Successes and challenges experienced in implementing a measurement program in small software organizations

Sylvie Trudel, Pascale Tardif

Centre de Recherche Informatique de Montréal (CRIM), Montréal, Canada

Sylvie.Trudel@crim.ca, Pascale.Tardif@crim.ca

Abstract:

In recent years, the authors have implemented measurement programs in several organizations of different sizes. Two of them were small software companies of approximately 12 employees, which were mostly developers. Although these two organizations were similar in size and technology, the differences in the issues they were facing led to completely different approaches for their measurement program. This paper is about the steps taken to implement these measurement programs, both including functional size measurement with COSMIC, effort, schedule, and defects. It also describes what was done to ensure the success of each program and, most importantly, the challenges that were faced during their implementation and maintenance, as well as some of the solutions proposed to answer these challenges.

Keywords

Measurement program, metrics program, small organizations, functional size measurement, COSMIC

1 Introduction

Whether a software company is large or small, their managers need specific information in order to make sound decisions about their projects. This information usually includes data about effort, schedule, quality, and size. While large organizations count on dedicated personnel to define, collect, analyze, and report information, many small organizations must rely on their sole manager and owner, or on one of their team members, to define and manage their measurement program and its results, among many other tasks. This situation can jeopardize the success of implementing a measurement program, especially when delivering within aggressive schedules, getting new business contracts, and managing the company's growth.

The following sections present the cases of two similar software organizations of 12 employees that have put in place measurement programs supporting project decisions.

1.1 Characteristics of the two small software organizations

The first small organization develops and maintains manufacturing management modules integrated as add-ons to a commercial off-the-shelf accounting system, built on client-server architecture. Nine out of their 12 employees are software developers with an attrition rate of approximately 2 developers per year. The company was started 19 years ago by its president, who was writing code up until 4 years ago but who now acts as the project manager most of time. Their modules are installed in 65 manufacturing industries which have 1 to 30 users for their system, the average being 4 users per customer site. Their biggest issue was quality, having to deal with an increasing number of defects and the effort required fixing them. Before implementation of the measurement program, the defect issue was only known through management's perception since no data was available to analyze the situation. They had little success in implementing a measurement program themselves and were seeking help on that matter.

The second small organization mainly develops and maintains a sophisticated financial package for a large organization that manages loans for assets acquisition. Their system is utilized by approximately 300 users throughout Canada, also built on client-server architecture. The company was started 20 years ago by its president who is now acting as project manager. All of their 12 employees are developers, of which two are analysts dealing with requirements development. They were missing deadlines on several features given their short bi-weekly release cycle. Quality was not an issue since they usually have less than two defects per release found once the release is into production and fixed within half a day. Nevertheless, some of their potential projects were lost to major outsourcing organizations in India from 2001-2002 and they needed to be more competitive. They had learned about the existence of the CMMI [1, 2] and were concerned about applying its best practices to increase their efficiency and productivity.

1.2 Characteristics of their projects

Similarities other than team size and software architecture can be observed in both cases. They define a project as being a set of one or more related features laid out to develop or modify a part of existing modules or new modules. The average project effort is approximately 150 hours; some bigger projects are more than 800 hours. They experience cost overruns in half of their projects. They both use spreadsheets to document requirements, including interface mock-ups, planning data, design decisions, and test cases. They mostly have projects roughly defined and planned to keep the whole team busy for six upcoming months.

1.3 Characteristics of the managers' working schedule

The managers are not employees paid 40 hours per week to perform tasks laid out in their job description. They are the company owner and work between 60 and 90 hours per week. On top of ensuring daily operations and delivering quality software to their customers, they have to handle all other company aspects such as marketing, sales, communication, financing, accounting, human resources, training, and growth, simply because they cannot afford extra personnel to take care of these aspects.

Some twenty years ago, they started their companies themselves and were able to sustain growth at their pace, welcoming a new employee every 18 months on average. Their working schedule is quite full. As a result, these managers have needs for information that can be obtained through an efficient measurement program.

1.4 Requirements for a measurement program in small organizations

To have success in implementing a measurement program, the program must, of course, fulfil the managers' information needs. But it also requires the effort needed to sustain it kept as low as possible. In fact, the effort saved by managers for performing their decision-making process in all aspects covered by the measurement program should outweigh the program cost.

2 Case no. 1: dealing with quality issues

A half-day mini assessment of their software development process revealed that it was fuzzy and poor in quality control activities. The software team had grown significantly over the last three years. Several hundred defects were identified in each release. The manager had few insights into these defects and could hardly make sound decisions for prioritizing them.

Their objectives were first to reduce the number of defects, and then to increase team's productivity, but they did not know what were their defect density and their productivity at that point in time.

2.1 The solution approach

The approach used is based on the Personal Software Process (PSP) [3], along with other process improvement methods known by the authors, such as the IDEAL^{SM} model [4].

Step 1: Stabilize the software process

The first step taken was to document and stabilize the software development process. The process phases were clearly defined along with expected work products, activities, and responsibilities. Half a day every other week was dedicated to process improvement where team members gathered to communicate and share their improvement experiences. Due to small projects at the time, the process was stabilized within three months. Project management measures were adapted to answer "how much effort is spent at each phase" and "how much effort is spent correcting defects".

Table 1 provides the 15 phases that were documented as part of the software process. Each of these phases is systematically applied on every project. When a new project is created in the timesheet system, all these steps are also created as tasks. Tasks 1 to 14 may be performed consecutively (waterfall cycle) or iteratively but effort is always measured per project, not per iteration, because it was not relevant for the organization to oversee the effort of a single iteration.

No.	***Process phases***	***Description***
1	PAEX	Analysis of customer requirements (Phase Analyse des EXigences)
2	PAFO	Functional analysis (Phase Analyse FOnctionnelle)
3	PATE	Technical analysis (Phase Analyse TEchnique)
4	PRAN	Analysis review (Phase Revue de l'ANalyse)
5	PDES	Design (Phase DESign)
6	PRDE	Design review (Phase Revue de DEsign)
7	PEST	Estimation (Phase d'ESTimation)
8	PRES	Estimation review (Phase Revue de l'EStimation)
9	PCTU	Construction and unit testing (Phase Code et Tests Unitaires)
10	PRCO	Code review (Phase Revue de COde)
11	PTBB	White-box testing (Phase Test Boîte Blanche)
12	PTBN	Black-box testing (Phase Test Boîte Noire)
13	PVTE	Tests verification (Phase Vérification des Tests)
14	PDUS	User documentation (Phase Documentation USager)
15	PFIN	Project finalized (Phase de FINalisation)

Table 1: Process phases and their description

Defects can be identified in any of these phases. The last phase (PFIN) accounts for defects found by developers and users once the software is in production. The manager needed to know in which phase every defect was injected in the process in order to focus process improvement activities on phases which require them the most. Defects could be injected in one of the analysis phases (PAEX, PAFO, PATE), the design phase (PDES), the estimation phase (PEST), or the code and unit test phase (PCTU). Defects in the user documentation (PDUS) are not recorded as software defects and are handled separately by a technical writer.

There was already a spreadsheet database to record defects that contained the following information: unique defect number, project number, affected customer, software version where it was found, date the defect was identified, severity, defect description, date the defect was fixed, and software version where it was fixed. A decision was made to improve the defect database by adding columns to record the following data, among other improvements:

- Defect categories (see Table 2 below).
- Phase in which the defect was identified.
- Phase in which the defect was injected (determined through investigation).
- Effort to fix the defect.

Thus, effort spent fixing defects is recorded twice: in the defect database for every defect and in the timesheet system as part of the phase where the defect was identified (e.g., PTBB or PTBN). By doing so, the defects database readily answers "how much effort is spent fixing defects", "in which phase defects are mostly injected", and "what categories of defects are mostly found".

Cat.	***Description***	***Example***
1	Missing	Missing items in a phase (e.g., PAEX, etc.)
2	Irrelevant	Irrelevant items in a phase
3	Incorrect	Incorrect or imprecise items in a phase
10	Documentation	Comments, messages, manual, etc.
20	Security	Locking errors, user management, access permission
30	Packaging	Configuration management, build. etc.
40	Assignment	Declaration statements, duplicates, objects or variables initialization, freeing memory, range (array), boundaries (variables), scope, etc.
50	Interface	Procedure call, references (parameters), files, display, printing, communication, formats, contents, etc.
60	Checking	Error messages, inadequate conditions, exceptions not handled, etc.
70	Data	Structures, contents, etc.
80	Function	Pointers, loops (off-by-one, increments, recursivity), algorithms, calculations, etc.
90	System	Performance (speed), memory usage, etc.

Table 2: Defect categories, inspired by those found in the PSP

		Defect categories															
		nil	***blank***	***1***	***2***	***3***	***10***	***20***	***30***	***40***	***50***	***60***	***70***	***80***	***90***	***All***	***%***
Phases	***PAEX***	1	0	6	1	4	0	0	2	0	1	0	1	0	0	***16***	0.6%
	PAFO	6	0	29	1	19	0	1	1	1	2	3	1	6	1	***71***	2.8%
	PATE	18	0	9	0	3	0	2	2	2	4	1	5	2	4	***52***	2.1%
	PDES	13	1	74	8	4	0	2	3	23	291	10	12	5	0	***446***	17.8%
	PEST	0	0	5	0	0	0	0	0	0	0	0	0	0	0	***5***	0.2%
	PCTU	216	14	60	10	8	26	45	5	198	600	321	110	275	27	***1915***	76.4%
	All	***254***	***15***	***183***	***20***	***38***	***26***	***50***	***13***	***224***	***898***	***335***	***129***	***288***	***32***	***2505***	***100%***
	%	10.1%	0.6%	7.3%	0.8%	1.5%	1.0%	2.0%	0.5%	8.9%	35.8%	13.4%	5.1%	11.5%	1.3%	***100%***	

Table 3: Number of defects injected by phase, per defect category

		Defect categories															
		nil	***blank***	***1***	***2***	***3***	***10***	***20***	***30***	***40***	***50***	***60***	***70***	***80***	***90***	***All***	***%***
Phases	***PTBB***	1.25	0.00	10.25	2.25	0.50	0.00	0.00	0.00	12.25	0.50	3.75	2.50	1.50	0.00	***35***	1%
	PTBN	13.00	0.00	7.75	0.00	0.25	0.00	0.00	0.00	3.75	10.75	0.00	2.25	1.00	2.00	***41***	1%
	PVTE	133.00	14.65	19.50	0.25	0.00	3.50	59.75	8.00	62.50	375.60	138.49	50.50	136.00	29.00	***1031***	28%
	PFIN	125.50	40.50	221.75	7.50	100.00	18.25	108.50	35.50	232.00	395.00	378.00	178.50	640.50	64.00	***2545***	70%
	All	***272.75***	***55.15***	***259.25***	***10.00***	***100.75***	***21.75***	***168.25***	***43.50***	***310.50***	***781.85***	***520.24***	***233.75***	***779.00***	***95.00***	***3652***	***100%***
	%	7.5%	1.5%	7.1%	0.3%	2.8%	0.6%	4.6%	1.2%	8.5%	21.4%	14.2%	6.4%	21.3%	2.6%	***100%***	

Table 4: Effort to fix defects by phase, per defect category (hours)

From November 2003 on, new defects were added to the improved database. Table 3 and Table 4 provide defect related data from November 2003 to August 2006. Table 3 clearly shows that most defects were injected during the code and unit test phase.

Step 2: Introduce functional size measurement

The second step was to introduce functional size measurement. An earlier attempt with IFPUG [5] led the team to abandon this method they considered too costly to sustain. The COSMIC-FFP [6] method was thus adopted. Implementing COSMIC [7] was rather simple for them since it only required adding one worksheet to their analysis spreadsheet where functional requirements were written in a way to easily identify functional processes and data movements.

The team was measuring size for estimation purposes and for projects comparison. The manager wanted the software process to be predictable because most projects were quoted as firm fixed price. Thus, knowing the average effort spent per size unit became valuable. Training was provided to the manager and team members on the COSMIC method.

From that point, new projects were created and performed. Defect, size, schedule, and effort data was recorded for 25 projects. Size was measured from the functional requirements early in the process. The manager then compared actual data of size and effort. He was not fully satisfied with the correlation and standard deviation since he needed a more precise estimation model. He verified the functional size measurement of all 25 projects and found the following issues:

- Different individuals were obtaining different sizes of the same project due to misunderstanding of the method. Developers had a clear tendency to measure from the developer's point of view when the user point of view was required.
  To resolve that issue, they decided to limit functional sizing to two individuals who seemed to have a common understanding of COSMIC and they mostly obtained similar results (less than 2% difference) when adopting the user's point of view.
- Project size was different between the initial analysis phase and the final phase because requirements are changing during the project execution (e.g., data validations requiring extra "read" data movements, data groups being added).
  This issue was resolved by measuring the size twice during the project:
  - Initial size: right after the user graphical interface mock-ups are done (end of analysis phase).
  - Final size: at the end of the project, to compensate for the missing details in the requirements documented during the analysis phase.

- For estimation purposes, it made no sense for them to use one single point (size unit) for data movements on data groups that had a large number of attributes because software features handling large data groups require more effort to develop. This was the main reason for the imprecision of the correlation between effort and size.

  To resolve that issue, they closely examined data groups and data movement on the 25 projects and developed a local extension to COSMIC, where data groups are "weighted" as one extra point for every set of 12 attributes for "exit" and "read" data movements (i.e., a "read" or an "exit" of a data group of 1 to 12 attributes is counted as one, of a data group of 13 to 24 attributes is counted as two, etc.) They recalculated the correlation factor between final size and effort which was then more than 0.90. They knew at that point that their productivity model was stable. Table 5 provides their productivity models in hours per functional size unit for the overall project effort, the programming effort, and the rework (i.e., effort to fix defects). However, their local extension to COSMIC does provide a bigger size than the standard COSMIC method and makes it difficult to compare to other organizations.

	Projects 1 to 25
Overall productivity	1.94 hours/size unit
Programming productivity	1.12 hours/size unit
Rework effort per size unit	0.33 hours/size unit

Table 5: Performance data for the first 25 projects

Step 3: Do a Pareto analysis of defects found

The third step was to do a Pareto analysis of the defects found using the injected phase. Table 3 clearly shows that most defects were injected in the code and unit test phase and, for most of them, they were the result of errors made by programmers.

Based on actual data obtained from the measurement program, a decision was taken to introduce peer reviews as a method for identifying defects early in the process. The team developed checklists to help them recognize specific defect types as outlined by the Pareto analysis of defects.

Then, considering defect data and productivity data, the manager set the following business objectives:

- Increase the overall productivity by 10%.
- Decrease the number of defects by 20%.

The team continued recording measures and indicators for 11 more projects before looking at the results six months later.

2.2 The results

Within six months, they achieved a productivity increase of more than 11% and a drop of the effort required to fix a defect by 61%, as shown in Table 6. They were also able to reduce the number of defects by 33%.

	Subset of projects			*Improvement*
	A: 1 to 25	*B: 26 to 36*	*C: 1 to 36*	*between A and B*
Overall productivity	1.94	1.73	1.82	11%
Programming productivity	1.12	0.91	1.04	19%
Rework density	0.33	0.13	0.26	61%

Table 6: Performance data for the first 36 projects

The six following project indicators from the PSP approach were identified and are used in every project for decision making:

1. Defect density = number of defects / final size
   Usage: monitor project quality and take necessary actions when it deviates from internal range of values.
2. Rework density = total effort for fixing defects / final size
   Usage: monitor waste of effort induced by fixing defects and improve peer review process when required.
3. Overall productivity = total effort / final size
   Usage: monitor that project performance is within internal range of values.
4. Schedule delivery = number of calendar days / final size
   Usage: predict and communicate project delivery date to customers.
5. Completeness of requirements = final size / initial size
   Usage: adjust estimation model based on average completeness of requirements.
6. Accuracy of estimates = actual effort / estimated effort
   Usage: adjust estimation models when required.

The project manager and the analyst are able to count functional size of an average project in less than an hour, which they consider as a low measurement cost. Functional size is very useful for them since it is part of the majority of their project indicators.

Measurements have been collected on all organizational projects. Improvements were made on estimation contingency factors through feedback from actual measures of their projects. Hence, for estimation purposes, they have to increase their initial functional size measure by 20% (average difference between initial size and final size), apply their productivity model, and then add effort contingency for customer and management issues. Now, they rarely have cost overruns on their projects.

2.3 The challenges

As the manager decided to implement a measurement program, the first challenge was resistance to change from employees. It was alleviated through communication and process insights. It took time for some employees to understand the concept of "quality is free" and the measures were enlightening to them. Nobody likes to be measured, specifically on productivity. But the productivity indicator is really of a project, not an individual.

The second challenge was the rigour required to sustain the measurement program which still is an every day challenge. But whenever they lack rigour, project results drive away from their standard project performances. They understood that project control comes with measures.

The third challenge they faced was to be able to compare their productivity with other organizations, such as projects found in the ISBSG database [8]. Because they need their local extension to COSMIC, they intend to divide their functional size measure in two: the first part being measured as the standard COSMIC method, and the second part being what they add to account for data groups containing a large number of attributes.

3 Case no. 2: concerned with applying best practices

This small organization had a stable but undocumented process. They did not face any big issue related to software development but small irritants were observed, namely in software development estimation and customer communication. They wanted to learn about the CMMI, assess their practices for project management, engineering, and support process area categories, and start improving their process on a continuous basis. Their process improvement motivation and objectives were to:

1. Improve quality in general: reduce the number of defects, deliver on time, deliver or exceed customer expectations, while including innovation.
2. Improve quality of life while working: from fire-fighting and day-and-night shifts to smoother 8 to 10 hours per day of work, 5 days per week.

3. Manage growth through a normalized process: reduce rework, have all team members apply the same consistent and repeatable process.
4. Be able to delegate some tasks from management to team members within the process framework.

They already had a measurement program that included effort and schedule measures, and its main purpose was for billing the customers at the end of every project. Their biggest customer's demand for features was growing, so was the team size. They wanted to continue to be result oriented, i.e. always seeking customer satisfaction through project progress meetings also used to obtain new projects. They also have specific meetings to obtain a clear understanding of customer needs.

Their largest software product was written in Visual Fox Pro (VFP). They have undertaken a system technology re-engineering from VFP to C#, done in small steps at a time, starting with the business logic layer while the graphical user interface (GUI) is still in VFP. New modules are generally developed in C#. The actual plan is to migrate the whole application to C# in the next 3 years.

3.1 The solution approach

The approach used is based on the CMMI. The continuous representation was adopted since the organization was not seeking any acknowledged "maturity level".

Step 1: Document and start improving the process

Initial discussions highlighted that the process was known from team members but it was undocumented. As team size was growing, it became important to document the process for communication purposes. The manager required the process documentation to be as light as possible. The team was able to graphically document the whole process on only five pages, supported by a few templates for expected work products.

The naming for process steps was partly inspired from steps described in the Guide to the Project Management Body of Knowledge (PMBOK) [9]:

1. Initiate project: gather customer needs, reach common understanding of requirements, and develop requirements documentation.
2. Plan project: establish estimates and schedule, and obtain "go ahead" from customer.
3. Execute project:
    a. Develop software: as per requirements, perform unit testing.
    b. Test software on internal test environment.

    c. Package software and install on customer’s test environment.
    d. Supervise acceptance testing on customer’s site and obtain “go live” from customer.
    e. Promote software into production environment.

4. Control project: fill timesheets (actual effort data) and monitor progress, issues, and action items.
5. Close project: archive project data, bill customer, and receive payment.

With a documented process, it was easier to identify where improvements could be made because the team now had a clear view of their process. Improvements were made, namely on customer communications and supervision of acceptance testing, in order to alleviate or remove identified irritants.

Step 2: Provide training on best practices

Small half-day workshops on CMMI were held every other week with the focus on one CMMI process area per session. The scope was established and agreed upon to the following CMMI process areas:

Project management category	***Engineering category***
Project planning	Requirements management
Project monitoring and control	Requirements development
Risk management	Technical solution
Integrated project management for IPPD	Product integration
	Verification
	Validation
Support category	
Configuration management	
Process and product quality assurance	
Measurement and analysis	

Actual practices were looked at in detail, starting with project planning and project monitoring and control process areas. Gap assessments were performed for each process area within scope, and notes were taken on every practice on strengths and weaknesses. Improvement suggestions were noted and made prior to the next workshop.

The two-week delay between workshops allowed enough time for the team to implement improvements. Feedback was provided at the beginning of the next workshop and improvement adjustments were recommended and made.

Step 3: Improve existing measurement program

Actual effort is measured by project phase using a home grown timesheet system that integrates other project attributes such as start and end dates, project type, and estimated effort. Effort estimates are based on core software tasks defined in the project spreadsheet. Analysis effort, part of the initiate project phase, is always recorded as performed, prior to planning and estimating. Percentages are added for system testing, acceptance testing, and project management. The total effort estimate becomes a “not to exceed” quotation for the customer. The manager’s motivation for an accurate productivity model was that he did not want to ever exceed that amount because it could not be billed to the customer. But he also wanted the estimate to be kept as low as possible because he gets the project only if the cost of developing it is outweighed by the return on investment the customer will get.

As project management training sessions were held, it became clear that effort and schedule measures were insufficient to fulfil the manager’s information needs, and that a size measure was required to improve their estimation efficiency. The COSMIC-FFP method was chosen and training was provided to the analyst and project manager. The analyst was responsible to measure the size of current and new projects. A productivity model was established based on functional size and effort. It was mostly used to validate estimates made on a task-effort basis.

Important productivity differences were observed among projects, ranging from 3 to 5 hours per size unit. Investigation was made to explain this variation to find out that differences come from the technology used where VFP project productivity averages 2.5 hours per size unit (i.e. Cfsu), and C# project productivity averages 4.5 hours per size unit, once the learning curve was over. One of the difficulties was when they have to integrate both technologies (i.e., VFP for GUI and C# for business logic) because it is more complicated and requires more effort. Also, a lot of effort must be spent on creating stored procedures for new database tables. Many projects developing new features were actually using existing tables along with their stored procedures. In these cases, the required effort per size unit was lower than the productivity model for projects where most tables were new. The productivity model was adjusted not only based on functional size but also on the number of new tables and stored procedures that need to be created. Calculating expected project effort based on these attributes was automated with macros into the spreadsheet template.

The team productivity model behaves as if new development occurs only for the first functional process (i.e., creating GUI, business logic, and persistence layers, plus stored procedures) and as if it switches to maintenance mode thereafter (i.e., stored procedures are already created and used as in an evolutive or adaptative software maintenance).

3.2 The results

Functional size is now part of their project definition template as a worksheet and macros were developed to raise a flag when a significant difference occurs between their traditional estimate and the productivity model estimate based on functional size. The list of current and upcoming projects now contains functional size along with estimates for comparison between projects and reporting purposes.

3.3 The challenges

The ever growing demand for features from their customer led the analyst to abandon functional size measurement on several projects due to lack of time. Their first challenge was to continue functional size measurement even under time constraints. They intend to experience different ways to measure functional size and expect to gain speed.

Their second challenge was to keep-up with the required rigour. They intend to put in place more project resources to ensure that analysts will have the necessary time to perform measurement. A second analyst was hired but has not been trained yet on the COSMIC-FFP method.

Their third challenge was to improve measurement usage to manage and take decisions. Their decision process is still performed through perception and not enough based on measurement. For example, they used to wait a long time for a customer’s decision on change requests priority, and they have recently added an “expected benefit” attribute on every change request based on number of impacted users, effort per year expected to be saved per user, and effort to develop the change request. Now, change requests priority is automatically calculated and sorted descending on this expected benefit.

4 Conclusion and future work

Even if both organizations had similarities, they were facing different issues. It was clear that one measurement program could not be designed to suit any small organization that intends to use it to fulfil their information needs in order to support their software management decisions. However, similar measures were defined and used including functional size with the COSMIC-FFP method, effort per project phase, and schedule delivery.

Training and resources are key issues to sustain a measurement program. Management support is mandatory for success and managers should be trained as well on what a measurement program can do for them, such as gaining control and confidence over project results.

Implementing a measurement program was done through half-day workshops every other week, in both cases. This pace allowed teams to implement improvements with few disturbances of their normal working schedule of software development. As process improvement experts, it is important to be ready to work within a flexible schedule when helping small organizations in their quest to develop software more efficiently.

Acknowledgement

The authors wish to thank Daniel Murray, president of SIGM Inc. (dmurray@sigm.ca) and Michel Martel, president of Analystik inc. (michel.martel@analystik.ca), for their precious collaboration and information sharing.

References

1. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI(R)-Guidelines for Process Integration and Product Improvement, Addison-Wesley Professional, Boston, MA, 24 February 2003
2. Basque, R.: CMMI-Un Intineraire Fléché Vers Le Capability Maturity Model Integration, Dunod, 2004
3. Humphrey, W.: A Discipline for Software Engineering, Addison-Wesley Professional, Boston, MA, 31 December 1994
4. McFeeley, R.: IDEAL: A User's Guide for Software Process Improvement, Handbook CMU/SEI-96-HB-001, Software Engineering Institute, 1996. [Accessible on line at http://www.sei.cmu.edu/publications/documents/96.reports/96.hb.001.html]
5. ISO/IEC 20926: Software engineering -- IFPUG 4.1 Unadjusted functional size measurement method -- Counting practices manual, Geneva, Switzerland, 2003
6. ISO/IEC 19761: Software engineering -- COSMIC-FFP -- A functional size measurement method, Geneva, Switzerland, 2003
7. Abran, A. et al : Measurement Manual, The COSMIC Implementation Guide for ISO/IEC 19761:2003, version 2.2, January 2003
8. International Software Bechmarking Standards Group (ISBSG), http://www.isbsg.org
9. Project Management Institute, A Guide to the Project Management Body of Knowledge (PMBOK® Guide) - Third Edition, The Project Management Institute, 2004