Comparison of CMM Level 2 and eXtreme Programming

Jerzy R. Nawrocki[1], Bartosz Walter[1], Adam Wojciechowski[1]

[1] Poznan University of Technology, ul .Piotrowo 3A, 60-965 Poznan, Poland
{Jerzy.Nawrocki, Adam.Wojciechowski}@put.poznan.pl,
Bartosz.Walter@cs.put.poznan.pl

Abstract. Lightweight software development methodologies promise an easy way to deliver products of high quality without excessive cost. On the contrary, classical heavyweight processes are well-defined and proven, but require a lot of effort. Two approaches: eXtreme Programming (XP) and CMM Level 2 have been used in joined industry-academic software projects run at the Poznan University of Technology. Running concurrently those two software approaches allowed us to compare them on the basis of experimental data. After the projects were completed, major risk factors connected with both approaches have been collected and some improvements have been proposed.

1 Introduction

In 1960's big software producers, such as IBM, encountered serious problems concerning software development. Since then a lot of ideas, methods and tools have been proposed to make software development easier, more predictable, and more effective in the sense of customer satisfaction. Structured programming [7], modularity [17, 26], structured analysis [5], object-oriented programming [23], object-oriented analysis [3], CASE tools and many others have been proposed to produce software of better quality.

In this quest for quality, many software developers and researchers advocated to look to other, more mature, areas of engineering to bring from there some useful ideas and adapt them to the context of software development. One of such proposals was the ISO 9000 suite developed around 1987 with classical consumer goods in mind [24] and then used also for software development. Unfortunately, ISO 9000 in many cases proved to be too restrictive, resulting in too much paper work, and too 'heavy' [21].

Capability Maturity Model (CMM for short) developed a few years later at the Software Engineering Institute (Pittsburgh, USA), much better suited the needs of software developers [18]. It was supported with two other products of SEI: Personal Software Process [10] (PSP) and Team Software Process (TSP) [11]. CMM was developed in a few versions oriented toward different aspects of software development: Systems Engineering, Software Engineering, Integrated Product and Process

Development, People CMM, and so on. In 2000 first three CMM models have been combined into so called CMM Integration (CMMI for short) [4].

Unfortunately, CMM models and PSP are also perceived by many people as too heavy and too bureaucratic. So, no wonder that in late 1990's so called 'lightweight' software development methodologies appeared. Perhaps the most popular one is Extreme Programming (XP for short) proposed by Kent Beck [1]. In XP artifacts are reduced to code and test cases. There are no inspection meetings. Planning horizon is very short, so advanced planning tools (such as Function Point Analysis) are getting useless. Many people admire XP. Among them is Tom de Marco, who expressed his great support for XP during his keynote speech at the joint IEEE Metrics and ESCOM 2001 conference (see also [6]).

The problem is that we know very little about lightweight methodologies. There are books presenting the idea and practices [1, 2, 12] but we lack experimental data that would clearly assess when one approach is better than another. Many years ago Tom De Marco said: 'You can't control what you can't measure'. Unfortunately, we still lack measures concerning XP. The only data that have been published so far concern pair programming [16, 25, 14] but those experiments are very limited.

The aim of the paper is to compare some aspects of exemplary XP and CMM Level 2 implementations (Level 2 of CMM is, like XP, project-oriented, while higher levels of CMM are oriented toward the whole organization) and to describe important risk factors related to both methodologies. Recent publication by Paulk [19] gave some light into the subject, though it was focused rather on mapping between XP and CMM practices. In this papers experimental data that has been collected during laboratory-like experiments and some came from student projects developed as part of the Software Development Studio [13] is presented. Next section describes the Software Development Studio and the student projects. The student projects have been split into two groups: SDS-CMM and SDS-XP. Section 3 describes practices adopted by the CMM group, while Section 4 concerns the XP group. Then, in subsequent sections we discuss main risk factors that can serve as comparison criteria. In Section 5 software maintenance issues are analyzed. In Section 6 both approaches are compared from the point of view of customer involvement. Section 7 deals with pair programming and presents results of laboratory-like experiments that aimed at comparison of pair programming with individual programming. The last section contains conclusions and directions of further research.

2 Software Development Studio

Software Development Studio is a 2 years (4 semesters) long academic module run at the Poznan University of Technology. It is intended to teach students the software development in projects for external customers (either industry companies or different units of the university). Every project involves eight students: four from 3rd year, two from 4th and two from 5th year of study. Students play different roles in two different, one-year long projects in subsequent years, gaining experience in program-

ming, software project management and quality assurance. The educational outcome of the project is bachelor thesis written by 3 rd year students.

The projects are proposed and defined by prospective customers. Project subjects vary very much, from scientific ones (e.g. *Assembling DNA sequences*, *Managing users in meta-computing environment*) to commercial ones (like *Defect tracking system*, *Multi-project management at a software company*).

In year 2000 all eleven projects were divided into two Project Areas: SDS-XP and SDS-CMM, devoted respectively to eXtreme Programming and CMM Level 2. Since some of the projects were run concurrently according to both processes (by different project teams), we could observe the impact of methodology on the process quality.

XP defines a very different set of roles and responsibilities than the classical methodologies. Therefore, team structure and roles were different in both Project Areas. Moreover, the processes had to be suited for academic environment, what resulted in some modifications of the original approaches.

3 Implementation of CMM Level 2

Software Engineering Institute's (SEI's) Capability Maturity Model[SM] (CMM) is a framework representing improvement paths for software organizations. It is a domain-independent hierarchical structure built on five levels; except level 1 (Initial), each maturity level is decomposed into a number of Key Process Areas (KPA). Each KPA indicates a cluster of activities that performed collectively, help to achieve a set of goals important for organization's software process.

Among the five levels, the Level 2 (*Repeatable*) is directly project-oriented, whereas the others focus on the whole organization. It is important that the first step towards process maturity concentrates on individual projects and team internal communication, leaving more general issues for higher levels. CMM puts stress on initial identification of the crucial for the process and product quality practices. This allows to concentrate on the most important success factors.

The Level 2 comprises 6 Key Process Areas: *Requirements Management, Project Planning, Project Tracking and Oversight, Subcontract Management, Quality Assurance* and *Configuration Management*[18].

At university it is difficult to be fully compliant with all the areas defined at the Level 2. Some deviations are inevitable. For example one of the areas – *Subcontract Management* - has been entirely omitted, since the SDS project teams must develop the software by themselves. Therefore the process defined for SDS-CMM does not meet all the CMM requirements, but it is close to them.

The team is organized hierarchically: 4 students from 3rd year of study are *Programmers and Designers* (PDs), 4th year students act as *Project Managers* (PMs), whereas the oldest and most experienced ones (5th year of study) play roles of *Quality Assurers* (QAs). Moreover, a senior management layer is established for every Project Area (Project Area Managers are university teachers). Detailed responsibilities assigned to roles are presented at Table 1:

Table 1. SDS – CMM project team - roles and responsibilities.

Role	**Responsibilities**
Customer	Requirements Management, Acceptance tests
Quality Assurers (5th year)	Inspections, QA, Configuration Management (*Software Cnfiguration Control Board*, SCCB)
Project Managers (4th year)	Planning, Progress tracking, Configuration Management (*Software Configuration Management Board*, SCMB)
Programmers & Designers (3rd year)	Design, Coding, Unit testing

In order to support SDS-CMM Project Area projects, two university modules were introduced: Requirements Engineering and Quality Assurance. Besides, every project team is assigned a dedicated university supervisor to support the project and assess the educational outcome. This Project Area follows a modified CMM-based process presented at Table 2:

Table 2. Most important activities in SDS – CMM process implementation

Level 2 Key Proces Area	**Most important activities implemented**
Requirements Management	• requirements are documented, reviewed and managed; • IEEE/ANSI Std 830-1993 is used;
Project Planning	• plan is documented and managed; • size and effort are estimated; • risk factors are identified, monitored and mitigated;
Project Tracking and Oversight	• the plan is a basis for progress tracking; • effort, schedule and progress are tracked; • formal reviews are perfomed twice a semester;
Quality Assurance	• the SQA group reviews documents and activities; • Fagan-like inspections are used;
Configuration Management	• a repository of software baselines exists; • CVS is used;
Subcontract Management	not implemented

4 Extreme Programming Process for SDS

Extreme Programming is relatively new (about five years old) lightweight methodology of software development. It was proposed in response to problems concerning rapidly changing (dynamic) requirements and new technologies. In opposition to

CMM, XP offers a light approach, going on the edge of chaos. It is suited for small software teams (up to 10 people).

The general assumption in XP is that the customer, not the documentation is the heart of project. It means that customer is a privileged team member, and he is able to make all business decisions, which drive the project development process. This leads to instant orientation towards customer's goal: whenever the project objectives change, the focus of the project is changed as well.

XP also prefers informal, oral communication within the team in place of written documentation. This allows both for avoiding documentation maintenance and speeds up the process.

Here the most distinctive features of XP are listed:

- The software development is split into a number of small increments (each one about 2-3 weeks long), giving the developers fast feedback from the customer; planning is performed just-in-time, only for the forthcoming increment.
- The customer role is crucial; he provides the programmers with user stories (informal, customer-written description of the functionality) and he is capable of changing them at any time. They also specifies the aceptance tests.
- Quality assurance is test–centered: writing every piece of code should be preceded by preparation of a set of test cases for the implemented function. If a defect is detected, the programmer writes new test cases that detect the defect if it reappears.
- Programming is done in pairs. This is a technique thought as substitution for reviews and inspections – a pair of programmers work on a single piece of code, so they can review it continuously.
- The code is co-owned by all team members, so that everyone can change anything. It requires a defined coding standard;

For the purpose of SDS we implemented a modified version of the XP approach. The roles and responsibilities for the SDS-XP Project Area are listed in Table 3.

Table 3. SDS – XP project team - roles and responsibilities.

Role	**Responsibilities**
Customer	User stories, Business decisions, Acceptance tests
Testers (5th year)	Implementing and running acceptance tests
Coach (4th year)	Managing the team
Tracker (4th year)	Risk management, Progress tracking
Programmers (3rd year)	Development, Coding, Unit testing

The project time span was divided into 2 releases, each consisting of 2 iterations, each iteration took approx. 4 weeks.

In year 2000 SDS-XP Project Area consisted of 5 projects.

5 Software Maintenance Issues

Software maintenance is a real test for the organization's software process: it shows whether the knowledge about projects is kept and can be recalled at further stages at low cost. In real environment the maintainability (and its cost) is a very important factor for the software process assessment.

Although SDS projects last one year only, it is long enough to experience problems with maintainability. In both project areas several artifacts were produced, and keeping them up-to-date was one of key success factors.

In case of SDS-XP Project Area, the reduction of documentation to code and test cases makes the software maintenance particularly difficult. The process for that project area assumed only two written artifacts: user stories (it was our deviation from XP – we asked our students to keep user stories documented) and the code with accompanying test cases. Since no other documents existed, it was hard for teams to keep the track between changing user stories, the code implementing them and the associated test cases. Changes in source code, which might trigger redefinition of test cases, were difficult to follow and track. Since we did not have a board for user stories, the person designated to maintain the artifacts, became a key team member.

The other drawback was lack of an intermediate layer similar to the architectural design in our CMM implementation process. The system metaphor (informal system description understood by all team members including the customer), despite its simplicity and customer-readability, is not well suited for maintenance.

One of symptoms of problems in project maintenance in XP were the difficulties the 3rd year students had in writing their bachelor thesis at the end of the project. Thesis describes the process and system developed during the project. Several students could hardly recall details of the work after just three months. Unfortunately, refactoring was not a common practice in SDS-XP projects, and this could be the reason for the maintenance problems we encountered.

In order to improve the process, we suggested several changes into our XP implementation, which were implemented next year. First, for each user story a business rationale should exist. This gives a chance to keep all user stories tied to the customer goal. Secondly, some kind of architectural model appears inevitable. We suggest to use UML as a unified and commonly used by software engineers language, though it may not be readable for the customer. On the other hand, it improves the communication within the team if programmers are not used to do it orally.

In XP changes drive the process. Beck claims that "*maintenance is really the normal state of an XP project*"[1]. Unfortunately, infrequent refactoring and lack of written artifacts make inexperienced programmers to forget the specifics of user requirements and to miss the goal. We found it difficult to implement XP in newly formed teams, which was the case.

In the SDS-CMM Project Area the maintenance was based on the documentation. The students in this project area had significantly less problems with writing their bachelor theses. Moreover, one of the SDS-CMM projects was continued by a new team and the new team did not report important problems.

This shows that the risk connected with maintenance in XP is much higher than for the CMM-based approach. Code and test cases seem to be insufficient.

6 Customer Involvement

In XP, customer is the central person of a team. His presence *in the team* (not *beside the team*) is a prerequisite for customizing the project to the needs immediately after the needs arise. The customer should make or approve every decision concerning the project scope and schedule. Therefore, the insufficient customer involvement seems to be an important risk factor. Lack of the legitimated decision maker in the team leads to decision guessing. Then the pass of acceptance tests is in danger.

XP assumes on-site customer. Unfortunately, in our experiment there was no single project where customer would be available all the time. Also in other XP experiments [8, 20, 22] customer was not continuously present. To manage this, our project teams organized on-line chat sessions with customers, which provided a partial solution for customer absence. Many teams had to go to the customer's premises to get necessary knowledge, which was not so effective.

In most SDS-XP projects the close contact with customer resulted in a higher customer satisfaction, while in one case lack of decision-maker hold the project for a few weeks. Therefore we conclude that a close, personal contact with customer and their instant presence is a must for XP-like processes.

For SDS-CMM Project Area it was not assumed that customer will be available all the time. Customers were supposed to meet the team quite frequently at the initial stages (before coding) – at least once a week, and once a fortnight during the implementation phase. For that projects we observed noticeably lower actual customer involvement: sometimes they did not have time for reading documents, which led to costly refactoring at further stages. On the other hand, most of customers attended at least the inspection meetings, making important remarks. In the SDS-CMM Project Area none of the customers changed their requirements once they had been defined.

It appears again that the assumptions made in XP process, concerning the customer involvement, are much stronger and needs appropriate risk management. Although all projects in SDS-XP Project Area succeeded in general (products were delivered with only minor delays), it appears that fast decision making and almost immediate feedback from the customer is crucial for lightweight methodologies.

7 Pair programming

Pair programming is one of the most important and characteristic features of XP. It means that all the production code is programmed in pairs. While one programmer is in control of keyboard and enters the code, the other looks at screen and checks if the code is correct. After some time they switch the roles. Next day they change their partners.

Kent Beck claims that pair programming removes the need for inspection meetings because there is continuous inspection – all the time there is someone checking quality of the produced code [1]. However, there are some problems concerning pair programming:

Not everybody likes working in pairs. In a software company that decided to join our experiment to provide large-scale empirical data one of very clever programmers did not want to program in pairs at all. He could not stand someone watching him working.

Even if people are ready to program in pairs, it does not mean that everybody will be able to work with anybody. It is common observation that with some people we have very good contact and communication is very effective, while with other people it is very hard to communicate. In case of programming it is even more difficult. It can happen that two people have very good social contacts, but while they start to work together one of them is much faster than the other, and after some time they cannot work together because they work at very different speed.

Thus, we cannot assume that *all* the production code is pair programmed (in experiment described by Schuh [22] pair programming took 30% of developer time). Consequently, one cannot remove inspections. It is important to realize that inspection meetings serve two purposes:

- It is well known that through inspections one can assure quality.
- Moreover, inspection meetings make all the parties present at the meeting responsible for prospective defects. That is of great importance from psychological point of view. When a defect appears, it is easier to focus attention on what to do and prevent people from blaming anybody else.

If there is a person in a project team that hates pair programming, inspection meetings can be the only solution. Obviously, they should be well organized and not too long (Fagan advocated inspection meetings lasting not longer that 1.5 hour [8]). It means that frequent inspections are better than rare. If inspection meetings are organized frequently, the size of artifacts subject to inspection is relatively small. Then inspection meeting can be short and more effective. While observing student projects we have found that inspection meetings should be once a week (students working on SDS projects are expected to spend about 15 hours per week on a project). In industry setting inspection meetings should be probably even more frequent (at least twice a week).

Even if programmers are ready to program in pairs, it does not mean that they will be allowed to. Our students reported that in one company senior management was very enthusiastic about XP but very skeptical about pair programming (similar observations were reported in [9]). From management point of view pair programming was waste of time and money. Why to assign two people to a task that one person can do? It is an important question concerning pair programming: what is overhead connected with pair programming? Is pair programming effective from economical point of view? That question seems to be still open. Very few experiments concerning pair programming have been performed till now. According to an experiment described by John Nosek [16] pairs consumed 43% total effort more than individuals. Unfortunately, that experiment concerned only one very short program-

ming assignment (the allotted time was 45 minutes). The second experiment has been conducted at the University of Utah [25]. The results were more optimistic. The authors report that '*the pairs completed their assignments 40% to 50% faster*'. That means that total effort in case of pairs was not greater than in case of individual programmers. On the other hand, from our experiments follows that pairs need at least 50% more effort than in case of individual programming (see [14] for more details). So, it is not obvious if pair programming really pays-off and further research is necessary.

8 Conclusions

After completion of all projects in both Project Areas (SDS-XP and SDS-CMM), several conclusions can be deduced. First of all, XP has some critical risk factors, which can lead to project failure. The factors we described appeared to be important in SDS projects, that do not claim to be real industry ones, although in many aspects (like full-time students involvement, progress tracking) they were very similar. Therefore, special care should be paid when introducing XP into organization's practice. We suggested several improvements that allowed the teams to finalize the projects successfully (i.e. with delivering both the product and master thesis without excessive delay). Anyway, it proved to be non-trivial for newly-formed teams, which members are inexperienced and do not know one another very well.

On the other hand, the SDS-CMM process implementation seems more stable and less failure-prone, but also the work is more tedious (e.g. keeping the numerous documents consistent). In the SDS-CMM Project Area, the customer involvement, appropriate CASE tools and strict process definition seem to be the most important success factors.

The comparison of the processes shows that SDS-CMM process is less failure-prone, since it puts stress to the process organization, whereas XP process stresses personal team members involvement and experience. Still, the impact of risk factors we encountered may be dependent on the team itself. Anyway, XP appears to be an interesting alternative for teams with some experience in working together, members of which can communicate easily.

References

1. K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, Boston, 2000.
2. K. Beck, M. Fowler, Planning Extreme Programming, Addison-Wesley, Boston, 2001.
3. G. Booch, Object-Oriented Analysis and Design with Applications, Redwood City, Addison-Wesley, 1991.
4. CMMI Models, http://www.sei.cmu.edu/cmmi/products/models.html (February 2002)

5. T. DeMarco, Structured Analysis and System Specification, Englewood Cliffs, Prentice-Hall, 1978.
6. T. DeMarco, The Agile Organization, http://www.escom.co.uk/conference2001/papers/keynote-tom-demarco.pdf (February 2002)
7. C. Elgot, Structured programming with and without GO TO statements, IEEE Trans. Soft. Eng., No 1, 1976, 41-53.
8. M. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol. 15 (1976), No. 3.
9. J. Grenning, Launching Extreme Programming at a Process–Intensive Company, IEEE Software, vol. 18, No. 6, 27-33.
10. W. Humphrey, A Discipline for Software Engineering, Addison-Wesley, Reading MA, 1995.
11. W. Humphrey, Introduction to the Team Software Process, Addison-Wesley, Reading MA, 2000.
12. R. Jeffries, A. Anderson, C. Hendrickson, Extreme Programming Installed, Addison-Wesley, 2001.
13. J. Nawrocki, Towards educating leaders of software teams, **in**: P. Klint, J. Nawrocki (eds), Proceedings of Software Engineering Education Symposium SEES'98, Scientific Publishers OWN, Poznan, 1998, 149-157.
14. J. Nawrocki, A. Wojciechowski, Experimental evaluation of pair programming, **in**: K. Maxwell, S. Oligny, R. Kusters, E. van Veenendaal (eds), Project Control: Satisfying the Customer (Proceedings of ESCOM 2001, 2-4 April 2001, London, UK), Shaker Publishing, 2001, 269-276, http://www.escom.co.uk/conference2001/papers/nawrocki.pdf (February 2002)
15. J. Nawrocki, B. Walter, A. Wojciechowski, Toward maturity model for eXtreme Programming, Proceedings of the 27th EUROMICRO Conference, Los Alamitos, IEEE Computer Society, 233-239.
16. J. T. Nosek, The case for collaborative programming, Communications of the ACM, vol. 41 (1998), No. 3, 105-108.
17. D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, vol. 15, No. 12 (1972), 1053-58.
18. M. C. Paulk et al., The Capability Maturity Model: Guidelines for Inproving the Software Process, Addison-Wesley, Reading MA, 1995.
19. M. C. Paulk, Extreme Programming from a CMM Perspective, IEEE Software, vol. 18, No. 6, 19-26.
20. Ch. Poole, J. W. Huisman, Using Extreme Programming in a Maintenance Environment, IEEE Software, vol. 18, No. 6, 42-50.
21. J. Seddon, The Case Against ISO 9000, Oak Tree Press, Dublin, 2000.
22. P. Schuh, Recovery, Redemption, and Extreme Programming, IEEE Software, vol. 18, No. 6, 34-41.
23. B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading MA, 1985.
24. R. Tricker, B. Sherring-Lucas, ISO 9001:2000 in Brief, Butterworth-Heinemann, Oxford, 2001.
25. L. Williams et al., Strengthening the case for pair programming, IEEE Software, vol. 17 (2000), No. 4, 19-25.
26. N. Wirth, Programming in Modula-2, Springer-Verlag, Heidelberg, 1982.