Time Constrained Requirements Engineering with Extreme Programming – An Experience Report

Erik Lundh
Compelcon AB
Helsingborg, Sweden
erik.lundh@compelcon.se
www.compelcon.se

Martin Sandberg
netMage AB
SE-224 64 Lund, Sweden
martin.sandberg@netmage.se
www.netmage.com

Abstract

Requirements engineering is a core practice within eXtreme Programming. Practising teams learn to not abandon any of the interconnected core practices. The continuous feedback teaches the team that they will develop at a faster and more reliable pace with the full set of practices. This paper shares an experience of how teams could accelerate development out of necessity, while maintaining their established core process, including requirements engineering. In this paper we discuss some issues of timing, continuous requirements engineering and sustainable pace.

1. Introduction

Contrary to common belief, rapid development does not mean that you have to throw requirements engineering out of the window.

We believe that most developers would rather use a minimal core development process that includes efficient requirements engineering, than start out by trying to pick and choose from a larger, arguably richer process framework. A minimal core process, that the team trusts, will be used even when time pressure usually would tempt them to cut corners.

We find support for our point of view in experiences that the netMage team gained from adopting the agile methodology eXtreme Programming (XP) [1],[2].

There are several papers and articles that discuss the strengths and weaknesses of XP. We will point out some benefits with respect to requirements engineering that teams like netMage have attained from XP.

One objection to Extreme Programming is that requirements engineering is overlooked as a development phase and that problems are solved by pushing the problems forward to the next iteration, as they occur. A reason for these assumptions might be that the XP literature does not describe requirements engineering as an independent phase. However, requirements engineering is a core practice in XP and plays a significant role throughout the XP development cycle.

2. Requirements Engineering in XP

2.1. Requirements Engineering

Requirements engineering is the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. There are a number of generic activities common to all requirements engineering processes [4]:

- Elicitation
- Analysis
- Validation
- Management

Elicitation is a definition of the system in terms understood by the customer. *Analysis* is a technical specification of the system in terms understood by the developer. *Validation* is concerned with showing that the requirements define the system that the customer wants. Requirements *management* is the process of managing changing requirements during the requirements engineering process and system development.

2.2. Extreme Programming

Extreme Programming is an agile methodology that has gained increasing popularity and acceptance in the software community. XP promotes a discipline of

software development principles of communication, feedback, simplicity, and courage. It is designed for use with small teams who need to develop software quickly in an environment of rapidly changing requirements.

The four requirement engineering activities listed in section 2.1 above, are core activities of XP. Duncan states in [5] that "The primary vehicle for requirements elicitation in XP is the addition of a member of the customer's organization to the team." We agree on the importance of customers in the team, but we also believe that the Planning Game is the primary vehicle for requirements engineering in XP.

The team has one or several members that are customers. The customer team member brings the team knowledge on what the users want and a business perspective. The customer could come from sales and marketing or a domain expert from a representative customer or, in contract or in-house development, a domain expert from the customer who pays for the project.

The practice to make customers part of the team is often referred to as "onsite customer", since the early XP projects were contract work. We use the shorter "customer" to define any team member that generates functional requirements. Usability experts are often great additional customers in a team.

The team elicits functional requirements expressed as user stories from team members that are customers.

In the planning game, developers tell customers how much work they can do in an iteration. The developers then select and prioritise the functionality they want in this iteration. Developers then define, order and prioritise engineering tasks to implement the desired functionality. The customers decide what the product should do, the developers how to do it and how much they can do in each iteration.

Figure 1 A Planning Game

Since at least one customer is always present, or available on 15 minutes call, throughout the development he or she is available to answer questions and clear up ambiguity. The following is based on Don Wells extremeprogramming.org[2] description of XP. We add our own experiences of the development process in XP and make an emphasis on the requirements engineering and customer activities.

2.2.1. Elicitation and Analysis during Planning Game

Each planning game, except the very first one, begins with a delivery of the work products of the previous iteration, often including installation and demonstration of the current product.

At the first iteration: If there is an old or similar product the team uses a demo of it as a starter for the first planning game. The demo is important and the only exception is at the very first iteration of a new product with no legacy.

The customers have to sign off on the acceptance tests of each implemented user story in the previous iteration. Each iteration of the XP development cycle starts with a Planning Game at the *iteration-planning meeting*.

The customers of the team write or select new *user stories* (Figure 2) based on the fresh impressions of the current product. Customers often bring prewritten stories to the meeting, but the selection and priorities are usually affected by impressions from the current product.

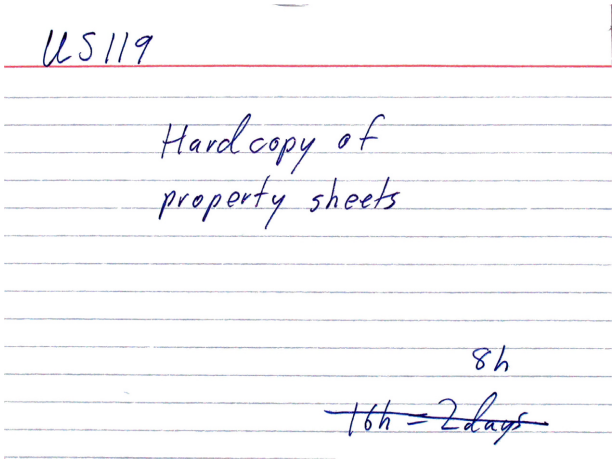

Figure 2 The User Story US119 with final estimate 8h

Each user story is written on a new index card. Each user story is uniquely numbered, like receipts in a simple bookkeeping scheme. Our numbering scheme is simple: We mark the first user story of a project "US1". The next US2, US3, etc. We keep track of numbering for user stories (USxx), acceptance tests(ATxx) and engineering

tasks (ETxx)., We have a single place were we note the highest used number for each of these. Most teams appreciate the numbering since it brings an intuitive feel for the size of things.

The team always adds, generates and prioritises among user stories based on the product's current status at the planning game at hand. All user stories get prioritised in three basic "lanes"(Figure 3): Must Have Now, Can Wait, and Nice To Have. The long-term release plan (Figure 4), usually introduced after the team has grown confident with the constant reprioritization and small releases, contains an additional number of milestone releases or "lanes", with user stories in priority order for each of these milestones.

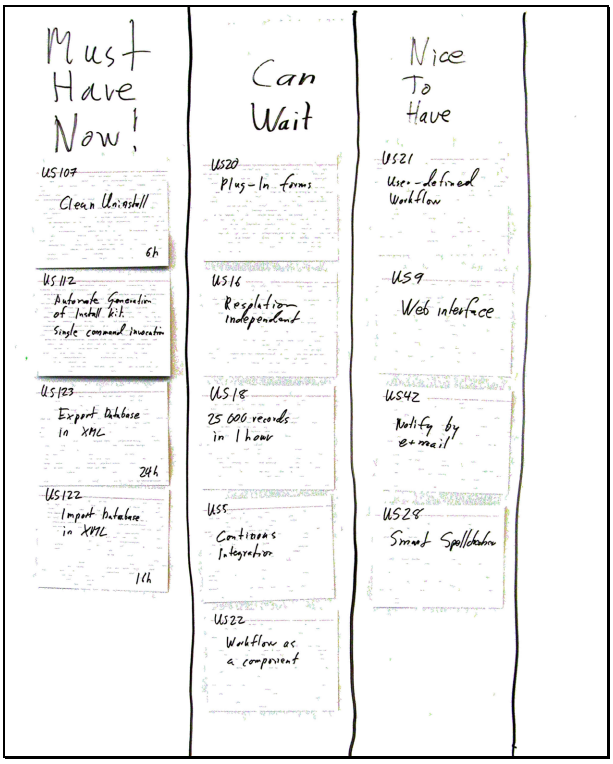

Figure 3 Basic three lanes requirements priorities

Typical high-priority user stories in each "milestone lane" affect other teams and subprojects, such as hardware development, or are much anticipated by key customers in beta test programs.

Customers present user stories to developers in priority order from the "Must Have Now" row. Developers make rough time estimates that are written on each user story's index card. If a story cannot be estimated it goes back over the table to the customers to be rewritten, split or down-prioritized to a later iteration.

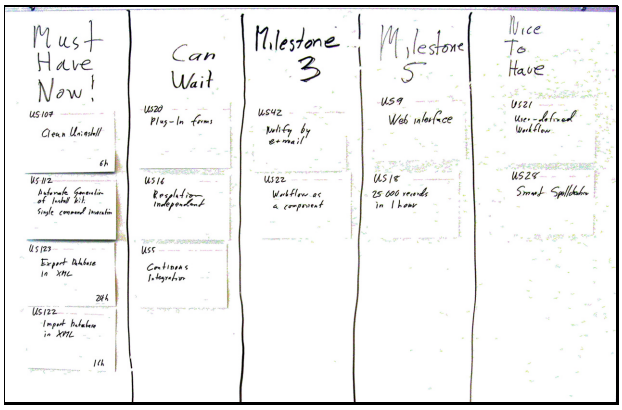

Figure 4 Release plan with milestone lanes

The developers tell the customers how much work they can do in this iteration. Different teams have different ways to define units of work. Some uses "ideal time", some use points. The important thing is that the developers have a scaling factor, *velocity* in XP terminology, between available workdays times available developers and the amount of work that the team can do. The team does not count activities outside the project into their estimates. They use the velocity factor as a heuristic that gets adjusted after each iteration. Things like sick-days or off-site company activities do not affect the velocity, it affects the number of available workdays. But normal day-to-day activities affect the velocity. The work that can be done is the available developers multiplied by the available calendar days divided by the velocity.

Based on these rough estimates, the customers then choose a set of prioritized stories that fill up the iteration. They usually prioritize a few extra at the end to cover for the case where things go smoother than estimated.

After that the developers start to break down the stories in engineering tasks and make an estimate on each task. The engineering tasks are uniquely numbered throughout the project (ET1, ET2, etc). The engineering task is also marked with the number of the user story that it implements part of.

The task estimates are summed up to each user story. The customers now have another opportunity to reprioritize among stories based on the more detailed estimates. Again, it is recommended to always have a few extra stories broken down into engineering tasks, if things speed up.

The iterations are time-boxed for several reasons. That means each iteration has a fixed duration and ends at a

fixed date. If the team runs out of stories before the end of the iteration, we need to cast an extra planning game. The planning game has an overhead of a couple of hours for the whole team. Better to have a few extra stories at the back of the iteration plan. We don't want to change the pace of the time-boxed iterations to make the team's estimation heuristics more meaningful. It is also much easier to synchronize the iterations with other projects if we stick with a regular time-boxed pace of equal-sized iterations.

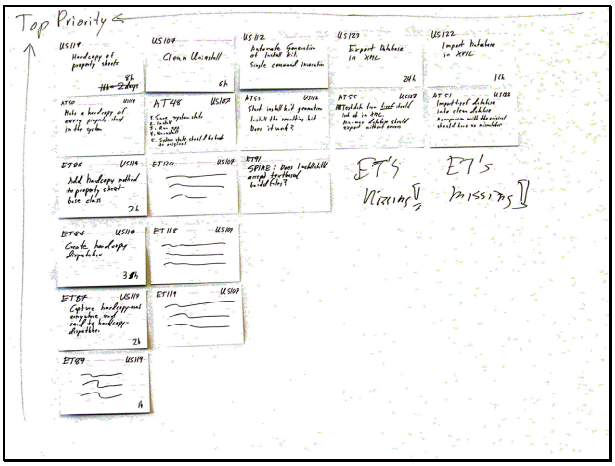

Figure 5 An iteration plan develops during planning game

A software release often consists of a number of iterations (Figure 6). An *iteration-planning meeting* is called at the beginning of each iteration to run a Planning Game that produces that iteration's plan of programming tasks.

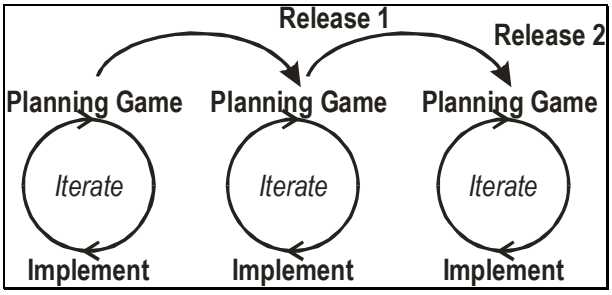

Figure 6 Relationship between iterations and releases

Each iteration is one to three weeks long. Iterations usually have a fixed interval during a project, to improve estimation heuristics and provide time-boxing and fixed releases as benefits to external stakeholders. The customer chooses a number of user stories for the iteration. The developers translate the user stories into engineering (programming) tasks that realize them. The tasks are written down on cards. The time to complete the tasks is estimated by the developers and written on the task cards, then the estimates are added up for the whole user story. Then the customer prioritises the user stories in order of the most valuable first. These task cards are the detailed plan for the iteration.

Documentation is produced throughout the XP development cycle. The mandatory documentation consists of the user stories, programming tasks and acceptance tests that are all documented on paper cards. Everything can of course be documented electronically for later review but what typically gets documented electronically are the acceptance tests as they are the definition of the system and needed for regression testing. Additional documentation, such as, requirements analysis documents written according to a client's preferred standard, for example, IEEE; system documentation, quality assurance documents, user manuals etc can be defined in user stories and delivered together with the system after each iteration. Documentation is not taken for granted in an XP project. Instead it is a deliverable that gets prioritised according to business value. Short cycles and intense development, often with a system-wide scope; give high visibility to which documents are essential during the project. The customers have to put business value on necessary documents that are needed to maintain the end product and further develop it in new generations.

Figure 7 Requirements repository - XP style

2.2.2. The customer's perspective

An XP project starts with the customer writing *user stories* on paper cards. The stories are written by the customers as things that the system needs to do for them.

Writing user stories is the XP practice to elicit requirements from customers. They are in the format of about three sentences of text written by the customer in the customer's terminology without technological-syntax. That is, details of specific technology, data base layout,

and algorithms should be avoided. The stories should be focused on user needs and benefits as opposed to specifying GUI layouts. User stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. However, user stories have a lot in common with low-fi usability requirements engineering such as paper prototyping [9]. When the time comes to implement the story the developers could go to the customer on the team and receive a detailed description of the requirements face to face. User stories also drive the creation of the acceptance tests. The team assigns one developer, the Tester role, at the outset of each iteration with the added responsibility that good acceptance tests get developed together with the customer.

Acceptance tests (Figure 8) are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. Many teams ask the customers, already at the planning game, to specify at least one acceptance test for each user story that the customers include in the upcoming iteration. The customer specifies scenarios and criteria that determine when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works. Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release. A user story is not considered complete until it has passed all its acceptance tests. Acceptance tests should be automated so they can be run often.

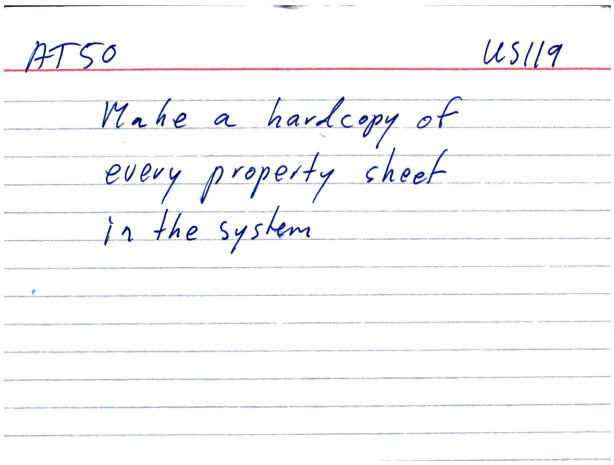

Figure 8 AT50 is an acceptance test for US119

2.2.3. The developer's perspective

During the planning game, the developers on the team break down each user story into one or several *engineering tasks* (Figure 9). The tasks are ordered in a row under each user story. If several stories depend on the same task, the task ends up under the most important story. This is the XP way of resolving requirements dependencies. The placement and prioritisation of engineering task are the responsibility of the developers. The engineering tasks are numbered.

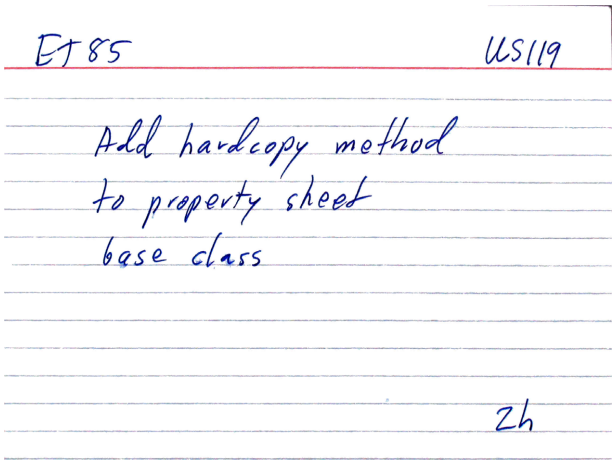

Figure 9 Engineering task ET85 implements part of US119

Each iteration a team member takes the responsibility of the Tracker role. The tracker keeps track of actual effort spent on each engineering task, and write the down on the tasks index card. The tracker also collects other metrics that the team agree they have use for. The tracker updates the velocity factor at the end of the iteration.

Spike solutions are created to figure out answers to tough technical or design problems. Spikes are often created to reduce uncertainty in estimates. A spike solution is a very simple program to explore potential solutions to implement functionality expressed in user stories or engineering tasks. The spike is a small proof of concept or benchmark program in itself, which only addresses the problem under examination, and ignores all other concerns. Spikes could also be for example a script that creates and exercises a database, giving a feel for the performance or the size of an implementation in code. Spikes should not be good enough to keep, so we plan to throw them away. The goal is to reduce the risk of a technical problem or increase the reliability of a user story's estimate.

During the iteration a developer picks the engineering task with the highest priority. The developer invites a peer to

pair up with for the task. They start to work on the task, maybe with a quick design session at the whiteboard, but the first code written is always unit tests that deepen the understanding of the functionality the task should produce. New questions might arise that had not been obvious during the planning game and that have not been explained in the user story. As the customer is part of the team, they walk over to her, ask the questions and receive the needed input. The unit tests are written first, followed by production code with the desired functionality. As the understanding of the task deepens, the tests might be improved. Developers continuously implement functionality to meet current requirements and then refactor the code to clarify its intent, make it more maintainable, etc [8]. To implement and refactor is to recognize and embrace stepwise refinement, write or modify just enough code to solve the task at hand, and then refine the solution to a certain finish. The technical definition in the words of Martin Fowler[8] "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure." After a while refactoring becomes a state of mind and the team refactors their planning, user stories and tests as well as the product code. In some cases there is no clear line between optimisation (in a broad sense) and refactoring.

When all tasks to fulfil a story are finished the developers alert the customer to have it acceptance tested. The user stories and acceptance tests make up the customers functional and non-functional requirements in XP.

3. The project – XP in fast forward

3.1. Background

netMage is a software company developing enterprise solutions that create, aggregate and distribute enterprise information. This experience report deals with the work on the first release of a product, which amounted to six months, during June 2001 – December 2001.

The project involved eleven people: six developers, a technical architect who doubled as acceptance tester; the customer group comprising of a sales representative, a product manager, usability engineer and an administrator. The product was developed using the Microsoft .NET framework, when .NET was still a beta release and a new 3rd party platform from the company Anoto for digital paper functionality, which was also a beta release. Both these platforms were subject to change during the course of the project.

netMage had previously used a traditional software development methodology influenced by RUP (Rational Unified Process) and related heavyweight methodologies and quality assurance practices. The methodology had involved a requirements process and a development process where results were shown to the customer only at the late stages of the project. The character of the company's products, such as, corporate portals did not fit in well with the methodology in place.

The company had a few developers who were very technologically savvy but not so experienced in working according to software processes and in teams. Martin Sandberg, the Quality Manager, tried to guide the developers in the direction of software best practices through introducing a traditional development methodology with its associated processes. It proved difficult to make the less experienced developers understand the need and usefulness of these practices. If you do not understand why you are doing something you are inclined to do it half-heartedly or you do not do it as it was intended. Some developers were also coding cowboys who worked more individually than as a team. Due to the waterfall-like development process market input came in at the very beginning of the project during requirements elicitation and then they went away while the system was developed to return when it was to be delivered often discovering that their expectations had not been met or that their view on the upfront requirements had changed.

Martin wanted to change the methodology to something more agile and iterative and went to a conference on XP, organized by Erik Lund, to get new ideas on how to improve the process and the working culture in the company, especially the cooperation between the marketing and development departments. After having learnt more about XP at the conference and through literature Martin was very pleased to see that XP involved many of his own experiences from successful practices and projects and added a few more practices, which he believed in but had not tried himself in any project. What especially appealed to him was that XP provided a minimalist core set of team processes "out of the box", which were ready to start using and that there was enough literature to support the introduction of the process without having to write and adopt a process.

When a new project was started with new technologies combined with vague and changing requirements; Martin suggested they should try XP, which is designed for use with small teams who need to develop software quickly in an environment of rapidly changing requirements. Management understood the shortcomings of the existing methodology and agreed to try XP on the new project.

Erik Lundh was contracted as the external XP coach while Martin Sandberg acted as internal onsite coach. The project started off with an introductory training in XP,

followed by coaching during the course of the project. Erik usually introduces a new XP team with two or three half-day XP seminars with at least a few days digestion time in between. The last seminar often ends with a planning game of the first iteration of the team's first XP project. Erik typically only coach onsite half a day at a time during projects. This way he keeps the external perspective of the project. He never allows a team to include him as a development resource in the planning. In Erik's experience, an external coach should be present at every planning game to share the team's view of priorities and be able to give the proper coaching support during the following iteration.

3.2. A time constrained experience

The aim of the project was to develop a web content management solution for refining handwritten messages (Figure 10). The messages are created using digital paper and digital ink [3]. Before the project was started a minor feasibility study was carried out. The requirements for the functionality were elicited gradually during the course of the project. The team chose a web messaging system as its metaphor.

Four and a half months into the six-month project, XP was an established backbone process for the team. Every team member knew and trusted the core practices and how they interacted. The team felt productive, confident and proud of their work. They had experienced "flow"[10] on several levels, as individuals, in pairs and as a team. Pair programming time was referred to as "flow time". One day a new potential client showed interest in the product. A meeting with the client was scheduled for the following day, as the client would be passing through on other business. Word of this meeting reached the product manager after business hours. In order to make a good impression on the client it was desirable to show a new feature that was still only a concept in the mind of the customer group within the team.

The team was highly motivated and knew that a successful demonstration the next day would be very good for the team as well as the company. The team put their trust in the work pattern that they now felt comfortable with. No one went to the office to work through the night until the deadline. They knew that a focused team effort, within their everyday minimalist process, the next day would be more likely to succeed. This was a significant vote of confidence in the team's established agile process.

The following morning, during the stand-up meeting, all project members were, as usual, briefed about the status. But this morning was special. It was decided to put all other work aside and do the most of the opportunity presented to the team. The two-week iteration was then only halfway through. A few new user stories were identified and written down to wrap-up the current state of the iteration. Then the requirements for the new feature were written in new user stories. The user stories were broken down into programming tasks, time estimated and prioritised. The planning was done in matter of minutes instead of hours. The fallback strategy was to demonstrate the version of the software built on the previous day.

Figure 10 The Anoto pen

As the developers started on their tasks, the customers, some of them with web design and human computer interaction skills, created the graphics and the interaction for the new feature. Later in the day that input was given to the developers who implemented the feature, assisted by the customers. Integration of the pair developers' work was carried out and everything was acceptance tested. The team finished the work just in time for the product manager to demonstrate it to the client in the evening.

Some clean up, refactoring, was needed on the following day but the implementation had resulted in "the simplest thing that could possibly work".

XP in Fast Forward (XPiFF) was later repeated in additional scenarios and projects.

4. Discussion

4.1. Is XP complete?

We have seen some papers and presentations regarding the limitations of XP. Usually a non-practitioner tries to compare XP to one or several process frameworks such as the Unified Process. It is our belief that such comparisons have a limited value. eXtreme Programming is a condensed minimalist set of development practices, presented with an approach that caters to the developer, and have a scope that teams and individuals can grasp, at least after a minimal period of practice. XP is a core pattern and a backbone that teams can rely on. Teams that use only XP "by the book" or pick and choose among practices are less successful than teams that have experienced full XP as described in the literature and by the community and then grow their own core process on that experience.

The first XP project at netMage was a focused effort that involved the whole company and all of its resources. Martin Sandberg notes that in current projects, the staff is much more open to methods and processes from "heavy-weight methodologies". Martin introduced, after the successful first XP project, certain practices that adapt and complement the teams core XP process to better handle a portfolio of multiple projects. The team embraces them in a way that was not possible before their successful XP process experience.

4.2. What made XPiFF work?

- A functioning team with confidence in its lean and well understood process.
- Several months of experience with the process at a normal pace.
- A well defined, familiar problem domain and a well defined product metaphor.

It is clear that the netMage team would not have been able to make a one-day iteration at the outset of the project. The product domain, the .NET environment and XP were all new to them. But after just a few months of practice with their integrated core XP process, in this new and complex environment, it was still more natural to stick with the process than to "hack away" when faced by time constraints and stress.

4.3. The relation between XP and CMM level 5

Mark Paulk has compared the Capability Maturity Model (CMM) and XP in detail in [6].

Paulk also suggests in [7] that a well functioning XP team could, *as a team - not the whole organization,* qualify for a successful CMM level 4 assessment. CMM level 4 is the managed level, with measurements guiding decisions. Few software organizations in the world have qualified above level 3 or 4. Note: A team performing at CMM level 4 or even level 5, the optimising level, is not the same as a whole organization being at the same level. CMM assessors trained by SEI do not look at a single team; they look at an organization.

Do teams, officially assessed as high-maturity, succeed in following their process under time pressure?

The legendary group that builds the NASA space shuttle software had a large influence on the definition of the higher levels of the CMM model. Normally shuttle software is built in cycles of months. We have heard stories of this team being able to run their full process in fast forward, cutting down a release cycle to days, on several occasions. We wanted to get the facts straight and Sweden's leading CMM assessor, Fredrik Westin, told us to contact Ted W Keller, the former project coordination manager for the shuttle software. We asked Mr Keller for the true facts of one of the stories about shuttle software changes "in fast forward" that we heard. Mr Keller read an early draft of this paper and came back with a kind reply, and a better story:

"Perhaps a better example would be another (of the many) incidents I frequently shared in such presentations, which had to do with an in-flight (hardware) failure of an attitude control jet on the Shuttle during mission STS-68 in 1994, a problem, which, left unresolved, would have prevented the Radar Mapping objectives of the mission from being accomplished and likely would have prematurely forced termination of the mission. Our team was able to exploit our Mature SEI CMM Level 5 Process in real-time *(during the mission)* to analyze the options for modifying the flight control code, to "design" a very simple code change to reassign the functionality of other attitude control jets that would result in equivalence to the original capability (before the jet failed), to make the code change, to have the code change inspected by our own team as well as all NASA crew, control, and safety experts, to execute the complete governance and configuration control processes to allow such a change to get into the flight code, to fully test the change and re-test the resulting flight control system in simulators and in code execution test cases, to conduct and complete the NASA flight readiness certification of the change, to formally deliver the revised code to NASA, and support NASA in

the transmission and installation of the revised code into the executing code onboard the Shuttle in orbit. The successful execution of this process enabled the mission to continue and succeed. The software process steps that were executed in this incident usually required 3.5 weeks to complete in an expedited path, just due to the built in prerequisites for such life-critical code. On this particular occasion, the entire process (no step missed, no short cut taken, except the deviation from the standard process which required 3 day advance notice and review materials for the reviews) was completed in less than 8 hours, in order to save the mission. For this, our team received NASA awards and were invited to be in the NASA Mission Control Room for the landing of the mission, which we had saved. One of our programmers was invited to climb the ceremonial ladder and hang the mission plaque on the wall, a tradition at NASA signifying another successful Shuttle mission." – *Ted W Keller, IBM Global Services*

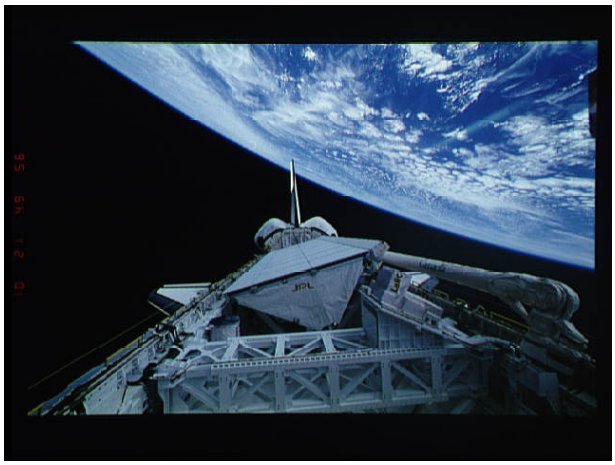

Figure 11 Space shuttle cargo bay on mission STS-68

The shuttle team executed *every step* of their well-known field-proven CMM level 5 process, but very fast. The only deviation from the standard process was review meetings not being announced three days in advance.

We do not suggest that the netMage team is at level 5, but we observe some CMM level 5 characteristics as described by SEI-trained CMM assessors.CMM level 5 is the highest maturity level: optimising. Among the characteristics: continuous improvement and change, ability to successfully change or adapt processes and tools even during projects.

What we do think is that XP mature teams quickly. Mark Paulk in [6] suggests that XP and CMM complement each other. According to Paulk CMM tell you *what* you need to do to mature on each level, but very little on *how*. XP on the other hand focuses on how. The XP how-to approach counts on the team to gain insight in what they are doing from actual experience.

Erik Lundh is currently coaching XP pilot projects on teams in a large development organisation, working closely together with Fredrik Westin, CBA IPI, SW-CMM and P-CMM assessor XP give a team a quick, well-defined path to a certain level of maturity, while CMM give a whole organization a roadmap to maturity. In an organization with the majority of its staff in software development, we think that process orientation and process improvement will be much more appreciated by both development and market organization when they have an XP experience.

5. Summary

The experiences described in this paper point out that the adoption of a core agile process, XP, gave the team at netMage a backbone to rely on when the development needed to be accelerated temporarily. They were able to engineer requirements under time-constraints, both old and new ones.

As XP is a condensed process every team member understands it and can act upon requirement changes. A successfully established XP team does not abandon core practices because of time constraints. They are more confident with executing their familiar condensed minimum-overhead process at a faster pace.

netMage now has a core methodology that it can build upon and trust when change is swift or the process needs to be sped up. The project members also know that they can rely on everybody always knowing what to do, as everyone has experienced the full methodology.

Give a team a full development cycle process they can quickly master, that makes sense, and gives visible results. That team will stick with their core process even in difficult times, and will be much more receptive to sensible process improvements. They have experienced success and put trust in their process, but have most certainly already made small improvements and adaptations on their own.

XP is a lean team-oriented software development process. Once the team has a gut feeling for the XP practices and rules, you can supplement them with methods, techniques and tools that can produce even better business value according to the characteristics of the particular project.

6. About the authors

Erik Lundh, Compelcon AB

Erik Lundh has 20 years of experience of product oriented software development.

Erik is an independent that still knows the ins and outs of code, but prefers to offer his consultancy time as a mentor and coach to companies that develop software for products. He serves as an advisor to management and as a director at the board of select companies.

Erik currently stimulates improvements by acting as coach to several XP-teams working with product oriented software development.

Martin Sandberg, netMage

Martin is in his seventh year of experience in development and management of component based software and systems integration projects, as well as in various software development methodologies and quality standards. Martin was the product manager in the project and also acted as internal coach. Prior to this project Martin was the Quality Manager of netMage.

7. References

[1] Kent Beck, *eXtreme Programming – Embracing Change*, Addison Wesley 1999.

[2] http://www.extremeprogramming.org

[3] http://www.anotofunctionality.com

[4] Ian Sommerville, *Software Engineering 6th ed.*, Addison-Wesley 2000.

[5] Richard Duncan, *The Quality of Requirements in Extreme Programming*, Mississippi State University.

[6] Mark Paulk, *Extreme Programming from a CMM Perspective,* Software Engineering Institute 2001, online at: http://www.sei.cmu.edu/cmm/papers/xp-cmm-paper.pdf

[7] Mark Paulk's comments in IEEE Dynabook eXtreme Programming 2000, online at: http://www.computer.org/SEweb/Dynabook/PaulkCom.htm

[8] Martin Fowler, *Refactoring*, Addison Wesley 1999

[9] The INUSE Handbook section 3.3.1: *Paper Prototyping.* (One of several descriptions of the method.) Available online at http://www.ejeisa.com/nectar/inuse/6.2/3-3.htm

[10] Mihaly Csikszentmihalyi, *Flow: The Psychology of Optimal Experience,* HarperCollins 1991