplier indicates how much extra or less effort a project needs due to the rating for this particular effort multiplier. A *nominal* rating corresponds to no extra effort. Therefore, we will group the projects according to the rating given to the effort multiplier EM_i. For each group of projects with a given rating we will calculate a regression line and compare this line with the regression line for the projects with a *nominal* rating. In other words, as the slope of the regression line is directly proportional with the value for the effort multiplier for a particular rating, we will compare the slope of each regression line with the slope of the *nominal* regression line (this is the regression line through the projects that have a *nominal* rating for the effort multiplier). This will provide us with information about the effect of the different ratings of the effort multiplier.

For example, assume the COCOMO II-value for a *high* rating of the effort multiplier is equal to 1,10. This means that according to the COCOMO II-model, a project with a *high* rating for the effort multiplier will need 10% more effort than a project with a *nominal* rating for this effort multiplier. Comparing the slopes of the *high* regression line and the *nominal* regression line will provide us the information whether the effect of this effort multiplier predicted by the COCOMO II-model can also be detected in the data of the analyzed projects.

When we get the same ratio (in the example 1,10), we can conclude that the effort multiplier has indeed an effect on the effort and that the amount of this effect measured in the analysed projects is comparable to the estimated effect by Cocomo II. However, when we get a significantly different ratio, we can conclude that there is an influence but that the effect of this effort multiplier is different in this company. This indicates an area of investigation as there might be an opportunity for productivity improvement. For example, assume we get the ratio 1,20. This means for the analyzed projects, a project with a *high* rating will not need 10% more effort as prescribed by the Cocomo II-model, but needs 20% more effort than a *nominal* project.

On the other hand, when we get a ratio equal to 1, we can conclude that the effort multiplier does not have an effect on the effort. And therefore, we should not credit projects with a *high* rating with more effort in the estimation.

In summary, comparing each regression line with the *nominal* regression line yields two types of information: Firstly, when the slopes are different this demonstrates that the effort multiplier has indeed an effect on the effort. Secondly, the comparison of the values also shows whether the effect is comparable to the amount prescribed by the Cocomo II-model.

As explained in the previous chapter, the most powerful information can be found in the cost drivers of the Cocomo II-model. This information can be revealed with the report we described in section 8.2. In this section we will illustrate the reports we obtained for the effort multipliers in the model. Because we only have 22 projects and additionally, these projects are scattered over the several ratings for each effort multiplier, we are aware that this amount of data is insufficient to provide enough confidence that the slopes capture the correct effect of the effort multiplier. Therefore we did not use the slopes as an absolute value, but we used the reports as input for a discussion with the experts in the several domains of the effort multipliers. When the slope corresponds with the feeling/experience of the experts, we certainly have an indication about the real effect of this effort multiplier in this company.

Remark that the slopes of the regression lines do not have the same value as the values for the effort multipliers in [15]. The reason is that the factor A is also included in the slope as well as the fact that within the company we measure in weeks rather than months and one month consists of 148 hours, while in the Cocomo II-model, a month consists of 152 hours. However, these factors are constant for every project, so when we compare the slope of two regression lines, these factors will be neutralized.

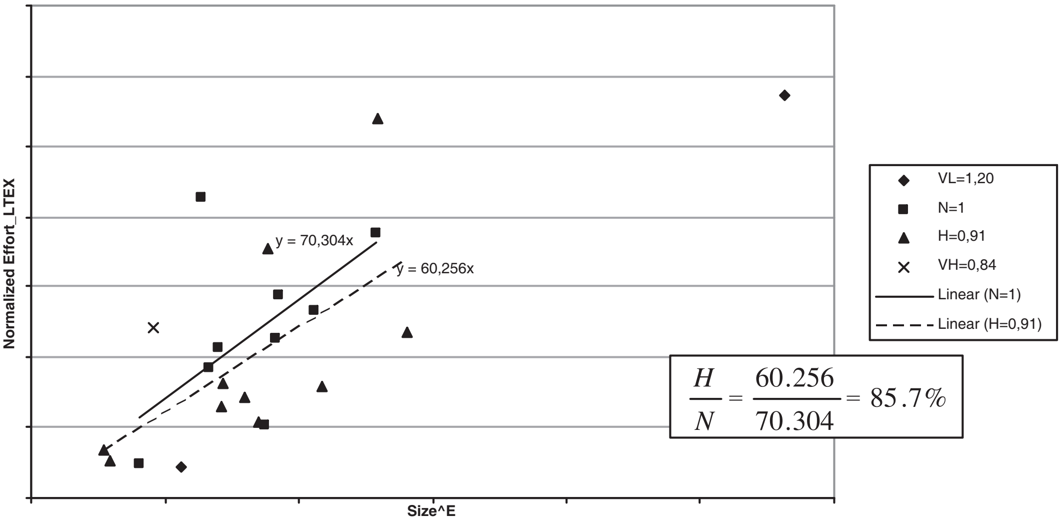


Figure 9: LTEX report

8.3 Reports with KBC-data

8.3.1 Language and Tool Experience, LTEX

Figure 9 shows the report for effort multiplier Language and Tool Experience (LTEX). Two regression lines are plotted: one for the *nominal* rating of LTEX and one for the *high* rating. The *high* slope in proportion to the *nominal* slope shows that a team with an overall language and tool experience of 3 years with the languages and tools used in the project (*high* rating) will only need 86% of the effort compared with a team with an overall experience of 1 year (*nominal* rating). This is a slightly larger effort reduction than Cocomo II predicts (91%).

As the effort reduction with more language and tool experienced teams is slightly larger than according to the Cocomo II-prediction, it might be advisable to aim for a higher rating for this effort multiplier. This means that we should strive to put together teams with a larger overall language and team experience. However, the company deals with a technology switch at this moment and in addition it employs a lot of external people who are not familiar with these technologies and therefore need education. With respect to this cost driver, at this moment, there is no action for productivity improvement possible.

Platform Experience, PLEX Figure 10 shows the report for effort multiplier Platform Experience (PLEX). Two regression lines are plotted: one

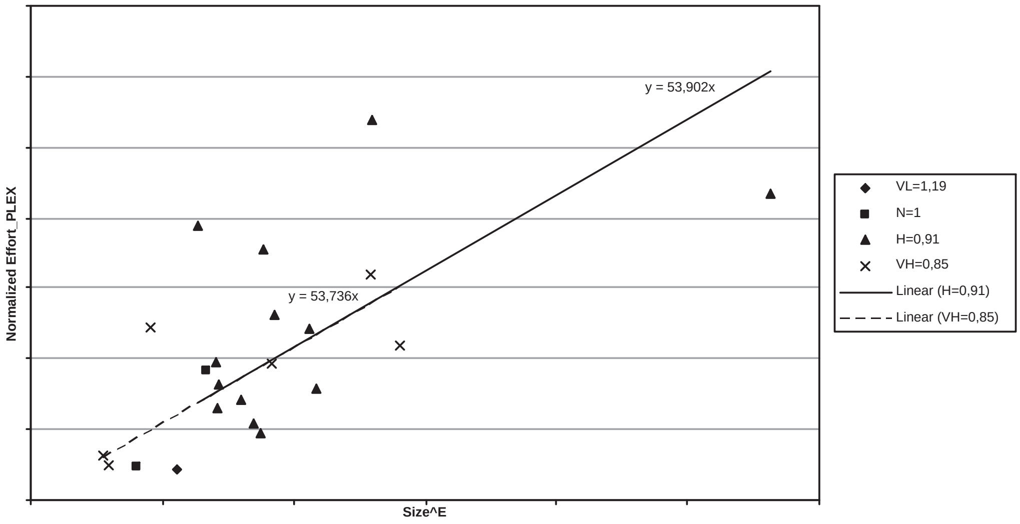


Figure 10: PLEX report

for the *high* rating of PLEX (i.e. an average of 3 years experience with the platform) and one for the *very high* rating (i.e. an average of 6 years of experience with the platform). There is not enough data to plot a *nominal* regression line to compare our slopes with. However, it is clear that the slopes of the *high* and *very high* rate are equal, which means that a rating of *high* and of *very high* have the same influence on the effort. Cocomo II, on the other hand, predicts an additional effort reduction of 6% for projects with a rating of *very high* for PLEX compared to projects with a rating of *high* for this effort multiplier.

As the equality of the slopes of the regression lines indicates, the platform experience has no further influence on the effort and productivity within this company once the team reaches an average of three years of experience. It seems plausible that after 3 years of experience with the same platform you know everything about this platform. However, as there is not enough data available about the less experienced teams, we can not generalize this observation and state that this effort multiplier has no influence at all in the company. Most likely, also in this company, there will be an effort reduction compared with projects with a *nominal* or lower rating. Nevertheless, teams with an average experience of more than 3 years should not have a further reduction of the allowed development time than teams with an average

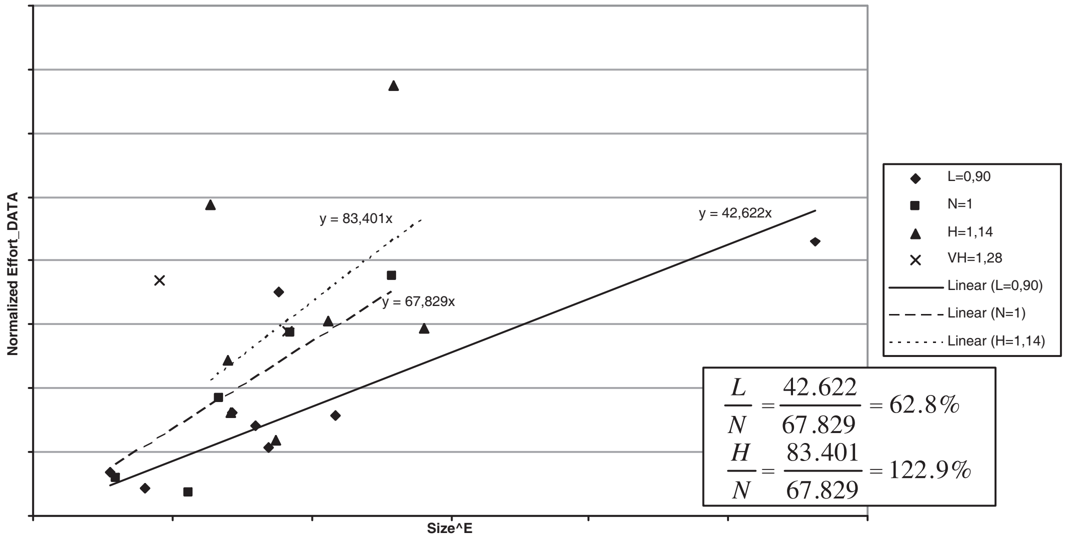


Figure 11: DATA report

experience of three years.

8.3.2 Database Size, DATA

Figure 11 shows the report for effort multiplier Database size (DATA). This effort multiplier captures the effect of large test data requirements on product development. Three regression lines are plotted: one for the *low* rating, one for the *nominal* rating and one for the *high* rating. Comparing the slopes shows that this effort multiplier has more effect than predicted by COCOMO II. A *low* rating (use a copy of production data in the accept environment as test data) will lead to a reduction of 37% in the effort, while a *high* rating (create considerable amount of own test data) will lead to an increase of 23% in the effort compared to the *nominal* rating (create a minimum of own test data). These in- and decreases are larger than predicted in the COCOMO II-model (10% decrease for *low* rating and 14% increase for *high* rating).

Although we can not take the slopes as absolute numbers, from this report we can state that the DATA effort multiplier has a serious effect on projects in this company and therefore definitely needs attention. An inquiry with the testing team revealed that there are indeed some issues with test data, but they had no idea about the amount of effect on projects. A possible explanation for the values is that the required effort to find and manage test data is substantial in this company. Additionally, everyone works on the same test data from the infrastructure systems. This means, once a project has used this test data, afterwards most probably many updates have been made to the data by other projects. As a result, the original test data is lost and therefore it is no longer usable for the own project. For example, suppose you need to test software on account numbers and you need accounts with a negative balance. When an account has initially a negative balance, it is possible that after the test run this account has a positive balance and therefore it is no longer useful. The problem is that after one or more test runs, one does not know anymore whether the balance of the accounts in the test data set is positive or negative. This leads to additional effort required to re-create a usable set of test data over and over again.

Because of the identified influence of this test data factor, the company realized it was necessary to undertake action to improve the test process. The report provided enough convincing material to build a business case in order to receive the permission from the CIO to spend the necessary resources to facilitate the retrieval and creation process of physical test data. The test team identified three potential actions: firstly, the construction of master and work test data bases. The master test data base is a stable set of test data that can be copied to a work data base for the test cases in projects. This should resolve the problems connected with the fact that everyone uses the same test data: the master data set always allows returning to the initial (and known) status of test data. Secondly, a set of standard queries on infrastructure systems can be supplied to more easily retrieve test cases. Thirdly, test data creation scripts can be provided to (re)initialize a set of test data. This third option is an alternative for the first one that can be very expensive if it leads to an extra test environment. However, this third option might have the restriction that the scripts can only be applied to relative simple structures of physical data. These three options will be further investigated to decide which one can be implemented.

8.3.3 Main Storage Constraint, STOR, and Execution Time Constraint, TIME

Figure 12 and Figure 13 show the reports for effort multiplier Main Storage Constraint (STOR) and Execution Time Constraint (TIME). In the company, they don't make a difference between these two parameters and therefore the two measures are considered as a measurement for taking into account performance issues. For the STOR effort multiplier (Figure 12), a *high* rating regression line and a *nominal* rating regression line are plotted. The ratio between the slopes indicate that a project with a *high* rating (some extra attention for main storage issues) and *nominal* rating (no extra attention for main storage issues) leads to an increase of 73% in the effort, which

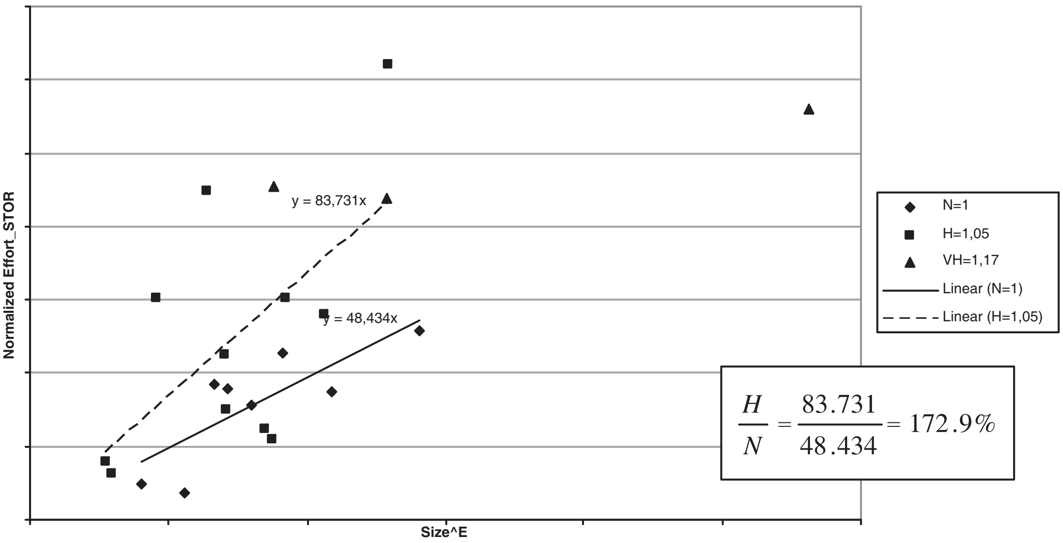


Figure 12: STOR report

is a large difference with the 5% predicted by the Cocomo II-model.

For the TIME effort multiplier (Figure 13), three regression lines are plotted (*nominal*, *high* and *very high*). Similarly as with the STOR effort multiplier, we see a large increase in effort when extra attention is needed for execution time issues. A *high* rating leads to an increase of 97% and a *very high* rating leads to an increase of 120% in the effort compared with the *nominal* rating. The Cocomo II-model predicts an increase of 11% and 29% for respectively a *high* and *very high* rating.

According to the experts, the values we retrieve with these reports are higher than what they experience in practice, although they agree that taking into account limitations with respect to main storage and time execution means extra design work for the project team. Nevertheless, these reports are definitely an indication that dealing with performance issues needs improved assistance in this company. At the time the measurements took place, there was not enough knowledge about performance issues present. Also project teams would spend too little attention to performance issues during the functional design stage, which would lead to rework in further stage in the development cycle. Furthermore, there was no procedure available for the set up of online performance tests. And finally, the complexity of systems and transactions constantly increases. All these factors can give an explanation why performance is an issue that needs attention.

Currently, there is already ongoing work, mainly through coaching, to improve the knowledge about performance in project teams. As it is identified

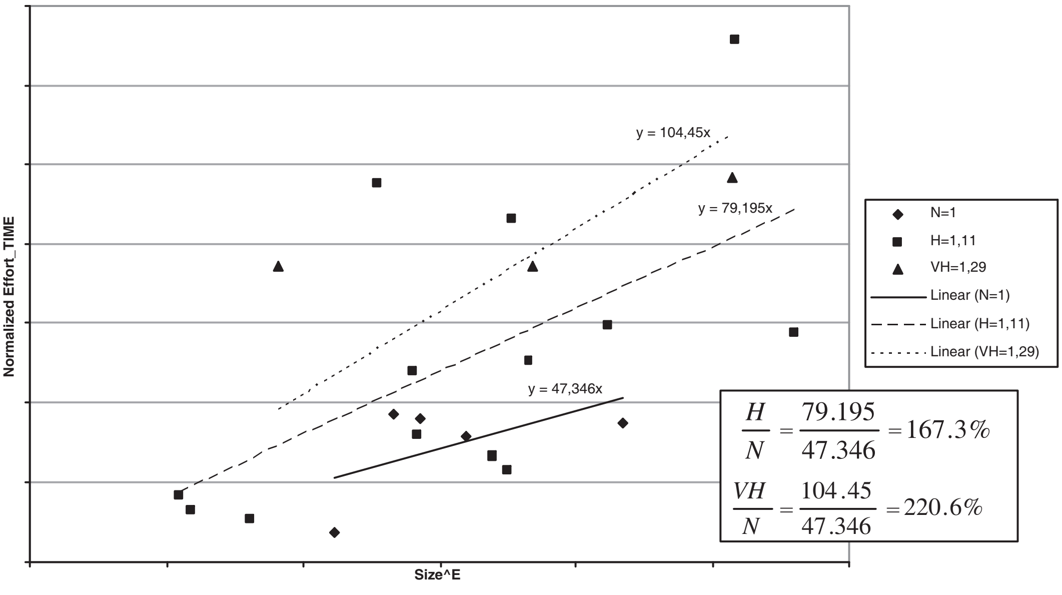


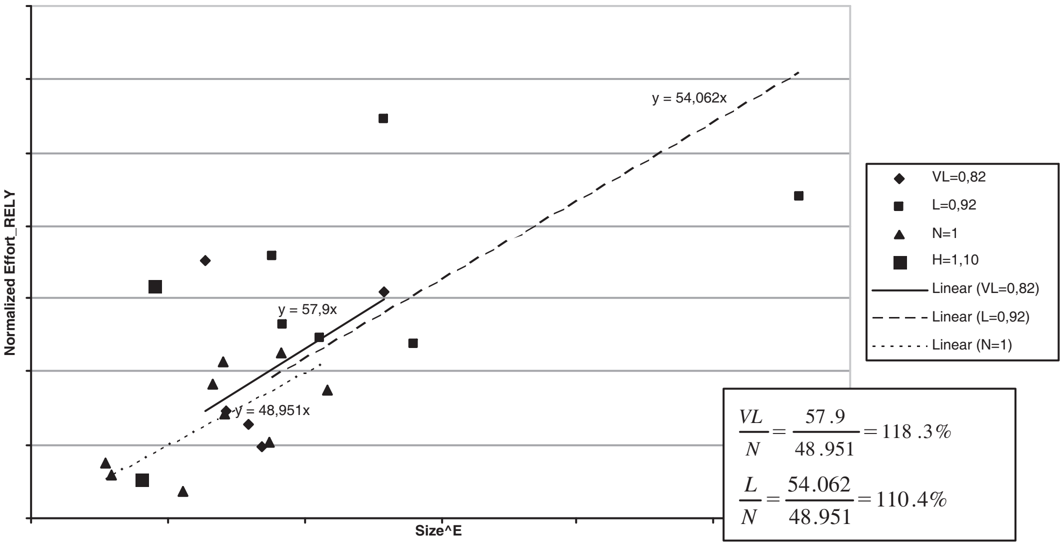
Figure 13: TIME report

that performance issues are not tackled from the beginning of the development cycle, a new initiative has been taken to have more attention for performance issues during the project inception phase, more particularly at the moment the requirements of the physical architecture are defined.

8.3.4 Required Software Reliability, RELY

Figure 14 shows the report for effort multiplier Required Software Reliability (RELY), this effort multiplier measures the extent to which the software must perform its intended function by means of the effect of a failure of the software. Three regression lines are plotted: one for the *very low* rating (slight inconvenience), one for the *low* rating (low, easily recoverable losses) and one for the *nominal* rating (moderate, easily recoverable losses). These regression lines lay in a reverse order than we would expect when we look at the values Cocomo II predicts. An application for which a failure causes only a slight inconvenience needs more effort than an application for which a failure causes moderate losses.

A possible explanation is that this effort multiplier has no effect in this company. All the applications in the mainframe environment work with the same 'business logic' in a relative strong integrated environment. Consequently, all applications are more or less equal critically. Although the effect of an error might be less, the project needs to deliver a certain amount of



$$\frac{VL}{N}=\frac{57.9}{48.951}=118.3\%$$

$$\frac{L}{N}=\frac{54.062}{48.951}=110.4\%$$

Figure 14: RELY report

quality that is standard for all the projects. From this we would expect to see no difference in the slopes of the effort multiplier. Hence, this is no explanation why the projects with a lower rating need more effort than the projects with a *nominal* rating.

8.3.5 Product Complexity, CPLX

Figure 15 shows the report for effort multiplier Product Complexity (CPLX). Two regression lines are plotted: one for the *low* rating and one for the *nominal* rating. These regression lines lay in a reverse order than prescribed by Cocomo II. A project with *low* complexity needs 66% more effort than a project with *nominal* complexity, while Cocomo II prescribes that a *low* rated project needs 13% less effort compared to the *nominal* rated project.

The results for this effort multiplier are contra-intuitive. We can not give an explanation why this effort multiplier has this strange influence. The CPLX effort multiplier is subdivided in several criteria, which result in several questions in the questionnaire. To define the rating of the CPLX effort multiplier, a weighted average of these questions is used. Maybe this weighting had an influence on the result.

8.3.6 Developed for Reusability, RUSE

Figure 16 shows the report for the effort multiplier Developed for Reusability (RUSE). This effort multiplier takes into account the extra effort needed

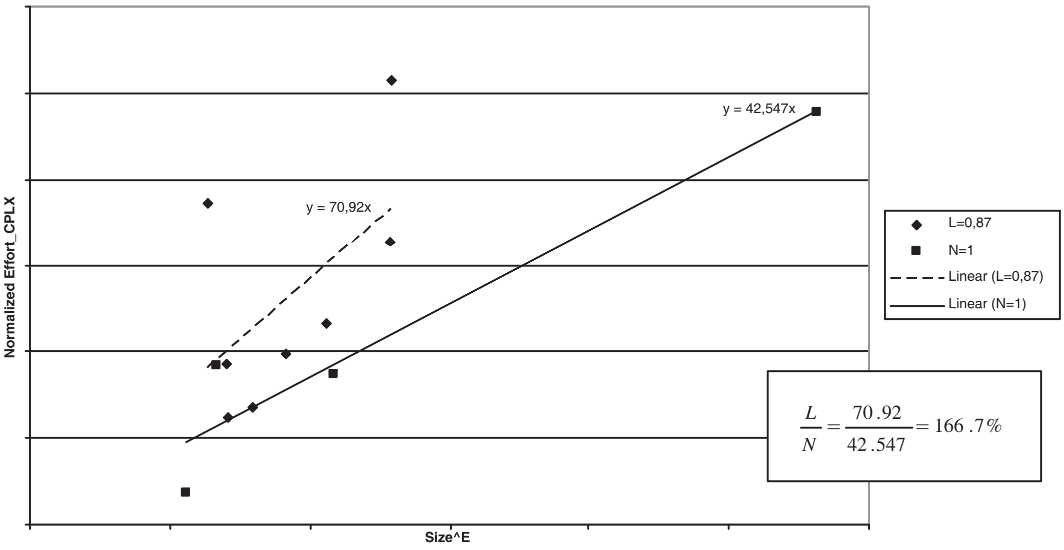


Figure 15: CPLX report

to develop a reusable module. As can be seen from the slopes, making a module reusable needs more effort than prescribed by Cocomo II. A *low* rated project needs 24% less effort and a *high* rated project needs 37% more effort than the *nominal* rated project compared to 5% less and 15% more effort Cocomo II predicts.

From the slopes we can conclude that it takes a lot of effort to make an application multi-company. This is also confirmed by the core team of technical design. For such applications, more functional and technical design is needed. Additionally, to develop reusable components means the application has several connectors. Often these connections need to be in place before the actual delivery of the application due to the planning of the connectors. As a consequence, the project has intervening 'tactical' deliveries before the actual release which is less efficient than a 'strategical' delivery at the release date.

Possible improvement actions that will be further investigated are knowledge exchange with respect to reusability, improve the functional documentation and adjust the planning on the strategical solution rather than the tactical ad hoc deliveries.

8.3.7 Documentation Match to Life-cycle Needs, DOCU

Figure 17 shows the report for the effort multiplier Documentation Match to Life-cycle Needs (DOCU). As can be seen on the figure, only one regression

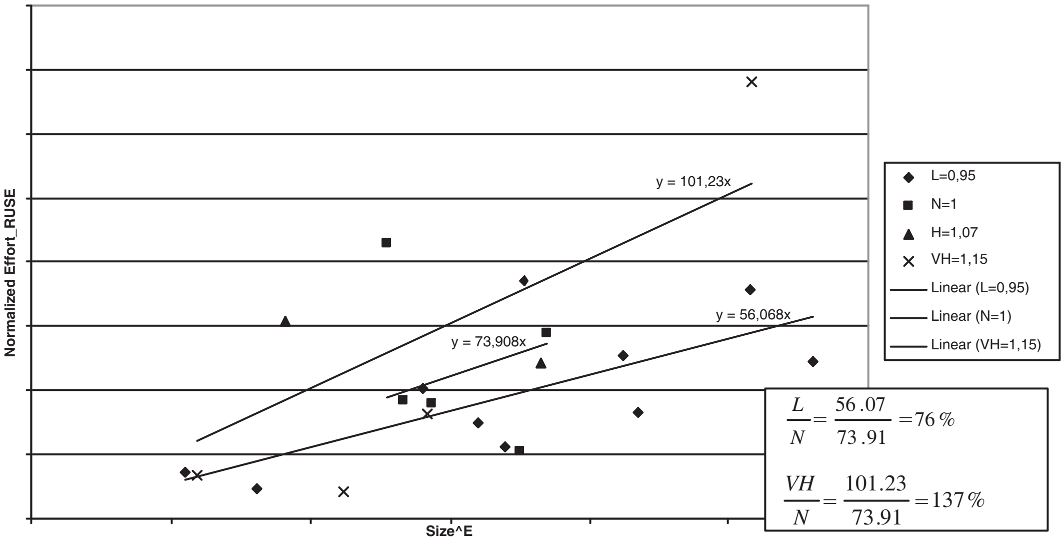


Figure 16: RUSE report

line can be plotted, the *nominal* regression line (all fazes are documented). As there is only one regression line, no interpretation can be given with respect to the influence of this effort multiplier. Nevertheless, we can remark that there might be an explanation why almost all projects have a *nominal* rating. The ratings of the effort multipliers are determined by the project leaders. It is a responsibility of the project leader that all fazes are well documented, hence, it is not surprising that most projects get a *nominal* rating from the project leader.

8.3.8 Application Experience, APEX

Figure 18 shows the report for effort multiplier Application Experience (APEX). Two regression lines are plotted: a *nominal* regression line and a *high* regression line. Comparing the slope of the *high* regression line with the slope of the *nominal* regression line, we see that this effort multiplier has almost the same effect as prescribed by the Cocomo II-model. A project where the average team experience with the applications is 3 years, will need 11% less effort than a project with an average team experience of 1 year with this kind of applications. Cocomo II predicts an effort reduction of 12%.

Recently, there are a lot of new employees in the company and also about 600 external members. Additionally, new knowledge domains are created (e.g. leasing, credit risks). No further actions are planned to increase the average team experience with respect to the application type. At this mo-

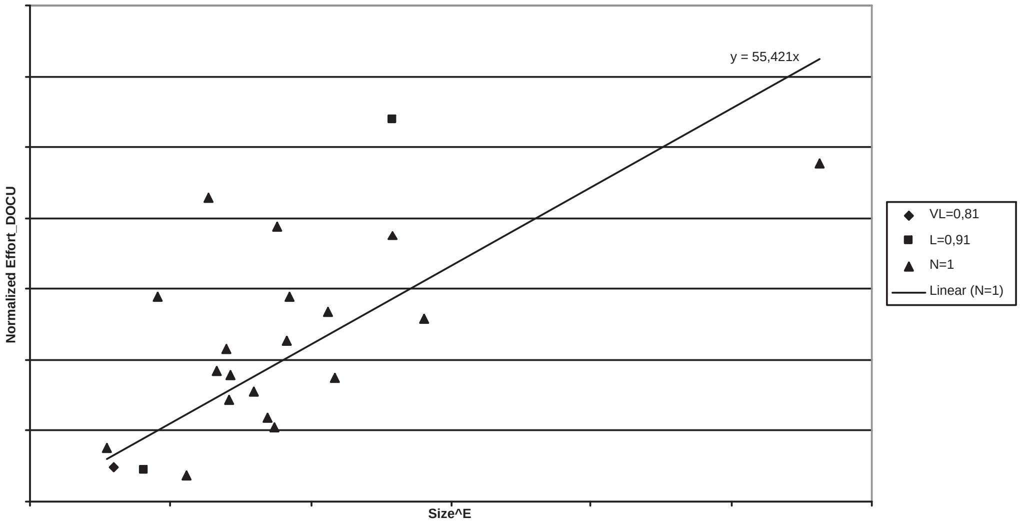
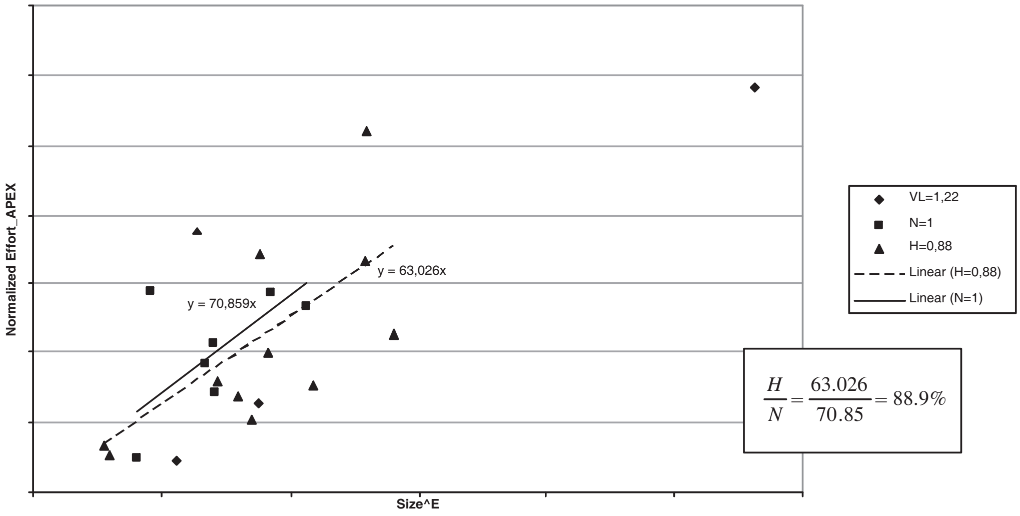


Figure 17: DOCU report



$$\frac{H}{N} = \frac{63.026}{70.85} = 88.9\%$$

Figure 18: APEX report

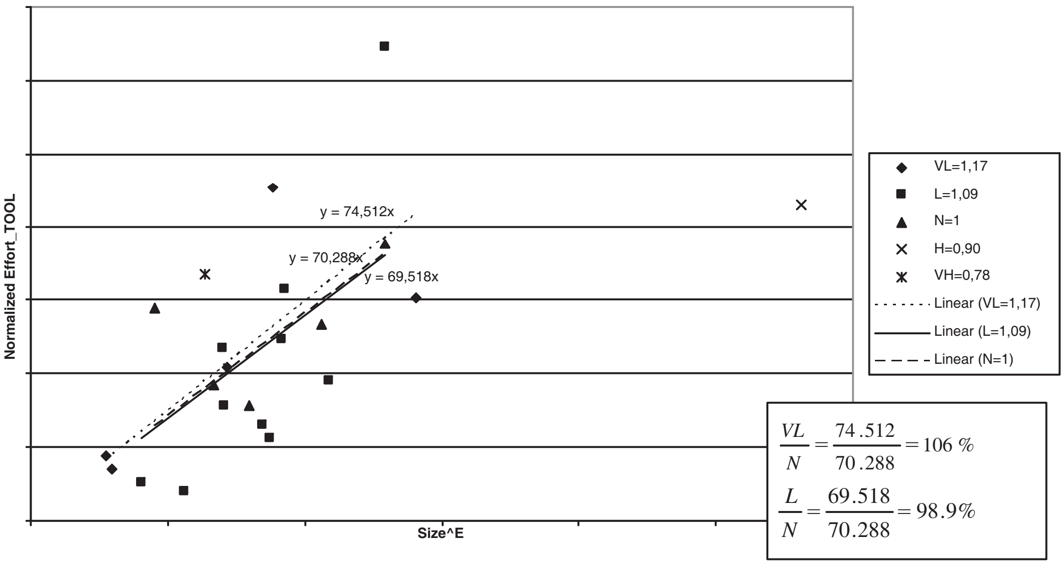


Figure 19: TOOL report

ment, in the company they strive for an experience of at least 4 or 5 years in a domain.

8.3.9 Use of Software Tools, TOOL

Figure 19 shows the report for effort multiplier Use of Software Tools (TOOL). Three regression lines are plotted: *very low*, *low* and *nominal*. The slopes of the *low* and *nominal* regression line are almost equal, while COCOMO II predicts an increase of 9% in effort for the *low* rated project. Also the *very low* rated project needs less extra effort (+6%) compared to COCOMO II (+17%).

From this we can conclude that the use of software tools has no strong influence on the projects at KBC. This seems plausible, a significant productivity gain can only be reached from strongly integrated tools which are not often used at KBC.

8.4 Assumption violation?

In the previous reports, we produce the report for each effort multiplier at the same time. As a result, we assume implicitly that the influence of the other effort multipliers is estimated correctly by the COCOMO II-model. As can be seen in the reports in Section 8.3, the retrieved values for the effort multipliers within this company do not always correspond with the COCOMO II-values.

	RESL	TEAM	STOR	LTEX
B*Size	0.76731			
PREC	0.76761	0.79899		
TIME			0.65921	
APEX				0.71017

Table 9: Highly correlated parameters

Consequently, that assumption is violated. How big is the influence of this violation? To investigate the influence, we first calibrate all parameters in the Cocomo II-model with the experience data using a multiple regression technique.

$$\text{Effort} = \text{A} \cdot \text{Size}^{\text{B}+0.01\cdot\sum_{j=1}^{5}\text{SF}_j} \cdot \prod_{i=1}^{n} \text{EM}_i$$

By taking logarithms on both sides of Cocomo II-formula, we linearize the equation. Using our data set of 22 projects for a multiple linear regression, we can estimate the β and γ - coefficients in following formula:

$$\log(\text{Effort}) = \beta_0 + \beta_1 \cdot \log(\text{Size}) + \beta_2 \cdot 0.01 \cdot \text{SF}_1 \cdot \log(\text{Size}) +$$

$$\ldots + \beta_m \cdot 0.01 \cdot \text{SF}_m \cdot \log(\text{Size}) + \gamma_1 \cdot \log(\text{EM}_1) + \ldots + \gamma_n \cdot \log(\text{EM}_n)$$

As explained before, some of the parameters are given a fixed value for all the projects within the company (see table 5). These parameters/values will be absorbed in the estimation of the constant parameters: β_0 and β_1. Before performing a regression, the correlation between the different parameters is calculated, see table 9. For the highly correlated parameters (threshold value of 0.65 as in [17]), we formed new aggregated parameters: the scale factors PREC (precedentedness) and TEAM (team cohesion) are aggregated into PREC-TEAM computed as $0.01 \cdot (\text{PREC}+\text{TEAM}) \cdot \log(\text{Size})$; the scale factor RESL (risk resolution) is aggregated with the constant exponential parameter into RESL-SIZE computed as $0.91 + 0.01 \cdot (\text{PMAT} + \text{RESL}) \cdot \log(\text{Size})$; the effort multipliers STOR (storage constraints) and TIME (execution time constraints) are aggregated into the parameter TIME*STOR calculated as $\log(\text{STOR}\cdot\text{TIME})$; the effort multipliers LTEX (language and tool experience) and APEX (application experience) are aggregated into the parameter APEX*LTEX calculated as $\log(\text{LTEX} \cdot \text{APEX})$.

The results for the multiple linear regression for the independent variable log(Effort) are shown in table 10. As can be seen, only the intercept, RESL-SIZE and TIME*STOR are significant parameters in our model. As few

Coefficient	Estimate	error	t-value	$Pr > \|t\|$
Intercept	3.24072	0.33506	9.67	<0.0001
RESL-SIZE	0.38825	0.13691	2.84	0.0195
PREC-TEAM	0.72050	3.62193	0.20	0.8467
FLEX	0.07311	5.50276	0.01	0.9897
ln(RELY)	1.57696	1.93337	0.82	0.4358
ln(DATA)	2.02789	1.90953	1.06	0.3159
ln(CPLX)	1.22632	2.11497	0.58	0.5763
ln(RUSE)	1.55788	2.14439	0.73	0.4860
ln(DOCU)	2.76709	2.85579	0.97	0.3579
ln(TIME*STOR)	3.19435	1.18152	2.70	0.0243
ln(APEX*LTEX)	-1.51476	1.67537	-0.90	0.3895
ln(PLEX)	2.26019	3.47506	0.65	0.5317
ln(TOOL)	-1.47937	1.47876	-1.00	0.3432

Table 10: Coefficients Estimates for Multiple Linear Regression

Coefficient	Estimate	Error	F- value	$Pr > F$
Intercept	3.12575	0.20844	224.88	< 0.0001
RESL-SIZE	0.34353	0.07723	19.79	0.0003
ln(DATA)	2.37148	0.70165	11.42	0.0033
ln(TIME*STOR)	2.63722	0.73568	12.85	0.0021

Table 11: Coefficients Estimates for Stepwise Regression

DATA	low L	nominal N	high H	very high VH
a priori value	0.9	1	1.14	1.28
data determined value	0.778	1	1.364	1.795

Table 12: Ratings for effort multiplier DATA

TOOL	very low VL	low L	nominal N	high H	very high VH
a priori value	1.17	1.09	1	0.9	0.78
data determined value	1	1	1	1	1

Table 13: Ratings for effort multiplier TOOL

parameters are significant, we performed a stepwise regression instead. The results are shown in Table 11.

With the estimates of the β and γ -coefficients in the stepwise linear regression (see table 11 or zero in the case of a non-significant parameter), we can calculate new values for the parameters in the Cocomo II-model. The new values for a scale factor are obtained by multiplying the original value with the coefficient: $\beta_j \cdot \mathrm{SF}_j$. The new values for an effort multiplier are obtained by taking the exponent of the coefficient multiplied with the logarithm of the value of the effort multiplier: $\exp(\gamma_i \cdot \log(\mathrm{EM}_i))$. As an example, Table 12 and Table 13 show the old and new values of effort multipliers DATA and TOOL respectively.

A database of only 22 projects is obviously not enough to estimate 13 parameters. The amount of data points should be large relative to the number of model parameters. Consequently, in our case, each data point has a substantial influence on the calculation of the parameters. This means that one point can distort the estimations. Therefore, we will not use the estimated values but a weighted average with the a priori values set by Cocomo II to determine the new values of the cost drivers and constant factors A and B. This technique is also used in [17], [16]. In analogy with [17] the weights we use are 10% estimated value and 90% a priori value. Due to the strong economy of scale we discovered within this company, we will not use the a priori value set by the Cocomo II-model, but the value we obtained by the first calibration of the constant parameters A and B, i.e. $A = 15.16$ and $B = 0.33$.

With these new parameters, we produced once more the reports for the influence of the effort multipliers. The results of the adjusted slopes of the regression lines can be found in Table 14. Compared with the conclusions

EM	slope	adjusted report	initial report	Cocomo II
RELY	L/N	1.159	1.104	0.92
	VL/N	1.14	1.183	0.82
DATA	L/N	0.684	0.628	0.90
	H/N	1.286	1.229	1.14
CPLX	L/N	1.34	1.667	0.87
RUSE	L/N	1.026	0.76	0.95
	VH/N	1.795	1.37	1.15
TIME*STOR	H-N/N-N	1.441	1.729	1.11
	H-H/N-N	1.657	2.89	1.1655
APEX	H/N	0.902	0.889	0.88
PLEX	H - VH	+6%	∼=	+6%
LTEX	H/N	0.89	0.857	0.91
TOOL	VL/N	1.087	1.06	1.17
	L/N	0.995	0.989	1.09

Table 14: Comparison of the slopes

based on the first reports (where the assumption was violated), we see:

- RELY:The regression lines are still in a reverse order compared to the *nominal* line. However, the slopes of the *low* and *very low* regression lines are almost equal. This fits into our hypothesis that this effort multiplier has no influence in the company.

- DATA:The effect we measured in our initial report is still present and even a little stronger for the *high* rated projects.

- CPLX: We still can't give a reasonable explanation for the reverse order of the regression lines, although the effect is weakened a little compared to the initial report.

- RUSE: From the initial report, we concluded that it takes a lot of effort to make an application multi-company. However, from the new slopes, we see *low* projects are comparable to *nominal* rated projects. The effect for *high* rated projects on the other hand is even larger than measured in the initial report. Hence, it is still necessary to pay attention to the identified actions.

- TIME and STOR: As expected, the STOR and TIME effort multipliers influenced each others, hence we measured in our initial report an accumulated effect. By combining these effort multipliers into one effort multiplier we obtained more realistic results. Nevertheless, the effects are still significantly larger than predicted by COCOMO II. A plausible explanation for the strong correlation between these two parameters is the specific environment of this company. Probably the execution time constraint will only be present due to huge amount of data that needs to be processed rather than time consuming calculations. Hence a high rating for the execution time constraint will induce also a high rating for the storage constraint. This could explain why these two cost drivers are highly correlated within this company and need to be combined into one cost driver in contrast with the COCOMO II-model.

- APEX: The effect we measure is comparable to the effect in the initial report.
- PLEX: The effect we measured in the initial report, namely, that once there is more than 3 years of experience with the platform and languages, this cost driver has no extra influence, does not hold anymore. We now measure a comparable effect as the COCOMO II-model prescribes. However, no actions were identified with respect to this effort multiplier.
- LTEX: The effects we measured in the initial report are a little weakened.
- TOOL: The same conclusions as with the initial report can be taken.

Overall, we can state that the conclusions we made with the first reports still hold. Hence, although the assumption of a correct estimation of the parameters not under investigation is violated, the results and conclusions we draw from the reports still hold. However, as we only take into account 10% of the new calibration, this weighted average can still have an influence on our assumption. In order to have a more confident calibration, we need more data. However, the company already has his ROI and are not interested in putting more effort just to refine the calibration. Additionally, waiting for more data from new measurement, one can object that all kind of differences due to the time between the two measurements can also have an influence. In other words, waiting for a new measurement, provides us with two incomparable data sets that should not be mixed up into one data set.

8.5 Measure Improvement

Now we have identified possible actions, we need to find a way to measure the results of these actions. In other words, how can we capture the productivity improvement with this report? An improvement in productivity can result in two effects.

Firstly, an action leads to a shift in the rating of an effort multiplier. This improvement will not be visible in the report as the estimated effort by the Cocomo II-model will incorporate this improvement by giving another rate to the effort multiplier. However, a simple report with the frequency of each rating of the effort multiplier enables us to detect this shift in rating.

Secondly, an improvement might lead to an effort reduction rather than a shift in the rating. This change will be visible in a change in the slope of the regression lines. However, when a company performs several actions on different effort multipliers, it is not clear to which action the measured improvement can be attributed. Indeed, suppose a company identified two actions A_1 and A_2 on two different effort multipliers EM_1 and EM_2 that lead to a global effort reduction of 10%. How much of this reduction can be assigned to action A_1 and how much to action A_2? The reduction in the actual effort will lead to a reduction in the Normalized Effort for both effort multipliers EM_1 and EM_2. As a result, for both effort multipliers, there will be a change in the slopes. How much of the change we measure in the slope of EM_1 is actually an effect of action A_1 and how much is the result of the action on the other effort multiplier? In summary, the question remains how we can deduce to which action we can attribute the effect in productivity we measure. To solve this question we would need projects for which only action A_1 was applied and a set of projects for which only A_2 was applied. As management is aiming at maximal productivity improvement, the company is not willing to invest in such experimental set up and all projects will be allowed to benefit from all software process improvement actions.

9 Useful Reports for the Future

The objective of the reports is not an analysis of the projects on an individual basis, but rather an instrument for management to evaluate the productivity of the overall software development departments. The main goal of the measurement system is to provide management with a trend report. Management also would like an insight in the opportunities where and how the productivity can be improved. And finally, measure the improvements.

9.1 Workload in relation to Lines of Code

In this graph the actual workload and the workload according to the Cocomo II-model are plotted against the lines of code. The graph also displays a line denoting the *company-standard*. This is the workload predicted by Cocomo II when all the effort multipliers and scale factors are set to standard value for the company (i.e. initially the *nominal* rating, except for the parameters with a fixed value, later the standard value). Filtering should be possible with respect to the development environment (mainframe or client based), the domain where the project has been developed (banking, insurance,), the type of the program (an investment or continuity file) and the organizational unit (team responsible for the project, the development service and development management).

An example of this report can be seen in Figure 6. This report has been drawn with the 22 projects of the first measurement results. At the moment, not much interpretations can be given to this report. Hopefully, with more measurements, we can identify trends or a kind of projects that are less/more productive than others.

9.2 Influence of an Effort Multiplier

As seen in section 8, this report provides a lot of information to detect possible areas for productivity improvement in the company. Also in the future, this report can be used to detect whether improvement actions resulted in a positive way or whether the (negative) effect of a specific effort multiplier is still present.

9.3 Frequency of Cost Drivers

A frequency table of each effort multiplier and scale factor give an indication about the most frequent rating for each cost driver. This can give, in the first place, an indication of the company-standard rating. Secondly, this report can also serve as a detection of possible outliers. And thirdly, as explained in section 8.5, this report can help to detect a shift in the rating of a specific effort multiplier due to an productivity improvement action. Figure 20 shows the frequency table for the effort multiplier DATA.

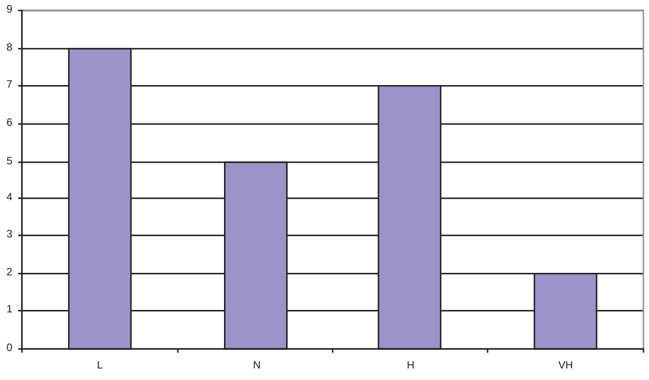


Figure 20: Frequency table of Effort Multiplier DATA

9.4 The productivity as a function of the number of teams that registered workload

As mentioned during the set-up phase of this measurement project, in this company, projects are organized around one project team and possibly several interface teams. By plotting the number of teams participating in the project against the productivity rate (Cocomo II-calculated effort/actual effort), management wants to capture the influence on productivity of working with multiple teams in stead of one team. This means, they want to see whether working with multiple teams reduces the productivity significantly or not. An example with the dataset of 22 projects can be seen in Figure 21. We can make two observations in this report. Firstly, most projects work with a large amount of interface teams. We notice that most of the projects work with 5 to 15 interface teams. Secondly, we clearly see a decrease line in the productivity for increasing number of interface teams.

9.5 The productivity as a function of the period

Besides plotting the effort against the lines of code, one can also plot the productivity rating (Cocomo II-calculated effort/actual effort) in time. This trend report should help management to see whether the productivity improves. The same filters are possible as in the first report: the development environment (mainframe or client platform), the domain where the project has been developed (banking, insurance,), the type of the program (an investment or continuity file) and the organizational unit (team responsible for the project, the development service and development management).

A remark for all these reports is that we need a good baseline to calculate

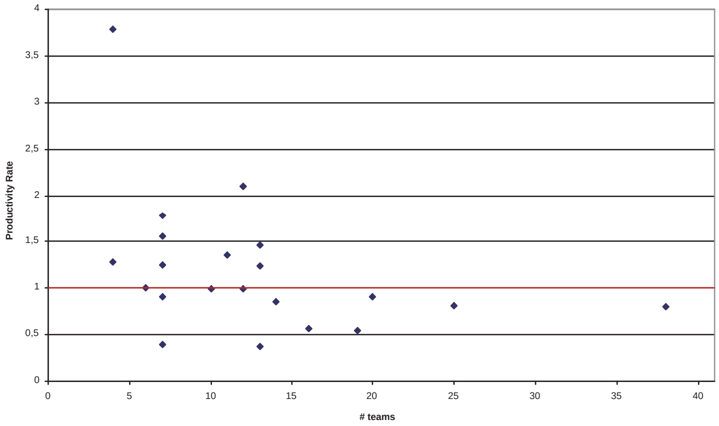


Figure 21: Productivity versus number of teams

the relative productivity (i.e. Cocomo II-calculated effort/actual effort). Hence, we first need a good calibration where we can start from. Once we recalibrate our formula and parameters, we loose the initial benchmark and we can no longer compare the productivity of our projects with the former projects.

10 Conclusions

The main innovative contribution of this research paper is the use of an estimation model as productivity measurements. Rather than comparing output with input and judging about productivity on the basis of historical data, we define productivity as a comparison of the estimated time with the actual time. This approach creates benchmark possibilities with the estimation model as reference and norm. However, as experienced in our case study, the best reference frame is a reference frame adapted to the own environment. Nevertheless, collecting own data takes time.

The relative productivity only gives one view on productivity. Even more important is finding ways and opportunities to improve productivity. Cocomo II is useful for this as it contains several cost drivers. With the report introduced in this paper, maximum information can be retrieved from our measurement data with respect to these cost drivers. The report identifies the actual influence of each cost driver on the company's projects.

As illustrated in the case study, even with a limited amount of project data, much knowledge can be gained from comparing the slopes of the regression lines in the graph between $Size^E$ and the Normalized Effort. Such comparison gives an indication of the influence of the effort multiplier. These reports do not identify the actions you can undertake to improve the productivity. Yet they are an instrument to identify places where problems occur and improvement is possible. When results of the report correspond with the feeling and experience of experts, there is a clear indication of productivity improvement opportunities. The report can be used to build a business case and to visualize to management that it is important to invest in software process improvement to reduce (negative) effects of particular effort multipliers.

The results obtained with these reports, namely the new values for the effort multipliers, can be used to calibrate the effort multipliers in the Cocomo II-model. In case the model is used as an estimation tool, using the obtained values for calibrating the model improves the prediction accuracy, even with limited data. In our case, the model is mainly used as a productivity measurement instrument. We want to measure productivity, identify areas of improvement and finally measure improvement. By calibrating the model, the initial benchmark is lost and new measurement results are no longer comparable with formerly obtained measurements. New calibrations make it impossible to measure improvements. Therefore, we did not yet use the obtained results to calibrate the effort multipliers.

Nevertheless, there are still some comments about the report. At this time, we produce the report for each effort multiplier at the same time. As a result, we assume implicitly that the influence of the other effort multipliers is estimated correctly by the Cocomo II-model. Although the calibration of the model (section 8.4), we notice that these factors are not always correctly estimated in relation with this company and hence we don't take into account the new knowledge we receive from the reports. One solution would be to collect more project data to fine tune our report and calibrated factors. As the company already reached its ROI, they are not motivated to collect more data just for fine tuning. Another approach is to use a 'step-wise regression' technique to incorporate the extra knowledge we receive from the reports. This means, not to produce all the reports at the same time, but starting with the report with the highest variation from the original values and then use these new values in the next report for the other effort multipliers.

References

[1] Cosmic FFP. `http://www.cosmicon.com`.

[2] History of functional size measurement. `http://www.cosmicon.com/historycs.asp`(visited 24/09/2007).

[3] IFPUG. `http://www.ifpug.org`.

[4] ISBSG. `http://www.isbsg.org`.

[5] ISO/IEC 19761:2003(E) Software engineering COSMIC-FFP A functional size measurement method. 2003.

[6] ISO/IEC 20926:2003(E) Software engineering IFPUG 4.1 Unadjusted functional size measurement method Counting practices manual. 2003.

[7] ISO/IEC 14143-1:2007 Information Technology - Software Measurement - Functional Size Measurement - Part 1:Definition of Concepts. 2007.

[8] S. Abrahão, G. Poels, and O. Pastor. Comparative Evaluation of Functional Size Measurement Methods: An Experimental Analysis. Technical report, Working paper 2004/234 at Faculty of Economics and Business Administration-Ghent University, Belgium, February 2004.

[9] S. Abrahão, G. Poels, and O. Pastor. A functional size measurement method for object-oriented conceptual schemas: design and evaluation issues. *Software and Systems Modeling*, 5(1):48–71, 2006.

[10] A. Abran and P.N. Robillard. Function points: a study of their measurement processes and scale transformations. *Journal of Systems and Software*, 25(2):171–184, 1994.

[11] A. Abran and PN Robillard. Function points analysis: an empirical study of its measurementprocesses. *Software Engineering, IEEE Transactions on*, 22(12):895–910, 1996.

[12] A.J. Albrecht and J.E. Gaffney Jr. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–647, 1983.

[13] J. Baik, B. Boehm, and B.M. Steece. Disaggregating and Calibrating the CASE Tool Variable in COCOMO II. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 1009–1022, 2002.

[14] B.W. Boehm. *Software Engineering Economics.* Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[15] B.W. Boehm, B. Steece, and R. Madachy. *Software Cost Estimation with Cocomo II with Cdrom.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2000.

[16] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *Software Engineering, IEEE Transactions on*, 25(4):573–583, 1999.

[17] B. Clark, S. Devnani-Chulani, and B. Boehm. Calibrating the COCOMO II post-architecture model. *Proceedings of the 20th international conference on Software engineering*, pages 477–480, 1998.

[18] N. Fenton and S.L. Pfleeger. *Software metrics: a rigorous and practical approach.* PWS Publishing Co. Boston, MA, USA, 1997.

[19] R. Grable, J. Jernigan, C. Pogue, D. Divis, U.S.A.M. Command, and R. Arsenal. Metrics for small projects: Experiences at the SED. *Software, IEEE*, 16(2):21–29, 1999.

[20] T. Harbich and K. Alisch. Accuracy of Estimation Models with Discrete Parameter Values shown on the Example of Cocomo II. *Proceedings of Software Measurement European Forum (SMEF) 2007*, pages 313–324, 2007.

[21] DR Jeffery and G. Low. Calibrating estimation tools for software development. *Software Engineering Journal*, 5(4):215–221, 1990.

[22] C. Jones. *Programming productivity.* McGraw-Hill, Inc. New York, NY, USA, 1985.

[23] C.F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987.

[24] B. Kitchenham. Making Process Predictions. *Software Metrics: A Rigorous Approach, Chapman & Hall*, 1991.

[25] B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[26] B.A. Kitchenham. The question of scale economies in softwarewhy cannot researchers agree? *Information and Software Technology*, 44(1):13–24, 2002.

[27] T. Kralj, I. Rozman, M. Heričko, and A. Živkovič. Improved standard FPA methodresolving problems with upper boundaries in the rating complexity process. *The Journal of Systems & Software*, 77(2):81–90, 2005.

[28] D. Longstreet. Function Point Training and Analysis Manual. *Longstreet Consulting*, 2001.

[29] R. Marwane and A. Mili. Building tailor-made software cost model: intermediate TUCOMO. *Information and Software Technology*, 33(3):232–238, 1991.

[30] T. Menzies, Z. Chen, J. Hihn, and K. Lum. Selecting Best Practices for Effort Estimation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 883–895, 2006.

[31] R.E. Park et al. *Software Size Measurement: A Framework for Counting Source Statements*. Carnegie Mellon University, Software Engineering Institute, 1992.

[32] G. Poels. Functional Size Measurement of Multi-Layer Object-Oriented Conceptual Models. *Proc. 9th International Object-Oriented Information Systems Conference, Geneva, Switzerland*, pages 334–345, 2003.

[33] A. L. Rollo. Functional size measurement and cocomo - a synergistic approach. In *Proceedings of Software Measurement European Forum (SMEF) 2006*, 2006.

[34] Luca Santillo and Harold van Heeringen. Proposals for increasing benchmarking data quality of projects measured in cosmic. In *Proceedings of Software Measurement European Forum (SMEF) 2008*, 2008.

[35] CR Symons. Function point analysis: difficulties and improvements. *Software Engineering, IEEE Transactions on*, 14(1):2–11, 1988.

[36] Kristien Van den Branden and Jeroen Vanuytsel. Toepassingen van Computers in Management: Functiepuntanalyse bij KBC. Technical report, K.U.Leuven, 2004-2005.

[37] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. Mining software repositories for comprehensible software fault prediction models. *J. Syst. Softw.*, 81(5):823–839, 2008.

[38] S. Yenduri, S. Munagala, and L.A. Perkins. Estimation Practices Efficiencies: A Case Study. *Information Technology,(ICIT 2007). 10th International Conference on*, pages 185–189, 2007.