Maturing Extreme Programming Through the CMM

Jonas Martinsson

Master's Thesis in Computer Science
October 2002

CODEN: LUNDFD6/NFCS/NFCS-5248/1—35/2002

Department of Computer Science
Lund University
Lund, Sweden

Abstract

Extreme Programming (XP) and the Capability Maturity Model (CMM) are two recently emerging models addressing the difficulty of developing and delivering high-quality software products. The CMM introduces 5 levels of maturity and gives guidelines for *what* to do at an organizational level, while XP gives explicit guidelines for *how* to develop software at a project level. Together these models form a comprehensive framework for structuring the software development organization. Drawing from previous researches in software engineering as well as personal experiences of coaching XP teams this paper analyzes and gives practical suggestions for how to handle the combination of the two models. The paper shows that it is not only possible, but also advantageous to use XP as a springboard for reaching the highest CMM maturity level and assuring the long-term goals of the software development organization.

Table of contents

1 Introduction

The software industry has throughout its history been ridden with an exceptionally high rate of failed projects. The most common symptoms of the software crisis are overrun budgets and schedules, volatile cost and time estimates, final products crippled with bugs, and increasingly complex systems. To better cope with the rapidly changing environment of software development, different models have been tried with varying success. Two relatively recent developments in this field are Extreme Programming (2000) and the Capability Maturity Model (1991).

Extreme Programming (XP) is a lightweight methodology for small software development teams, introduced by Kent Beck [1]. In the last years a strong interest in the model has mushroomed within the computing industry. At the same time as it is based on sound software engineering values, it also evokes strong feelings due to its extreme use of some practices.

The Capability Maturity Model (CMM) runs in the same vein as XP in regard to aspirations, but its origins are a lot more formal. It was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in a response to a request to provide the government of USA with a method for assessing the quality of software contractors.

It is often said that the software development industry has an exceptionally wide gap between best practices and used practices. One reason for this can be the lack of knowledge about existing comprehensive and practical processes. This paper suggests XP as a foundation for building a mature software organization and improving upon it through modifications of the recommendations in the CMM.

Below you will first find a brief introduction to the general field of process control models. Next, the two software development models, Extreme Programming and the Capability Maturity Model, are defined and explained. Readers who are already familiar with the two models can skip sections 3 and 4. Sections 5 and 6 contain critique of how well XP fits in the CMM framework, as well as actual suggestions for how to certifiably improve the maturity of organizations practicing XP. Comparisons between XP and the CMM have been published before [2] [3] [4] [5] but not on the same level of detail as in this paper.

2 Defined vs. Empirical Process Control Models

In the industrial process theory field, it has long been understood that there are two major branches for controlling all processes: defined and empirical [6]. These two diverse approaches lend themselves to different types of product development.

The defined process control model requires that every segment of work towards the final product is completely understood. Given a well-defined set of inputs, identical outputs will be produced on every occasion. The empirical process control model, on the other hand, anticipates the unexpected. It is used for processes that do not have a well-defined set of inputs, and will not generate the same outputs every time.

In an interview [7] with the most highly respected experts in industrial process control, led by Babatunde Ogannaike, amazement regarding the widespread use of the defined process control model in the software development industry was voiced: "…systems development has so much complexity and unpredictability that it has to be managed by [an empirical] process control model…" Software development is typically a chaotic environment, with changing, or lack of, well-defined requirements, changing environments and workflows that cannot be arbitrarily started or stopped. Attempting to control a process like this with a defined control model will not be the right approach, according to industrial process theory.

Still, all of the major software development methodologies, such as Waterfall [8] or the Rational Unified Process [9] that have appeared over the years are approaching the problem with a defined process control model. These models put a large amount of effort in specifying the system before the implementation. Specification of the requirements and the design need to be completed in strict order before the process of actually building the system can start. The problem with this approach is that it will be very costly to change the requirements or implementation details late in the project.

During the year of 2001, the Agile Alliance was created, harboring a great number of well-renowned experts in the computing field. The organization put forward the Agile Manifesto, which among other things values responsiveness to change over following a plan [10] – in other words an empirical process control model. Maybe the most attractive of the agile methodologies the Agile Alliance are advocating is Extreme Programming. Without explicitly expressing an adherence to the empirical process control model, XP molds itself exceptionally well to it.

3 Extreme Programming

Extreme Programming is a lightweight methodology for managing rapid development of high-quality software with little overhead. It is defined by values and practices, which are examined in greater detail in sub-sections 3.2 and 3.3 below.

3.1 The Cost-of-Change Curve

The heart of XP lies in its assumption that the classic graph of the exponentially increasing cost-of-change curve can be flattened with the advent of today's modern development techniques and technologies.

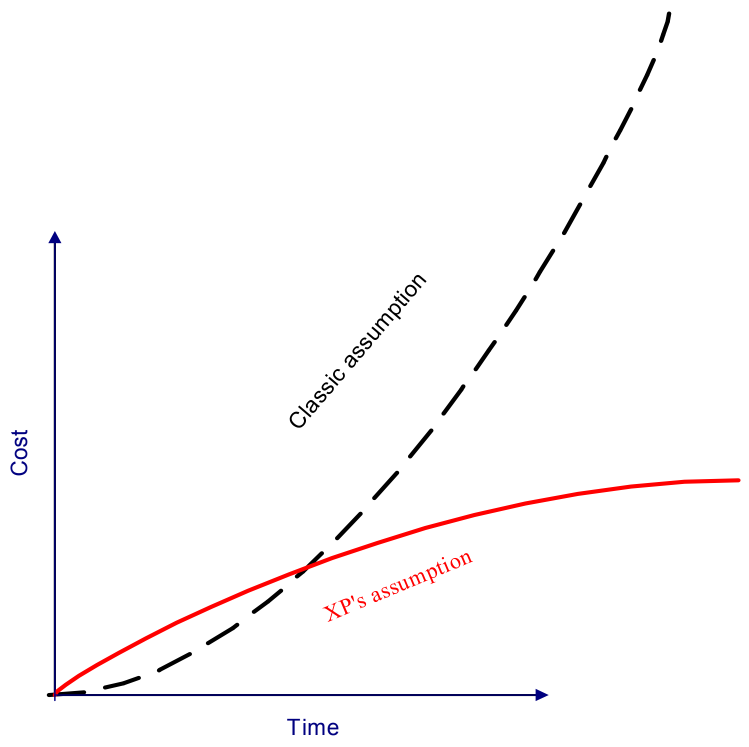

Figure 1 Flattening the cost-of-change curve

The cost-of-change diagram (Figure 1) depicts the increasing cost of changing the requirements or the design of a product over time. This has to do with the common use of a defined process control model, where all the design and requirements are done at the very start of the project. To make changes at a later time will require a more fundamental re-design. It may be argued that it has not been feasible for computer science to adopt an empirical process earlier in its history, due to the inherent non-modifiable nature of software development. It is not until recently that we have seen technologies such as object-oriented programming, relation databases, design patterns [11] and refactoring [12] emerge and become fully accepted in the computing industry. It is these very techniques that are the cornerstone for enabling the empirical process, which allows for effective and rapid changes to be made in software design throughout the lifetime of a project.

3.2 Values

One of the attractive features of XP is its recognition of the humans working in the project, and their well-being. XP names four values which always should be remembered as a guiding light in search for a good software development process. These values help XP team members from "reverting to their own short-term best interest" in benefit for "long-term social goals", according to Beck. The values are

- communication;
- simplicity;
- feedback;
- courage.

Project mishaps can usually be traced back to information not being *communicated*. The miscommunication can take many forms: programmers failing to convey design changes to co-workers, customers not being asked the right questions about domain decisions, etc. XP strives to keep the information flowing by using practices that will force communication to happen: unit testing, pair programming and task estimation. There is also a coach, who will make sure that the practices are followed.

Simplicity is all about believing in the flattened cost-of-change curve. Keeping a software design simple takes a lot of effort. As a programmer, it is very easy to go for elegant generic solutions to problems that might be re-used later on. The problem with this approach is that as the code base grows, so does the complexity of the design. It is also not certain that the generic approach will be needed later. Because we believe in a flattened cost-of-change curve, it will usually not cost us much more effort to implement the more generic features later, if and when required.

Concrete, unbiased *feedback* is a very valuable asset in a software project and Extreme Programming offers these metrics through unit testing, task estimation and acceptance tests. XP also offers the customer a snapshot of the product in between the short iterations of the project. From here it is possible to steer the project in new directions by adding, changing and removing requirements.

The last of the values is *courage*. With the first three values in place, functioning as a safety net, it is possible for the programmer to courageously make decisions and try different approaches for the implementation. Simplicity supports courage, because in a simple system it is much easier to be bold. It works the other way around too; courage enables simplification of the system.

3.3 Core Practices

XP recognizes 12 successful practices in software development that it takes to extreme levels; if a practice is good it should be addressed all the time. The 12 practices are:

- the planning game;
- small releases;
- metaphor;
- simple design;
- testing;
- refactoring;
- pair programming;
- collective ownership;
- continuous integration;
- sustainable pace;
- on-site customer;
- coding standards.

3.3.1 The Planning Game

The planning game is perhaps the most innovative of the XP practices, in comparison with established software practices. As a basis for the planning game XP introduces four control variables to manage a project; these are:

- cost;
- quality;
- time;
- scope.

Cost represents the total amount of money the company is willing to spend on resources for the project. This can result in a different number of allocated programmers, or new and more efficient development tools. It can also be reflected as an enabling or disabling of overtime. It is important to note that there is often not an immediate gain in project success by investing more money in a project, while reducing a project's budget will often lead to immediate deterioration. A discussion of the intriguing cogs that govern these principles lies outside the scope of this paper, but can be further examined in *The Mythical Man-Month* [13].

Quality is something that is pretty difficult to realistically concretize in terms of project goals. It is maybe for this reason that quality often will be the variable that will suffer when a project manager or customer tries to fix the other three. Quality goals can be set as number of bugs in the code or application, but is inheritably difficult to measure and verify. Therefore quality can often be visualized in the finished project as a result of how successfully and realistically the other variables have been managed.

Time dictates the various deadlines for the project and how much time is allocated for each release.

Scope decides which features should be implemented in the project and which ones that should be left out.

These variables will decide what the final product will look like. It is important to understand that the variables interact, similar to U-tubes. Changing one of the variables will lead to a change in one or more of the others. In a less mature company it is not uncommon for project managers to try to fix these variables as constants, but this will most likely lead to serious problems.

The requirements for the project are translated into *user stories*. The user stories are usually described as small chunks of functionality, of value to the customer, defined in only a few sentences. The reason for not specifying the requirements more rigidly is to follow the XP value of *simplicity*. Instead it is said that a user story is a promise for further conversation between the customer, dictating the story, and the programmers, implementing it. The stories are estimated in terms of time they will take to implement, test and verify, and finally prioritized and scheduled into the project plan by the customer. For simplicity's sake the user stories are kept on index cards, for easy accessibility. The size of the user stories should be kept so that a few stories will fit into each *iteration*.

An iteration is a period of 1 to 4 weeks, which begins with an *iteration planning meeting*. During this meeting the customer selects the highest prioritized stories to be implemented, and has also the right to make changes to the remaining user stories. The programmers break down the selected user stories into *engineering tasks*, which are best described as units of the system that are essential for it to work as desired. Most engineering tasks are derived from user stories, although this is not mandatory. The programmers estimate these tasks and in the cases where it is not possible to give an accurate estimate, a prototype will be developed to give insight into the technical problem. The programmers finally sign up for the estimated tasks during the iteration planning meeting.

Instead of actual calendar time or man-hours an abstract unit is used for estimating the workload of a user story or an engineering task. At the end of each iteration the number of average completed units per iteration will be used to update the project plan. This metric is called velocity and will give the customer and management a good indication of when the project will be finalized. In many projects abstract metrics will yield better estimated than more traditional ones [14].

The practices of the planning game enable XP projects to be continuously changed, which is exactly what the agile software development aspirations are about.

3.3.2 Small Releases

At the end of each iteration a functional and running version of the software system is running and it will be possible for the customer to schedule releases between each iteration. This is a tremendous benefit for the customer, who will be able to follow the progress of the software product and steer it in the desired direction. It is easy to see the advantage of this system compared with a more traditional approach with a delivery months later without any continuous feedback to and from the customer.

3.3.3 Metaphor

The metaphor in XP is used to describe the system being built in a way that is not too technical for the non-programmers, but at the same time might give the programmers a new and more insightful way of looking at the system. It can also be used to facilitate naming conventions for classes, objects and variables, and even suggest problems and solutions that might not have been anticipated otherwise. In many ways the metaphor is a very abstract and difficult notion in the Extreme Programming methodology, and its advantages can be questioned. Many XP teams are using the “na¬ve metaphor” (i.e. the actual software product) [15], which yields no advantages at all over not using a metaphor.

3.3.4 Simple Design

Simple design is in much a direct translation of the XP value s*implicity* into a software practice. It calls for the programmers to make design decisions for the current problem, not a more generic solution that might look more elegant and perhaps be used in a future solution. It is a common error among programmers to opt for elegant, generic solutions, which is not the most efficient way of solving the problem at hand.

XP is making a bet that more time will be gained in the long run by using a simple approach today. When and if a situation arrives which will require a more generic solution it will be implemented at that time. There are two quotes that are repeated *in absurdum* by XP advocates to explain simple design and perhaps they are the clearest explanation of the practice:

- “Do the simplest thing that could possibly work”;
- “You ain’t gonna need it”.

With experience, discipline and an understanding of the fundamentals behind this practice it is possible to break away from the rule of simple design. This holds true for most of the 12 core practices.

3.3.5 Testing

All production code written in the project is accompanied with *unit tests*, which are being run continuously throughout the development cycle. Unit tests are code that is being executed to test the functionality of the production code. These tests are completely automated and initiated with a start of a button or maybe even automatically on the integration system when new versions of source code are being checked in. While the idea of unit tests are not new to software development, the XP idea of test-driven design and test-first programming is something of a revolution. The practice is to write the tests for a method before the actual implementation, which aims at conveying the functional intent of the method and thereby also producing more solid class interfaces with a clearer knowledge of the way a caller will use the method. Before source code is checked in to the common repository all unit tests must first run without a single failure. When a bug is found in the system it means that somewhere someone forgot to write a test, and the correct procedure to stamp out the bug is to first create a test which will reproduce the bug and then make the test suite run again at 100%. Areas of the system where similar bugs may reside are identified and tested.

As a complement to unit tests there are *acceptance tests*, which are specified by the customer and will help him as well as the programmers to verify that a user story has really been implemented completely. The acceptance tests should preferably be automated as well, although this requirement is not as strict as in the case with unit tests.

3.3.6 Refactoring

Martin Fowler's book *Refactoring,* with the subtitle *Improving the Design of Existing Code*, catalogues commonsense programming patterns (similar to, but at a lower level than the more famous *Design Patterns* by Gamma et al [11]) for producing well-written object oriented source code. There is also a web site dedicated to the refactorings in the book and more recent discoveries [16]. Refactorings are fundamental in XP due to the fact that design and re-design of the code is happening all the time. Without adherence to well-written source code the baseline would soon become unmaintainable, but with continuous refactorings it is possible to change the design at any point during the project without impairing the readability of the source code. There is also a strong tie between refactorings and unit testing; the tests will give the programmers courage to make refactorings because if functionality is altered by a design change the unit tests will immediately tell them so.

3.3.7 Pair Programming

Perhaps the most eye-catching practice of the 12 is *pair programming*. It states that all production code in the project is written by groups of two people, using one computer and one keyboard. One of them, the *driver*, is typing, while the other one, the *partner*, is performing constant code review and focusing more strategically about the design. While many student assignments at universities are written in pairs it is a very uncommon sight in the professional arena. Research has given that pair programming yields more solid code with fewer bugs in about 20% more man-hours (thus, in less calendar time). [17] [18] A typical XP pair programming session might last for about 2 hours with frequent role changes between the driver and the partner.

3.3.8 Collective Ownership

In an XP project there is no programmer that is more or less responsible for a certain piece of source code than someone else. As soon as a programmer locates a defect in the code he should not wait until the programmer who produced the defect will become available to fix it, but instead tackle the problem himself (as usual by writing a unit test to recreate the defect and fix it together with a pair programming partner). The benefit of this particular practice is besides timesaving that knowledge about the system will spread within the team, and that no single programmer will be indispensable for understanding a certain domain of the source code. The practices of pair programming and unit testing will prevent collective ownership from equaling chaotic hacking where anyone could impair the source code on a bad day.

3.3.9 Continuous Integration

By integrating the entire system multiple times every day there will not be the "integration hell" so often experienced in more traditional software projects, where the code freeze period will get prolonged as the integration issues mount.

3.3.10 Sustainable Pace

Sustainable Pace, or 40-hour week, as this practice was originally named[1] is there to prevent burnout and keep the programmers happy. As the original name suggests, the practice typically calls for 40-hour workweeks, where there is no overtime allowed. In exceptional cases XP suggests that overtime will be allowed, but never for two consecutive weeks.

3.3.11 On-Site Customer

A customer should be present, or at least available to the programmers all the time. Because user stories are nothing but promises for further conversation this requires the programmers to have the customer at hand to ask about details in the user story implementations that were not resolved or identified at the iteration planning meeting. Beck is saying that verbal communication face to face is much less likely to produce misunderstandings than a requirements definition in a typical traditional software project.

Of course it might not always be easy to persuade a customer to be constantly present in a software project, and in these cases a solution needs to be found. Usually, in situations like this, it is common for the customer or project manager to appoint a proxy for the customer who will be able to make decisions and answer the programmers' questions.

3.3.12 Coding Standards

Coding standards will facilitate the programmers' work and cause less grievances between them concerning the collective source code ownership. These coding standards should include everything from a naming standard for classes, methods, variables, files, etc, to where the curly braces should be positioned. XP does not recommend any specific coding standard, but says that one should exist.

[1] The name change took place on the Yahoo! Groups Extreme Programming mailing list [19] during the year 2001.

4 The Capability Maturity Model

CMM, or the Capability Maturity Model, is a set of guidelines for how to improve and measure the capability of software organizations. The CMM has been developed and improved by hundreds of leading computing experts, under supervision and direction from SEI. The first version of what would later become the CMM appeared in 1987, when the SEI released a brief description of the software process maturity framework. Four years later the first actual version of the CMM was released. With feedback from a multitude of different areas in the computing world a revised and upgraded version was released in 1993 [20]. The CMM does not give rules or *explicit* guidelines for how to conduct software development but rather advises how to organize a software organization so that it can improve and better predict its final results.

Basically, the CMM consists of 5 maturity levels, where companies will fit in according to their maturity, and 18 key process areas (KPAs) consisting of 52 goals, which will need to be addressed for organizations wanting to ascend to higher maturity levels. The levels are outlined below.

4.1 Maturity Levels

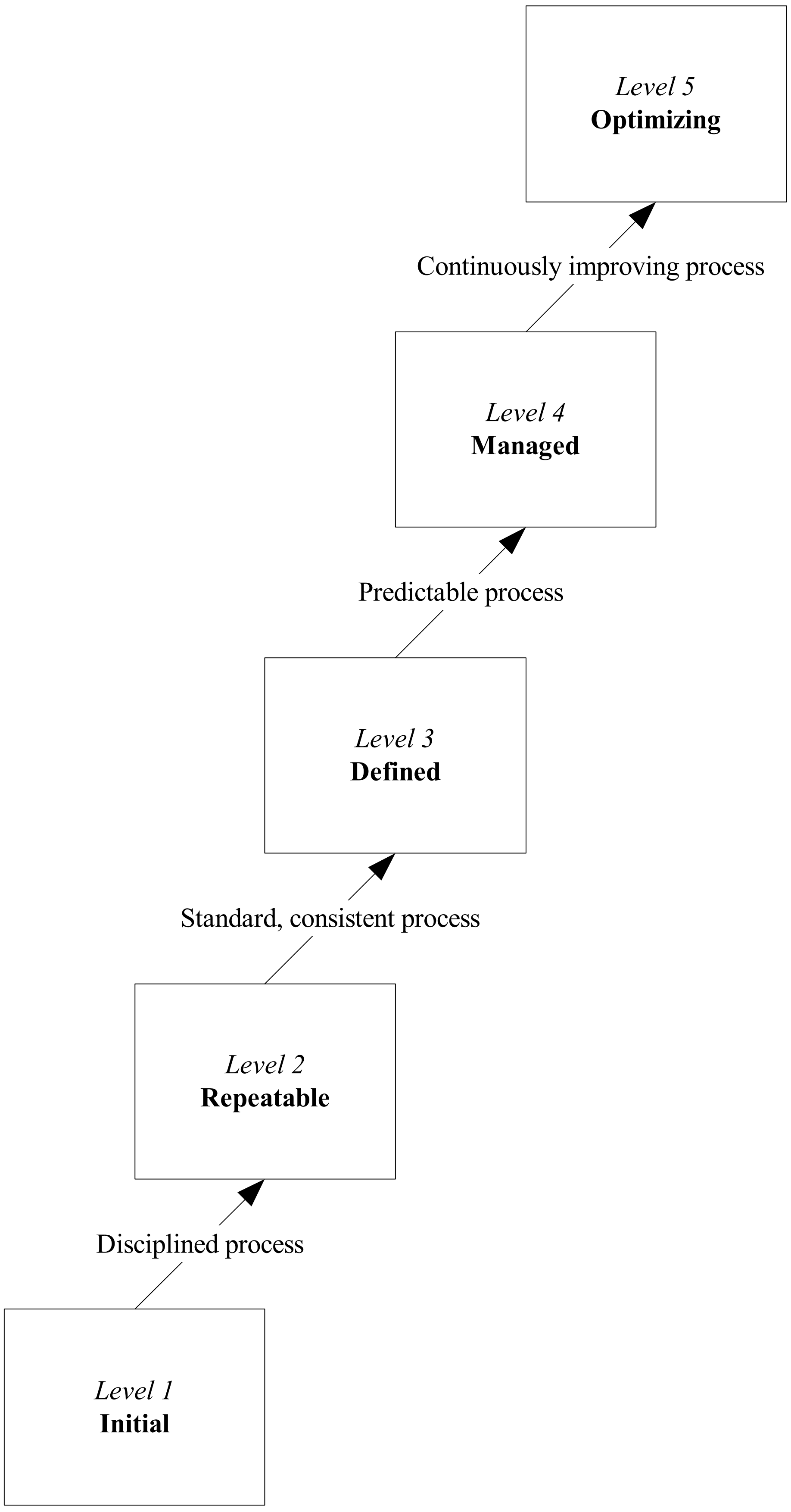

Figure 2 The five levels of software process maturity according to the CMM

Figure 2 shows an overview of how the CMM levels are defined. A more detailed explanation of what separates the levels is given below. Because each maturity level in the CMM forms a foundation on which to build the next level, trying to skip levels as an organization matures is usually counterproductive.

4.1.1 Level 1: Initial

It is said that about two thirds of all software companies in the world would belong to the *initial* level if they were to be classified [21]. Level 1 is characterized by an organization that relies on individual heroics from developers and managers for their success, and over-commitment is common. It is also not possible for the organization to repeat earlier successes unless the same heroes are involved in the next project as well. The company may have planned procedures but are abandoning them during crisis. In this *ad hoc*-environment companies still develop products that work, but it is very common with overrun budgets and schedules.

4.1.2 Level 2: Repeatable

At the *repeatable* level there are policies for how to manage the software projects, as well as established procedures to follow these policies. The planning and managing of new projects are based upon the experiences from earlier similar projects. Projects are utilizing efficient processes that are:

- defined;
- documented;
- practiced;
- trained;
- measured;
- enforced;
- improvable.

Realistic project estimations and commitments are made, based on results from previous projects and the specific requirements of the current project. A software manager monitors the estimates for time, cost and scope continuously and will identify problems as they arise. As multiple projects with common functionality are developed a baseline with controlled integrity is also maintained. Standards for software projects are defined and the organization verifies that they are followed. Processes between projects do not need to be identical in level 2 organizations, but an organization-level policy for establishing an appropriate management process per project must exist.

4.1.3 Level 3: Defined

At the *defined* level a standard process for software development and maintenance, involving both engineering and management areas, is documented and used throughout the organization. Projects then customize the standard process to fit the unique characteristics of their particular requirements. A well-defined process can be identified by its “readiness criteria, inputs, standards and procedures for performing the work, verification mechanisms (such as peer reviews), outputs, and completion criteria.” [22] While the CMM calls this level defined it should not be confused with the defined process control model discussed in section 2.

A group within the organization is responsible for software process activities and an organization-wide training program is used for ensuring that the staff has the needed skills for carrying out their responsibilities.

4.1.4 Level 4: Managed

At the *managed* level the organization sets up a database for analyzing metrics measuring the quality of both the software products and the processes for all projects. Projects control the quality of their products and processes by narrowing the variation in their performance to fall between acceptable limits. When an exceptional situation occurs and the pre-defined limits are exceeded steps are taken to identify, address and correct the problem. The risks of entering a new application domain will be recognized and carefully managed. The products produced by a level 4-company are of predictably high quality.

4.1.5 Level 5: Optimizing

Just a handful of software companies around the world have reached the 5th and final CMM level: *optimizing*, where for example NASA and Motorola belong. In this level the whole organization is focused on continuous process improvement, and has the ability to identify weaknesses and correct them proactively. The quality measurements of the process are used to evaluate proposed changes to the organization's software process. Innovations in the computing field that will improve the process are identified and implemented in the whole company. Defects in the process are analyzed to pinpoint their causes and lessons from this are communicated throughout the organization.

4.2 Key process areas

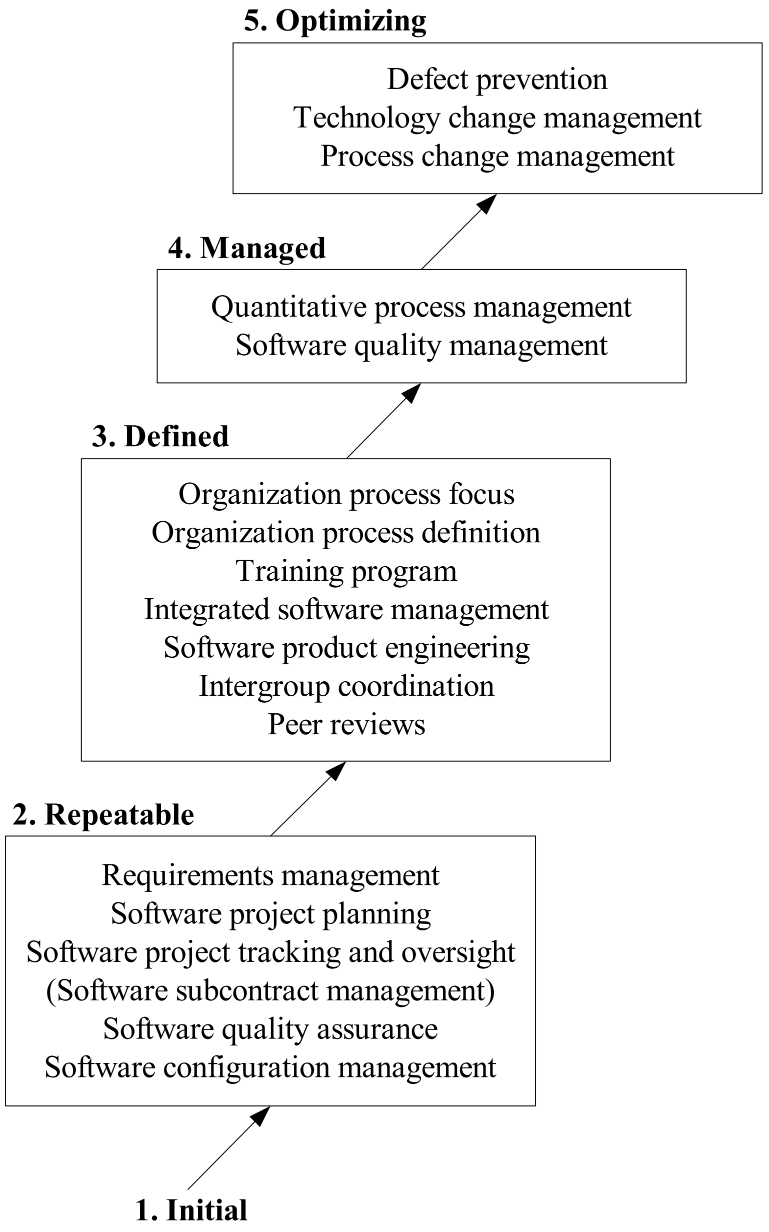
5. Optimizing
Defect prevention
Technology change management
Process change management
4. Managed
Quantitative process management
Software quality management
3. Defined
Organization process focus
Organization process definition
Training program
Integrated software management
Software product engineering
Intergroup coordination
Peer reviews
2. Repeatable
Requirements management
Software project planning
Software project tracking and oversight
(Software subcontract management)
Software quality assurance
Software configuration management
1. Initial

Figure 3 The key process areas according to maturity level

With the exception of level one, each of the CMM levels is connected with several key process areas (KPAs), where organizations should focus their attention for improving their process (Figure 3). If a company aims at reaching a certain maturity level it must address all KPAs at that level and below. Each of the KPAs has goals that need to be addressed for the KPA to be satisfied.

Definitions of the KPAs and their respective goals a presented in sections 5.1 to 5.18 below and not here, to facilitate referencing when analyzing the CMM with XP. The presentation of the CMM is not meant to be inclusive, but more of an overview for the purpose of successfully being able to compare and analyze it side-by-side with XP. A fuller description is given in SEI's comprehensive book on the CMM [22].

5 CMM Maturity of the XP Process

The most glaring limitation of XP, seen from a software organizational perspective, is its design for relatively small development teams, and that it does not actively address management responsibilities. It is often said that XP is designed *by* developers, *for* developers. Kent Beck suggests in his XP manifesto [1] that teams with ten programmers or less are clearly feasible, but twenty will probably be too many for the process to run successfully out of the box. Similar open issues arise when it comes to synchronizing and managing multiple XP projects.

Recent developments and research have tried to remedy these problems, from where for example *Scrum* [7] has emerged, which is probably the model most similar to XP's values and goals. While Scrum is receiving a lot of attention among XP advocates it is still a very marginal development in the computing universe overall. The same cannot be said about the CMM however, which is in widespread use in many highly developed software organizations, especially in USA and Asia. The origin of the CMM has a much larger base of credibility than Scrum's, considering the number of years invested and the amount of computing experts involved in the development of the respective models.

The CMM and XP are complementing each other in the respect that the CMM is saying *what* to do in general terms, but not *how* to make it possible, while XP is an *implementation model* with explicit practical guidelines for a software organization at the project level. These reasons make the CMM a prime candidate for cooperation with XP in many computing organizations.

It is important to understand that the CMM is not mandating how to achieve the goals of the KPAs, although it is suggesting possible routes to reach each goal. Alternative practices may accomplish the goals of a key process area. Since many of the suggested practices for reaching the goals are focusing on heavy documentation (while XP is not) these suggestions have been left out from this paper. It is important not to confuse the suggested practices with the goals, as is often the case when people are criticizing the CMM in various contexts.

In the subsections below, the 18 KPAs and their goals, which are quoted verbatim, will be analyzed in regard to how well they agree with the XP process model.

5.1 Requirements management

The purpose is to establish an understanding and agreement between the customer and the development team regarding the customer's requirements for the project.

The goals are:

1. System requirements allocated to software are controlled to establish a baseline for software engineering and management use.
2. Software plans, products and activities are kept consistent with the system requirements allocated to software.

Requirements management is primarily addressed during the iteration planning meetings where customer and the rest of the XP team discuss the requirements. The critical steps in the iteration planning meeting are [23]:

- Presentation of user stories from the customer.
- Brainstorming of the user stories and engineering tasks by the team.
- Estimation of and signing up for engineering tasks.

The specific XP practices that are dealing with the issues of requirements management are the user stories, on-site customer, and continuous integration. It is interesting to note that XP in a sense goes even further than the CMM goal and aims for delivering what the customer wants, and not only his requirements. In other words, XP acknowledges that in a software project it is often the case that a customer is not always successful in translating his wants and needs into a formal requirements definition. The on-site customer in XP facilitates a dialog between the customer and the rest of the team concerning domain-specific details that arise as the programmers are delving deeper into the project implementation. At the end of an iteration, or during the iteration planning meeting, the team's demonstration of the current state of the project, is also assisting the customer into further specifying and communicating future requirements.

5.2 Software project planning

The purpose is to establish sensible plans for software engineering and for managing the software project.

The goals are:

1. Software estimates are documented for use in planning and tracking the software project;
2. Software project activities and commitments are planned and documented.
3. Affected groups and individuals agree to their commitments related to the software project.

Software project planning is again primarily addressed during the iteration planning meeting, where estimates are written down on index cards. During the meeting the activities and commitments are well defined by the XP team, in the form of user stories and engineering tasks. The overall plan for the whole project is addressed by the XP practice of small releases in combination with building a release plan based on the user stories together with the customer. The short-cycle planning strategy in XP is close to the heart of Watts Humphrey's, the father of the CMM, proverb: "If you can't plan well, plan often." [2]. The last goal of this KPA, the agreement to commitments, are evidently taken care of through the practice of the programmers themselves signing up for the tasks to be implemented during the upcoming iteration.

5.3 Software project tracking and oversight

The purpose is to provide adequate visibility into the actual progress so that management can take efficient actions when the project's performance deviates significantly from the project plan.

The goals are:

1. Actual results and performance are tracked against the software plans.
2. Corrective actions are taken and managed to closure when actual results and performance deviate significantly from the software plans.
3. Changes to software commitments are agreed to by the affected groups and individuals.

Software project tracking and oversight is partially addressed by XP's planning game practice. XP does not state any practice based upon documentation, but it is important to make clear that it is not frowned upon either. In Fowler's and Beck's work "Planning Extreme Programming" [24] the authors point out the danger of software project metrics becoming "the end instead of the means". They suggest that metrics on a more detailed level than project planning should only be used in problematic areas of the project.

At the end of each iteration (or more often if required) the tracker will collect metrics for the project performance during the iteration. It is very common for XP projects to use a spreadsheet application or simpler tools like pen and paper for tracking project-planning metrics such as estimates and actual achievements as well as the overall project plan. This habit will clearly satisfy the first goal of the KPA.

It is the coach's job to make sure that the programmers are working efficiently and effectively, and in instances where a programmer is not on par with his abilities or estimates it is important for the coach to find a solution to this as soon as possible. Although there are no formal rules for how to achieve this in XP (which is not very strange since the analysis of the reasons behind getting behind the estimates often belong more in the field of social science than computer science), it is something that is given much focus in the XP literature. By additionally defining a clear communication path of the metrics to management goal number two should be satisfied.

Changes to the software commitments, or user stories, during an iteration are handled through communications between the developers and customer, with assistance from the coach. If for example a developer realizes that an estimate is wrong, it is possible to break up the user story into two or postpone the user story to the next iteration.

5.4 Software subcontract management

The purpose is to select qualified software subcontractors and manage them effectively.

The goals are:

1. The prime contractor selects qualified software subcontractors.
2. The prime contractor and the software subcontractor agree to their commitments to each other.
3. The prime contractor and the software subcontractor maintain ongoing communications.

Software subcontract management is a unique KPA in that it is only relevant for computing organizations outsourcing part of their development, and is the only KPA that is not required for achieving a CMM rating. There is nothing in XP that mentions software subcontracting, and nothing in particular that prevents it. It is likely that subcontracting management will not be applicable in the target organization, and for this reason it is omitted from this paper.

5.5 Software quality assurance

The purpose is to provide management with appropriate visibility into the process being used by the project and of the product being built.

The goals are:

1. Software quality assurance activities are planned.
2. Adherence of software products and activities to the applicable standards, procedures, and requirements is verified objectively.
3. Affected groups and individuals are informed of software quality assurance activities and results.
4. Noncompliance issues that cannot be resolved within the software project are addressed by senior management.

Software quality assurance in the form of planned quality assurance activities are clearly satisfied by XP through the practices pair programming, which is much like running constant code reviews, and test-first programming. Usually there is not much gain by having dedicated quality assurance personnel in an XP project since quality is so central in the regular programming sessions.

The problem for XP is to accomplish the objective verifications of adherence to a project's standards, procedures, and requirements. XP assures the quality on a social level through peer pressure, which often is very successful, but may crumble under external pressure. This is an issue that must be addressed explicitly by an organization that wants to use XP and at the same time be rated as CMM level 2 or higher.

The presentation of results for quality assurance activities are commonly lived up to by XP projects by posting a graph of the percentage of test failures for each build. These should typically stay at a constant number of zero percent. Similar presentations could be made for additional metrics that may need to exist in order to satisfy the second goal of the KPA.

The final goal can easily be achieved in most projects and is neither favored nor obstructed by the XP process.

5.6 Software configuration management

The purpose is to establish and maintain the integrity of the products throughout the project's life cycle.

The goals are:

1. Software configuration management activities are planned.
2. Selected software work products are identified, controlled, and available.
3. Changes to identified software work products are controlled.
4. Affected groups and individuals are informed of the status and content of software baselines.

Software configuration management is implicitly satisfied by XP projects through the practices continuous integration, collective ownership, refactoring, and small releases. It is clear that configuration management activities are planned in XP because it states that "daily builds are for whimps", and opts for very frequent check-ins of the source code to the repository. Furthermore, refactorings are pushing the source code in the direction of a larger baseline, with more classes and code in common.

The CMM suggests establishing a board and a group with the authority for managing the software's baselines. This will not be necessary in an XP environment, because the responsibility is jointly laid on all the developers. The remaining goals, to control software work products and communicating their status, are also implicitly supported by XP, which goes a long way when it comes to maintain mature software configuration practices.

5.7 Organization process focus

The purpose is to establish the organizational responsibility for process activities that improve the overall process capability.

The goals are:

1. Software process development and improvement activities are coordinated across the organization.
2. The strengths and weaknesses of the software processes used are identified relative to a process standard.
3. Organization-level process development and improvement activities are planned.

Organization process focus is loosely addressed by XP on the team level through various suggestions of how to introduce the model, but it does not touch on the organizational level at all. However, XP teams, and in particular multiple XP projects that are to be synchronized within an organization, would benefit tremendously from a process-conscious management with rules and guidelines for how to manage, analyze, and actively improve the software development process. The realization of this KPA, in combination with many of the other KPAs of the third level, is perhaps the key to running XP projects successfully in a larger environment.

5.8 Organization process definition

The purpose is to develop and maintain a usable set of process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization.

The goals are:

1. A standard software process for the organization is developed and maintained.
2. Information related to the use of the organization's standard software process by the software projects is collected, reviewed, and made available.

Organization process definition is provided by the XP literature and through various Internet resources. The KPA's two goals is perhaps most easily satisfied by buying copies of the XP books, preferably Extreme Programming Explained [1], Extreme Programming Installed [23], and Planning Extreme Programming [24], reviewing them and making them available within the organization. For companies focusing on multiple software development methodologies a wider and more organizational-conscious approach will need to be taken.

5.9 Training program

The purpose is to develop the skills and knowledge of individuals so they can perform their roles effectively and efficiently.

The goals are:

1. Training activities are planned.
2. Training for developing the skills and knowledge needed to perform software management and technical roles is provided.
3. Individuals in the software engineering group and software-related groups receive the training necessary to perform their roles.

Training program is not directly implemented by XP, but all of its goals are indirectly and partially achieved by the practice of pair programming, which is a fertile ground for giving new co-workers and team-members hands-on experience in learning the skills and intricacies required in the actual project. It is also a valuable tool for training junior programmers good and proven habits from the more experienced staff. The software engineering group, which is referred to in the 3rd goal is defined by the CMM as the "collective of individuals…who have responsibility for performing the software development and maintenance activities (i.e., requirements analysis, design, code, and test)". These activities are all covered within the pair programming practice, except for requirements analysis, which is not explicitly addressed by XP. The KPA of training program is given more or less constant attention within an XP team.

While pair programming is giving the programmers a tool for sharing and spreading knowledge within the team, it can also be used for spreading knowledge about new technologies. In conjunction with the implementation of KPA 17, Technology change management, it will be possible to phase in knowledge about new technologies through the pair programming practice.

5.10 Integrated software management

The purpose is to integrate the software engineering and management activities into a coherent, defined process that is tailored from the organization's standard software process and related process assets.

The goals are:

1. The project's defined software process is a tailored version of the organization's standard software process.
2. The project is planned and managed according to the project's defined software process.

Integrated software management is not addressed by XP. Because XP is a flexible process it should adapt well to customizations, which are very common in real-life XP projects.

5.11 Software product engineering

The purpose is to consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent products effectively and efficiently.

The goals are:

1. The software engineering tasks are defined, integrated, and consistently performed to produce the software.
2. Software work products are kept consistent with each other.

Software product engineering is satisfied by many XP core practices, and is more or less the heart and soul of XP: to efficiently develop products using sound software engineering principles. Essentially, each of the twelve core practices strives towards reaching the two goals of this KPA.

However, for some specific types of projects, XP's core practices may not be enough. In particular, software systems that have high demands for critical real-time performance are often said to require big design up-front to minimize risks, and this is directly contrary to XP. Also, the systems requirement for ensuring stability and functionality of such a project may not allow for merciless refactorings and design changes once the analysis has been done. This will shatter XP's assumption of a flattened cost-of-change curve and therefore render it impractical for the software problem at hand. In a more traditional software organization though, it is beyond a shadow of a doubt that XP fulfills the aims of this KPA.

5.12 Intergroup coordination

The purpose is to establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently.

The goals are:

1. The customer's requirements are agreed to by all affected groups.
2. The commitments between the engineering groups are agreed to by all the affected groups.
3. The engineering groups identify, track, and resolve intergroup issues.

Intergroup coordination is well expressed in XP's value of communication and manifests itself in the core practices of the planning game, metaphor, pair programming, collective ownership, and on-site customer. The common XP practice of having a *daily stand-up meeting* will help the intergroup coordination. In this meeting the team takes five minutes in the morning to discuss issues and plan for the upcoming tasks.

Goals number one and two are specifically handled during the iteration planning meeting, where programmers themselves sign up for and discuss the engineering tasks. Goal number three is not actually solved by any specific XP core practice, but it is the coach's clearly stated responsibility to identify and resolve issues that may arise within the team. The assigned tracker role of each XP team could be augmented into tracking intergroup issues, which would further help to satisfy the third goal completely.

5.13 Peer reviews

The purpose is to remove defects from the work products early and efficiently. An important corollary effect is to develop a better understanding of the work products and of defects that might be prevented in the future.

The goals are:

1. Peer review activities are planned.
2. Defects in the software work products are identified and removed.

Peer reviews are fully satisfied through pair programming and testing. It is important to note that pair programming does not equal peer reviews, in that it does not offer the same structural framework. However, the added benefit of "peer coding" should amply outweigh the lack of a framework. The core practice of testing, which among other things mandates the incorporation of identifying tests for all defects found, will ensure that the KPA's goal number two is met.

5.14 Quantitative process management

The purpose is to control the process performance of the project quantitatively, where the measurements represent the actual results achieved from following a software process.

The goals are:

1. The quantitative process management activities are planned.
2. The process performance of the project's defined software process is controlled quantitatively.
3. The process capability of the organization's standard software process is known in quantitative terms.

Quantitative process management is a KPA that relies on a defined software process [see section 5.10] to exist and XP does not address this. Due to the requirement of having KPAs of underlying CMM levels satisfied for being able to reach the next level, it is assumed here that a defined software process do exist for the project.

The heart of the matter as of to which degree XP satisfies this KPA, is what the CMM mandates the quantitative control to consist of. The CMM states that quantitative control "implies any quantitative or statistically based technique appropriate to analyze a software process, identify special causes of variations in the performance of the software process, and bring the performance of the software process within well-defined limits." [22] A metric like this exists by default for any XP team in the form of user story and engineering task estimates, in conjunction with how many units of work the team and each individual programmer have been able to deliver during each iteration.

These metrics are collected and analyzed by the XP team's tracker at least once during every iteration, which is enough to satisfy goal number one.

Goal number two is partially satisfied in the planning game by a practice often referred to as "yesterday's weather", which implies that an XP team should schedule exactly as much work (or as many "units") as the team's average velocity, no more and no less, given that external factors such as manning and critical resources are approximately the same. The velocity will also serve as the foundation for updating the project plan to be realistic and adhere to the historical performance of the team. Because XP does not deal with management decisions it does not state how management should react to this, but it would be safe to assume that resources could be reallocated, or other actions taken, as a response to changes in these metrics. Defining a clear communication path of the metrics to management would be the necessary last step in satisfying goal number two.

The third goal is not addressed by XP and will need to be implemented. How this should be implemented will to a large extent depend on what the defined software process consists of.

5.15 Software quality management

The purpose is to develop a quantitative understanding of the quality of the project's products and achieve specific quality goals.

The goals are:

1. The project's software quality management activities are planned.
2. Measurable goals for software product quality and their priorities are defined.
3. Actual progress toward achieving the quality goals for the software products is quantified and managed.

Software quality management is not addressed by XP and its goals will need to be satisfied for an XP organization to reach level 4.

5.16 Defect prevention

The purpose is to identify the cause of defects and prevent them from recurring.

The goals are:

1. Defect prevention activities are planned.
2. Common causes of defects are sought out and identified.
3. Common causes of defects are prioritized and systematically eliminated.

Defect prevention is addressed through the core practices of testing, continuous integration, and pair programming. Specifically, goal number one is satisfied by the principles of test-first design and writing acceptance tests that verifies that the software is fully functional and without defects.

If a defect is spotted, usually during pair programming or integration, a test case will be written to reproduce the defect. By trying to find common denominators in the code similar bugs might be identified. This satisfies goal number two, and part of goal number three.

The third goal, when it comes to prioritizing defects is not addressed by XP, because it states that there should never exist any known defects in the code at any time. When a defect is identified it always has the highest priority over anything else, and code with known defects are never checked in to the common repository. Since XP is focusing very hard on stable source code and baselines, it should not be an issue in an XP team to prioritize defects, and therefore the first part of this goal should be deemed irrelevant in this context.

5.17 Technology change management

The purpose is to identify new technologies (tools, methods, processes, etc.) and transition them into the organization in an orderly manner.

The goals are:

1. Incorporation of technology changes is planned.
2. New technologies are evaluated to determine their effect on quality and productivity.
3. Appropriate new technologies are transferred into normal practice across the organization.

Technology change management is not addressed by XP and its goals will need to be satisfied for an XP organization to reach level 5.

5.18 ***Process change management***

The purpose is to continually improve the process used in the organization with the intent of improving software quality, increasing productivity and decreasing the cycle time for product development.

The goals are:

1. Continuous process improvement is planned.
2. Participation in the organization's software process improvement activities is organization-wide.
3. The organization's standard software process and the projects' defined software process are improved continuously.

Process change management is not addressed by XP and its goals will need to be satisfied for an XP organization to reach level 5.

5.19 Table of Summary

Level	Key Process Area	Goal 1	Goal 2	Goal 3	Goal 4
2	Requirements management (RM)	√	√		
	Software project planning (SPP)	√	√	√	
	Software project tracking and oversight (SPTO)	√	≈	√	
	Software subcontract management (SSM)	N/A	N/A	N/A	N/A
	Software quality assurance (SQA)	√	×	√	×
	Software configuration management (SCM)	√	≈	≈	≈
3	Organization process focus (OPF)	≈	×	×	
	Organization process definition (OPD)	√	√		
	Training program (TP)	≈	≈	≈	
	Integrated software management (ISM)	×	×		
	Software product engineering (SPE)	√	√		
	Intergroup coordination (IC)	√	√	√	
	Peer reviews (PR)	√	√		
4	Quantitative process management (QPM)	√	≈	×	
	Software quality management (SQM)	×	×	×	
5	Defect prevention (DP)	√	√	≈	
	Technology change management (TCM)	×	×	×	
	Process change management (PCM)	×	×	×	

Table 1 Degree of satisfaction for the 52 CMM goals by implementing an XP process.

√	= XP core practices and roles fully satisfy the goal.
≈	= XP core practices and roles partially or implicitly satisfy the goal.
×	= XP core practices and roles do not satisfy the goal.
N/A	= Not applicable.

6 Achieving CMM Maturity Using XP

As seen in the previous section an XP project is not even reaching a maturity of the CMM level 2 by default. However, implementing many of the suggested practices from the XP literature that are not core XP practices *per se*, will satisfy the goals that would normally only be reached partially or implicitly (marked with ≈ in Table 1). Guidelines for how to achieve this are found in the previous section in this paper. Only the CMM goals that have not been met at all by the implementation of an XP process (marked with ×) will be examined in sections 6.1 to 6.4 below, in an effort to aid XP advocates to mature organizations' software process in a certifiable manner. Maturing the organization this way will also facilitate the running and coordination of multiple XP projects. Each of the subsections below is devoted to climbing *one step* on the CMM ladder and it is assumed that the organization at hand is implementing all the XP core practices and roles.

6.1 Reaching Level 2

Two missing goals need to be reached for an XP organization to reach CMM level 2. These are:

1. Adherence of software products and activities to the applicable standards, procedures, and requirements is verified objectively. (SQA Goal 2)
2. Noncompliance issues that cannot be resolved within the software project are addressed by senior management. (SQA Goal 4)

The process area of *software quality assurance* will need to be improved upon and become more formalized in XP teams striving to reach CMM level 2. Specifically, an objective scrutiny of the software products and the activities will need to be fashioned. Examples of metrics that could be appropriate for objectively verifying the products and the process are:

- release plan adherence,
- percentage of test cases that are running successfully,
- number of acceptance tests that are running successfully,
- length of pair programming sessions,
- individual velocity,
- team velocity,
- velocity, compared with estimates.

Not all of these metrics are required to fulfill the KPA of SQA; it is more sensible to carefully examine which areas are likely to assist management in getting visibility into the state of the project at the most appropriate abstraction level. This level is not necessarily the same for different types of projects within the same organization. The responsibility for these activities could very well lie on the coach, with the assistance of the tracker. However, it is important to keep these responsibilities separated from the developers and testers. The coach or tracker that wants to take on the tasks of SQA measurements can therefore not be assigned for programming tasks within the same project.

It is also important for the fulfillment of the SQA KPA to convey the metrics through defined channels to the affected parties and senior management. The metrics are most conveniently posted to the team using a white-board or an exposed wall in a central location. When metrics out of the ordinary occur it is important for the coach to communicate these findings, either directly to the affected party or during the daily stand-up meeting, and that these issues will be resolved. If no satisfactory solution is found the issue is brought to the customer or project manager, depending on the nature of the issue. If no solution is to be found at this level, senior management will be presented with the issue.

6.2 Reaching Level 3

Four missing goals need to be reached for an XP organization residing on CMM level 2 to reach level 3. These are:

1. The strengths and weaknesses of the software processes used are identified relative to a process standard. (OPF Goal 2)
2. Organization-level process development and improvement activities are planned. (OPF Goal 3)
3. The project's defined software process is a tailored version of the organization's standard software process. (ISM Goal 1)
4. The project is planned and managed according to the project's defined software process. (ISM Goal 2)

Common for the key process areas that are to be dealt with at level 3 for an XP team (OPF and ISM) are the needs for an *organization process definition*. This will typically be the methodology of Extreme Programming, as laid out in the books and on various Internet resources. (A collection of comprehensive sources can be found in the reference section at the end of this paper.) However, companies should not feel restricted to the usage of the 12 core practices, but utilize the fact that XP lends itself very well to customizations, as long as the values and the spirit of the core practices are acknowledged. A software organization that does not exclusively use XP as the implementation model will need to address the coordination between the different models in its organization process definition at an organizational level.

A mature o*rganization process focus* will need to be developed within the company for it to be better able to customize and improve the *de facto* process. Goals number 2 and 3 of the OPF call for insight into the actual process used, and to compare it with the *organization process definition*. The CMM suggests establishing a *software engineering process group*, consisting of software engineering representatives, responsible for carrying out the tasks of managing the process used at an organizational level. It will be appropriate for this group to assess the process every few years, and from the results of this assessment come up with a comprehensive action plan for improving the organization process definition. The action plan will serve as a primary input for driving the development and improvement of the process forward according to a scheduled plan.

The 2 goals of *integrated software management* call for the *organization process definition* to be tailored and streamlined for each project's individual characteristics, into the project's defined software process, using data and lessons learned from previous projects together with changes proposed by the project as guidance. The XP process may be modified in a number of different ways, to better suit a specific project; for example:

- ❑ changing the iteration length;
- ❑ using different granularities when estimating user stories and engineering tasks;
- ❑ modifying the frequency of releases;
- ❑ using different types of metaphor(s);
- ❑ customizing the role and location of the on-site customer.

It is important to note that the process is not to be modified without restrictions. For example, it should still be possible to compare feedback data about the process between different projects within the organization. The defined process definition should also be reviewed by the software engineering process group and project manager, and approved by management, before documented and communicated to all affected parties. It is important to carefully control and manage all changes to the different process definitions. Because the defined process definition is still connatural with XP, its core practices and roles will assert the satisfaction of goal number 2 of integrated software management.

6.3 *Reaching Level 4*

Four missing goals need to be reached for an XP organization residing on CMM level 3 to reach level 4. These are:

1. The process capability of the organization's standard software process is known in quantitative terms. (QPM Goal 3)
2. The project's software quality management activities are planned. (SQM Goal 1)
3. Measurable goals for software product quality and their priorities are defined. (SQM Goal 2)
4. Actual progress toward achieving the quality goals for the software products is quantified and managed. (SQM Goal 3)

In order for the organization to take advantage of *quantitative process management* it must use the process performance from earlier projects and analyze these. For the XP organization it will involve extracting the estimates and actual implementation times of all user stories that may have any general significance. The velocities of the projects are also gathered in a database and analyzed. This data will serve as a model for making future estimates to similar user stories. The data should be stored in a catalogued format that caters for quick retrieval and little overhead during the iteration planning meeting. The software engineering process group should be responsible for managing the data and regularly update it as new technologies and development practices emerge. By analyzing the difference between estimates and actual time spent on user stories or tasks, additional insight and knowledge into the planning game might be yielded. Other data, such as the number of failed acceptance tests, and number and severity of defects after release, should also be managed centrally within the organization.

Satisfying QPM's goal number 3 would indeed make the XP organization more mature, by making it better able to give an initial estimate for the first iteration's velocity, and thereby better fit to give a customer the cost and time frame for a given set of requirements up-front. The inability in standard XP projects to give this estimate is often quoted as a major obstacle for selling in XP to a customer or to senior management. It would still be impossible to give an exact estimate of the total cost and time of a potential project, but not more so than in any other software process model. XP would of course still enable the customer to alter the requirements very late in the project, resulting in an updated project budget.

Software quality management calls for a quantitative understanding of the software products' quality, seen from the perspective of the organization, customer and end user. The XP organization normally approaches the problem of satisfying the customer and end-user's quality goals in different ways than what the CMM suggests. XP strives to please the customer through frequent and continuous product releases, acceptance tests and communication throughout the development lifecycle, while the CMM suggests surveys, focus groups and product evaluations to find out the quality needs before the project starts. While the approaches are fundamentally different, they still try to solve the same problem. The trouble with the XP approach, seen from a CMM perspective, is that it will not lend itself very well to extracting measurable data that can be analyzed and processed during the entire development process. Fundamentally, there are two questions that will need to be efficiently and comprehensively answered:

1. What are the quality requirements, quantitatively?
2. How do we translate the quality requirements into process practices?

Question number one is answered by the CMM through suggested surveys, focus groups and product evaluations to find out the customer and end user's goals. XP does not even try to answer the same question, because it puts the responsibility of finding this answer on the customer while the CMM puts this responsibility on the software organization. It will be next to impossible for an XP organization to satisfy the KPA of SQM without taking responsibility for finding the answer to what the quality requirements are. Therefore, the XP organization will do best by implementing the suggested practices by the CMM mentioned above.

Question number two is answered by the CMM through suggestions like measuring how long the system is running on average between failures. XP would do best in answering this question through implementing the CMM suggestions into acceptance tests for the affected user stories. In order for the quality requirements to be translated into acceptance tests, the requirements will have to be formulated ahead of the project start (or at least ahead of formulating the corresponding user story.) It will be very important to communicate and agree on the quantitative quality requirements with the customer so that these will be successfully translated into acceptance tests. The practices above would satisfy SQM goals number one and two.

Remaining is SQM goal number three, which is somewhat tricky because XP is not taking the approach envisioned by the CMM. By continuously running the acceptance tests for the implemented user stories and track their status, a quantified measurement of the progress towards the quality goals would be obtained. It will also be necessary for this data to be continuously routed and available to the affected people in charge of the quality goals, and for them to take appropriate action to steer the measurements in line with the goals. This practice should be enough to satisfy the final remaining goal for reaching CMM level 4.

6.4 Reaching Level 5

Six missing goals need to be reached for an XP organization residing on CMM level 4 to reach level 5. These are:

1. Incorporation of technology changes is planned. (TCM Goal 1)
2. New technologies are evaluated to determine their effect on quality and productivity. (TCM Goal 2)
3. Appropriate new technologies are transferred into normal practice across the organization. (TCM Goal 3)
4. Continuous process improvement is planned. (PCM Goal 1)
5. Participation in the organization’s software process improvement activities is organization-wide. (PCM Goal 2)
6. The organization’s standard software process and the projects’ defined software process are improved continuously. (PCM Goal 3)

The key process area of *technology change management* is relatively straightforward to implement in the XP organization. However, it should be noted that when a company is using TCM, it might eventually become a non-XP organization or a hybrid if an emerging new process is evaluated as superior.

TCM is best realized in the XP organization by assigning upon the software engineering process group, with the help of experienced staff with expertise in specialized areas, the responsibility to regularly probe the market for new technologies that could be suitable for the organization to adopt. Examples of areas are computer hardware, operating systems, software development platforms, development tools, programming languages, and formal methods. New and unproven technologies that are of potential value for the maturity of the organization will be tested in pilot projects, from where metrics will be collected and analyzed. The technologies that pass the test and are approved by management will be incorporated into the organization process definition or other corresponding organization-wide document, and properly communicated and taught to the affected personnel.

Process change management is also a key process area that is relative straightforward to implement as it can be addressed in more or less the same manner regardless of implementation model. PCM is realized through the establishment of training and incentive programs for making *everyone* within the organization aware of their responsibility and privilege to identify and communicate process improvement opportunities. The responsibility for performing and coordinating these activities once again fall upon the software engineering process group, which also develops and maintains a plan for process improvement according to a documented procedure. The suggested process improvement opportunities are examined and if found interesting tested in pilot projects, and thereafter analyzed. When actual improvements have been identified, verified and approved, these changes are made to the organization's standard software process and the defined software processes, as well as communicated through training courses within the company.

7 Summary

XP and the CMM are clearly complementary. Although XP does not even satisfy the requirements for reaching the second level of the CMM, it is satisfying many key process areas in different levels and using sound practices that facilitates the satisfaction of many more. By making improvements to the organization as described in sections 5 and 6 above, it will be feasible for an XP-practicing company to reach the highest level of software development maturity.

The most important and critical measure will be the establishment of a software engineering process group, which will be responsible for the whole organization's compliance with many of the key process areas. The most difficult area to implement will be that of software quality management on level 4, but as has been shown it does not conflict with the framework of XP.

According to SEI, the CMM requires a long-term commitment and it may take ten years or more to reach the final level. According to this paper much of the work has already been done by practicing the XP process. Is this a paradox?

No, it is important to note that introducing XP in its strictest sense is not accomplished overnight. The process for cultivating this development process within an organization might very well take years. Implementing the remaining KPAs as described will also be a painful and time-consuming process. In this light XP should be regarded as an appealing option and a springboard for a software development company in search of an empirical process control model to reach higher maturity. By introducing the measures described here, most companies would be likely to improve their position on the market and reach a state where their long-term goals would be stable, and their software products would be of a predictably high quality.

8 References

[1] Kent Beck, *Extreme Programming Explained*, ISBN 0-201-61641-6, Addison-Wesley, 2000

[2] Mark C. Paulk, *Extreme Programming from a CMM Perspective*, http://www.sei.cmu.edu/cmm/papers/xp-cmm-paper.pdf, 2001

[3] Hillel Glazer**,** *Dispelling the Process Myth*, http://www.stsc.hill.af.mil/CrossTalk/2001/nov/glazer.asp, CrossTalk, 2001

[4] Mark C. Paulk, *Agile Methodologies and Process Discipline*, http://www.stsc.hill.af.mil/CrossTalk/2002/oct/paulk.asp, CrossTalk, 2002

[5] Ron Jeffries, *Extreme Programming and the Capability Maturity Model***,** http://www.xprogramming.com/xpmag/xp_and_cmm.htm, 2000

[6] Babatunde A. Ogunnaike and W. Harmon Ray, *Process Dynamics, Modeling and Control*, ISBN: 0-19-509119-1, Oxford University Press, 1994

[7] Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum*, ISBN: 0-13-067634-9, Prentice Hall, 2001 (a relevant excerpt from this book is freely available at http://www.controlchaos.com/Excerpt.pdf)

[8] Royce W, *Managing Development of Large Software Systems: Concepts and Techniques*, Proceedings of WESCON, 1970

[9] Rational Software whitepaper, *Rational Unified Process: Best Practices for Software Development Teams*, http://www.rational.com/products/whitepapers/100420.jsp, 1998

[10] Agile Alliance, http://www.agilealliance.org

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, ISBN: 0-20-163361-2, Addison-Wesley Pub Co, 1995

[12] Martin Fowler, *Refactoring*, ISBN: 0-20-148567-2, Addison-Wesley Pub Co, 1999

[13] Frederick P. Brooks, *The Mythical Man-Month*, ISBN: 0-20-183595-9, Addison-Wesley Pub Co, 1995

[14] Jonas Martinsson, *Evaluation of Extreme Programming Pilot Project at Labs2*, http://www12.brinkster.com/jonasmartinsson/docs/EvaluationOfExtremeProgrammingPilotProjectAtLabs2.rtf, 2002

[15] William C. Wake, *The System Metaphor*, http://xp123.com/xplor/xp0004/index.shtml, 2000

[16] Refactoring Home Page, http://www.refactoring.com

[17] Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries, *Strengthening the Case for Pair-Programming*, http://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.PDF, 2000

[18] Alistair Cockburn, Laurie Williams, *The Costs and Benefits of Pair Programming*, http://www.cs.utah.edu/~lwilliam/Papers/XPSardinia.PDF, 2000

[19] Yahoo! Groups mailing list for XP, http://groups.yahoo.com/group/extremeprogramming/

[20] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, *Capability Maturity Model, Version 1.1*, IEEE Software, Vol. 10, No. 4, July 1993, pp. 18-27.

[21] Terttu Orci, Astrid Laryd: *CMM for Small Organizations, Level 2*, Technical Report UMINF 00.20, Department of Computing Science, Ume˥ University, http://www.cs.umu.se/~jubo/Projects/QMSE/QMSE_English_V1.0.3.pdf, 2000

[22] Mark C. Paulk, *The Capability Maturity Model*, ISBN: 0201546647, Addison-Wesley Pub Co, 1995

[23] Ron Jeffries, Ann Anderson, Chet Hendrickson, Kent Beck, *Extreme Programming Installed*, ISBN: 0201708426, Addison-Wesley Pub Co, 2000

[24] Kent Beck, Martin Fowler, *Planning Extreme Programming*, ISBN: 0201710919, Addison-Wesley Pub Co, 2000