Published in the Proceedings of Smef 2006

COCOMO II applied in a banking and insurance environment: experience report.

Lotte De Rore, Monique Snoeck, Guido Dedene

Abstract

This paper presents an experience report about the set-up of a measurement environment using COCOMO II for software development projects in a company in the banking and insurance area. This set-up is part of a larger research project on managing efficiency aspects of software factory systems. As in a practical environment it is not always possible to faithfully follow the theory, this article describes the decisions we made and our experiences during the whole set-up process.

1. Introduction

With their program Expo 2005, the ICT-department of a bank and insurance company had not only 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.

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. Hereby it was not the intention to analyse projects on an individual basis, but rather to look for trends in the whole ICT-development area. Productivity is measured as size versus effort, effort being measured as working hours spent to the development of software. Measuring size however is quite more complex. Historically, lines of code (LOC) have been the first measurement for software size. But because of the many paradoxes that it yields [1], more user oriented size measurements have been developed, most of which are based on the notion of delivered functionality. For this project, two function point based measurement methods were considered: IFPUG [2] and COSMIC-FFP [3]. In addition we also considered the use of COCOMO II [4]. 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 that 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 FPA 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 [5] 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 [6]. As a result of this experiment, we concluded in this company there are no artefacts 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 [3]. It has the advantage of offering a more simple way of counting units of functionality that could be easier to automate than the IFPUG counting rules. It was therefore still a candidate to consider. On the other hand, there is however not a lot of historical data available allowing the transformation of COSMIC-FFP into effort estimations 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 Capers Jones [1] 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. Indeed, the IFPUG-method is still considered to be mathematically flawed, because it classifies user functions on an ordinal scale (simple, medium, complex) and then subsequently uses operations that are (theoretically) not allowed on an ordinal scale [7]. The COCOMO II-model on the other hand has been developed using statistical models and is therefore considered mathematically more correct. For COCOMO II, the project size is seen from the point of view of the implementation, this in contrast 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.

3. 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. The effort is expressed as person-months (PM) and can be retrieved with the following formula:

$$PM = A \text{ x } Size^{E} \text{ x } \prod_{i=1}^{n} EM_i$$

$$\text{where } E = B + 0.01 \text{ x } \sum_{j=1}^{5} SF_j$$

A and B are constant factors, namely A =2.94 and B = 0.91 for COCOMO II, while EM represents the effort multipliers and SF the scale factors. The size of a project is expressed in KSLOC.

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 -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.

3.1. 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 need therefore 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 spending to WP for the different projects goes from 8% to 14% of the total workload (in comparison with COCOMO II, where the plans and requirements phase amount between 2% -15% with an average of 7%). 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 in the company they work with ‘interface teams’. 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 intern 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 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.

3.2. 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 a later section in the paper. Here we discuss the correct count in order to match with the project's time registration. The PDS records the effort spent to 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 accept. 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, cannot be restored at a later point in time.

In addition to establishing the correct moment of counting, there should also be an inventory of all the modules that were created or modified during the project. In case of update of existing code, we also need before and after situation. Finally, the 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 section further in the paper.

3.3. 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. 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 need 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.

4. Set up of the measurement environment

4.1. Scale factors and Effort multipliers

The different descriptions of the values for the scale factors and effort multipliers were transformed into multiple-choice questions. The possible answers were adjusted to the terminology used within the company. Some answers were left out, because they are not possible for the kind of projects that appear in the company. Other questions get a fixed answer for all the projects. Because the company works with different platforms (mainframe based development for the front-end of the headquarters and business logic and open systems based development for the front-end of the distribution channels) a slightly different questionnaire was produced for the different platforms. Some of the project characteristics are subdivided in multiple criteria in order to determine the appropriate value. Each criterion forms a question, but also a summary question is included and serves as a control question.

The questionnaire is to be sent to the project leader via e-mail after the completion of the project. For each question a normative answer has been determined. Projects that differ too strongly from this norm are evaluated for correctness.

4.2. Lines of code

As far as possible the guidelines to count the lines of code described in the book of COCOMO II [4] are followed, but with the condition that the count should happen automatically. In the company, they work on different platforms (mainframe based development for the front-end of the headquarters and business logic and open systems based development for the front-end of the distribution channels) and on these platforms even with different 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 open systems the tool Clearcase. As explained before, one of the major concerns was to rule out the paradoxes inherent to using LOC as size measurement. Capers Jones [1] 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.

4.2.1. Resolving paradoxes resulting from to 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 [1] 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 centre of the company, several modules have been written in the different program languages in order to compare them with each other. That way, correct conversion factors could be derived that are specifically applicable for projects written by programmers in this company. For COBOL environments (mainframe), the conversion factors are:

1 line COBOL = 2.1 lines APS = 9.9 lines VA/G

For the JAVA environment (open systems), 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 en SygelJSP is about 50% more voluminous than handmade JSP. Reduction factors have to be applied to the generated code.

SYGELJAVA/1.1 = JAVA
SYGELJSP/1.5 = JSP

The counts on mainframe and the counts on open systems 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.

4.2.2. 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 open systems platform.

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 open systems projects however this is more difficult. There are only two releases per year for open systems software, 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.

4.2.3. 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, then more modified lines of code will be counted when the lines of code are counted at each delivery than when 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.

4.2.4. Resolving paradoxes resulting from different programming styles

The problem with using lines of code as a basis for our measurements is the paradoxes. Some of the paradoxes have been countered by using conversion factors and reduction factors for generated code as explained before.

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 qualitative 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. Having senior programmers review the code of junior programmers under their supervision also stimulates uniformity of programming style.

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.

4.3. 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 lines of code delivered. Modules that are reused cannot 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. In [4] 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:

$$\text{Equivalent KSLOC} = \text{Reused KSLOC} \times (\text{AA} + 0.3 \times \text{IM})/100$$

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 have 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 open systems 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% instead 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.

4.4. Reports

Just like the count of the lines of code, reports should be generated automatically. The objective of the reports is not an analysis of the projects on an individual basis, but rather a help for the 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. Several reports can be drawn from the data collected by the measurement system:

The actual workload in relation to the lines of code. In this graph the actual workload and the workload predicted by 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 the effort multipliers and scale factors are set to nominal, except for the parameters that have been given a constant value for the whole company. Once a company-wide normative value has been determined for each effort multiplier and scale factor, these values can be used for the 'company-standard'. Filtering is possible with respect to the development environment (mainframe or open systems), the domain where the project has been developed (banking, insurance, …), the type of the program (an investment or continuity file) and the organisational unit (team responsible for the project, the development service and development management).

The productivity as a function of a project characteristic. For each scale factor and each effort multiplier a separate graph can be derived. Here the productivity (workload according to COCOMO II / actual workload) is plotted against the value of the effort multiplier or scale factor. All the projects above 'productivity = 1'-line are more productive than COCOMO II predicts and all the projects beneath the 'productivity = 1'-line are less productive. This report shows the influence of the specific project characteristic and whether it is interesting to work on this specific characteristic in order to improve the productivity. Here also the same filters can be applied as in the previous type of report.

The productivity in function of the workload spent on a development phase with respect to the total workload. This graph plots the percentage of the workload spend on a particular development phase with respect to the total workload against the achieved productivity. With this report one seeks for the optimum percentage to spend on each particular development phase and for the influence on productivity when too much time is spent on some phase, e.g. the functional design or the technical design.

The productivity as a function of the number of teams that registered workload. As mentioned above, projects are organized around one project team and possibly several interface teams. With this report, management wants to capture the influence on productivity of working with multiple teams on one project.

The productivity as a function of the period. This is a trend report to help the management to see whether the productivity improves. The same filters are possible as in the first report: the development environment (mainframe or open systems), the domain where the project has been developed (banking, insurance, …), the type of the program (an investment or continuity file) and the organisational unit (team responsible for the project, the development service and development management).

Reuse. Reuse is an important issue for the company and since the danger exists that reuse will be penalized in stead of stimulated by the way the count happens, a report is drawn where the amount of reuse is mapped in order to make it able for management to follow up the amount of reuse within the projects.

5. Calibration of the model

The COCOMO II-model has been created by using statistical models. By including the data retrieved from the own projects, the model can be calibrated 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 [4] to 1 instead of 2.94.

For further calibration it is too soon, more experience data is needed. Because the company works with different measurements of the lines of code for mainframe and open systems 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 open systems projects on the other hand.

6. Points of excellence and limits about COCOMO II

In this section we elaborate some good points about the use of COCOMO II, but also some minor points and limits.

In setting up the measurement environment, we found that the extensive list of project characteristics (the scale factors and effort multipliers) that are used in the COCOMO II-model are on itself already very useful. Even without using the answers for calculating an effort estimation, the questionnaire can perfectly be used to detect outliers. As such reasons can be found why a project is more or less productive than the others. The questions can also be used as points of special interest before the project starts. They can be used to answer questions like: how can you reduce the duration of the project? Which values of a certain effort multiplier result in an increase or decrease of the duration of the project? How can we improve our scale factors and effort multipliers?

The COCOMO II-model is a quick and easy way to measure the productivity. Once we had figured out how to count lines of code, it was just a fill-in exercise to compute the expected duration of the project. You even can automate the measurements and this was the deciding factor in this case. Once the measurement environment is set up, only a little work has to be done to collect data concerning the productivity of projects. The experience database can be completed quickly and analysis can be performed on these experience data.

Right from the start the values of the scale factors and effort multipliers deliver a good point of comparison with other companies/projects because they are stipulated using a lot of experience data from several companies. But since we collected experience data from our own projects, a better predictive model could be obtained for the company by using the historical data for a calibration of the COCOMO II-formula. That way, we get a better look on the productivity of the projects within the company. On the other hand, a model calibrated to the own company makes the comparison with projects outside the company less straightforward.

Because we wanted to count the lines of count in an automated way, it was not possible to follow all the guidelines to count the lines of code completely. This has as a consequence that the comparison with other projects by the use of the COCOMO II-formula, is not totally correct. By deviating from the guidelines, we loose the benchmark possibility. But as long as the count within the company in a consistent way, you still can compare projects within the company.

Finally, the most important challenge was to deal with the paradoxes associated with lines of code. By deriving company-specific conversion factors, by keeping mainframe and open systems platforms separate and by the way we deal with reuse, we believe we managed to resolve this issue. It will however remain an important point of attention in the future use of the measurement environment.

As a result we can conclude that although it seems that COCOMO II offers a great opportunity to benchmark your projects with other projects outside your company, in practice, this is difficult to realise because of the large variations in counting lines of code, the use of different languages and coding practices across companies and the measurement of effort (e.g. how many working hours in a person-month, how many months is a year?). In summary, whereas resolving the LOC-paradoxes in the context of one company appeared to be largely feasible if one sets the scope to portfolios of applications rather than to individual projects, resolving these paradoxes across companies is largely an illusion.

7. Acknowledgements

This paper has been written as part of the ‘KBC-research chair’-project on ‘Managing efficiency aspects of software factory systems’ sponsored by KBC bank and insurance.

8. References

[1] Capers Jones, “Programming productivity”, McGraw-Hill, New York, 1986
[2] http://www.ifpug.com/
[3] http://www.cosmicon.com/
[4] Barry W. Boehm, “Software Cost Estimation with COCOMO II”, Prentice Hall, New Jersey, 2000
[5] http://www.isbsg.org/
[6] K. Van den Branden, J. Vanuytsel, “Toepassingen van computers in management: Functiepuntanalyse bij KBC”, Leuven, 2004-2005
[7] N. Fenton, Shari, L. Pfleeger, “Software metrics, a rigorous and practical approach”, 2nd edition, PWS Publishing, Boston, 1997