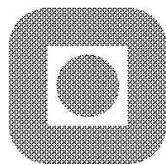


How to Combine Software Process Improvement and Quality Assurance with Extreme Programming

Diploma thesis

by

Håvard Julsrud Hauge



IDI, NTNU

June 10, 2002

Abstract

Extreme Programming (XP) has since its introduction around five years ago achieved a great interest in the international information technology community. It is an agile software development methodology that handles rapid changes in requirements and environments. We have investigated how to combine software process improvement and quality assurance with XP and found that there are several aspects that supports these areas. However, to enable improvements in the XP process we must make sure that XP is adapted to suit local conditions instead of recommend concrete solutions. There is no definite answers to how XP should be performed, it will always depend on current conditions and will change during a project. Continuous learning through experience and knowledge sharing are important for improvements. We have suggested some extensions to XP to make improvements in XP more efficiently; iteration feedback meetings after each iteration, Post Mortem Analysis before and after projects, and usage of the Goal-Question-Metric method and developer diaries to get some documented information from a project. When it comes to quality assurance there is several challenges for both developers and the customer to succeed with XP. We have suggested to include dedicated quality responsible into the development team to ensure system quality and customer satisfaction. We have also highlighted that all tests for the system should be well organised to handle the rapid pace of development and changes. Automating the tests is central. Two XP development projects are used as case studies to support our theories for software process improvement and quality assurance in XP.

Preface

This report is a Diploma thesis written during the final semester at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The author is a fifth year student at the Software Engineering Group at IDI.

The background for this thesis is the interest of a rather new and agile development methodology called Extreme Programming and the challenges related to it. My desire for learning more about Extreme Programming and the need for an investigation of how software process improvement and quality assurance can be combined with this method were the main motivations.

The mentor and teaching supervisor has been Tor Stålhane, Professor at the Department of Computer and Information Science at The Norwegian University of Science and Technology. I would like to thank him for all contribution and guidance.

Further I would like to thank Ole Johan Lefstad and Øivind Mollan for letting me take part of their XP project this semester, Jørgen Brudal Sandnes and Sverre Stornes for letting me learn about the experiences from their XP project, Nils Brede Moe and Tore Dybå at SINTEF for valuable discussions, and Trond Johansen at Proxycor in Trondheim for sharing his XP experiences.

Trondheim, June 10, 2002

Håvard Julsrud Hauge

Contents

1	Introduction	1
1.1	Introduction to the subject and problem	1
1.2	The method used to solve the problem	2
1.3	Overview of the thesis	3
2	XP, SPI, and QA: Background and motivation	4
2.1	Extreme Programming	5
2.1.1	What is XP – the history and the philosophy	5
2.1.1.1	Root, C3 project, and The XP Series	5
2.1.1.2	Philosophy	6
2.1.2	The values, principles, activities and practices	6
2.1.2.1	Four values	7
2.1.2.2	Principles and activities	7
2.1.2.3	The twelve practices	8
2.1.3	How XP is planned and carried out	9
2.1.4	An agile and lightweight methodology	10
2.1.4.1	From heavy to light	11
2.1.4.2	The Agile Manifesto and the ongoing debate	12
2.1.4.3	A self-adaptive process	13
2.2	Software Process Improvement	14
2.2.1	What is SPI?	14
2.2.2	Improvement principles and models	15
2.2.3	Measurement and information gathering	15
2.2.3.1	Goal-Question-Metric	15
2.2.3.2	Post Mortem Analysis	16
2.2.3.3	KJ / Affinity diagrams	17
2.2.4	Process improvement and quality	17
2.2.5	Process improvement vs. process innovation	18
2.2.6	Rapid Process Improvement	18
2.2.7	Personal and organisational learning	19
2.3	Quality Assurance	21
2.3.1	What is QA?	21
2.3.2	Total Quality Management	22
2.3.3	Quality standards	23
2.3.3.1	The ISO-9000 family	23
2.3.3.2	ISO/IEC-9126 and ISO/IEC-14598	24
3	Experiences with XP	27
3.1	Reports from the field	28
3.1.1	Short description of the projects	28
3.1.2	Experiences with the XP practices	29
3.1.3	How local modifications were performed	31
3.1.4	How XP affects quality and quality personnel	32
3.2	Research and use in academia	33
3.2.1	XP in university environments	33
3.2.2	The effects of Pair Programming	34
3.3	Interviews and personal contact with companies	37
3.3.1	SINTEF	37

3.3.2	Proxycom	37
4	Case studies.....	39
4.1	Case Study 1: A XP development project evaluated with GQM and PMA	40
4.1.1	Project description and measurements.....	40
4.1.1.1	The system to be developed.....	40
4.1.1.2	Project and iteration plan	41
4.1.1.3	The GQM abstraction sheet and logbook.....	41
4.1.2	Project environment.....	43
4.1.2.1	The developers and the customer.....	43
4.1.2.2	Office landscape and facilities	43
4.1.3	Limitations and constraints	45
4.1.4	Observations of the project and the iteration evolvement	45
4.1.4.1	The iterations.....	45
4.1.4.2	Learning and improvements of the process	47
4.1.4.3	Quality related work.....	50
4.1.5	Case study results.....	52
4.1.5.1	Results from the GQM measurements.....	52
4.1.5.2	Experience with XP practices	55
4.1.5.3	Learning and improvements	56
4.1.5.4	Quality assurance.....	58
4.1.6	Conclusion and recommendations.....	60
4.2	Case Study 2: A XP development project evaluated with PMA.....	63
4.2.1	Project description.....	63
4.2.2	Case study results and conclusions.....	63
4.3	Discussion of the case studies	66
5	XP, SPI, and QA: How to combine them.....	68
5.1	SPI and XP	69
5.1.1	Existing SPI elements in XP	69
5.1.2	SPI difficulties in XP and some solutions	70
5.1.3	How increase the SPI efficiency in XP	71
5.1.3.1	Local adoption	71
5.1.3.2	Continuous learning and experience transition	73
5.1.4	SPI extensions to XP.....	74
5.1.4.1	Iteration feedback meetings with KJ	75
5.1.4.2	Project analysis with GQM and logbook.....	76
5.1.4.3	Distribution of project experience with PMA and repositories.....	76
5.2	QA and XP	78
5.2.1	Existing QA elements in XP.....	78
5.2.2	QA difficulties in XP and some solutions to these	79
5.2.3	How to ease the introduction of QA in XP.....	81
5.2.3.1	Include a dedicated QA team or QA person.....	81
5.2.3.2	Use automated tests.....	82
5.2.4	QA extensions of XP	82
5.3	Discussion.....	83
6	Conclusion and further work	85
6.1	Conclusion	85
6.2	Further work.....	86
7	Abbreviations and acronyms.....	87
8	References	88

9 Appendix A: Experiences with XP in the industry i
10 Appendix B: XP resources vii

Figures

- Figure 2.1: The PDCA improvement cycle [12]. 14
- Figure 2.2: GQM hierarchical structure [13]. 16
- Figure 2.3: Degrees of process change in process improvement and –innovation [14]. 18
- Figure 2.4: A two level feedback system [25]. 20
- Figure 3.1: Stand-up meeting at the storyboard [38]. 31
- Figure 3.2: Workflow queues [38]. 32
- Figure 3.3: Programmer time [50]. 35
- Figure 3.4: Code defects [50]. 35
- Figure 4.1: Office environment. 43
- Figure 4.2: Arrangements around the developers. 44
- Figure 4.3: The XP practices on the wall. 46
- Figure 4.4: Pair programming. 46
- Figure 4.5: XP practices with experiences and suggestions. 48
- Figure 4.6: Feedback meeting. 49
- Figure 4.7: Discussion on a feedback meeting. 49
- Figure 4.8: Post-it notes from a KJ session. 50
- Figure 4.9: Man-hours of refactoring per release in CS1. 53
- Figure 4.10: Number of man-hours of pair programming per user story in CS1. 54
- Figure 4.11: Number of completed user stories in CS1. 54
- Figure 4.12: Suggested and introduced improvements to the XP practices in CS1. 56
- Figure 4.13: Number of suggested improvements per iteration in CS1. 57
- Figure 4.14: Assessment of how easy it is to make automated test for Java in CS1. 58
- Figure 4.15: Assessment of how easy it was to make automated test fir JSP and HTML in CS1. 59
- Figure 4.16: Number of lines of code per release in CS1. 60
- Figure 5.1: PMA before and after a project [66]. 77

Tables

Table 2.1: The XP Series.....	6
Table 2.2: ISO-9000 standards.....	24
Table 3.1: Experiences with the XP practices in the industry.....	30
Table 4.1: The daily log in Case Study 1 (CS1).	41
Table 4.2: Measurements after each iteration in CS1.	42
Table 4.3: Measurements after each release in CS1.	42
Table 4.4: Distribution of the man-hours used in CS1.....	52
Table 4.5: Categorical description of the iterations in CS1.....	53
Table 4.6: Experiences with the XP practices in CS1.	55
Table 4.7: Abbreviations of the XP practices.	57
Table 4.8: Experiences with the XP practices in CS2.	64

1 Introduction

This chapter will give an overall introduction to the thesis. As the name of the thesis indicates – *How to Combine Software Process Improvements and Quality Assurance with Extreme Programming* – this is an investigation of how to do process improvement and quality assurance when we are using the development method Extreme Programming.

We will give an introduction to the subject and explain the problem to solve in section 1. 1. In section 1. 2 we will present the method used to solve the problem and in section 1. 3 we will give an overview of the thesis.

1. 1 Introduction to the subject and problem

Extreme Programming (XP) is a development method that was introduced to the software community around five years ago. It has since then achieved a great interest in the international information technology community. The increasingly need for rapid changes in projects and their environment motivated the introduction of XP and other agile methodologies. More traditional methods – like the waterfall method – focus on predictability trough extensive planning and documentation of all parts of a project, and rapid changes does not fit them. XP handles the rapid pace of changes by doing the development in short iterations and no planning beyond these iterations. Instead of putting all planning and creativity at the start of a project, it is possible to be creative throughout the project lifecycle and change requirements, as it is needed. There are no or little documentation and the customer is included in the development team to be able to follow the project evolvment and decide what the team shall work with during the iterations.

Software process improvement (SPI) is generally about learning how to work better. The goal is to create a process or an environment where people can do a better job. How could we achieve this goal when we are using XP? What does the rapid changing environment influence on how we should do SPI in XP? The challenges for combining SPI with XP must be described and solutions must be presented to answer these questions.

Quality assurance (QA) goes beyond the process and focus more on the product. The general goal is to control that the system quality is taken care of and that the customer gets satisfied with the resulting product. How could this be possible when we are using XP? We have no strict project plan and no up-front requirements for the system to be developed. The challenges for combining QA with XP must be also be described together with the solutions, to understand how QA can be combined with XP.

This thesis will investigate these issues, present challenges and solutions, and give suggestions to how both SPI and QA effectively can be combined with XP.

1.2 The method used to solve the problem

To solve our problem of combining SPI and QA with XP we will use the following strategy. We will start with a presentation of the background theory of XP, SPI, and QA. Then some experiences with XP will be described. To be able to observe XP in real projects, we will present two case studies. Then we will discuss our problem and present solutions. Finally we will conclude with our findings.

The background theory will give a basis and motivation for the thesis. A state-of-the-art of XP will be given. We will also outline the agile nature of XP and how this method is planned and carried out. To understand XP's relation to SPI and QA we will also give short introduction to these areas.

Experiences are important. We will see how XP are used in the industry and in academia. By learning about these experiences we can be able to understand the positive and negative sides of XP. There are done some research of XP and its influence on learning and quality, which will be discussed. I have also been in contact with some companies and gathered more experience with the use and introduction of XP.

To be able to access quantitative and qualitative data about XP we will do two case studies. The first case study includes a detailed analysis done throughout a XP project, while the second case study is analysed after a XP project. My participation in the first case study will be to observe and contribute. This case study will be used to try out some ideas for SPI and QA in a XP project. The observations and results from these case studies will give both ideas and support for the problem to solve in this thesis.

The problem to solve in this thesis will finally be discussed and findings will be presented. The discussion will summarise our theories about how SPI and QA can be combined with XP. Based on theory, experiences, and empirical studies, we will also present some new ideas for SPI and QA combined with XP.

1.3 Overview of the thesis

This thesis is divided into eight chapters and two appendices. In addition to this introductory part, the four following chapters are describing theory, experiences, observations, and discussions about XP, SPI and QA and the combination of these. Then the conclusion and suggestions for further work are given. Abbreviations, acronyms, and references are then listed. At the end the appendices include XP experiences and further XP resources.

Chapter 1, Introduction.

Chapter 2, XP, SPI, and QA: Background and motivation. State-of-the-art of XP will be presented together with introduction to the areas of SPI and QA. All theory needed as basis for this thesis is presented.

Chapter 3, Experiences with XP. Experiences with XP in the industry is investigated. Research and use of XP in academia is presented. Own contact with companies is described.

Chapter 4, Case studies. Case studies of two XP projects are described. The Goal-question-Metric method is used in the first case study and Post Mortem Analysis is used to evaluate both.

Chapter 5, XP, SPI, and QA: How to combine them. A discussion of the problem to solve is done. Suggestions and findings are presented.

Chapter 6, Conclusion and further work.

Chapter 7, Abbreviations and acronyms.

Chapter 8, References. A list of literature and other sources.

Appendix A: Experiences with XP in the industry. Description of experiences with the XP practices.

Appendix B: XP resources. A list of XP resources on the Internet.

2 XP, SPI, and QA: Background and motivation

This chapter will present the theory used in this thesis. We will give a state-of-the-art description of Extreme Programming (XP) and give introductions to the areas of software process improvements (SPI) and quality assurance (QA). By learning about these areas we will be able to understand what XP is and how it should work. Further we will learn about the central focuses that are important for SPI and QA. The theory here will be the background and motivation for the rest of the thesis.

XP is described in section 2. 1, SPI is described in section 2. 2, and QA is described in section 2. 3.

2. 1 Extreme Programming

During the last five years, a new way of developing software has appeared in the software community. It is called Extreme Programming (XP). Since its birth at the C3 project at Chrysler around 1997 [1] / [2], the method has gained acceptance, the popularity has grown, and it is used more and more in the industry and academia today¹.

Section 2.1.1 will explain what XP is and describe its history and philosophy. The values, principles, activities, and practices are described in section 2.1.2. How XP is planned and carried out is described in section 2.1.3. Lightweight and agile methodologies – which XP is a part of – will be described in section 2.1.4.

2.1.1 What is XP – the history and the philosophy

XP is a lightweight methodology for small- to medium-sized teams developing software in the face of vague or rapidly changing requirements.

- Kent Beck [3]

The quote from Kent Beck gives a good description of what XP is. And for a good reason: He was the main founder of XP. XP is a development method for projects where not all requirements and features of a system can be defined and planned before the development starts. By taking the customer closer to the developers and only plan for short iterations, XP can handle rapid changes in requirements and environment. We can say the development team is able to be creative all along with the project and not only in an up-front requirement phase, as many other methods. We will in the following describe the history of XP in 2.1.1.1 and present the philosophy of XP in 2.1.1.2.

2.1.1.1 Root, C3 project, and The XP Series

The roots of XP lay in the Smalltalk community, and in particular the close collaboration of Kent Beck and Ward Cunningham in the late 1980's. Both of them refined their practices on numerous projects during the early 90's, extending their ideas of a software development approach that was both adaptive and people-oriented.

The crucial step from informal practice to a methodology occurred in the spring of 1996. Kent was asked to review the progress of a payroll project for Chrysler. The project was being carried out in Smalltalk by a contracting company, and was in trouble. Due to low quality of the code base, Kent recommended throwing out the entire code base and starting from scratch under his leadership. Along with (just mentioning a few) Ron Jeffries, Martin Fowler, and Chet Hendrickson, the result was the Chrysler C3 project (Chrysler Comprehensive Compensation), which since became the early flagship and training ground for XP [4, p.10]. The experience was published in the 'Embracing Change with Extreme Programming' paper in 1999 and the book 'Extreme Programming Explained: Embrace Change' in 2000. This book became the first publication of 'The XP Series', which is now counting six more titles, as this thesis is written – see Table 2.1.

¹ See chapter 3 for a description of the usage of - and experience with XP.

Year	Title	Description
2000	Extreme Programming Explained	The XP manifesto
2001	Planning Extreme Programming	Planning projects with XP
2001	Extreme Programming Installed	Get XP up and running your organisation
2001	Extreme Programming Examined	Best XP practices as presented and analysed at the recent XP conference
2001	Extreme Programming in Practice	Learn from the chronicle of an XP project
2002	Extreme Programming Explored	Best XP practices for developers
2002	Extreme Programming Applied	Delves deeper into XP Theory

Table 2.1: The XP Series.

2.1.1.2 Philosophy

The philosophy of XP is to develop software in a light, efficient, low-risk, flexible, predictable, scientific, and fun way [3]. The idea is to focus more on programming and testing, and less on documentation. With a close collaboration with the customer, the software is developed in short increments and release cycles. The direct and constant communication between the developers and customer helps the developers to understand what the customer want and the customer will always know how the software solution is progressing. One key to the strategy is to split business and technological responsibility. The project is driven by the customers business decisions, based on what give most in return. The developers are focused on technical solutions and must inform the customers about cost and risk.

The XP values, described in section 2.1.2.1 below, give more insight to the foundation that the XP philosophy is based on.

2.1.2 The values, principles, activities and practices

This section will describe the values, principles, activities, and practices of XP. These are the main artefacts of XP and are related to how XP should be performed. Most of the artefacts we find in XP are not new inventions, but put together, they represent a new way of developing software. XP is put forth as a methodology or a process description, trying to solve many of the problems that are common in the software industry.

Section 2.1.2.1 describes the four values that XP is based on, section 2.1.2.2 describes the principles and activities that concretise the XP values, and section 2.1.2.3 describes the twelve practices that tell you how XP should be brought into practice.

2.1.2.1 Four values

The central criteria for success with XP are a set of values. As shared goals for the team, they serve as indicators to tell if they are going in the right direction. The four values of XP are communication, simplicity, feedback, and courage [3, c.7]:

- **Communication:** Everyone involved are communicating on different levels to make sure that information, progress, and requests always are present. XP aims to keep the right communication flowing by employing many practices that can't be done without communication.
- **Simplicity:** The simplest thing that could possibly work (TSTCPW). You should only choose the simplest solution that is required today, and leave further advancements until later, when they are required.
- **Feedback:** Concrete feedback about the current state of the system is central, and this feedback works at different time scales: minutes/days/months. The essence is that developers get feedback about the state of their system and get feedback about the customer's decisions, and the customer get feedback on their requirements and the progress of the system. Then redirections in development and changes can be performed efficiently.
- **Courage:** XP encourages the taking of drastic and unexpected decisions and actions. Without the courage to do what is right and what should be done, it can be difficult to achieve the desired goals. Even if it involves throwing code away or breaking tests that have already been running and fixing the flaw.

A deeper value is also mentioned, one that lies below the surface of the other four – respect. XP is depending on members of a team caring about each other and what they are doing.

These values are important parts of XP, and used together they will serve as a good foundation for a project success. However, to be able to make them a natural habit, we must be more concrete on how they are achieved. This concretisation is done in the following sections.

2.1.2.2 Principles and activities

The XP values are distilled into concrete principles, which determine XP practices. The principles will help us to choose between alternatives. We will prefer an alternative that meets the principles more fully to one that doesn't. The XP basic principles are [3, c.8]:

- **Rapid feedback:** Any learning on how best to design, implement, and test the system must be fed back in seconds or minutes instead of days, weeks, or months.
- **Assume simplicity:** Treat every problem as though it could be solved the simplest way possible.
- **Incremental change:** Solve any problem with a series of small changes that make a difference. Big changes made all at once will probably not work.

- **Embrace change:** The best strategy is the one that preserves the most options while actually solving your most pressing problem.
- **Quality work:** You should enjoy your work: this is how you produce good software.

There are also some less central principles in [3, c.8], but the ones mentioned here are most important. Together they help us decide what to do in a specific situation. The four XP values and the derived principles are a basic for building a discipline of software development practices. However, before practices are identified, XP gives a list of activities, which are derived from the XP principles. The four basic XP activities are [3, c.9]:

- **Coding:** The basic activity of XP. The code is the central artefact and should be used for everything: learn about the system, communicate solutions, describe algorithms, express tests, and so on.
- **Testing:** Is an important activity. The tests should cover everything that could go wrong with the system, indicate if we have achieved our goal, and should be automated. Unit tests are written by the programmers, in order to prove that programs work the way they are expected to. Functional tests are written by the customer to convince themselves that a system as a whole works as it is expected to.
- **Listening:** XP develops rules that encourage structured communication and discourage communication that does not help: just saying “everybody should listen to each other” doesn’t help much.
- **Designing:** The activity of designing is part of the daily business of all programmers in XP in the midst of their coding. It is based on the concept “that a change of one part of the system does not always require a change in another part of the system”. In a good design every piece of logic in the system has only one home, logic is put near the data it operates on, and allows extension of the system with changes in only one place.

With these principles and practices in mind, it is time to learn the practices that constitute XP.

2.1.2.3 The twelve practices

The twelve XP practices is a framework for how things should be done. It is important to remember that no practice stands alone; it requires other practices to keep it in balance. The twelve practices in XP are [3, c.10]:

- **Planning Game:** Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- **Small releases:** Put a simple system into production quickly, and then release new versions on a short cycle.
- **Metaphor:** Guide all development with a simple shared story of how the whole system works.

- **Simple design:** The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- **Testing:** Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- **Refactoring:** Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplicity, or add flexibility.
- **Pair programming:** All production code is written with two programmers at one machine.
- **Collective ownership:** Anyone can change any code anywhere in the system at any time.
- **Continuous integration:** Integrate and build the system many times a day, every time a task is completed.
- **40-hour week:** Work no more than 40 hours a week as a rule. Never work overtime two weeks in a row.
- **On-site customer:** Include a real, live user on the team, available full-time to answer questions.
- **Coding standards:** Programmers write all code in accordance with rules emphasising communication through the code.

These practices are telling you how you should do XP. Moreover, by being part of an “extreme” team, you sign up to follow the rules. However, they’re just rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change. How XP has been carried out and experiences with XP will be described in chapter 3, and a discussion on how to improve and change the XP practices are done in chapter 5.

2.1.3 How XP is planned and carried out

The planning of a XP project is about deciding how the values, principles, activities, and practices described above should be put to life. In [3] and especially in [54] a description of how XP is planned and how a XP project is carried out is given. We will highlight some of these descriptions here.

The strategies of the XP project management are to decentralise the decision-making. By having principles as accepted responsibility, quality work, incremental change, local adoption, travel light, and honest measurements will give the developers the ability to decide and control their solutions. The managers have the Planning game practice to control the project, are using metrics as the basic XP management tool. The development strategy is radically different from the traditional view of the development process. Its motto is that in XP all activities are centred around programming.

There is also an idea of an “ideal” XP project. The lifecycle of this consist of a short initial development phase followed by a long-term simultaneous production support and refinement [3, c.21]:

- **Exploration:** Prepare production, practice writing user stories, estimate and experiment with technology and programming tasks.
- **Planning:** Run the Planning game practice; agree the smallest set of stories to be completed.

- **Iterations to first release:** Produce a set of functional test cases that should run at the end of the iteration.
- **Productionising:** You need one-week iterations, certify that the software is ready for production; implement a new test and tune the system.
- **Maintenance:** Simultaneously produce new functionality and keep existing system running, refactor is necessary or migrate to a new technology; experiment with new architectural ideas.
- **Death:** XP project ceases to exist, new functionality do not have to be added or can be delivered.

The balance between business decisions and technical decisions are important. Business people and technical people should only do them alone, respectively. Business decisions cover dates, scope, and priorities, and are done by the customer. Technical decisions cover technical solutions and estimates, and are done by the developers.

The first plan and the first estimates are difficult. The team learn to do this better when they got more experienced. By using the idea of Yesterdays Weather – earlier experiences [54, c.8] – estimates on user stories and team velocity will stabilise to be more accurate and correct. The team velocity is telling how much the team can do in one iteration, based on ideal time or other measuring units.

Further, the best advice for XP teams is to have the described XP theory in mind and start off with these given rules. Then the team will get to know XP better and they will hopefully improve and adjust their way of doing XP along with the project. This process philosophy is associable to the type of methodology that XP is, namely agile and lightweight. The nature of these kinds of methods is described in the next section.

2.1.4 An agile and lightweight methodology

XP is a unique way of developing software, but the philosophy behind it is shared with some other development methods. The similarities are called agile development or a lightweight methodology. Beside XP, these methodologies are: The Crystal Methods, Adaptive Software Development (ASD), Open Source, Scrum, Feature Driven Development (FDD), Agile Modelling, Dynamic System Development Methodology (DSDM), Lean Development, Pragmatic Programming, and possibly some more. See [4] for further introductions to some of them. Common to these methods are their respond to former and more heavy methodologies and the name 'light' describe the position well.

It is important to understand what kind of methodology XP is, what characterise it, and have knowledge to the debate around the new methodologies. Subsection 2.1.4.1 will therefore tell more about the transition from earlier methodologies to new ones, subsection 2.1.4.2 describes the debate that this transition has raised, and subsection 2.1.4.3s describes how these new methods are adaptive and suited for constant change.

2.1.4.1 From heavy to light

Software development can be chaotic and projects have often run out of control. Methodologies are used to manage software projects. Traditional methodologies want to achieve a disciplined process with the aim of making software development more predictable and efficient. These methodologies have focused on detailed planning, with upfront analysis and design to determine system requirements. Many of them are based on the waterfall model where a project goes through phases from analysis through design and implementation to testing. Detailed documentation is also required to support these methodologies.

By being both extensive and detailed, these methods are criticised and often referred to as heavy methodologies. The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down.

As a reaction to these methodologies, a new group of methodologies have appeared in the last years. Although there's no official name for them, they are referred to as light methodologies – signalling a clear alternative to the heavyweight methodologies. The reaction lies in being less bureaucratic when it comes to long-term planning and extensive documentation. These new methods attempt to strike a compromise between no process and too much process, providing just enough process to gain a reasonable payoff. In short, they are more agile¹ by nature. The most immediate difference is that they are less document-oriented, usually emphasising a smaller amount of documentation for a give task. In many ways they are rather code-oriented: following a route that says that the key part of documentation is source code.

In [4] the deeper differences between heavyweight and lightweight methodologies are described:

- *Light methods are adaptive rather than predictive.* Heavy methods tend to try to plan out a large part of the software process in great detail for a span of time. This works well until things change. Therefore, their nature is to resist change. The light methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.
- *Light methods are people-oriented rather than process-oriented.* They explicitly make a point of trying to work with peoples' nature rather than against them and to emphasise that software development should be an enjoyable activity.

The differences clearly emphasize the values for light and agile methodologies. These values are more formally proclaimed in the Agile Manifesto, which is described in the next section.

¹ What agile means, is described in the next section 2.1.4.3.

2.1.4.2 The Agile Manifesto and the ongoing debate

In February 2001 several people sharing the same ideas on how software development should be done, came together and discussed this topic. They were all advocates for different new methods, and the creator of XP was also present. They set up and agreed on the Manifesto for Agile Software Development [5]. The manifesto is summarised with the following values:

- *Individuals and interactions* over processes and tools,
- *Working software* over comprehensive documentation,
- *Customer collaboration* over contract negotiation,
- *Responding to change* over following a plan.

That is, while there is value in the items on the right, the items on the left are valued more.

So what does agile mean? Dictionary definitions describe agility as nimble, quick, responsive, and active. In [6] they say agility, ultimately, is about creating change and responding rapidly to change. To be able to do this, an agile process requires responsive people and organisations. This is highlighted in [7], and it is central to focus on the talents and skills of individuals and mould process to specific people and teams, not the other way around.

A dominant idea in agile development is that the team can be more effective in responding to change if it can

- 1) reduce the cost of moving information between people, and
- 2) reduce the elapsed time between making a decision and seeing the consequences of that decision.

To solve (1), the agile team works to

- place people physically closer,
- replace documentation with talking in person and at whiteboards, and
- improve the team's amicability – its sense of community and morale – so that people are more inclined to relay valuable information quickly.

To solve (2), the agile team

- makes user experts available to the team or, even better, part of the team and
- works incrementally.

Not surprisingly there has been reactions to this new trend towards agility. There has been an ongoing debate on whether it is agile or plan-driven methods that is best for software development. The debate is summarised in [8] and the conclusion is that everything depends on the specific situation. The challenge is to find the combination that effectively supports the current project.

2.1.4.3 A self-adaptive process

Agile and lightweight methods are so far described as adaptive in the context of meeting the changing requirements from the customers. These methods are, however also self-adaptive. This means that the process is changing over time. A project that begins using an adaptive process won't have the same process a year later. Over time, the team will find what works for them, and alter the process to fit [4].

This self-adoption is more marked in some lightweight methods than others. It is a highlighted part of methods like ASD and Crystal, and a more indirect part in a method like XP. In XP it is suggested to follow the practices by the book for several iterations before adapting it. Nevertheless, as said in [9], if you're still doing XP in iteration six the way you were in the first iteration you're not doing XP.

The first part of self-adaptivity is regular reviews of the process. This can be done daily or within each iteration. In [10] the following questions are suggested to raise after each iteration:

- What did we do well?
- What have we learned?
- What can we do better?
- What puzzles us?

These questions will lead to ideas to change the process for the next iteration. In this way a process that runs into some kind of problems can improve as the project goes on, adapting the process to better suit the team that uses it.

Secondly, to deepen the process of self-adaptivity, it is also suggested to do a more formal review at the end of the project. By doing this, the experience adapted in the project also can provide learning beyond the team to the whole organisation. A broader description of such reviews is done chapter 2. 2. This kind of review is called Post Mortem Analysis (PMA) and is one method used in process improvement work.

Because these agile and lightweight methodologies are focused on people and how they work and communicate best, rather than being focused on a strict process, the methodologies can be self-adaptive. The combination of both being able to react and adapt to changes externally and internally are one of their advantages.

2.2 Software Process Improvement

This section will give an introduction to software process improvement (SPI) and related issues. SPI is important because most or all development processes have parts that can be improved. If we are able to improve our process, we will also be able deliver results faster and better.

We will define SPI in section 2.2.1, describe the basic improvement principles in 2.2.2, present methods for doing measures and collecting information in section 2.2.3, describe the relation between process improvement and quality in section 2.2.4, explain the difference between process improvement and -innovation in section 2.2.5, present the theory about rapid process improvement in section 2.2.6, and describe how personal and organisational learning affects the process improvement in 2.2.7.

2.2.1 What is SPI?

Software Process Improvement (SPI) and process improvement are in general terms about learning how to work better. The idea is to create a process or an environment where people can do a better job. The desired goal for a company is to be able to develop products that are cheaper and better, and that takes less time do deliver. How to achieve these goals is dependent on many factors, but common to most of the methods is that they are based on systematic and continuous improvements of the development process [11, c.2]. The Plan-Do-Check-Act (PDCA) cycle constitute this approach.

The PDCA cycle – see Figure 2.1 – developed by Walter Shewhart in the 1920s, provides the basic philosophy for a disciplined, cyclic approach to continuous improvement. PDCA is also referred to as the “scientific method” and the “Sheward cycle”. William Edwards Deming later introduced the cycle in his work with the Japanese industry in the post-war period.

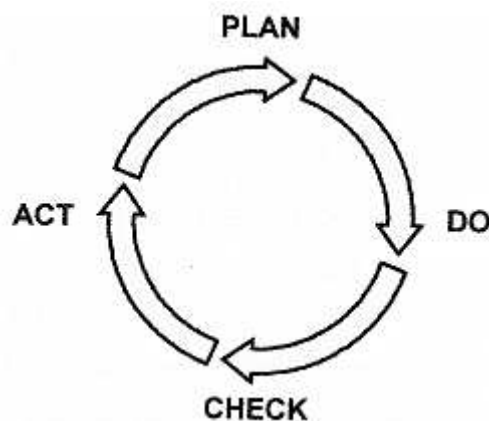


Figure 2.1: The PDCA improvement cycle [12].

The parts in the PDCA cycle are described as:

- Plan – *Plan activities*, i.e. define the problem and state the improvement objectives.
- Do – *Implement the plan*, i.e. identify possible problem causes, establish baselines, and test changes.
- Check – *Check the result*, i.e. collect and evaluate data.
- Act – *Improve the process*, i.e. implement system change and determine effectiveness.

2.2.2 Improvement principles and models

Two basic approaches to SPI have emerged. These are referred to as the “benchmarking” and the “analytic” approach, or “top-down” and “bottom-up” process improvement. While the benchmarking approach seeks to meet a “best practice” defined model, the analytical approach assumes that the organisation’s individual goals and experiences must drive the process change. The most well known benchmarking approach to SPI is the IDEAL model [12, p.28], and the most prominent example of the analytic approach to SPI is the Quality Improvement Paradigm (QIP) [12, p.29].

2.2.3 Measurement and information gathering

To support the improvement process, doing measures, and collection information, there exist several methods. Most relevant to this thesis is GQM (Goal-Question-Metric), PMA (Post mortem Analysis), and KJ (Kawakita Jiro). We will give short introductions to these methods here.

2.2.3.1 Goal-Question-Metric

The Goal-Question-Metric (GQM) method is a goal-oriented approach for measurements in software projects to select and implement relevant measures and indicators. The strength of this method lies in its ability to collect, organise, document, and exploit the knowledge and experience that exist in a company [11, s.11.4].

The GQM model has three levels [13]:

- 1) **Conceptual level (goal):** A goal is defined with respect to various models of quality. From various points of view, relative to a particular environment. Typical objects of measurements are products, processes, and resources.
- 2) **Operational level (question):** A set of questions is used to characterise the way the assessment/achievement of a specific goal is going to be performed, based on some characterising model. Questions try to characterise the object of measurement with respect to a selected quality issue, and to determine its quality from the selected viewpoint.

- 3) **Quantitative level (metric):** A set of data is associated with every question in order to answer it in a quantitative way. The data can be either objective or subjective.

A GQM model is a hierarchical structure starting with a goal as shown in Figure 2.2. The goal is subsequently refined into a set of questions, and each question is then refined into metrics. The metrics reflect the actual data needed to answer the questions. The same metric can be used in order to answer different questions under the same goal.

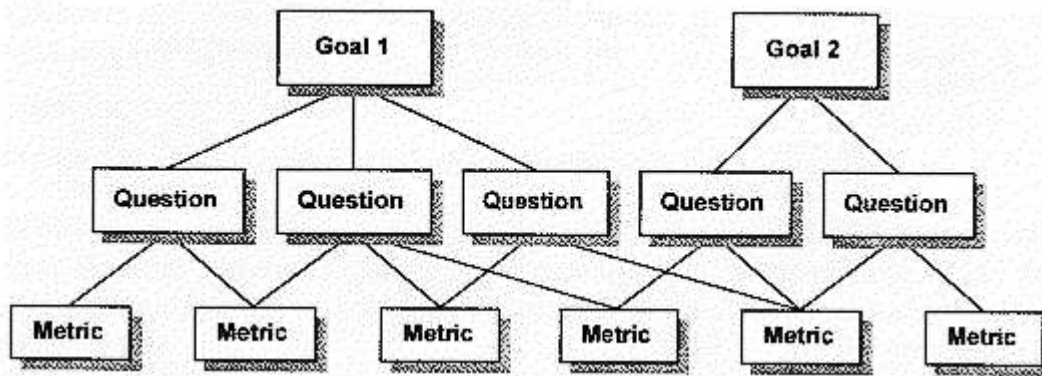


Figure 2.2: GQM hierarchical structure [13].

The GQM hierarchy is organised in a GQM abstraction sheet. This is used in the first case study in chapter 4. 1 and described further in [11, s.11.4].

2.2.3.2 Post Mortem Analysis

Post Mortem Analysis (PMA) is a review done in the end of a project to sum up good and bad experiences. The purpose is to reflect and learn from the past project to help improve the team's process on future projects. Information that is gathered during a PMA is coming from the participants themselves and is therefore suited for collection of information that is not documented in other ways.

A PMA can be arranged in many ways [14] and we will therefore not give concrete descriptions of PMA here. However, PMA will be used and described during the project in the first case study in chapter 4.

2.2.3.3 KJ / Affinity diagrams

The KJ method is named after the Japanese anthropologist Kawakita Jiro who developed it. The KJ method, also referred to as affinity diagrams, is an organised and focused brainstorming technique [15]. KJ is suited for define and clear out imprecise or unclear problems by organising qualitative data as ideas or thoughts.

KJ is performed in the following steps [16, c.3]:

- 1) *Problem definition.* Formulate the issue or problem to discuss.
- 2) *Brainstorming.* The participants write down ideas and thoughts related to the issue on small notes. The notes are put on a board on the wall
- 3) *“Clean” the ideas.* The notes are explained. Rewrite unclear notes.
- 4) *Group the ideas.* Everyone take part in grouping ideas in belonging groups.
- 5) *Formulate names of the groups.* A name that describe the ideas in the group.
- 6) *Assessment.* Everyone take part in prioritising the groups. The result should be documented in some way.

The most important issues can be analysed further by Root Cause Analysis, but this is a further extension.

2.2.4 Process improvement and quality

Process improvement and quality are related to each other. Process improvement is a means for creating products with high quality [11, s.2.3]. Quality is primarily a relation between the customer and a product or service. Does the product fulfil the requirements and requests from the customer, or is the quality satisfying enough to keep the customer coming back? (See section 2. 3 for further description of quality and quality assurance). Process improvement on the other hand wants to establish processes where the employees can do a better job, in other words develop products with sufficient quality.

It is said “the quality of a software product is largely governed by the quality of the process used to develop and maintain it” [17, p.8]. Thus, the relationship between the development process and the outcome can be described as:

Quality (Process) \Rightarrow Quality (Outcome)

However, the same change to a process does not always lead to a similar change in outcome. The relationship depends on the particular context (e.g. experience, organisational size, or environmental turbulence).

2.2.5 Process improvement vs. process innovation

There are several approaches for improving the processes in an organisation. The two strategies that imply changes to the organisation and its processes are process improvement and process innovation. In Figure 2.3 the degrees of change to the software process are shown for each of these approaches.

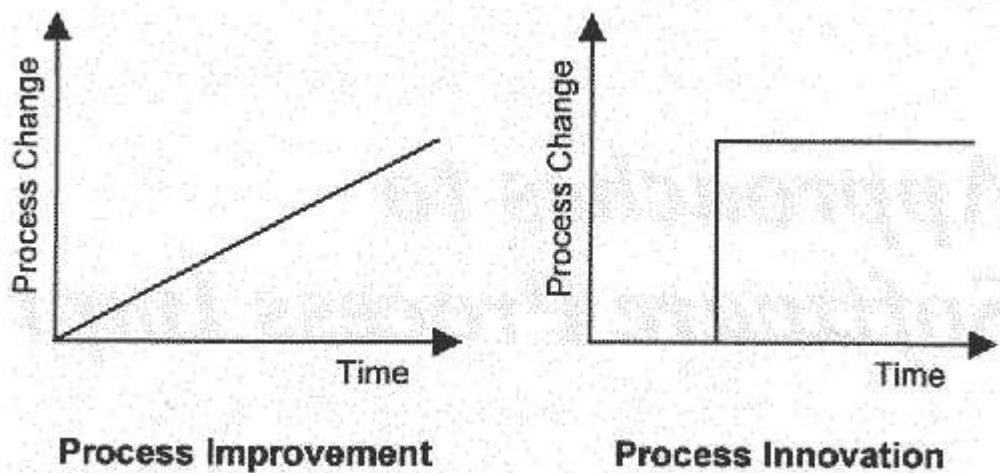


Figure 2.3: Degrees of process change in process improvement and –innovation [14].

While process improvement is a strategy that involves continuous or incremental changes, process innovation is a strategy that involves radical changes with entire replacements of processes.

2.2.6 Rapid Process Improvement

The business environment and software development methods are in many fields changing rather quickly and in unpredictable ways. Software development practices are changing to meet changing business needs. These changes may arise from market fluctuations, mergers and re-organisations, changes in business strategy, changes of management, and changing technology. Process improvement must also change to meet the needs of this new environment. A look-ahead of more than two years is of limited value for SPI because it is rarely clear what will be required then. A more appropriate approach to process improvement called Rapid Process Improvement (RPI) has therefore been developed [18]. RPI deliver many simple improvements – each in a short time box –, evaluate actual results, learn from fast failures, and build on success.

The Rapid process improvement approach is intended to deliver:

- results rapidly and cheaply, allowing benefits to be accrued as soon as possible
- learning from mistakes without incurring major costs
- process improvements to respond to changes in the business environment

RPI has a number of characteristics that are required and built in:

- *Clearly focused on solving real problems.* This may be a technical or business problem that affects the developers.
- *Desired results stated explicitly and preferably measurably.* What will the situation be like when the problem is resolved?
- *Speed.* Scale and partition tasks to deliver useful results in days or weeks, not months.
- *Results driven.* Work to achieve results rapidly and then act on the results. Do not track activity, track results and evaluate them.
- *Use a model to guide the work but do not be driven by it.*
- *Highly visible.* The work being performed should be visible to those affected, not just the results.
- *Owned locally.* Those affected by the changes own and control the changes. Process improvement is rarely successful if done to people; it is done by people. Process improvement is about changing behaviour and expectations and those affected must be involved and in control.
- *Local Accountability.* Results (good and bad) should be accountable locally rather than be remote committees or management steering groups.
- *Opportunistic.* If there is a chance to make improvements or fix a problem, do it now!

To perform rapid process improvement activities that exhibit the characteristics described above the, right tools and techniques for process improvements are required. These are described in [18, c.4].

2.2.7 Personal and organisational learning

To succeed with process improvement and SPI, some sort of personal and organisational learning must take place. It is said that “continuous improvement programs require a commitment to learning¹”. In the ever-changing technology and business environment we have today, it is also needed to learn how to constantly monitor, evaluate, and align with the dynamics and complexities of this environment. Learning in this context could therefore include both the individuals’ capacity to acquire, transfer, and interpret knowledge with the collective group of the organisation, and the ability to utilise this learning in a work-based environment [19].

Both in [20] and [21] it is stressed that organisational learning is closely connected to personal learning. Learning in individuals can be transformed into more general improvements that will lead to success and prosperity for organisations. [22] further contends that learning and conflict, and hence further learning, can be seen as inevitable in organisations as it is in individuals. Organisational learning is as natural as individual learning, where organisations attempt to adapt and survive in a competitive and uncertain world. There are many ways of learning [23]: congenital learning, experimental learning, vicarious learning, grafting, and searching and noticing.

¹ D. Garvin. Building a Learning Organisation. Harvard Business Review, pp. 78-93, July / August 1993.

In order for information to be interpreted it must be disseminated. The dissemination of information and knowledge can take many forms. Organisational memory is the repository where knowledge is stored for future use. [24] contend that a lot of learning also takes place in more informal ways. Often, learning in an organisation takes place by members sharing stories or anecdotes of actual work practice as opposed to what is mentioned in formal job descriptions or procedure manuals. The major challenge for organisations is to interpret information and create an organisational memory that is easily accessible [23]. [20] uses the following types of organisational learning: single loop, double loop, and deuterio learning. Single-loop learning occurs when errors are detected and corrected and the firm carries on with its present policies and goals – see Figure 2.4.

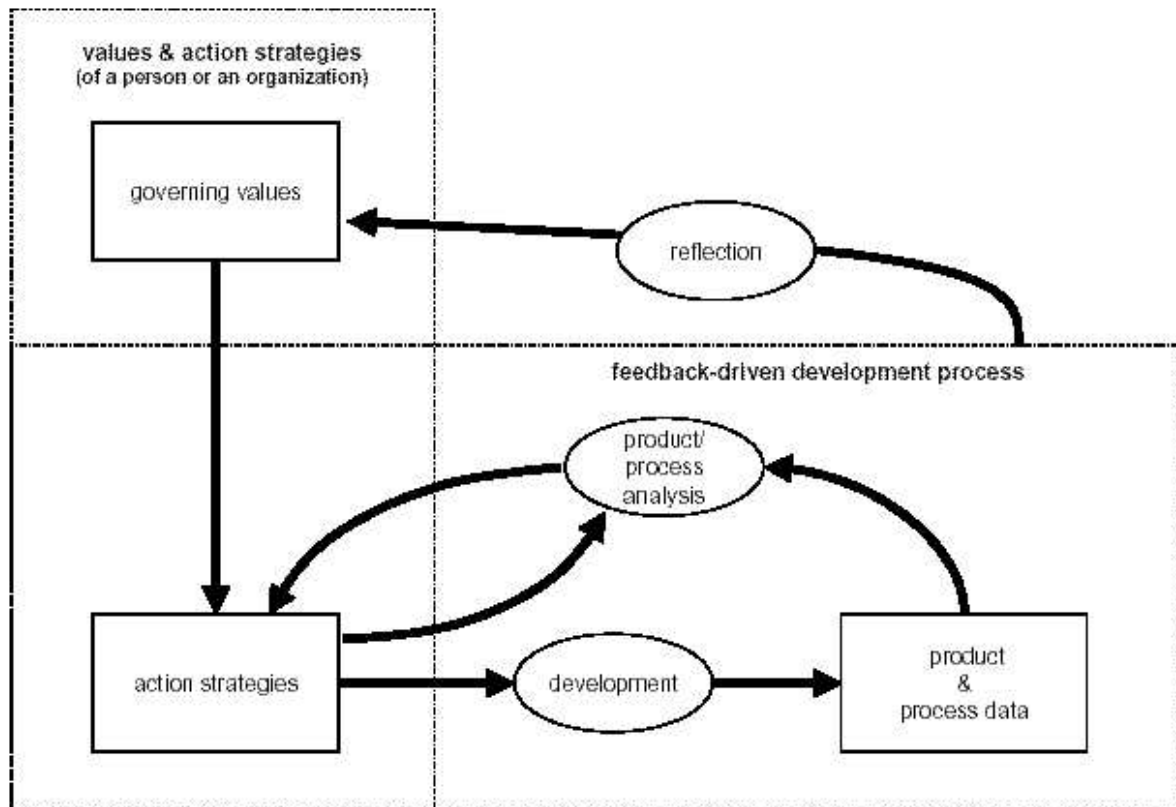


Figure 2.4: A two level feedback system [25].

When an organisation carries out single- and double-loop learning they are considered to be deuterio learning, or learning how to learn.

2.3 Quality Assurance

To make sure that projects and product quality are under some degree of control, most companies are doing some sort of quality assurance (QA). This chapter will give an introduction to QA and what it implies in the context of software development.

QA and quality is defined in section 2.3.1. Total Quality Management (TQM) is assumed as an overall foundation in this work and will be described in section 2.3.2. Standards are often used as yardsticks to assure quality and examples of such will be given and described in section 2.3.3.

2.3.1 What is QA?

According to the IEEE standard for software quality assurance plans [26], quality assurance (QA) is defined as follows:

- **Quality Assurance (QA):** A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.

The central goal is to make sure that the quality of the product and the process is assured. QA can both be aimed to achieve certain degree of quality of essential system attributes or to simply achieve customer satisfaction. To understand what this means, we have to define the meaning of quality.

We have all an idea of what quality is, but there are several ways of defining it. A definition previously used in the software business was "in accordance to the specification". Modern quality theories, including TQM, are putting the users of the products and services in focus of all quality work. It is up to the users to decide what good quality is and quality is defined as those features of the product or service that results in satisfied users. Quality can therefore be defined both by its own characteristics and how the users perceive these:

Quality:

- The totally of characteristics of an entity that bear its ability to satisfy stated and implied needs [27, c.4].
- Properties of a product and service that gives satisfied customers and users throughout its whole lifetime [11, s.2.3].

In [28] there are defined categories of quality characteristics. These essential product quality attributes are functionality, reliability, usability, efficiency, maintainability, and portability. See section 2.3.3 on quality standards for descriptions of these categories. [29] also focus on the user. Here it is stated that quality is relative and that quality means value to a person.

2.3.2 Total Quality Management

Total Quality Management (TQM) has its origin from Japan in the fifties, and is a management style used to lead the company towards Total Quality. Total Quality is a vision that describes the perfect company – a company with predictable technical, administrative, creative, and social processes – where all products satisfy the users requirements and expectations. Such level of perfection may not be possible to achieve in practice, but the vision shall stimulate the company to strive for continually improvement through creative achievements at all levels [11, c.9]. ISO-9000 defines TQM like this:

- *“Total Quality Management is one form of quality management which is based on the participation of all members of an organisation.”*

A quality-managed company is characterised by its focus on “quality first”. Economical goals are necessary for every company, but quality goals are more easily agreed among all the employees than the economical goals. Quality is therefore prioritised above short-term economic profit in quality-managed companies. The philosophy is that profit will follow when the quality goals are achieved.

TQM involve all aspects of quality and include a general framework based on the following areas [11, c.9] / [30, s.1.3]:

- *Customer oriented:* The focus on the customer is based on long-term relations, rather than on short-term focus on shareholders, e.g. the owners. Customer and quality first, but shareholders are not ignored.
- *Focus on the process:* To understand and improve the process is an important means to achieve quality, rather that just controlling the results.
- *Continuous improvement:* Is connected to the process and is based on the PDCA cycle. It is normally performed in three steps: standardisation, quality improvement, and quality innovation. These are described below.
- *Measurement and analysis:* TQM uses several techniques to discover, prioritise, and solve problems. The TQM techniques consist of two sets: the seven quality techniques (used to analyse and present data) and the seven management techniques (primary used in planning).
- *Human factors, planning, and management:* TQM involves building a quality culture into the entire organisation. The human factors and processes are just as important as the technical processes. Everyone is responsible and must cooperate to meet the common goals.

TQM include three main approaches or levels in a quality-managed organisation [30, p.39]. They are all part of the continuous improvement process:

- 1) *Quality maintenance (standardisation):* Involve quality management and assurance, and is about carrying out certain tasks due to prescribed standards in existing processes. It is system-oriented and represents management for “status quo”.

- 2) *Quality improvement (evolution)*: Involve small changes on existing standards, processes, and products, and requires training of quality management and problem solving. Low cost.
- 3) *Quality innovation (revolution)*: Involve larger changes in standards, processes, products, and technology, and is entirely dependent on a trustfully and creative working environment. Cost may be high.

Each of these levels is part of the continuous improvement that TQM is based on. The levels are achieved stepwise to meet the desired improvement. Standardisation is followed by improvement, which again are followed by innovation. After the innovation it is necessary to stabilise the process with standardisation.

2.3.3 Quality standards

When developing software there is always some sort of risk involved. The customer may not get the product in time or to the agreed price, the product itself may not meet the given expectations, or the product may simply not be good enough when it comes to stability or other non-functional requirements. Conformance to standards can ensure both customers and developers that the product is developed under controlled circumstances and that the quality of the resulting product is satisfying.

In this section we will introduce the ISO-9000 standards, ISO/IEC-9126, and ISO/IEC-14598. While the ISO-9000 standards primarily focus on the quality process and organisation in general, the ISO/IEC 9126 and ISO/IEC 14598 are mostly focused on software product quality and how this could be evaluated. The importance in evaluating both the process and the product is stated in [31]. The assumption that a professional development process is a prerequisite to achieve quality products is generally accepted. This was also stated in section 2.2.4 above. However, a good development process alone cannot guarantee a high-quality product. When product quality is important it is also necessary to consider the product itself. The process view and the product view should therefore complement each other.

2.3.3.1 The ISO-9000 family

In 1987 the International Standard Organization (ISO) published a set of quality standards called the ISO-9000 family (partly revised from 1994 to 2000). The ISO-9000 standards gives directions to the quality process, and are used for internal quality management purposes and external quality assurance purposes [30, c.2.2].

The ISO-9000 standards will not be described further here, but the most central standards are listed here, together with their purpose [32]:

Standard / Guideline	Purpose
ISO 9000:2000, <i>Quality management systems - Fundamentals and vocabulary</i>	Establishes a starting point for understanding the standards and defines the fundamental terms and definitions used in the ISO 9000 family which you need to avoid misunderstandings.
ISO 9001:2000, <i>Quality management systems - Requirements</i>	This is the requirement standard you use to assess your ability to meet customer and applicable regulatory requirements and thereby address customer satisfaction. It is now the only standard in the ISO 9000 family against which third-party certification can be carried out.
ISO 9004:2000, <i>Quality management systems - Guidelines for performance improvements</i>	This guideline standard provides guidance for continual improvement of your quality management system to benefit all parties through sustained customer satisfaction.

Table 2.2: ISO-9000 standards.

2.3.3.2 ISO/IEC-9126 and ISO/IEC-14598

The ISO/IEC-9126 [28] and ISO/IEC-14598 [27] standards are both focusing on software product evaluation. Used in conjunction with each other, they provide a complete framework for evaluating software product quality both objectively and quantitatively.

ISO/IEC-9126-1 suggests a hierarchical quality model with six quality characteristics and several attached sub-characteristics [28, c.5] / [31]:

- *Functionality*: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. Sub-characteristics are suitability, accuracy, interoperability, compliance, and security.
- *Reliability*: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a specified time period. Sub-characteristics are maturity, fault tolerance, and recoverability.
- *Usability*: A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users. Sub-characteristics are understandability, learnability, and operability.
- *Efficiency*: A set of attributes that bear on the relationship between the software's performance level and the resources used under stated conditions. Sub-characteristics are time behaviour and resource behaviour.
- *Maintainability*: A set of attributes that bear on the effort needed to make specified modifications. Sub-characteristics are analysability, changeability, stability, and testability.
- *Portability*: A set of attributes that bear on the ability of software to be transferred from one environment to another. Sub-

characteristics are adaptability, installability, conformance, and replaceability.

In addition, ISO/IEC-9126-2 and ISO/IEC-9126-3, describes (respectively) external and internal metrics that should be used in software quality evaluation. The internal metrics measures the internal attributes of the software related to design and code. These are most important to development managers and can predict the values of corresponding external measures. The external metrics represent the external perspective of software quality when the software is in use. These measures apply in both the testing and operation phases, and represent the quality in terms that are relevant to users.

Based on these quality characteristics and metrics, the ISO/IEC-14598 standard is intended for use by developers, acquirers, and independent evaluators as guidance and requirements for software evaluation. It consist of six parts, where part one gives a general overview, parts two and six relate to corporate or department level evaluation management and support, while parts three, four, and five give requirements and guidance for evaluation at the project level [31] / [27]:

- **Part 1: General overview.** This part provides an overview of the other parts and explains the relationship between ISO/IEC-14598 and ISO/IEC-9126. It defines the technical terms used in the standard, contains general requirements for specification and evaluation of software quality, and clarifies the concepts. Additionally it provides a framework for evaluating the quality of all types of software products and states the requirements for methods of software product measurement and evaluation.
- **Part 2: Planning and management.** Provides requirements and guidelines for a support function responsible for the management of software product evaluation and technologies necessary for software product evaluation. The responsibilities of this support function include people motivation and education relevant to the evaluation activities, preparation of suitable evaluation documents, standards, and response to queries on evaluation technologies. The main targets for evaluation support are the software development and system integration projects, which include software acquisition, both at a project and organisational level.
- **Part 3: Process for developers.** Provides requirements and recommendations for the practical implementation of software product evaluation when the evaluation is conducted in parallel with development and carried out by the developer. The evaluation process described defines the activities needed to analyse evaluation requirements: specify, design, and perform evaluation actions; and conclude the evaluation of any software product. The evaluation process is designed to be used concurrently with the development. It must be synchronised with the software development process and the entities evaluated as they are delivered.
- **Part 4: Process for acquirers.** Contains requirements, recommendations, and guidelines for the systematic measurement, assessment, and evaluation of software product quality during acquisition of off-the-shelf software products, custom software products, or modifications to existing software products. The evaluation process helps meet the objectives of deciding on the acceptance of a single product or selecting a

product. The evaluation process may be tailored to the nature and integrity level of the application. It is also flexible enough to cost-effectively accommodate the wide range of forms and uses of software.

- **Part 5: Process for evaluators.** Provides requirements and recommendations for the practical implementation of software product evaluation when several parties need to understand, accept, and trust evaluation results. The process defines the activities needed to analyse evaluation requirements; specify, design, and perform evaluation actions; and conclude the evaluation of any kind of software product. The evaluation process may be used to evaluate already existing products, provided that the needed product components are available, or to evaluate products in development. This part may be used by testing laboratories when providing software product evaluation services.
- **Part 6: Documentation of evaluation modules.** Describes the structure and content of an evaluation module. An evaluation module is a package of evaluation technology for a specific software quality characteristics or sub-characteristics. The package includes descriptions of evaluation methods and techniques, inputs to be evaluated, data to be measured and collected, and supporting procedures and tools. This part should be used by testing laboratories and research institutes when developing evaluation modules.

3 Experiences with XP

Since the birth of XP at the C3 project around 1997¹, XP has gained wide attention and popularity, as well as scepticism and critics. To understand XP more deeply and to see its positive and negative sides in practical use, this chapter will describe some experiences with introducing and usage of XP in different kinds of companies, projects, and academic environments. Through these experiences we will be able get more insight in what parts of XP that are more troubled to adopt than others and how such difficulties could be solved.

We also want to highlight the importance of sharing experiences. Learning from the success and failures of others is a quick way to learn and enlarge our horizon [33]. By describing actual situations together with the problems that occurred and how these were worked out, we are able to avoid doing all the possible failures again and again. It is also important, however to be aware of that experiences are influenced by many factors. Things may not turn out the same way next time, because the circumstances may be different. With these words in mind, we can now turn our attention to the field of XP.

Section 3. 1 presents several experience reports and XP projects from the industry, section 3. 2 presents research and usage of XP in the academia, and section 3. 3 presents some personal contact with companies that have experiences with XP.

¹ The C3 project and the history of XP are described in section 2.1.1.

3. 1 Reports from the field

The experiences with XP that are published vary in many ways, and a direct comparison of them is not meaningful. The projects vary in size and length, the solutions vary in complexity, and the market domains, and thus the customers, vary from safety-critical to light web-applications. To make the experiences easily accessible we must do some sort of categorisation. Central to the way of performing XP is the defined practices. We will therefore look at each practice and describe the associated experiences.

Common to all the experience reports is that modifications are done to the XP-practices along the projects. To be able to relate the experiences to their circumstances, we will describe the studied projects in section 3.1.1. In section 3.1.2 we will present the experiences sorted per XP practice. We will see how the modifications were performed in section 3.1.3 and see how the use of XP has affected quality in section 3.1.4.

3.1.1 Short description of the projects

Most of the reports that are available, mainly describes how XP was introduced or adopted and not how an existing and mature XP project was improved. In other words, they describe how XP is used as a means for process innovation¹, rather than how process improvements are done in XP. Consequently, they will not give immediate answer to the main questions in this thesis, but they will all give insight to the experiences with XP and how XP could be adjusted to local environments. Nevertheless, in many cases it is the introduction to XP that is relevant. To just be aware of the problems that can occur may help a fresh team to avoid them.

We have studied five projects and some experience reports, which all are summarised here:

XP was introduced to a 50-person team during a period of one and a half year [34]. The application being built was a comprehensive enterprise resource solution for the leasing industry, and the code base were well over 500,000 lines of executable code. They experienced that XP was a valuable and effective approach to software development, given that (i) all participants are engaged and give a conscientious contribution and (ii) it must be adapted as necessary for projects that do not fit the “small team” limits recommended by its founders. The experiences from this project is also described in [35], [36], and [37]. [35] describes experiences from a developer’s view of the project, [36] describes how database administration could be combined with XP, and [37] gives detailed descriptions of one specific iteration.

XP was adopted by the IONA Technology’s Orbix Generation 3 maintenance and enhancement team, including around 30 developers during five months [38] / [39]. Each iteration was two weeks long. Some problems were experienced, but by adopting the XP practices into the local settings the result was successful. XP was experience to be a viable and successful model for teams involved in pure maintenance and enhancement of a legacy code base.

[40] describes how XP was introduced into a pilot project in a company that is developing safety-critical systems. In a company used to heavy formal documentation and completely reliable software, there was two main ‘roadblocks’ for XP: required documentation and

¹ Process innovation is described in section 2.2.5.

reviews. The solution was, aside from making use-cases as user stories and clean and simple source code, to make a high-level documentation. The documentation was needed to define product requirements, sustain technical reviews, support the systems maintainers, and avoid mismatch of interfaces between groups. Doing the reviews after each iteration, covered the other requirement. Together with the documentation the reviews could be done efficiently, and the issues that needed to be changed, were given as stories in the next iteration. The report doesn't have a complete conclusion, since it was written only after practising XP for six months. However, with the few mentioned adjustments to XP, XP seemed to give positive effects to the team and product quality.

The project described in [41] is using a lightweight methodology similar to XP, called Fast Cycle Time (FCT). They refer to their development as a practical application of XP and describe how they experienced some of the XP practices. The project is a consulting engagement implementing a new call centre application. The main conclusions are, with good communication, requirements can be gathered without an on-site customer, iterations of three weeks works fine, and a full-time quality responsible ensures that the required quality level for frequent software releases is met.

As a "saving tool" XP was used in the further development process of a troubled project [42]. To meet the delivery date, the system had many shortcomings and the go-alive system was a result of many compromises. Afterwards, the client had many features that still needed to be implemented and the system needed to be heavy refactored. To manage this situation, XP was used as a basis for handling customer needs and development practices. The usage of XP was successful, even though all they didn't adopt all the practices.

Some reports are describing experiences not related directly to specific projects [43] / [44]. Nevertheless, they give suggestions and insight to possible extensions and improvements of some of the XP practices. For that reason they also will be used in the following.

3.1.2 Experiences with the XP practices

Since the experiences that we have gathered are rather extensive we will only summarise these experiences here. A more detailed description of the different experiences can be found in Appendix A.

In Table 3.1 all the XP practices are listed together with experiences related to them. The reference to the respective experience in Appendix A are put in parenthesis after each experience here, e.g. like (1).

XP practice	Experiences
<i>The Planning Game:</i>	<ul style="list-style-type: none"> ▪ Large teams should not have too long meetings (1) [35] ▪ The iterations to plan should not be longer than two weeks (2) [35] ▪ User story estimates should be increased when complexity grows (3) [34] ▪ The customer should be “coached” the suit XP (4) [34] ▪ The customer could be split to represent the real users and the client (5) [43] ▪ The customer role can be played by in-house representatives (6) [38] ▪ User stories can be documented by use-cases (7) [40] ▪ All influence parties should take part of the Planning Game (8) [36] ▪ A requirement team may replace the customer (9) [41]
<i>Small releases:</i>	<ul style="list-style-type: none"> ▪ The releases should be kept as small as possible. Has shown to be successful from one to four weeks (1) (2) (4) (3) [35] [38] [40] [41]
<i>Metaphor:</i>	<ul style="list-style-type: none"> ▪ Natural for customer communication (1) [38] ▪ UML-diagrams were used instead of a metaphor (2) [40] ▪ Unrealistic in large systems (3) [35]
<i>Simple design:</i>	<ul style="list-style-type: none"> ▪ Continuous refactoring of design is required to keep it simple (1) [35] ▪ If you know the design must be generalised, do it immediately (2) (3) [34] [40] ▪ Could not be practiced when the system already was complex (4) [38]
<i>Testing:</i>	<ul style="list-style-type: none"> ▪ Successfully in (1) (2) (4) [38] [40] [35] ▪ Setting up unit tests was extensive (3) [4] ▪ Still problems after passing unit tests (5) [34] ▪ Difficult to write complete functional tests (6) [34] ▪ QA responsible did the acceptance tests (7) [41]
<i>Refactoring:</i>	<ul style="list-style-type: none"> ▪ Keeps the simple design (1) [35] ▪ Should be performed regularly to avoid large refactorings (2) (3) [34] [40] ▪ Improves maintainability and stability of the code (4) [38]
<i>Pair programming:</i>	<ul style="list-style-type: none"> ▪ Difficult when not used to it (1) [38] ▪ Only for production coding (2) (3) [40] [35] ▪ Include other experts, e.g. database personnel or filed experts (4) (5) [36] [37]
<i>Collective ownership:</i>	<ul style="list-style-type: none"> ▪ Successfully and preferred in (1) (2) (3) [38] [40] [35]
<i>Continuous integration:</i>	<ul style="list-style-type: none"> ▪ The testing suite must be automated to make this work (1) [38] ▪ Important when there are several pairs (2) [40] ▪ Integration time must be part of estimates (3) [34]
<i>40-hours week:</i>	<ul style="list-style-type: none"> ▪ Not adopted yet due to overtime habits (1) [38] ▪ Unproblematic (2) (3) [40] [35]
<i>On-site customer:</i>	<ul style="list-style-type: none"> ▪ Could be replaced by in-house representatives (1) (2) [38] [40] ▪ Can be a whole team if the project is large (3) [35] ▪ Could have own requirement team at the customer site (4) [41] ▪ Not possible in small projects (5) [44]
<i>Coding standard:</i>	<ul style="list-style-type: none"> ▪ Already in place (1) (2) [38] [40] ▪ Done more informal (3) [35]

Table 3.1: Experiences with the XP practices in the industry.

3.1.3 How local modifications were performed

In general, the modifications done to the XP practices in projects described in section 3.1.1 above are performed to redesign XP to suit the needs and circumstances. It is also common to these projects that the modifications most often were performed using the communication and feedback that is a natural part of XP. The fact that XP is an adaptive¹ methodology also makes it natural for XP to adjust to the situations at hand. Below, we describe some more concrete examples.

To understand and manage capacity, ideal time estimates and actual time to completion were monitored in [38]. These metrics were used to improve the estimations for the next iteration or release in the planning game. It was also useful as an improvement means to visualise the activity and productivity on storyboards. By having stand-up meetings at the storyboard, everyone got an idea what the whole team was doing and they could more easily involve themselves in other parts of the system. An example of a stand-up meeting is shown in Figure 3.1.



Figure 3.1: Stand-up meeting at the storyboard [38].

The workflow queues improved dramatically, because people's attention got more focused to the unverified stories. In Figure 3.2 the storyboard was introduced in February 2001 and the number submitted – the not verified – stories sunk significant after only one month.

¹ See section 2.1.4.3 for a description on adaptive processes.

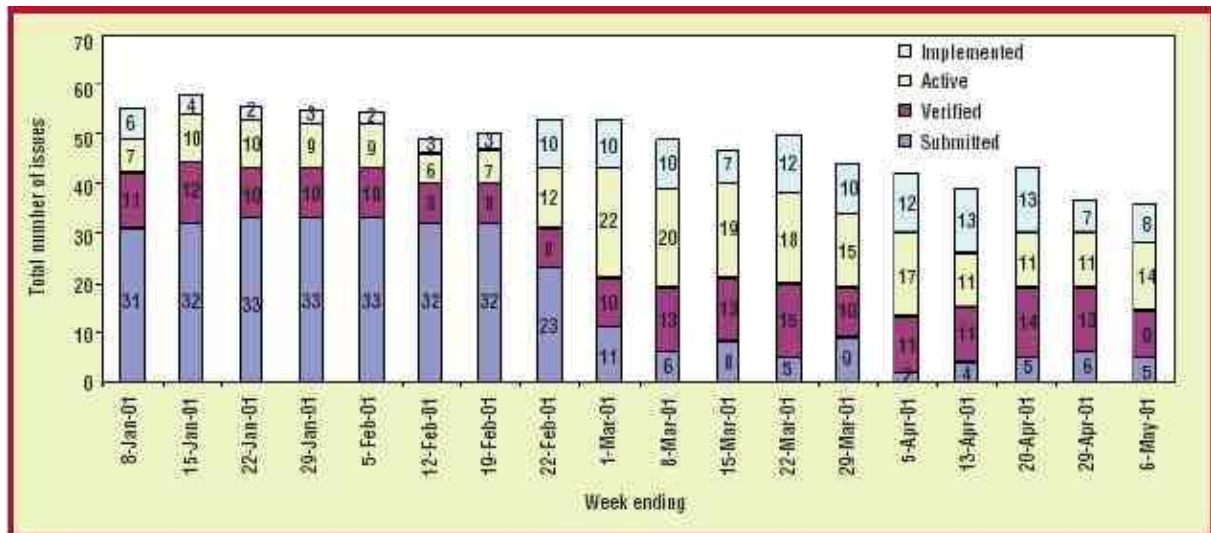


Figure 3.2: Workflow queues [38].

Recognising and respond to “bad smells” is important [34]. The metaphor of a “bad smell” is rewritten from meaning, “an early warning signal that something in computer code need to be rewritten”¹ to “early warning signal that certain aspects of an entire software development process need to be ‘refactored’”. When aspects of the development practice are recognised as inefficient, it is important to discuss how the problem may be solved. This may seem as just common sense, but experience tells us that in many cases things are flowing in the wrong directions before actions are taking place.

The relation between XP and improvements – SPI – are discussed further on chapter 5.

3.1.4 How XP affects quality and quality personnel

By tracking the status of all the stories and make the results visible to the participants, as described above, is also beneficial to quality. In addition to the increased performance, the prioritised stories are finished faster and not postponed. This give constantly more value to the customer during the project, which is motivating and positive for both the customer and the developers.

In large projects, the QA team cannot be replaced or removed [35]. Developers are often too biased in their understanding of how the system works or should work to be able to fill the role as quality assurance teams. The QA team should therefore consist of testers that are not part of the XP development team.

In smaller projects the QA responsibility can be given to one full-time Quality Engineer (QA) [41]. This can give benefits and relive much of the administrative burden of the quality process from the rest of the project team. The QE is responsible for controlling that the system quality is acceptable at every release. Quality metrics and automated build and test systems are central. By defining measurable quality metrics (results from different kinds of tests) and granting release-blocking authority (prioritise quality before new functionality in front of a release) to the QE, have improved overall quality and acceptance rates.

A further discussion of the relation between XP and quality and QA will be done in chapter 5.

¹ In Martin Fowler’s book *Refactoring*. Addison-Wesley, 1999.

3.2 Research and use in academia

There are done some research and use of XP in academia. In [45] there is a discussion about whether XP is suitable for software engineering education (SEE). It is stated that several of the XP practices are very valuable in the context of a general SEE. However, since some universities requires students to take only one software engineering course. If this singular course only used XP, students would lack the skills for documenting and designing larger projects. Moreover, there does not exist any empirical evidence showing whether the students learn and know more about software engineering using XP or not. However, it is made clear that students should experience other sides of software development than 'heavyweight models' that require much documentation by being introduced to a 'lightweight method' as XP. XP offers potential benefits for SEE by both being more focused the developing process and the pedagogical appeal of the practices.

Section 3.2.1 describes examples on how XP was introduced to university courses and how the experiences with of XP were in such environments. Section 3.2.2 investigates the effects of using pair programming by describing some results from research done on this field.

3.2.1 XP in university environments

In [46] XP was used in a 4th year team design project course. Traditionally the waterfall or V-shaped models were used in this course. The experiences of using XP instead were both positive and negative. The XP paradigm was understood as an ideal framework to complete a software project, but experienced that implementing this ideal framework within a real project in university environments caused some difficulties:

- Pair programming hampered the progress. The involved students had different time-schedules and it was often impossible to find someone to pair with.
- Refactoring took too much time at each deliverable. The students had only vague experiences with incremental development, and the lack of planning and initial design resulted in an overabundance of code rework.

Beyond that, small releases, collective ownership, and unit testing, were experienced with great enthusiasm. The project got a satisfied customer and product with an extremely low defect rate. With more knowledge and experience of XP (which they gained during the project), the students felt that possible later XP projects would be much smoother.

In [47] experiences from using XP in a course with computer science graduate students are presented. The course included three simple tasks and a project. The project teams consisted of six students and all development work was done in pairs. The course instructor played the role as customer. After some initial difficulties, the teams adopted the XP method. The observations that were done during the course were:

- Pair programming was adopted easily and was an enjoyable way of code (25% good and 62% average). Other positive sides of PP were learning from partner (100% agreed) and more confidence in solution (75% agreed). However, there were some uncertainties around what type of work not to do in pairs.

- Design in small increments (“Design with blinders”) was difficult. Holistic design behaviour may be difficult to abandon and one should at least be trained to design in small increments.
- Writing test cases before coding was not easily adopted and was sometimes impractical. This was especially regarding to graphical interfaces. Making good test also needed a great deal of experience.
- Due to communication overhead, they believe that XP as is does not scale well. It is definitely meant for small team (6-8 members).
- XP requires coaching until it is fully adopted. The team members had tendencies with falling back on old habits.

In additions, there were some conclusions related to the university environment. The instructor must insist about team meetings. First, the informal information exchange cannot be replace by e-mail or other means of communication. Second, as a student project in a university course always suffers from a tremendous lack of time, it is preferable to provide a software skeleton at the beginning from which the development can start.

3.2.2 The effects of Pair Programming

PP has been a central part of XP from the start [48]. Even though the experience is positive, there are still scepticisms about this approach to programming. The idea of being two people sharing the same machine seems inefficient and bothersome. This scepticism, however, seems to be shared only by those who haven’t really tried it (at least according to the published experience). Some improper cases are reported, but the overall experience is positive. It is said that “Two programmers in tandem is not redundancy; it’s a direct route to greater efficiency and better quality.”¹

For years, programmers in industry have claimed that by working collaboratively, they have produced higher-quality products in shorter amounts of time. To validate these claims, it is performed quantitative studies to find the effects of pair programming (PP) [49].

[49] refer both to industrial research and results from case studies that show that solving a task in pairs take somewhere between 20-40% more time to complete than for individuals. However, the results also show that the code delivered by pairs contains far less bugs. In fact, to achieve the same level of quality (absence of bugs), the individuals had to spend considerable more time than the pairs. By working in tandem, the pairs completed their assignments 40-50% faster than the individuals when the basis of comparison was quality.

After the initial adjustment period, pairs only spent 15% more time than individuals [49]. Here, the resulting code has about 15% fewer defects for the pairs (these results is statistically significant with $p < 0.01$). See Figure 3.3 Figure 3.4 for the results.

¹ L.L. Constantine, Constantine on Peopleware, Yourdon Press, 1995.

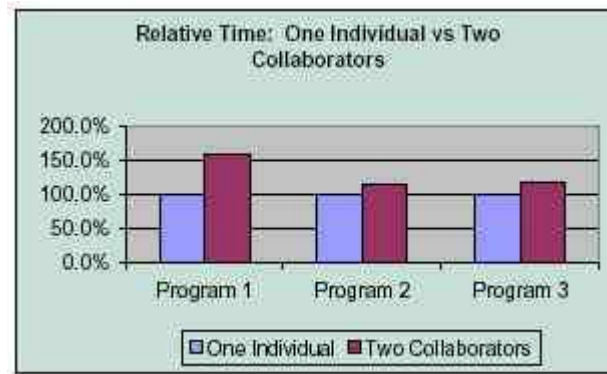


Figure 3.3: Programmer time [50].

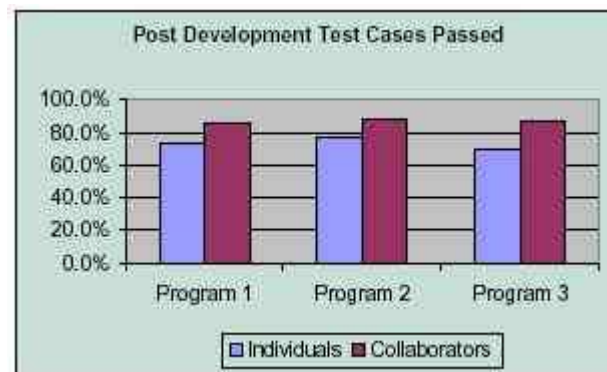


Figure 3.4: Code defects [50].

Further, the conclusions in [50] states that the initial 15 % increase in code development expense are not only recovered by the reduction in defects. The case study shows several other significant benefits of PP:

- Many mistakes get caught as they are being typed rather than in QA test or in the field (due to continuous code reviews).
- The end defect content is statistically lower (due to continuous code reviews).
- The designs are better and code length shorter (due to ongoing brainstorming and pair relaying).
- The team solves problems faster (due to pair relaying).
- The people learn significantly more, about the system and about software development (due to line-of-sight learning or direct communication).
- The project ends up with people understanding each piece of the system.
- The people learn to work together and talk together more often, giving better information flow and team dynamics.
- People enjoy their work more.

The extra costs of PP are thus repaid by many benefits and results in shorter and less expensive testing cycles, quality assurance, and field support.

Despite the positive response on pair programming from the XP supporters and these quantitative studies of it, there should be done more research to document the benefits of PP. Therefore, it is proposed an experimental framework to quantify benefits and costs of PP in [51]. Here, a framework and measurement plan for objective assessment of PP practices

introduced to selected parts of a company, are described. The results from this framework will be more complete when it comes to investigation of the software quality than earlier studies. Thus, with better documentation of the benefits, it may be easier to convince the sceptics to start using PP.

3.3 Interviews and personal contact with companies

As part of the work with this thesis, we have included some direct contact with companies that have experienced and used XP. This contact involves information from informal discussions to an experience report written after using XP in a project. We have been in contact with SINTEF and Proxycom, both in Trondheim.

The contact with SINTEF is described in section 3.3.1 and XP experiences from Proxycom are presented in section 3.3.2.

3.3.1 SINTEF

SINTEF (The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology) is with its 1700 employees, the largest independent research organization in Scandinavia. They supply intelligent and profitable solutions to their customers' problems. Their products consist of knowledge and related services based on research in technology, the natural and social sciences and medicine.

During the work with this thesis, the contact with SINTEF has occurred in various ways: from participating on an XP-seminar, through case studies¹, and to discussions with employees at the Trondheim office.

The contact with SINTEF has given general contributions to my understanding of XP and an insight of the introduction of XP in companies and projects.

3.3.2 Proxycom

Proxycom is a consultant company working with development and maintenance of IT-solutions and is located in Trondheim, Norway. To learn more about XP, they did an internal pilot project using XP [52]. The projects consisted of one customer (product responsible in an external firm), a project manager, and seven developers. The system to be developed was a web-based stock exchange-administration application. The man-hours worked were around 500, including planning and three iterations.

Following this project, all the participants answered a questionnaire, and the results from this questionnaire were summarised:

Positive experiences with XP:

- Frequent and short meetings was useful
- XP yield good customer co-operation
- Collective code ownership is favourable
- Pair programming is effective when it comes to learning, experience exchange, and quality

¹ The case studies are described in chapter 4.

Negative experiences with XP:

- Less time-effective than traditionally methods
- Difficult to find a pairing partner when the participants was part time in the project
- Difficult to include database management.

To reduce the negative effects of XP and increase the positive sides of XP, some improvements of the XP method was suggested:

- **Reduce the use of pair programming:** Pair programming should only be used for training and mentoring and when developing complex program modules. Simple and trivial modules should be developed by individuals. The developer who shall perform the development of the current module should determine the limit between complex and simple modules.
- **Developers should work full-time on the project:** This will eliminate the difficulties finding a partner to program with.
- **Do some data modelling and database-design upfront:** When the project includes databases and these databases is difficult to refactor in every iteration, the database design should be planned upfront (this was already carried out in this project).

To verify that the two first suggestions are useful, they will be used in the next project or deliverable. This has not been carried out yet, but the plan is to base the next project on the experiences gained here. They use an XP-handbook as guidance from project to project and modify this book when important adjustments are needed.

Based on the experience with XP, they also feel that it must be done some modifications to XP before in can be a trustworthy alternative to traditional methods. E.g. the absence of a requirement specification is problematic to many customers. As start it is suggested to use XP after the requirement specification phase.

4 Case studies

This chapter will describe two case studies where XP was used as the development methodology. The case studies describe two separate projects. Both project teams included two developers and one customer. The data collection in Case Study 1 (CS1) was far more extensive than Case Study 2 (CS2), so the main emphasis of the analysis and validity of results will therefore be given CS1.

We include these case studies to be able to observe different elements of XP and relate some of these to the work of doing SPI and QA together with XP. Investigating the XP process, we can confirm some of our suggestions of SPI and QA approaches to XP. Besides, will we get further input about how XP really works in real life, not only by second hand information, as described in chapter 3 above.

CS1 is described in section 4. 1. This project is analysed with the Goal-Question-Metric method and observed during the whole lifecycle of the project. At the end of the project we evaluated the project further with a Post Mortem Analysis (PMA). This case study was used to try out some extensions to the traditional XP practices.

CS2 is described in section 4. 2. This project is evaluated through interviews and a PMA after the project. This case study is, beside from getting more empirical experience with XP, used to compare some results from the first case study.

Finally we will have a discussion of the case studies in section 4. 3. Here we will sum some of the results and explain why XP turns out different in the two case studies.

4.1 Case Study 1: A XP development project evaluated with GQM and PMA

Case Study 1 (CS1) is a study of an XP development project where evaluation is based on observations along with the project, GQM metrics, and a finishing PMA. We will describe the project and the measurements in 4.1.1, the project environment in 4.1.2, a reminder of the limitations of the study in 4.1.3, our observations in 4.1.4, the case study results in 4.1.5, and give some conclusions in 4.1.6.

We will present the results from the case study, but more important to this thesis, we will present results according to how SPI and QA were performed during the project. Further we will try to conclude with what influence SPI and QA had to the performance and result of the project.

4.1.1 Project description and measurements

The goal of this project was twofold. The most important goal was to better understand the XP process by using XP to develop a system and compare this system with a similar system developed without XP. The GQM Method was used for measurements during the project. To support this goal, the second goal was to develop a system with a graphical interface for supporting projects and their process descriptions.

The project was launched at SINTEF (The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology) by two graduate students from NTNU (The Norwegian University of Science and Technology). The project leader came from SINTEF.

Next we will describe the system to be developed in 4.1.1.1, the project and iteration plan in 4.1.1.2, and the GQM measurements and the data collection in 4.1.1.3.

4.1.1.1 The system to be developed

The background for this project was a system, or more precise a prototype of a system, developed with Java and JSP. The system was developed by one developer during 168 man-hours and the development method was traditional with upfront requirements.

The idea was to develop a similar system using XP from scratch without looking at the old code or architecture at all, to see what influence XP has on these and other factors.

The system is a part of an intranet-solution for a company. It is a tool for supporting and reviewing projects and guidance to the project processes. A user shall be able to change properties of a project and collect information from other projects.

When we are referring to the already implemented version of the system and the one implemented in this project, we will call them system A and system B respectively

4.1.1.2 Project and iteration plan

The project lasted for two and a half months with iteration lengths of one week, with total seven iterations. During this period the developers had one main release in addition to the resulting product. Each iteration started with a planning meeting with developers and customer, and ended with a feedback meeting with everyone involved. Descriptions of these meetings will be given in section 4.1.4 below.

4.1.1.3 The GQM abstraction sheet and logbook

To retrieve information and support the evaluation of the project, the Goal-Question-Metric (GQM) method was used to make the measurement plan for the project. A diary were also written to both collect data to the GQM metrics and, in addition, other information during the project.

The developers and I defined the GQM abstraction sheet before the project started. The GQM goal was: "Analyse the XP process for the purpose of understanding its effect in a development project from the point of view of the developers in the following context: Intranet-based application at Company X". The GQM abstraction sheet and more details on the measurements can be found in [53].

Below we have listed what to register every day, after each iteration, and after each release.

What to register at the end of each day:

Hours refactoring
Hours pair-programming
Hours general project work
Number of changes in functionality per user story
Number of changes in priority per user story
Number of man-hours used per story
Number of planned stories
Number of active stories
Number of finished stories
Specified the hour usage
▪ Customer meeting
▪ Pair-programming
▪ Spiking
▪ Refactoring
▪ Acceptance testing
▪ Project administration
▪ Feedback meeting
Short textual description of the day

Table 4.1: The daily log in Case Study 1 (CS1).

A weekly feedback meeting was also used to gather information and data to the GQM metrics and to the development process. The organisation of this feedback meeting is

described in section 4.1.4.2. The data to gather during these meetings are described in Table 4.2 and Table 4.3.

What to register after each iteration:

Subjective assessment of this iteration
Subjective assessment of each XP-practice
Suggestions to changes of the practices
Number of used change-suggestions
Number of used change-suggestions, from earlier iterations
Subjective assessment of the customer availability
Subjective assessment of making tests for Java
Subjective assessment of making tests for JSP
Subjective assessment of making tests for HTML
Short textual description of this iteration
Number of changes of functionality in stories
Number of changed priorities of stories
Subjective assessment of the story size

Table 4.2: Measurements after each iteration in CS1.

What to register after each release:

Number of lines of Java code
Number of lines of HTML
Subjective assessment of customer feedback (after release)
Subjective assessment of JSP
Number of hours refactoring
Number of hours total

Table 4.3: Measurements after each release in CS1.

4.1.2 Project environment

The project environment includes the involved parties, office landscape and facilities, development tools and resources, a whiteboard, and other visual means.

4.1.2.1 The developers and the customer

The development team consisted of two graduate students from NTNU. Both students used this project as a part of their Diploma thesis. A Research Scientist at SINTEF represented the customer in the project. The customer was not physically located in the same room as the development team all the time, but could be reached either in his office “next door” or by telephone.

4.1.2.2 Office landscape and facilities

As recommended by the XP literature, e.g. [3, c.13], the office landscape was arranged as an open workspace. Figure 4.1 and Figure 4.2 shows how the office was arranged.



Figure 4.1: Office environment.



Figure 4.2: Arrangements around the developers.

Around the developers and their computers there were put several types of information on the walls:

- The active and upcoming user stories were placed in prioritised order on a whiteboard. See at the right of the picture in Figure 4.2.
- The coding standards and a figure of the current architecture were located right above the computer screens. See Figure 4.2.
- Descriptions of each XP practices were posted on the wall to constantly remind the developers about what the “development rules” were at all times. See at the left of the picture in Figure 4.2.

A table in the middle of the room was used for meetings. The developers, the customer, and everybody else stood around this table. The whiteboard could be seen from this location.

Due to the literature [3, p.88], the user stories were written on small cards and posted on the whiteboard. See Figure 4.2. Computers and needed software were installed at the beginning of the project.

4.1.3 Limitations and constraints

Case studies will usually have some sort of limitations or constraints that may limit the validity of the experiences or conclusions drawn from it. When the conclusions are generalised to also apply in other circumstances, the awareness of possible limitation or constraints are vital.

The limits and constraints in this case study were:

- **Experience with XP:** The developers had only theoretical experience with XP before the project started. This may influence how the XP-practices were used and adopted.
- **Team size:** It was only two developers and therefore only one pair. This limited for instance the use of continuous integration; it was only one code-base. Further, the communication inside the team had fewer barriers compared to larger teams.
- **Customer:** The customer had lots of programming experience and may have had too much control over the technical challenges and knew too much of the functionality from the beginning. From his experience with programming, the customer could for instance act when the estimates were too long or short. A less experienced customer may not have discovered misunderstandings of the required functionality that lead to wrong estimates.
- **Reference system:** (System A) The system that we compared to our work was developed by an engineer with less knowledge to object oriented design. This can cause larger differences to the code and architecture between the two solutions than strictly needed. A former C programmer will code less object oriented than developers' only knowing Java.

4.1.4 Observations of the project and the iteration evolvement

This section describes how the iterations evolved in 4.1.4.1, how the learning and improvement process worked in 4.1.4.2, and what the quality assurance work was like in 4.1.4.3.

4.1.4.1 The iterations

This section will give a short summary of the iteration evolvement. It was in all seven iterations. Each iteration started with a meeting with the customer and ended with a feedback meeting.

The starting point of the project was a description of the XP practices and how they should be carried out. These were posted on the wall to remind the developers about their development "rules" – see Figure 4.3. Then the customer gave an introduction to the system to be developed. During the first planning meeting, four stories were presented and estimated, and the development could start. The first iteration was, however, characterised by installing, configuring, and training of new software-packages. This had bad influence on the pair-programming progress. Nevertheless, the developers agreed on the system architecture.

This was posted on the wall along with the coding standard. Pair programming was enjoyable, but it was hard to get used to the test-first approach. Pair programming is showed in Figure 4.4.



Figure 4.3: The XP practices on the wall.



Figure 4.4: Pair programming.

The second and third iteration went much better, and the customer had to come up with new stories in the middle of the iteration to keep the developers busy. Some refactoring had to be done to remove deviation from the system architecture. The first release was presented to the customer and users after the third iteration. Several problems came up during the demo, mainly because the demonstrator was not familiar enough with the system “limitations”. The developer were getting more used to the testing and the unit test enforces good object-oriented design. Acceptance tests are more troubled. These are done manually together with the customer. There is no automatic tool available for GUI testing.

Iteration four and the reminding iterations were more troubled but also relative satisfactory. The customer wished a new and more dynamic GUI (Graphical User Interface). This was a risky decision, but the customer still wanted this change. Due to the new technology, the developers used much time on spiking. This resulted in great estimate overrun and few finished stories. For the future, the stories with this degree of new and unknown technology should be split down in smaller stories. A tool was found for testing the GUI in the fourth iteration, and was used throughout the project.

The last iteration included more refactoring than the previous. A little bit more refactoring during all the iteration could have avoided this. However, all in all there was not much refactoring compared to the amount of coding. Section 4.1.5 will show the proportion between pair programming and refactoring, together with the other results. The second release was delivered after this iteration. The final system was not ready to use but was a prototype and starting point for further development. This was ok for the customer, considering the great changes in technology during the project.

During all iterations improvement of the XP practices were suggested to make the practices more suitable for this project. How the improvements were performed is described in the next section.

4.1.4.2 Learning and improvements of the process

One of the goals for the students in this project was to adjust the XP practices to suit the project. By suggesting changes and introduce them into the development “rules”, a local version of XP would evolve. The improvements were performed on several levels, from the experience during the daily work to weekly feedback meetings. The feedback meetings were arranged as small PMA sessions¹, which was an extension of the stand-up meeting described in [54, c.20].

While the developers were working, they always tried to review what they experienced. When they felt that an experience was worth keeping, they wrote it down and posted it on the XP practices poster. If they felt that a change should be done to one of the practices, a suggestion for change was written down and posted. In Figure 4.5 there are showed examples of posted experiences and suggestions (in Norwegian under “Erfaringer” and “Forslag” respectively).

¹ PMA is described in section 2.2.3.

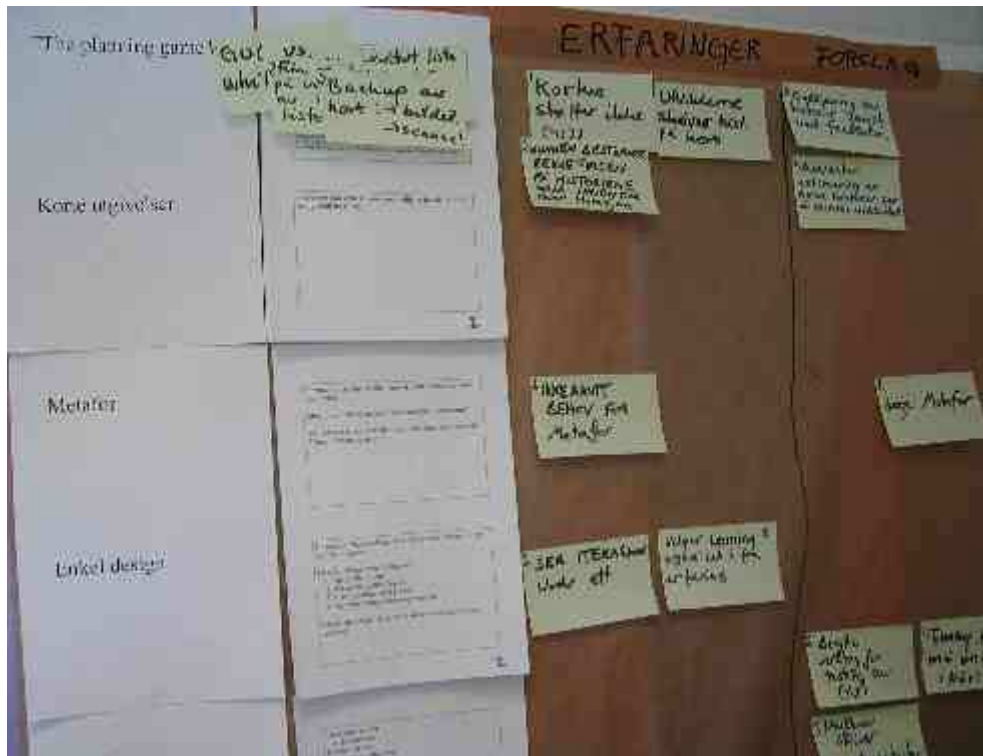


Figure 4.5: XP practices with experiences and suggestions.

To take care of other experiences, each developer wrote down a daily log. This log is described in section 4.1.1.3.

After each iteration, a feedback meeting was held. Present on these meetings were the developers, the customer, and myself and often the teaching supervisor – as spectators and contributors. The length of these meetings was around one hour. The agenda was as follows:

- 1) Summary of the iteration
- 2) KJ
- 3) Register data for measurements
- 4) Any other business

The pictures Figure 4.6 and Figure 4.7 are taken from one of the feedback meetings.

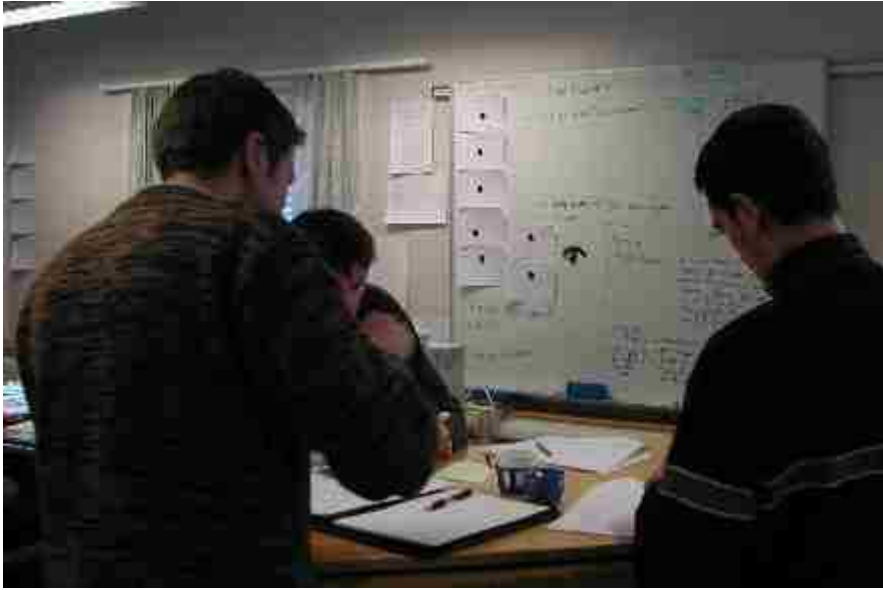


Figure 4.6: Feedback meeting.



Figure 4.7: Discussion on a feedback meeting.

After the iteration was summarised, we held a KJ session¹. Here the developers reflected on positive and negative sides of the last iteration, see Figure 4.8. This gave a lot of input to what should be improved. E.g. input to the discussion around this week's posted suggested changes and experiences, that was shown in Figure 4.5. Changes that were already taken in use were registered and useful suggestions were discussed. The suggestions that were agreed among the developers were introduced in the next iteration.

¹ KJ is described in chapter 2. 2.

they got more experienced, unit testing became a natural part of the development. Junit [55] was used to implement the unit tests for Java. Beside giving confidence to the code, unit testing also had two positive side effects. First, it assured that the code could be controlled easily after refactoring. If the refactoring demanded larger changes, the unit tests were rewritten first to avoid new bugs. Secondly, the unit test forced the code to be more object-oriented and thus gave an object-oriented design.

From the beginning, the acceptance tests were done manually by the developers together with the customer. The tests mainly consisted of testing the GUI. It was originally the customer's responsibility to write these tests, but it didn't work out. From iteration four these test were written and documented by the developers. Each acceptance test included the following items:

- User history number
- Iteration number
- Critical / Non critical for approval
- Short description of the test
- Actions:
 - Number
 - Command
 - Action description
 - Input data
 - Expected output data

Since the developers didn't have access to a testing tool, these tests were done manually until iteration five. The fact that they were documented, made it easier to redo them when needed. When a tool was found, the developers were given the responsibility of implement the tests. The tool Astra QuickTest [56] made it possible to make automated GUI tests. However, the developers did not get familiar enough with this tool to perform the automated test efficiently. In later projects, the developers should be familiar with a testing tool from the beginning.

Refactoring was used to clean up the code, remove duplicated logic, and keep the design simple. However, since the developers had suggested an overall design early in the project, it was not necessary with much refactoring. Refactoring was mostly needed to maintain the 3-tier design. Together with the unit testing, the refactoring resulted in a easy readable code and object-oriented design. This was satisfying to the developers and hopefully to the customer, e.g. for later extension development.

Another effective quality assurance means was the feedback that occurred during the Planning Game. When the customer had written the stories, the developers estimated them. To make sure that there were no misunderstandings between the customers and the developers, a re-negotiation was done. This improvement of the Planning Game was already introduced in iteration two. If the estimates had large deviation from what the customer had in mind, the scope and estimate of the story were re-negotiated.

Results and conclusion from the quality work is shown in the next sections, 4.1.5 and 4.1.6 respectively.

4.1.5 Case study results

This section presents the results from the case study. We will describe the results from the GQM measurements in 4.1.5.1, summarise how this project team experienced the XP-practices in 4.1.5.2, investigate the improvements done to the process in 4.1.5.3, and comment the quality assurance work in 4.1.5.4. Recommendations and conclusions are given in 4.1.6.

4.1.5.1 Results from the GQM measurements

The results from the GQM measurements are rather extensive and will not be presented here. Not all the results are relevant to the focus of this thesis and a selection will therefore be done. The complete collection of results can be found in [53]. Some of the results can also be found in the sections below.

The project lasted for seven iterations of one week each. The total number of man-hours used during this period was 424,5. Table 4.4 shows the distribution of the man-hours over the duration of the project.

Iteration	Man-hours				
	Pair programming	Spiking	Configuration/Installing	Customer meeting	Misc
1	16,5	17	31	2,5	5
2	26	13,5	17	2,5	0
3	36	9	0	1	4
4	31	27	0	1,5	0
5	25,5	11,5	0	0	5
6	29	32	0	2	0
7	43	30,5	0	2,5	3
Sum	207	140,5	48	12	17

Table 4.4: Distribution of the man-hours used in CS1.

These results were not far from what we believed when we started. Of the total number of man-hours used in the project of 424,5 the percentage per activity was:

- Ideal-hours pair programming: 48,8% Believed: 50%
- Spiking: 33,1% Believed: 25%-33%
- Configuration/Installing: 11,3%
- Customer meeting: 2,8%
- Miscellaneous: 4,0%

This means that the developers followed their plan with 40-hours weeks and maximum 8 man-hours pair programming per day. The share of spiking was believed to be up to one third of the time because we expected much new and unknown technology.

The code base was divided into Java, JSP, HTML, and JavaScript. The total lines of Java code were 1744 while total lines of JSP/HTML/JavaScript code together were 603. We expected these to be 900 and 3000 respectively. This huge deviation can be explained by the system architecture. A greater deal of the system logic and functionality was located in the Java code, near the server and not in the JSP or HTML code near the client. By using this kind of design we could remove duplicated JSP and HTML code, and this made the testing easier since it was more effective to do unit testing of Java code.

To see the most important part of each iteration, we categorised the iterations. These are showed in Table 4.5. The descriptions will be used to evaluate the GQM measurements of pair programming and completed user stories below.

Iteration	Categorical description
1	Installation and configuration
2	Programming
3	Programming
4	New technology / Spiking
5	Programming
6	New technology / Spiking
7	Refactoring

Table 4.5: Categorical description of the iterations in CS1.

The project consisted of two releases. We believed that it would be around 12 hours refactoring per release. The results were a bit different in release one and two in Figure 4.9:



Figure 4.9: Man-hours of refactoring per release in CS1.

Release one had 6,5 hours refactoring while release two had 18. The difference can be explained by the increased complexity in release two. There was also more refactoring at the end of the project than the average. The total time used for refactoring was also relative small: 24,5 hours or 6% of the total project time. This was as expected since the developers based the design and architecture on the knowledge they already had. Instead of only look at the current story and base the architecture on this, they used system architecture from a similar project and all the available stories as basis. A further comment on this can be found together with the simple design practice in 4.1.5.2.

Together with Table 4.5 of iteration characterisations, Figure 4.10 and Figure 4.11 show some interesting relations. Figure 4.10 shows the evolvment of the actual and planned man-hours of pair programming used per story, while Figure 4.11 shows the evolvment of the actual and planned completed user stories.

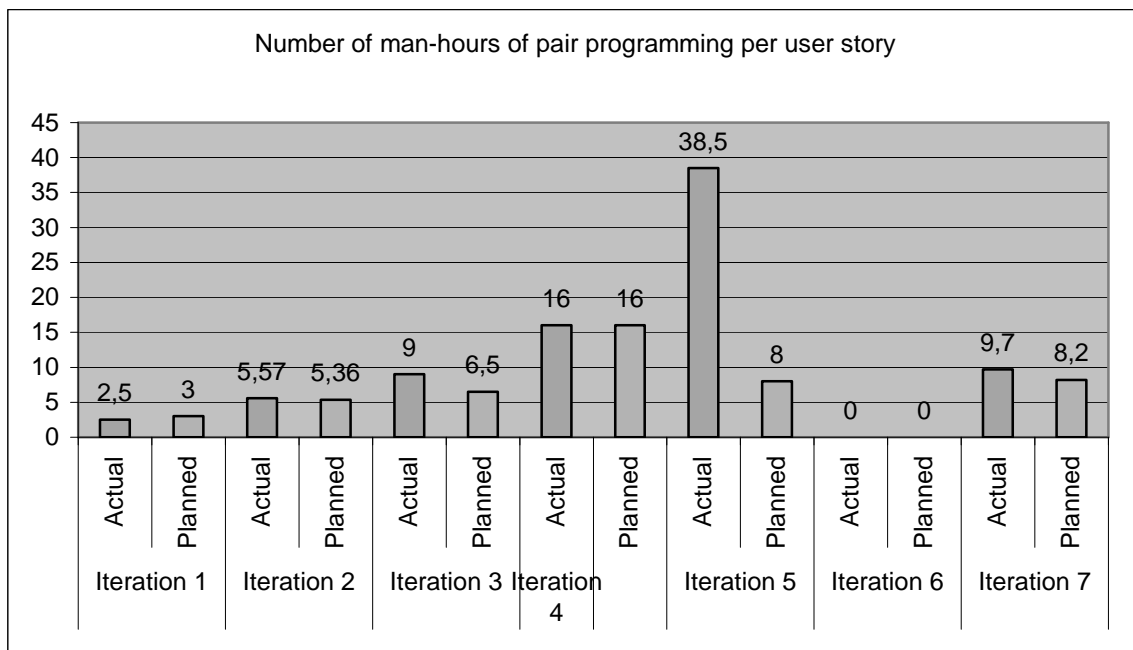


Figure 4.10: Number of man-hours of pair programming per user story in CS1.

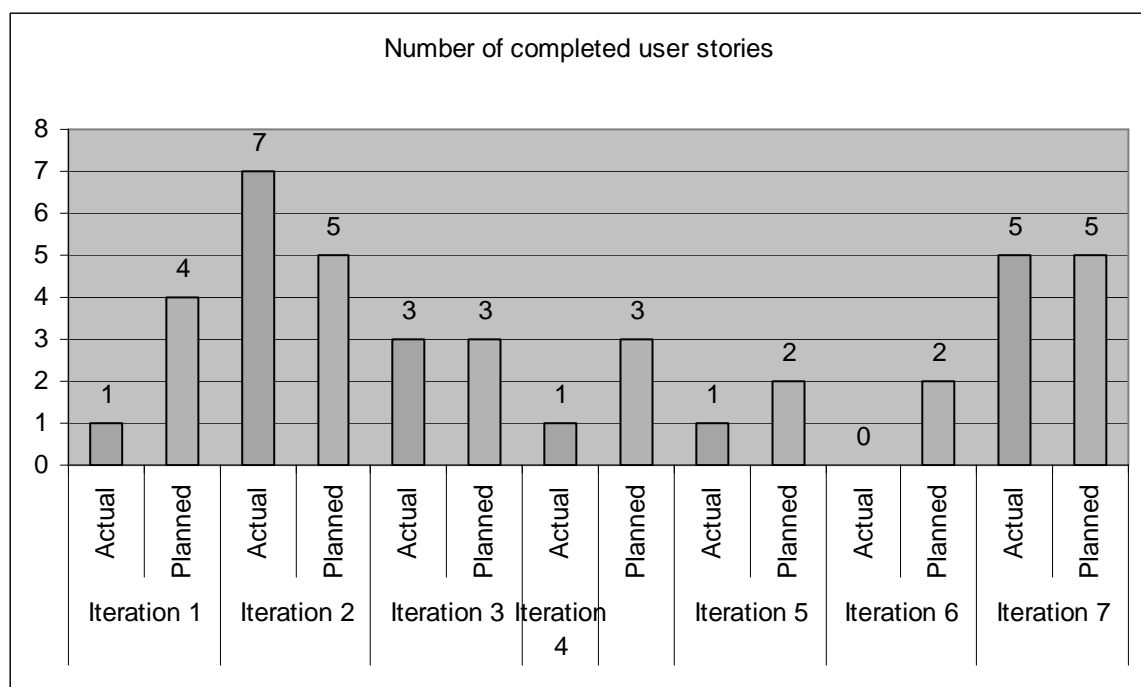


Figure 4.11: Number of completed user stories in CS1.

We can see that after iteration four with much new technology, the hours used to do pair programming were much larger than planned while completed stories were lower than planned. We can also see that iteration six, which also is characterised by new technology, had no hours pair-programming on stories and no completed stories. The log from this iteration tells us that the time was used on spiking and pair programming of spikes. The reason for this is that stories defined in these iterations were too large. When the stories include a large component of new technology, they should be split up into smaller stories. This was learned in iteration seven, where the estimated and actual values were merging.

4.1.5.2 Experience with XP practices

By continuous learning and suggesting improvements on each XP practice, the project team ended up with their own way of doing XP. This was an important outcome of the project. The improvement process of the practices is described in section 4.1.5.3. From the Q4 question in GQM abstraction sheet “How well are each of the current practices working now?” the general experience and opinion of the practices got better and better along with the project.

We will describe the experience with each XP practice in Table 4.6.

XP practice	Adoption status	Experience
The Planning Game	Adopted from start	Central for the communication between developers and customer. Re-negotiation was used to adjust estimates and scope of stories.
Small releases	Adopted from start	More difficult to deliver the last release, since the system complexity was higher.
Metaphor	Not adopted	A metaphor was tried out but not used. Due to the small team, there was no need for a metaphor.
Simple design	Adopted from start	The architecture design was based on all knowledge to the system, also to all available stories. This resulted in more general classes and methods, but no unnecessary functionality was ever added.
Testing	Adopted from start	Unit tests were always written before production code. The developers also ended up writing the functional tests, or acceptance tests, while the customer approved them. These tests should be automated, but this requires a suiting and stable testing tool.
Continuous integration	Not adopted	Since there was only one pair, all production code was located on one machine. Code was only integrated directly when pair programming.
Pair programming	Adopted from start	Gave constantly review of the code. All production code was made through pair programming. Spiking was partly done alone.
Collective ownership	Adopted from start	Positive.
Refactoring	Adopted from start	Should be done regularly throughout the project. Due to stable design, there was not much refactoring.
40-hours week	Adopted from start	Followed to the end of the project.
On-site customer	Partially adopted	The customer was seldom physically in the same room as the developers, but could respond quickly to requests directly or by telephone.
Coding standards	Adopted from start	More extensive comments should be written for complex code.

Table 4.6: Experiences with the XP practices in CS1.

A more extensive description these experiences and how the experiences changed during the project can be found in [53].

4.1.5.3 Learning and improvements

The case study gave us both insight to the learning process of an XP project and some important contributions to the work with XP and SPI. The learning was in general performed by gaining knowledge from failures and success of earlier experiences (partly due to the Yesterday's Weather¹ [54, c.8]), through pair programming, and by constantly reviewing of the current work. This result was partly expected, since this kind of learning is a natural part of XP.

An important contribution for this case study was the ability to track how improvements were suggested and introduced. These results are presented in Figure 4.12. The abbreviations used in the figure are listed in Table 4.7.

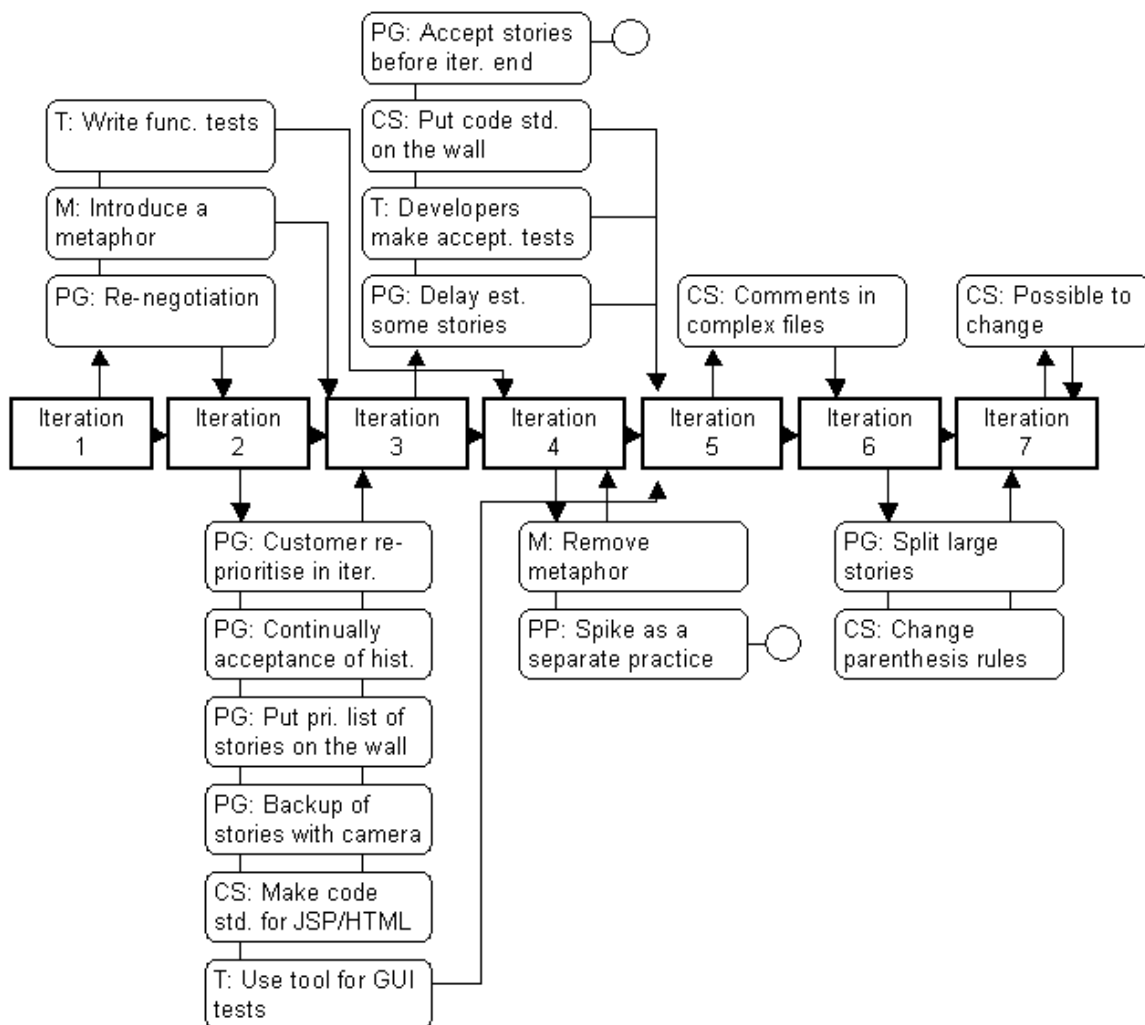


Figure 4.12: Suggested and introduced improvements to the XP practices in CS1

¹ Yesterday's Weather is a XP strategy, where you look at previous work and estimates.

PG – Planning Game	SD – Simple design	PP – Pair programming	40H – 40-hours week
SR – Small releases	T – Testing	CO – Coll. ownership	OC – On-site cust.
M – Metaphor	CI – Cont. integration	R – Refactoring	CS – Coding std.

Table 4.7: Abbreviations of the XP practices.

These observed results are not exactly like we expected. However, the results confirmed some of our theories. First to the surprise: We believed that constantly changes to the development process could cause instability and therefore make it difficult to introduce changes immediately after they were suggested. The result shows us that the main parts of the suggested improvements from one iteration were directly introduced into the next one. Only a few suggestions take more than one iteration to be introduced, while some were never introduced. There could be several explanations to these results. First of all, most of the improvements are small, and therefore easy to introduce. Major changes could be more difficult. Second, most of the suggestions were coming from the developers themselves. This makes them more enthusiastic to adopt the changes. Third, the team was small and it is easier to agree on changes. Forth, the feedback meetings made the reflections and communication efficient, and these conditions made the project more adaptive.

Large changes suggested by others than the developers were more difficult to introduce. If a change could represent major changes to the current practice or lead to more work, then they are not introduced immediately. Examples here are the introduction of a metaphor or performing automated acceptance tests. These changes would mean changing working habits and create more work to do, but could have been efficient in the long run. The problems with introducing these improvements can also be explained by the motivation of the developers. If they don't see the need for such changes, they are more critical to them. However, this could also be a positive sign. E.g. in this project, the introduction of a metaphor took some time and was immediately rejected, because there was no need for it.

We can observe from the statistics that there were more suggestions in the beginning than in the end. See Figure 4.13 for details.

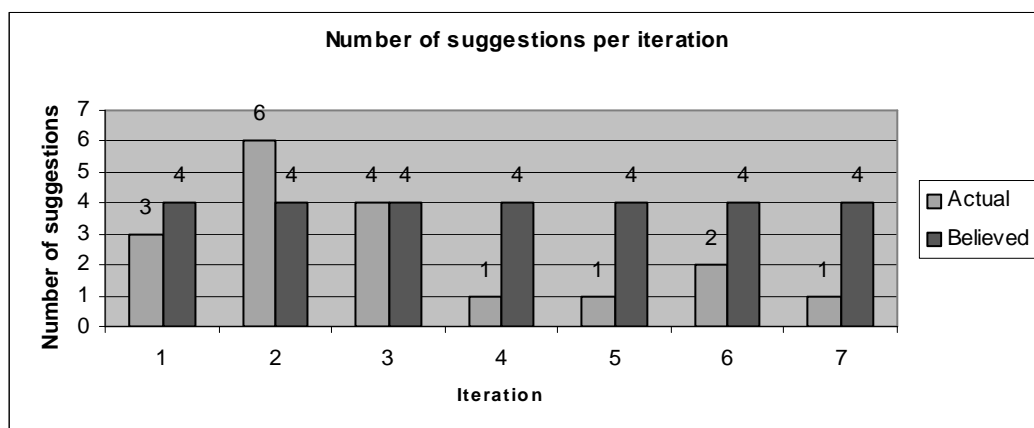


Figure 4.13: Number of suggested improvements per iteration in CS1.

We expected around four suggestions per iteration. It is natural that start-up adjustment results in more suggestions and that the process later will stabilise.

Another valuable contribution from this case study was to observe how the development process evolved and changed similar to the self-adaptive process we described in chapter 2. 1. The use of feedback meetings gave the whole team the possibility to discuss problems and ideas. This led to a development environment that was easy to change and improve along with the project. The central means in these feedback meetings was the use of KJ. KJ is an effective and informal way to share personal experiences and opinions. Together with a methodology like XP, which favours informal personal communication and experience- and information sharing, KJ is a well-suited method. The results, which are based on qualitative response from the project participants, are positive. We believe that the use of KJ in feedback meetings also can give beneficial support to XP-teams that are larger than ours.

4.1.5.4 Quality assurance

The Quality assurance (QA) work was focused on unit- and functional testing and customer satisfaction. The following results will reflect this focus. We will also present the results from the assessment of the difference between this system, System B, and the system developed by “traditional” methods, System A.

The results related to QA are measurements of the satisfaction of making automated tests. We measured the developer’s degree of satisfaction when they used automated tests for Java, JSP, and HTML.

The tests of the Java code were done with JUnit, and as we can see in Figure 4.14, the satisfactory were high during the whole project. This was like we had expected.

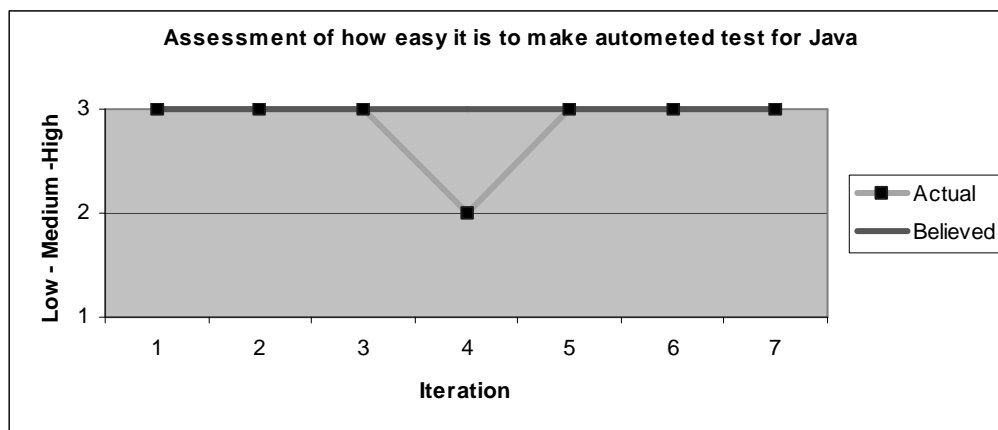


Figure 4.14: Assessment of how easy it is to make automated test for Java in CS1.

The results from the assessment of tests for JSP and HTML were more variable. Figure 4.15 shows the results.

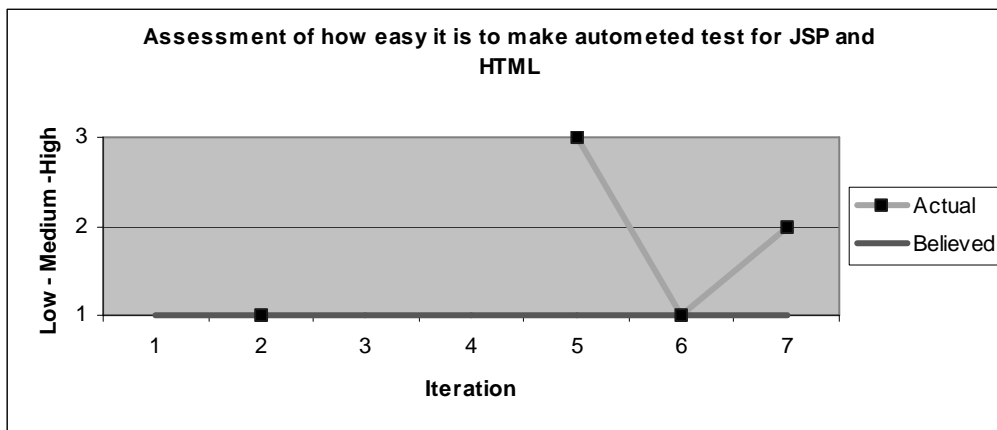


Figure 4.15: Assessment of how easy it was to make automated test fir JSP and HTML in CS1.

Our expectations were low during the entire project. The results were more variable from “low” in iteration two to “high” in iteration five. The reason for this is that there was no testing tool available until iteration five. The testing tool, Astra QuickTest, was easy to use but was a little unstable. This caused frustrations among the developers and the automated test of JSP and HTML, the GUI, were not very successfully.

The acceptance tests of the system were documented as described earlier and performed manually. Normally it would also be beneficial to track the results of the acceptance test as a means for controlling the quality during the project, but this was not done. Since this project was relatively small, it was not critically to automate the acceptance tests or track the results of these. The developers knew all the time how they were doing due to the accepted user stories, and had complete overview all the acceptance test. This would have been a lot different if the project had been larger and had included more than one pair.

The acceptance test can be used as a measurement of customer satisfaction. The acceptance test was accepted and we can therefore say that customer satisfaction was high. However, what about the final product? The system is not usable as it is today; it is a prototype for further development. Even though the goal for this project was to deliver a prototype, we believe that the customer hoped for a more usable one. This confirms one of the characteristics of XP; the customer must take a large part of the risk that is associated with the development. When the customer wished to try out a new solution late in the project, it involved a great risk for not meeting the expected functionality. The customer knew about this risk and is satisfied with the outcome. It was more valuable to the customer to have a unfinished product with new technology than a usable one with old and unsatisfactory solutions. This may be special for this project and can therefore not be generalised to apply to other customers or projects.

A major means for the customer to influence the development was the introduced re-negotiation in the Planning Game. Customer discovered misunderstandings early and was thus able to adjust the development before it was too late. This requires a customer with a great deal of insight into the domain and we must be aware that this solution may not be suitable for other circumstances.

We also performed a qualitative assessment of the quality of our system, System B, and System A, the former system. The difference between the systems can give us an idea of the effect QA has had to the XP process. None of the systems are fully usable systems, but System A has more functionality included. Some data from the code base of the different systems are showed in Figure 4.16:

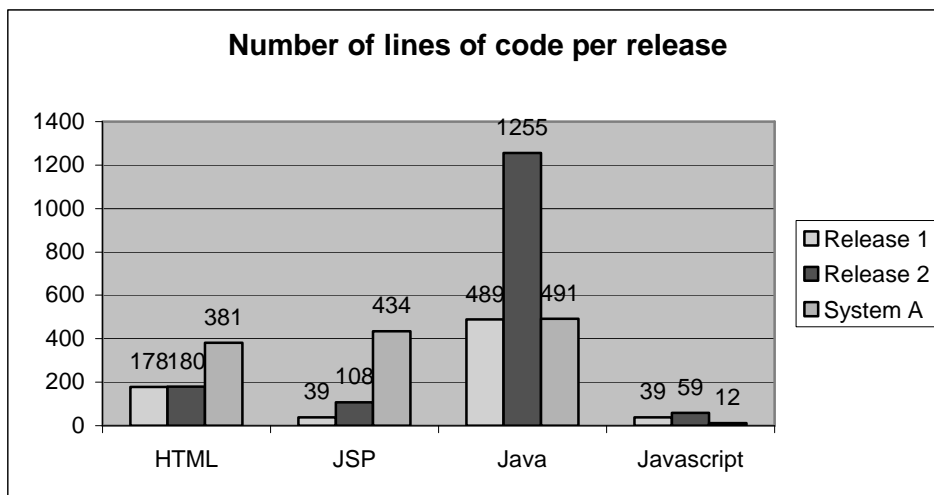


Figure 4.16: Number of lines of code per release in CS1.

System A had 182 man-hours and included one developer while System B had 242,5 man-hours and two developers. We can see that our solution has more Java code and less HTML/JSP/JavaScript code than the other system. Except for this difference in code location, we have analysed the system architectures. The results from both these studies show that our system is far more object-oriented and modifiable than the other. While a great deal of the system logic and functionality of System A is located in the client, due to more HTML/JSP, our system has this located in the server. This results in less duplicated code and the code is therefore more modifiable. The reasons for this difference are both the design of our system and the use of unit testing. It is far easier to automate the tests of the code written in Java, and this forces through more code in Java to satisfy the practices of XP. We will therefore say that XP and the QA had a positive effect on the system quality.

4.1.6 Conclusion and recommendations

This section presents our conclusions and recommendations from this case study. The reader must be aware of the limitations and constraints described in section 4.1.3 when interpreting the following. The conclusions and recommendations here are based on the results in section 4.1.5, and further background of the findings can be found in the subsections in the result section.

The most important recommendations and conclusions related to XP in general are:

- **Preparations.** The project should have an iteration null to arrange the project environment and set up all necessary software. Start-up problems will make it more difficult to follow the estimates in the first iteration than necessary. This may also affect the performance of acceptance tests. If a tool is ready before the project, it will be much easier to include this into the testing practice.
- **Office landscape.** The office area should be open and information related to the user stories, priority, coding standard, architecture, and so on should be visible and put on the walls.

- **Design.** The design should be planned, based on available knowledge from similar projects and on all the stories in the iteration, not only based on the current story. This will stabilise the design and reduce the amount of refactoring.
- **Stories.** The stories with higher degree of new and unknown technology or stories that are estimated to last more than half of the iteration, should be split in smaller stories. This will make it easier complete the stories without large estimate overrun.
- **Spiking.** Spiking on new technology was useful.
- **Refactoring.** Refactoring should be performed regularly throughout the entire project. This will avoid large refactorisations in the end.
- **Acceptance tests and tracking.** Making automated acceptance tests of the GUI were difficult because we didn't have a suitable tool. Instead we documented the tests and performed them manually with the customer present. Nor were the results of these tests tracked. Since the team was so small, this was not needed. The team had always the overview of their work and how they were performing. For larger teams we suggest both to perform automated acceptance test and to track these.

The most important recommendations and conclusions related to SPI and XP are:

- **Improvements.** Rapid improvement of the iterations is possible. When the developers themselves give suggest improvement, they are easier to include in the development process. This also applies when the suggested improvements are relative small. Changes that demand extended work and more responsibility are adopted more slowly, even if the developers understand and see the benefits.
- **Learning.** Learning during the projects was mainly done by gaining knowledge from failures and success of earlier experiences and by constantly reviewing the current work. The goal of both learning and improvement are to get better as fast as possible.
- **Feedback meetings with PMA and KJ.** Having a feedback meeting after each iteration, where reflections and suggestions can be shared among the participants, were beneficial to the whole XP- and development process. The use of KJ gave a good basis for these meetings. The meetings should not last more than hour.
- **Use of GQM and/or daily logs.** By tracking and analysing the different iterations, we can discover what we have trouble with. Use characteristics and measures to learn what you aren't good at, and focus the improvements on this. However, the metrics must not be too extensive, because most developers will get tired of having too many metrics to fill out.

The most important recommendations and conclusions related to QA and XP are:

- **Re-negotiation in The Planning Game.** After the user story features are defined and the estimations were determined, the developers and customer get together for a re-negotiation. If the

estimations are far from what the customer expected, the features should be discussed to make sure that there are no misunderstandings between the customer and the developers. Possible misunderstandings can then be discovered before it is too late and the implementation has started. The customer can make clear what they wanted and the developers aren't developing wrong features.

- **Unit test and design.** The use of unit test gave better design. This test approach forced the developers to use clean, object-oriented design. Such design is also easier to maintain and the unit tests therefore contributes to better code quality.
- **Customer and risk.** We experienced that the customer must accept a larger part of risk associated with the development. This is a part of XP that may trouble many customers. If the customer decides to make large changes of the directions during the project, the customer must also be ready to handle the unsuccessful changes.

4.2 Case Study 2: A XP development project evaluated with PMA

Case Study 2 (CS2) is a study of an XP development project where the evaluation is based on interviews and PMA. CS2 is similar to CS1 (Case Study 1) in many ways, e.g. team size and duration. We will present the results from CS2 here, and compare these results with the results from CS1. Section 4.2.1 will give an introduction to the project and section 4.2.2 will present the results.

4.2.1 Project description

This project was carried out at the same location as CS1, SINTEF. The developers were two graduate students from NTNU and the project was part of their Diploma thesis. The customer and project leader came from SINTEF.

The system that they developed was a prototype of a electronic management system for personal information. This kind of system is often called a universal profile system and the final system handled user-control of personal information and the transferring of this information between the customer and the business.

The project duration was two and a half months and was divided into five iterations. Each iteration lasted between one and four weeks. The office arrangement was similar to CS1 described in section 4.1.2.2. This case study did not have iteration feedback meetings or any planned improvement strategy.

The final system consisted of approximately 7000 lines of Java code, where 5000 were production code and 2000 were test code. A challenge during the whole project was the large amount of new and unknown technology.

More information about the project and the system can be found in the thesis [57].

4.2.2 Case study results and conclusions

We will present the results from the case study by describing the experiences of all the XP practices together. We will then describe other experiences and recommendations from the project. A discussion about these results and the results from CS1 will be presented in section 4.3.

The system ended up being a usable prototype. This was valuable to the customer, and the project was considered successful. XP was used as methodology because of its agility, and the use of XP turned out well. Due to the new technology in the project, the ratio between production coding and spiking were roughly half to half, or maybe more. This stipulation is based on after-the-fact estimates.

The experience with each XP practice is presented in Table 4.8:

XP practice	Adoption status	Experience
The Planning Game	Adopted from start	Central for the communication between developers and customer, but when the customer was not available we were forced to do the business priority. There should have been more strictly rules for setting iteration deadlines, since some iterations lasted longer than planned.
Small releases	Adopted from start	Except for the second iteration, all the iterations lasted approximately one week. The second iteration was prolonged because we got problems with understanding the technology, thus prolonging the spiking period. This also resulted in a late usable first-release.
Metaphor	Adopted from start	The metaphor was valuable for the project. It took three weeks to really get used to the use of a metaphor. The metaphor was called "Secretary".
Simple design	Adopted from start	Sometimes we oversimplified the design, with quick need for refactoring. However, most of the design was based on some ahead planning. Some patterns were used.
Testing	Adopted from start	Some code that untested, but the understanding and use of tests did not change. It gave a solid foundation to understand the code and gave us courage to alter the code base when needed. Unit testing is highly recommended. GUI test were not automated, since this was never important to the project.
Continuous integration	Adopted from start	CVS was used all the time. The developers were used to this practice from previous projects.
Pair programming	Adopted from start	There is little point in doing trivial tasks in pairs. When there were one or more tasks regarded as trivial we split the pair and did the developing in parallel. If there is no one to pair with, some production code was developed individually.
Collective ownership	Adopted from start	Positive, however, the knowledge of different part of the system was still unequal.
Refactoring	Adopted from start	Was done regularly, the developers were not afraid of doing changes to the code. A good development tool makes refactoring more easily.
40-hours week	Adopted from start	Followed to the end of the project. Spiking gave some overtime.
On-site customer	Partially adopted	The customer was seldom physically in the same room as the developers, but could respond quickly to requests directly or by telephone.
Coding standards	Adopted from start	Not important when there are only two developers, because all production code is developed on pair, and the coding standard becomes an informal agreement between them.

Table 4.8: Experiences with the XP practices in CS2.

The experiences with XP, conclusions, and recommendations from this project are:

- **XP and research.** There were both positive and negative experiences when the project goal is to search for new solution areas. While all the new technology and unknown features made it difficult to define stories and estimates, the pace of the iterations gave a positive feeling of progress and evolvment.
- **Unit testing and product quality.** The high degree of unit testing made the code cleaner and more robust. Object oriented design and clearly defined class interfaces were forced through by the tests. The developers also got more courage to do refactorings. Together with pair programming they feel that the testing practice significantly increases the product quality.
- **Acceptance test automated with scenarios.** The automated and scenario based functional tests were efficient. Defining scenarios in the system and basing the tests on these gave a suitable solution for acceptance tests. The quality impact was proved by the releases; they were delivered without problems or “last hour fixing”.
- **Metaphor.** The usage of a metaphor was a suitable way to describe the system. If it is possible to agree on a metaphor, it is recommended to use it. You will thus have the ability to discuss the system with the customer without using technical phrases.
- **Learning and pair programming.** Before the project started, one of the developers knew much more about the system to be developed than the other did. At the end of the project the knowledge had spread and they both was almost equally familiar with all parts of the system. The developers believe that unless they had practiced pair programming, this shared knowledge had not existed. Thus, by doing pair programming, the result was a collective ownership of the code and not a divided ownership.
- **Learning and project control.** Except for the customer there were no control or following-of of the developers and how they attended to the XP practices or improvements to these. This resulted e.g. in extended iterations deadlines and some evasion of the XP practices, as pair programming. Iteration feedback meetings was requested but never performed. The developers believe that such meetings would have increased their self-review of their practices and their learning, and done the performance of XP more efficiently.
- **XP and highly skilled personnel.** The developers experienced that their skills in the new technology were not good enough. Knowledge of this technology would have increased the development effectiveness. This is a problem, also in projects not using XP, but lack of experience are more visible in an XP project for, instance due to the rapid feedback from the development progress.

4.3 Discussion of the case studies

Since the two case studies were evaluated differently and CS2 had no logs or other documentation during the project, we cannot compare the cases directly. We can for instance not see how the learning and improvement strategies, or the quality work performed in CS1 had impacts of the results compared to results in CS2. However, we can see how experiences with XP were different and we can try to find the causes of these different experiences.

Even though we found several differences, most of the experiences with XP gained by both the project were similar. The reason for this is most likely the equal settings, technical skills, experience with XP, office environments, and customer. We will not highlight these experiences further here, only referring to the respectively result- and conclusion sections in each case study.

The different experiences we will discuss are:

- Degree of new technology
- Metaphor
- Continuous integration
- Automated and scenario based acceptance tests
- Re-negotiation
- Learning process.

The degree of new technology and unknown features was larger in CS2 than in CS1. CS1 had a previous system to refer to and based the work on this, and the customer knew much what was wanted. However, the solution required much new technology. In CS2, only the idea known from the beginning was accessible, and everything else was decided along with the project. The ratio of spiking also shows us this difference: in CS1 this was one third of the time while in CS2 it was approximately done the half of the time.

CS2 succeeded with the usage of a metaphor. This was tried but not used in CS1. In CS1 they could not find an appropriate metaphor and never found this practice necessary. The team was small and both the customer and the developers knew the system good enough to discuss as it was. The success in CS2 was explained by the fact the system is based on an idea, and this idea could easily be described by their metaphor. Hence, they knew the idea before the complete system and it was more naturally to use a metaphor.

Continuous integration was used to a larger extend in CS2 than CS1. The reason for this is probably that the developers in CS2 was used to integrate all code to a CVS and continued with this habit in the project. Otherwise continuous integration is not so important where you only have one pair. The developers are mostly programming on the complete production code all the time.

The acceptance tests were automated more extensively in CS2. In CS1 most of the stories was centred on user interaction with the system and since a suitable tool wasn't found, the functional test were documented and performed manually. This was a good solution for CS1. CS2 had less GUI to test and the functional tests were implemented as scenarios. Extensions to functionality in the user stories were continually included in the scenarios that were tested. Hence acceptance test were fully automated.

In CS1 they included re-negotiation in the Planning Game and succeeded with this practice. The customer could comment the estimates and adjustments could be done when necessary. This was not possible in CS2. Here the customer had no idea of what the estimates should be and was not able to judge the correctness of these. Thus, re-negotiation was never used in CS2.

The final difference we will discuss is the learning process due to the performance of XP and improvements of this. CS1 included iteration feedback meetings in their XP practice. This had many positive effects, e.g. the team could discuss their work and suggest improvements. CS2 saw the usefulness in these meetings but never introduced them. The reason for this was that they discovered the benefit of feedback meetings too late in the project and was more focused on improving their technical skills than their XP skills at that time. The absence of this meeting and the underlying review of the development process resulted in e.g. extended iterations deadlines and some evasion of the XP practices, as pair programming. The developers in CS2 believed that such meetings would have increased their self-review and learning, and done the performance of XP more efficiently during the project.

5 XP, SPI, and QA: How to combine them

Based on the theory on Extreme Programming (XP), Software process improvements (SPI), and Quality assurance (QA), and our experience with XP, this chapter will investigate how SPI and QA can be combined with XP.

XP is a developing method that is adaptive and has quality as one of the central goals. Much of what we can associate with XP theory and practice, have elements of SPI and QA included. This chapter will illustrate these elements, describe what actions that could enhance their effects, and give suggestions to what we can do to achieve a better development process and result.

Our question “XP, SPI, and QA: How to combine them?” need to be more specific to be answered. We will in the following change the questions to: What elements of XP support SPI and QA and which don't? How can we react to the difficulties and what extensions need to be done to increase the efficiency of SPI and QA in XP? To answer these questions we will discuss SPI and QA separately. The relations of both SPI and QA to XP will be described and possible solutions will be presented.

Even though SPI and QA are discussed separately here, we must always consider them both to fully succeed in the usage of XP.

Section 5. 1 will cover the combination of SPI and XP, while section 5. 2 will cover the combination of QA and XP. Section 5. 3 give a discussion and a summary of our findings.

5. 1 SPI and XP

This section will describe the relation and combination of SPI and XP. We have learned that SPI is about learning how to work better. SPI on XP will therefore be about making the XP development process work better and creating an environment where this improvement can be done effectively. XP has several elements that favour improvements on the process. These will be described in section 5.1.1. There are also some difficulties for SPI in XP, and these will be described together with some solutions in section 5.1.2. How to increase the SPI efficiency in XP is described in section 5.1.3, while some SPI extension to XP is presented in section 5.1.4.

5.1.1 Existing SPI elements in XP

XP contains several elements that favour SPI. The development process in XP is mainly based on the decisions made by the developers themselves, and this reflects how the improvement of the process is done. We will highlight the parts of XP that makes learning and constant improvement natural parts of their process.

The following SPI elements already exist in XP:

- Communication
- Pair programming and sharing of knowledge
- Open office landscape
- Information on the walls
- Rapid feedback
- Improved estimations
- Metrics

The communication between the participants is central for sharing knowledge and information. Communication in XP projects is informal and are performed on several levels: between the programmers during pair programming, among the developers during the stand-up meeting at the beginning of each day, and between all the participants during the planning meeting in the Planning Game. The communication between the customer and developers goes on all the time since the customer is available on-site and the acceptance test communicates customer satisfaction and development progress.

Pair programming and collective ownership must be mentioned together. These are important means for sharing knowledge of the complete system and programming techniques. Programmers can learn from each other and they can be familiar to all parts of the system. Hence, the learning process is a natural part and will continuously improve the performance of the team.

The recommended way of arranging the office landscape is central to the XP development process. It should be an open office landscape where the developers are located near each other. Since the amount of documentation is minimal, much of the project information is taken up from the desks and put on the walls. This includes user stories, coding standards, and results of central metrics, such as acceptance test results. The user stories are prioritised and arranged to visualise which stories that are new, which are under implementation and which are finished.

Short iterations, short releases, and continuous integration give continuous feedback to the team of how they are doing. This means that if something is wrong or inefficient, this can be discovered early and handled immediately.

Looking at the Yesterday's Weather are improvement means for the estimations. This means that when estimating a story, the developers look at similar stories when estimates are give. As the developers get more experienced, the estimates will improve.

The XP literature discusses important metrics. By making the results of these available to the developers, they will know how they are performing and what they should improve. It is suggested to track and measure resources, scope, quality, and time [58, c.18]. This is central to discover the difference between planned and actual values, to improve the planning and make these differences converge. This can also be related to the steering that is discussed in the XP literature [58, c.19]. The steering is about tracking and managing the four elements; resources, scope, quality, and time. Good steering in XP is the same as SPI in XP.

5.1.2 SPI difficulties in XP and some solutions

There are elements or characteristics of XP that gives difficulties to SPI and the development process. These are described here and some countermeasures are suggested. Solutions to many challenges described here will also be presented in more detail in the sections 5.1.3 and 5.1.4.

The difficult issues for SPI in XP are:

- Few SPI solutions fits all kinds of projects
- XP in large teams
- Fast introduction of XP
- Sceptical participants
- Rapid pace of iterations
- Lack of documentation

Since the practice of XP will differ from project to project, depending on the customer, the developers, the amount of participant, and the scope complexity, there will be few general SPI solutions that will apply to all projects. Good solutions in one project may turn out to be unsuitable in the next. Hence, the solution must be more focused on adapting the project to the current need as fast and smooth as possible.

The introduction of XP into large teams may cause several problems – see chapter 3. XP is best suited for small or medium sized teams, so too large teams can cause difficulties. However, there can be made many adjustments to XP to suit large teams, and these adjustments gives potentials for SPI. The lack of communication in large teams can for instance be solved by organising experience and knowledge sharing better. As suggested in chapter 3 we should introduce XP only to a small part of the team from the start and expand when things are working. This also apply to every new XP teams, namely to introduce XP slowly, and not expect that a complete introduction of XP will work perfect from the start. It is also suggested that one should try practices that cause problems for some iterations before they are rejected. The start-up problems may be solved when the experience has matured.

The participating employees can make difficulties to XP and the improvement. XP as a method is dependent of the peoples that are using it, so if these are sceptical to XP or in

other ways resist the use or changes of the practices, this can cause problems. In [59] different “problem people” are mentioned. This could be an employee that either is frightened or insecure by the close communication e.g. during pair programming, is insulted because other look at and change his “perfect” code, is rigid to changes, or is threatened by the loss of status because expertise must be shared among all developers. These problems and internal scepticisms can be solved by mentoring, education and information, let people contribute to the decisions, incentives, or not force people to take part of the XP team. This last suggestion should be avoided. However, consider the problems a really sceptical participant may cause, it could be the only solution.

The rapid pace of the iterations may cause difficulties, even if they are considered positive side of XP. The lack of documentation may also be a problem – see the next sections.

5.1.3 How increase the SPI efficiency in XP

All the mentioned SPI elements in XP are positive and beneficial for improving the process. However, the practice is not always as easy and glimmering as the theory. The experiences we learned from chapter 3 and 4 and the difficulties mentioned in the previous section, tells us that XP and its usage also have weaknesses. The challenge is to make the theory work effectively and avoid the difficulties. Thus, by give the correct advices and make the conditions favourable, the SPI efficiency on XP can increase.

When we are using XP and doing SPI on our process, it is important to remember that XP is a starting point, but this starting point is not a perfect process [37, p.21]. If we are doing XP the same way after a half-year as we did in the first months, we have failed. We must let the process evolve to make XP suit the project and our people to succeed with performing XP. By achieving a suited XP solution and well performing people we have succeeded with the SPI.

We will in the following focus on:

- The local adoption of XP in 5.1.3.1
- Continuous learning and experience transition in 5.1.3.2.

Many of the topics in these sections are related to each other and these relations must be taken to consideration when we do SPI on XP.

5.1.3.1 Local adoption

We have learned that XP can be a self-adaptive process in chapter 2. We learned that by reviewing our process along with the project we were able discover difficulties and improve ourselves. There are several aspects of local adaptations and how these can be act as means for SPI on XP.

We will discuss the following adaptation issues:

- Adaptation stages
- Slowly maturation of the XP practices
- Adaptation away from XP

- Participant influence and motivation
- Participant skills and improvising
- Use existing experience
- Get used to the rapid pace

The local adaptation of XP goes through several stages – referred to as maturity levels in [60] and [9]:

- Level 1 is where you are doing XP by the book
- Level 2 is where you have adapted XP for the local conditions
- Level 3 is where you are transcendence and don't care whether you're doing XP or not.

The goal for SPI is to move a team through these levels as quickly and smoothly as possible. In other words, SPI on the XP process is to get better faster by improve the adaptation pace. However, we must take some precautions: First, a team should not reject a XP practice before it is really tried out. A practice may turn out to be suitable for the team if it is given the chance to be matured. This is also emphasised by many XP advocates. Second, improvements for the improvement sake must be avoided. Even if we are struggling towards further improvements all the time, we must sometimes slow down and let things settle before we do further improvements.

Due to the process of adapting the XP-practices to perfectly suit the local conditions, we must ask ourselves: When is it not XP? This is discussed in [61] and the conclusion is that the practices will change but the values of XP will not. As long as the values communication, simplicity, feedback, and courage are still intact, the team is doing their development the XP way. The “Level 3” above describes this state.

How can we adapt XP more efficiently? By being good at using our experience and learn along with the project is important. We will try to answer this question by looking at several aspects of the adaptation of XP.

The local adaptation is performed to suit the current participants and is most often done through suggestions from these. Due to their ability to influence the process, the participants in a XP project will be motivated and thus be more eager to improve than if others dictated the improvements. This was also observed in the first case study in chapter 4. When for instance the developers are suggesting the improvements, they see the need of them and will try them out as fast as possible. This motivation and eager can hopefully also reduce the situations where the developers don't want to improve. It is problematic if the developers get satisfied with their way of doing XP, but still should be improved. This is a general challenge for all improvement work and is difficult to solve. The use of incentives or other motivation factors could solve the situation.

In a constantly changing environment the degree of a successfully adoption lies in the skills of the participants. In [62] they discuss the importance of improvising in unpredictable environments. The participants must be able improvise, that is, handle the changes and act upon these. The two most important challenges for an improvisational approach to SPI are exploitation and exploration. An appropriate balance between exploitation – the adoptions and use of existing knowledge and experience – and exploration – the search for new knowledge, either through imitation or innovation. The balance problem is to undertake enough exploitation to ensure short-term results and, concurrently, to engage in exploration to ensure long-term survival. A software organisation that specialise in exploitation will eventually become better at using increasingly obsolete technology, while an organisation that specialise in exploration will never realise the advantages of its discoveries. Both the

adoption process and the search for better solutions are described in this section and the use of experience and continuous learning are described in the next section.

It may be useful to use the experience already available in the organisation. Do the company have any experience on XP from earlier projects? If yes, use and extend it. Use suiting techniques for gathering and transferring the experience into the new project. If no, make sure that experience is available by using a coach or facilitator to avoid the start-up problems. A XP course for the participant can be favourable. Also introduce some XP practices at the time, not all at once. Be familiar with them before a complete XP-project is launched [3].

When adopting XP we must be aware of the theory and implications that are described for rapid changing methods like XP. When we are working with XP we must get used to its rapid pace of iterations. Improvements of the XP process have many similarities to the rapid process improvements described in [18] and summarised in section 2.2.6. Here, we will only repeat some of the advices. First the speed: All improvements are simple and delivering value rapidly. Lessons learned must be available early and the improvements must be focussed on solving real problems. Secondly, improvements must be owned locally. Process improvement is rarely successful if done *to* people; it should be done *by* people. These advices are the same as our own findings in this and next section, with bottom up improvements and continuous learning. We have also discovered that these elements are close related to the nature of XP. We experienced the importance of local ownership of suggestions in our case studies in chapter 4.

5.1.3.2 Continuous learning and experience transition

There are no simple solutions to SPI on XP. We cannot you that you should do XP this or that way. From the previous section we learned that it is the process of local adoption that is important. We must therefore get the circumstances right to make this adaptation proceed more efficiently. SPI on XP must base its work on continuous learning and experience transition.

We will discuss the following topics here:

- What continuous learning is
- How access experience outside the team
- How share experience inside the team

Continuous learning is part of XP's spirit through its values and partly through its practices. To make the process of continuous learning more efficient, we must, however, explain in more detail how this should be performed. [63] address this subject quite well, and we will base our findings on the suggestions give there. Continuous learning is about reflecting on our work and gradually improve it. XP address this like follows: Customers and developers learn continuously from iteration planning; developers learn continuously from pairing, swapping pairs, testing, and refactoring; coaches learn continuously by communicating with everyone; and the entire team learn continuously from feedback generated by the XP process. Central to the learning process is to gather and use experience available and base the improvements on this.

Since there exist little or no documentation in the XP method, the experience transition must be based on direct communication between those with experience and those without. There

are two sources for experience in a project: the experience that exist outside the team and the experience included in the team:

- (1) **Experience outside the team.** This experience can come from the company or elsewhere. The challenge is to introduce it and make this experience available.
- (2) **Experience inside the team.** This experience is either gained by the participant through former projects or along with the current project. The challenge here is to distribute this experience to all the participants.

We will focus on (1) by suggesting some SPI extensions to XP in the next section. This cover documenting experience after projects and use a meeting in front of a project to gather and learn about earlier experiences. This is gathers the experience that exists inside the company. Experience outside the company is more difficult to gather since it is not directly accessible. The solution may be to collect experiences through studies like the one in chapter 3, or by using resources on the web, e.g. XP repositories such as [64].

To address (2) we will explain what already exist in XP and what we must add. In traditional XP, pair programming and stand-up meetings are means for sharing experience and knowledge inside the team. This works to some degree, but may not be efficient enough. One solution is to share experience and knowledge more broadly after each iteration. This is suggested but not described in [54, c.27], and described more generally in [65, c.5]. Here, a reflection workshop is suggested. This should be performed often, not just at the end of the project. The general idea is to perform the meeting after each increment to address two questions: What did we learn and what can we do better. These reviews will make it possible to learn faster and not only take advantage of our experience in later projects, but to include them now. It will be important to gather experience and opinions on the current work to adjust the way things should be done in the next iteration. Hence, knowledge and experience from the previous iteration is input to the next one.

In the next section we suggest how these short feedback meetings should be arranged, by using the KJ¹ method.

5.1.4 SPI extensions to XP

Even though there is little information about SPI and XP available, there exist several good ideas and suggestion in several places. We have performed an extensive research and study of available literature and collected the “grains of gold”. Based on this we have done some further findings, and those related to SPI are described here.

We will describe to following subjects:

- Iteration feedback meetings with KJ in 5.1.4.1
- XP project analysis with GQM, including the usage of logbooks in 5.1.4.2
- Distribution of project experience with PMA and repositories in 5.1.4.3

¹ KJ is described in chapter 2.

5.1.4.1 Iteration feedback meetings with KJ

The iteration feedback meetings are used as means for SPI in the XP process. The idea is to use these meetings to share knowledge and experiences from the last iteration and to suggest improvements for the next one. The KJ method is suggested as basis for these discussions. The iterations feedback meeting and the use of KJ are tried out during the first case study described in chapter 4.

The feedback meeting should be arranged as follows:

- **Location.** The locations should be in the same room as where the development takes place. Here all needed information. e.g. the user stories, are directly accessible.
- **Participants.** Everyone involved in the project should participate. Hence, experience and knowledge from all part of the project could be shared.
- **Facilities.** The meeting should be informal. Everyone standing round a table is desirable but not required. Have coffee and tee available.
- **Duration.** The duration of a meeting is depending on the team size, but the meetings should not last for more than a half-hour, maximum one.
- **Agenda.** The direct communication is most important. To systemise the meeting it should include a short summary of the last iteration. Both developers and customer should contribute. Then a KJ session, discussing positive and negative aspects of the last iteration. Finally a discussion on improvements.
- **Responsibilities.** Responsible for the meeting should circulate among the participants. This responsibility will not include much work, but just having someone responsible is often necessary to keep things on track.
- **Conclusions.** The outcome from the meeting should be distributed and understood by all the participants before the meeting is finished. All improvements and adjustments must be understood by everyone to get effectively introduced. This distribution should only be communicated orally. However, it could be favourable to document the most important findings to make them possible to track later.

By arranging these meetings we are able to share experience and knowledge among all the participants of the project more efficiently. This is an important extension to the traditionally XP.

KJ can be used as a basis for several discussions during the feedback meeting. The central theme we have suggested has addressed these questions:

- Positive experiences in the last iteration
- Negative experiences in the last iteration

First, everyone present their opinions, written on post-it notes. Then the notes are systemised in groups of relevancy. The outcome from this process is used as the basis for further discussion. Positive experiences are highlighted while negative experiences are given improvement suggestions. The result is suggested adjustments for the next iteration.

These reflections and the wish to improve in the XP process¹ will ease the SPI of XP.

5.1.4.2 Project analysis with GQM and logbook

To be able to analyse the XP process in more detail, we suggest using the GQM method² as a basis. To gather information about the development involvement we suggest using a diary. In this way we will be able to collect both quantitative and qualitative data. The results can be analysed during the project or after it is finished.

We have tried this approach during the first case study described in chapter 4. The results from this method are dependent on the focus and questions we define for the GQM abstraction sheet. We must thus be relatively sure on what we want to find out when we are planning the metrics to be collected.

A possible outcome of the analysis is to discover relations in the development process that aren't directly obvious to the developers or other participants. We also want to find possible unexpected results. These deviations from what we believed can then be analysed further, and we can discover new parts of the process to improve.

The diary is used for data collection to the GQM metrics, but can also be used separately. By logging a few characteristics and opinions of the work done each day, the logbook can be used for analyses and documentations of the development process. The logbook should not be too extensive and it should not take more than a few minutes to fill it in each day.

5.1.4.3 Distribution of project experience with PMA and repositories

We have earlier described the importance of using experience from earlier projects. We suggest two ways of transferring this knowledge. The first is to perform a PMA³ before the XP project. Here the main lessons from other projects are identified and collected. Second, after the project is finished a second PMA is held. Here the projects is analysed and the experience is brought forth as recommendations for later projects. To be able to save experiences and knowledge from a XP project in a permanent form we also suggest documenting these and the recommendations in repositories.

We feel that these arrangements will increase the efficiency of experience transition and with that, makes the local adoption faster. To get an idea of how this should be arranged, we describe the PMA and the experience repository in more detail here.

In [66] some of these ideas are described in general terms and suggested used together with GQM. We will only focus on the PMA here. In Figure 5.1 we also see how feedback meetings – as described in 5.1.4.1 – are performed during the project.

¹ The characteristics of the XP improvements are described in section 5.1.3.1 above.

² GQM (Goal Question Metric) is described in chapter 2.

³ PMA (Post Mortem Analysis) is described in chapter 2.

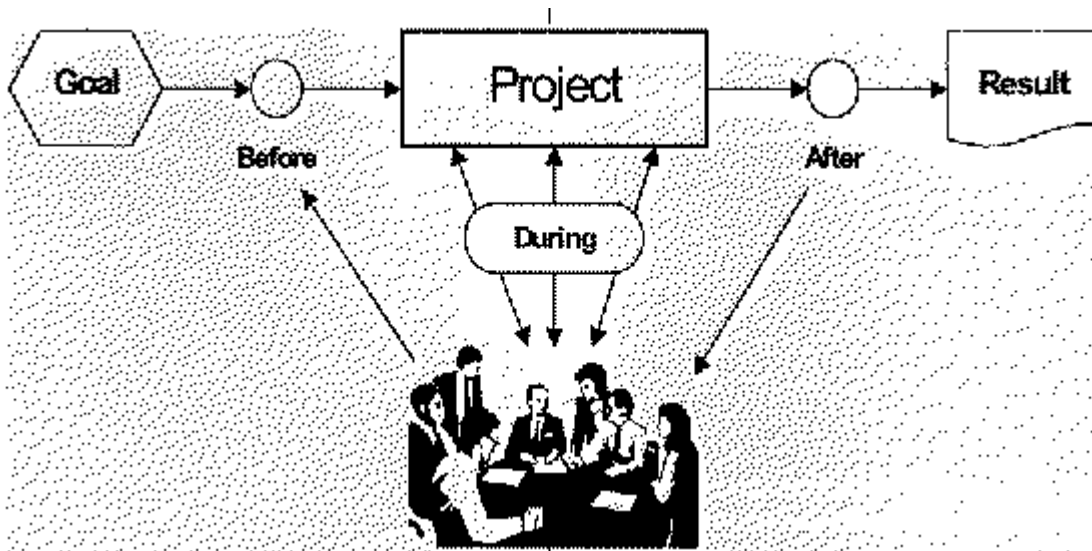


Figure 5.1: PMA before and after a project [66].

The PMA done before project start is motivated by asking who has done similar XP projects before and what can we learn from them. Experience repositories can also be used as input. Through a brainstorming – using KJ – and a discussion, the main lessons learned from other projects are identified and prioritised. The experiences and solutions that suit the current project are then introduced. At this point we must also be aware of the biases of earlier experiences [67]. The environment varies from project to project and this may have a large influence on how things turn out.

To assemble the experiences after the project, a PMA is performed. Here we are focusing on what went well and what did not go so well in the project. The brainstorming and discussion are performed in a way similar to the before-project-PMA. The outcome from this meeting is then given as recommendations for later projects.

These and other experiences may be stored in several ways. In chapter 3 we described how experiences were documented in an experience report. These should not be too extensive. The XP philosophy is not in favour of heavy documentation, so we should keep it as low as possible. Experience reports are useful for transferring knowledge and experience especially if the persons holding this knowledge and experience aren't accessible in later projects. This also applies to experience repositories. We suggest introducing repositories in companies to also make it possible for participants to insert experience during projects. An online repository database, available on the company's intranet would be beneficial. The benefits of such databases are often claimed as overvalued because they are "never" used. We believe that the nature of XP and its participants supports these kinds of solutions. People using XP are part of the problem solving and they are eager to find solutions to their problems. Sharing experience is also appreciated.

5.2 QA and XP

As we learned in chapter 2, QA is about making sure that the quality of the product is as required, and that the customer is satisfied with the result. XP contains several elements that assure quality. These will be described in 5.2.1. There are also several difficulties in the relation between QA and XP, and these will be described together with possible solutions in 5.2.2. How to ease the introduction of QA and possible extensions for QA to XP are described in 5.2.3 and 5.2.4, respectively.

5.2.1 Existing QA elements in XP

XP contains several elements that favour QA. Quality is central to XP and the assurance of quality is therefore implied parts of many of the practices.

The following elements and XP practices are described:

- Rapid feedback
- The Planning Game
- On-site customer
- Small releases
- Continuous integration
- Refactoring
- Pair programming
- Unit testing
- Acceptance testing
- Coding standards
- Simple design.

The overall QA in XP is taken care of by the constant and rapid feedback. The feedback are done on many levels, from unit testing giving feedback from the code to the developers, the continuous integrations giving feedback of the system state to the developers, the acceptance tests giving feedback of requirement fulfilment to customer and developers, to the Planning Game forcing feedback between the customer and developers about focus, priorities, and estimates. When working in iterations the projects, both the customer and the developers are able to adjust performance along with the project to give most value to the customer. This could also mean to end a project before it is completed, event though this is not often. An early closure of a project is cheaper for the customer than late closures, if the project never was “meant to be”. Furthermore, this constant feedback and rapid pace of iterations assures that the project can keep up with its time budget and is able to deliver a satisfying product on time. A shortening on time will give the customer an option to reprioritise the remaining features. In general we can say that problems are discovered along with the project and not in the end.

The Planning Game is central for QA in XP. Here, the customer is able control the evolvment of the project from iteration to iteration. The requirements are described in stories and implemented in the order the customer requires. The developers give feedback as estimations and finished stories. The customer is able to change the priorities and features all along the project to get most value out of the project. Acceptance tests are used by the customer to control the progress and for the developers to see how they are performing.

The on-site customer practice ensures a good communication between the customer and the developers. Hence, misunderstandings can be discovered and adjusted early and immediately.

Small releases give the customer a working version of the system. The ability to view the current system is valuable to the customer since the implemented features can be tested and adjusted.

Continuous integration is beneficial to the developers. When integrations are done often, it is less integration problems each time than if the code is integrated seldom. Bugs and system inconsistencies can be detected and fixed at early stages.

Refactoring can result in simple and modifiable code. These results are both beneficial for the system quality.

Pair programming results in fewer defects in the code. This is confirmed by research described in chapter 3. The reason for this is mainly the continuous code review while programming. Bugs and defects are easier discovered when one in the pair is concentrating of the resulting code. The programmer will be more concentrated on solutions and will oversee bugs.

Several levels of the testing practice are beneficial to quality. The unit tests ensure that the code does what we expect it to do. If the code is changed, e.g. during refactoring, we can confirm that the code is still working by running all unit tests.

Acceptance tests are central to QA. They ensure that the customer gets what's requested and that these requests are kept all along through the project. The primary measure of quality would be the customer's acceptance tests. If the program is passing all the tests that have ever been run on it, it is hard to argue that there is much wrong with it. Further, metrics of the test results will indicate how the development is doing, by seeing if tests still are passing or not.

Coding standards ensures clear and uniform code. This is beneficial to quality by making the code easy to read, and together with the simple design, the code base can function as part of the system documentation.

5.2.2 QA difficulties in XP and some solutions to these

In the previous section we listed the benefits of all the practices for the system quality and QA. How can XP then go wrong? We can say that the intentions of XP are good, but there are several barriers and situations that can result in QA difficulties. By doing SPI on XP as described in chapter 5. 1, we are able to improve the possibilities for doing QA. In this section we will focus on what parts of XP that may give difficulties when doing QA in a XP project. Some countermeasures are also described, but solutions to the challenges described here will also be presented in the next sections, 5.2.3 and 5.2.4.

We will describe the following difficulties for QA:

- Customer and risk
- The customer and the customer role
- Participant skills

- Design changes
- Customer approval
- Conformance to standards
- Non-functional requirements

The customer plays an important part of the development process and this may cause QA difficulties. The close inclusion of the customer in the project can result in enlarged risk for the customers themselves. There are only positive sides of this practice if the project runs without large problems. Problems occur, the inclusion of the customer involves more responsibility, and the customer must pay for the problem recovery. This challenge must be made clear to an inexperienced XP customer. If the customer knows about the risk responsibility, the decisions and priorities will be based on strategic foundations. This was for instance experienced in one of the projects described in chapter 3.

It is also important that the customer-representative know his or her role. The customer must represent the end users of the system to be developed. It is a problem for QA if the customer on-site is satisfied with the product and when the system is tried and delivered, many problems are reported. If the small releases are demonstrated to the real user and not only to the on-site customer, this problem can be avoided. By coaching the customer and warn about this problem we can reduce this problem. Sometimes, the practice of having an on-site customer is not possible. In these situations it is even more important to give good coaching and warn about the risk that is involved. Since the communication with the customer must be performed in other ways than directly on-site, the challenges mentioned here would be clearer.

Another issue that has a large impact on the quality and success of an XP project is the quality of the developers. Everyone involved are given much freedom and equally responsibilities. It is up to the developers to define and improve the development process. E.g. the amount of spiking before a solution is found depends heavily on the developer's technical experience and skills. In short, XP requires senior competence and above average developers to succeed. Since XP often is used to develop innovative systems the developers must also have cutting edge competence to solve technical issues fast enough. This is a problem for all parties including QA, because the time overrun may be considerable before a solution is found, if solution is found at all. There are no simple solutions to these problems, but it is important to be aware of them, and base the customer agreements and estimates on this will make it easier. It is not easy to admit that the participants are incomplete or inexperienced, but if overlooking this means that the project doesn't succeed, we must consider it. Nevertheless, if the team consist of one or some senior developers, the mentoring through pair planning and direct communication in XP will solve some of these problems.

The contrasts between simple design and refactoring may cause difficulties for QA. If a system evolves to be large and complex, and the design is based on architecture for the today-solution, large changes of functionality will involve large refactoring. This is experienced in several projects – see chapter 3 and the case study in chapter 4 – and the suggested solution is to do a little ahead planning of the design. Planning the design based on similar solutions and all available user stories will make the design more general and more resistant to large refactorings. Further, the customer must be more moderate when demanding changes to the existing system. The consequences must be considered. Rapid and small refactoring to remove duplicate code and architecture inconsistencies along the project will reduce the refactoring in the end of the project.

We have also found out that pure and hard-core XP only suits a minority of the customers. This assertion is based the experience with XP described in chapter 3. There are not many

customers that are willing to agree to the unpredictable circumstances that XP relies on. Some adjustments must often be made to a XP project to satisfy customers, see section 5.2.4.

Central to QA and Quality certification is conformance to standards. This was introduced in chapter 2. In particular, ISO 9000 certification is becoming more and more important for firms that want to make business with Government agencies. The difficulties and contrasts between XP and other agile methods and these certification standards are many. This is discussed in [68] and [69] but are not yet published, so this subject is postponed for further work.

Non-functional requirements, e.g. security or performance issues, are described as problems for QA in XP projects [70]. However, if the coverage of these issues is valuable for the customer, concrete requirements can be defined as user stories and included in the development. If, for instance one non-functional requirement was that the system should bear a load of 1000 simultaneous users, this can be described in a user story and a load test as acceptance test, could solve the problem.

5.2.3 How to ease the introduction of QA in XP

To be able to take advantage of all the quality elements that is part of XP and handle the difficulties described on the previous section, we must take some precautions and make some changes that make XP more suitable for QA. Just being aware of what may go wrong when using XP is a good start for QA. The awareness of these difficulties can bring out solutions that prevent the problems, and by having an ideal working environment, we may be able to avoid them. In the following sections we will represent some suggestions that may ease the QA in XP.

We will discuss these topics:

- Having a dedicated QA team or QA person, in 5.2.3.1
- Usage of automated test, in 5.2.3.2

5.2.3.1 Include a dedicated QA team or QA person

It is important for QA to have someone responsible for QA in the project. The most common suggestion – see chapter 3 – is to have a dedicated QA team or a QA person included in the XP team. This is also agreed in [71]. The QA team or QA persons are responsible for the overall quality control and for writing acceptance test (each developers are still responsible for writing unit tests). Taking part in the customers meetings and being present with the developers assure the overall quality. By having direct contact with both customer and developers, advices and suggestions can be given immediately and directly when it is needed.

The QA responsible should also have authority to hold a release back until important quality issues are solved. By having own QA responsible will also relive much of the administrative burden of the quality process from the rest of the project team. The developer's can then focus on what they know best, namely developing.

It is vital that the QA responsible is a resource and help for the developers, and not are viewed as a controlling instance that will become unpopular among the developers.

5.2.3.2 Use automated tests

To make the testing process effective, the tests should be run with automated testing tools testing suites. This is recommended by several projects in chapter 3 and in the case study in chapter 4. Unit tests and acceptance tests together constitute the practical QA in XP [71].

The unit tests are programmed during the test first principle. The XJUnit testing framework [55] is widely used and is recommended for automating the unit tests. XJUnit supports programming languages like Java, C++, Smalltalk, and so forth. If the system is large and the testing involves database operation, the execution time of the unit tests may be long [72]. To reduce execution time it may be smart to replace the database with an in-memory database.

To make the unit testing less expensive it is suggested to use patterns in the test code. ObjectMother [73] is a pattern used to make the maintenance of test objects easier. Its purpose is to build test objects and put them in certain states. This is often a large part of the testing process, so a general way of doing this is useful.

The acceptance test shall control that the implemented stories satisfies the requirements given by the customer. We recommend that the QA personnel write these tests. To be able to handle all the acceptance tests in a satisfying way, they should be automated [74]. The automation of these tests can either be done by programming own test suites or by using testing tools.

GUI testing can be an exception [75]. If the GUI is changing rapidly, there is too much overhead in updating all the testing scripts each time. It is suggested that the system functionality and system logic not are implemented near the GUI, and that the functional testing could be done on these lower levels.

In [76] they claims that it is the QA responsible that steers the project. We will partly agree to this by suggesting that the QA responsible has authority above the developers (see the previous section 5.2.3.1). It is also important that the QA responsible is ahead of the developers. By writing the acceptance tests before the story is put into development, the developers know exactly what they need to do to get the story approved, when they start the implementation.

5.2.4 QA extensions of XP

Many of the suggestions above are extensions to the XP environment. The suggestions are based on experiences from the industry in chapter 3 and from the case studies in chapter 4. Most of these extensions are practical solutions to the XP theory. By trying out XP, several suggestions and solutions to QA are introduced to improve quality and QA in XP. Since our findings of QA extensions to XP are similar to these suggestions and solutions, we will not describe these more detailed than already done in section 5.2.2 and 5.2.3.

5.3 Discussion

The discussion in this section will sum up the results and findings from this chapter. We will first point out some thoughts we have of XP and its combination with SPI and QA, and then we will summarise the results.

Since XP is a relatively new development method, there are not done much work, and hence not presented many publications about SPI or QA with XP. However, there exist several good ideas and suggestion on several places. We have performed an extensive research and done our own observations through case studies to present challenges and solutions for combining SPI and QA with XP.

XP is an agile and lightweight methodology with the programming code and the user stories as the main documentation. The development process is controlled in a large degree by the developers themselves. When doing SPI on XP it will therefore be most important to be better in adapting it and trust on local solutions. Experience and knowledge sharing is important to make this happen effectively, and finding good solutions for such sharing is central. The process of learning quickly and share experience are thus more important for XP than traditional methodologies. Our findings will be focused around these topics.

XP are focused on quality and the XP practices supports this focus. The rapid feedback makes it possible to discover problems early and enables the team to do something with these problems. However, there are several barriers and situations that can result in QA difficulties. The customer is as important for the result as the developers. This must be considered. The testing practice is also central for QA in XP. Our findings will focus on suggestions and solutions that are effective and possible to introduce for all kinds of XP projects

The most important results from our descriptions of possibilities and challenges when combining SPI and XP are:

- **XP has already several SPI elements.** The nature of XP and its practices includes several elements that favour SPI. The most important are the direct communication and knowledge sharing between the participants. The rapid feedback gives the team a constant indication of what to improve.
- **Few SPI solutions fit all kinds of XP projects.** The solution is to be more focused on adapting XP to the current need as fast and smooth as possible.
- **The local adaptation is important.** During maturation of the XP practices and small improvements in rapid pace, the team will adapt XP to suit their current needs.
- **Improvements must be motivated by the participants themselves.** This will increase the participant's eager to improve the process and make the adaptation of XP more efficient.
- **Continuous learning improves the team.** By reviewing our own practice we will discover possibilities for improvements. Pair programming and stand-up meetings ensures sharing of knowledge and experience.
- **Experience transition must be done effectively.** Experience outside and inside the team must be made accessible. This will improve the learning and adaptation process.

- **Iteration feedback meetings with KJ.** To improve the process, these meetings can be used to discuss the project and suggest improvements.
- **Do project analysis with GQM.** To be able to document the process a little bit more, use GQM together with developer diaries to collect information for further analysis. Unexpected relations in the project can be discovered and make further improvements possible.
- **Use PMA and repositories to share experience.** PMA before and after the project will make the transition of experience between projects more effective. Storing experiences in repositories during and after projects is also a solution. We have seen that developers are taking central parts of improving their process, and they will be eager to share and gather experience in such repositories.

The most important results from our descriptions of possibilities and challenges when combining QA and XP are:

- **XP contains several elements that favour QA.** The rapid feedback ensures that problems are discovered early and can be handled fast. Most of the practices ensure quality on many levels, from fewer bugs in code to customer satisfaction.
- **The customer can steer the development.** The customer can decide and prioritise what to do from start to end of the project, to get as much value out of the project as possible.
- **The customer role and real users.** The customer representative must also represent the real users and base the decisions on this. It is not a good idea to develop a system based on requests from the customer's managers if the real users don't agree on these.
- **XP involves more risk to the customer.** By taking such a central part of the development, the customer must take a larger part of the responsibility of something fails.
- **XP is dependent on qualified personnel.** Everyone involved are given much freedom and equally responsibilities. The qualifications are vital and must be taken into consideration when agreements and estimates are done.
- **There should be a dedicated QA team or QA responsible.** This is important to ensure and control that quality actually is taken care of. This will also relieve much work from the developers.
- **Use automated tests.** Both unit and acceptance test should run automatically to increase the efficiency of the testing process. As more automated they are, the more often they can be run.

6 Conclusion and further work

This chapter will conclude with our findings and own contributions, and give some indications of possible further work in the area of XP, SPI, and QA.

6.1 Conclusion

In this thesis we have learned that XP is a development methodology with positive and negative sides. XP has many elements that favour both SPI and QA and other parts that are more problematic.

The existing experience with the usage of XP, tells us that the XP theory and practices are a good starting point for many projects. However, to function effectively and succeed with XP, there must be done adjustments to this starting point in each project that uses it. XP handles both rapid change of requirements and environments, and the strategies in XP are therefore based on the current needs and circumstances. Since XP will turn out so different from project to project and will evolve and change from iteration to iteration, we must focus on being good at doing this – being adaptive – and not settle down with the way we are doing XP today. Solutions that were successful in other projects may not work out at all in another.

We must therefore concentrate the work with SPI in XP by make each XP project adapt to local conditions as fast and smooth as possible. The XP team must be aware of how it is performing by reviewing itself and search for better solutions. Continuous learning by using and sharing all available experience and knowledge is important.

When it comes to QA and XP we must focus on all the elements in XP that ensures quality and make sure that they are fully exploited. XP does not fit all kinds of projects or customers and this must also be highlighted. Our most important findings related to QA in XP are to extend the development team with a separate QA team or a QA responsible and automate all testing of code and functionality. When someone is responsible for QA it gives a certainty that the quality really is taken care of. Further, by automate the tests, the testing practice will be more effective.

To extend the learning process in XP and make the local adaptation more effective we have suggested three extensions to XP as means for SPI. First, to increase the knowledge and experience inside the whole team, we suggest introducing an iteration feedback meeting after each iteration. Here the last iteration is reviewed with the use of KJ to discover problems and introduce improvements. Second, to make deeper analysis of the process possible, we suggest documenting the process more broadly by using a developer diary and the GQM method to gather information about the process. Third, to make the experience from earlier XP projects more accessible, we suggest using PMA before and after a project and store experiences and recommendations in repositories during the project. The use of iteration feedback meetings was tried out in one of our case studies with success.

6.2 Further work

Since there are published a so minimal amount of theories and solutions on the areas covered in this thesis, I suggest that the most important now is to gain more empirical experience on these issues. By trying out the findings in this thesis, or other ideas, we will be able to find out how SPI and QA can be done more effectively. If we can succeed with this work, I think XP has a great future as a development method for many kinds of projects.

I will also remind that the relation between XP – and agile methods – and QA are discussed on several workshops at conferences as this thesis is written [68] / [69]. The outcome from these will give further ideas of what can be done in this area.

7 Abbreviations and acronyms

This is a list of abbreviations and acronyms used in this thesis.

ASD	Adaptive Software Development
C3	Chrysler Comprehensive Compensation
CS1	Case Study 1
CS1	Case Study 2
CVS	Concurrent Versions System
DBA	Database Administrator
DSDM	Dynamic System Development Methodology
FDD	Feature Driven Development
FCT	Fast Cycle Time
GUI	Graphical User Interface
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standard Organization
JSP	JavaServer Pages
LDUF	Little Design Up Front
NTNU	The Norwegian University of Science and Technology
PDCA	Plan-Do-Check-Act
PP	Pair Programming
QA	Quality Assurance
QDS	Quick Design Session
QE	Quality Engineer
QIP	Quality Improvement Paradigm
RPI	Rapid Process Improvement
SEE	Software Engineering Education
SINTEF	The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology
SPI	Software Process Improvement
TQM	Total Quality Management
TSTCPW	The Simplest Thing that Could Possibly Work
XP	Extreme Programming
YAGNI	You Ain't Gonna Need It

8 References

Links to some of the papers and resources can be found in Appendix B. Some references are specified more precisely with the following abbreviation in the text: c. – chapter, s. – section, and p. – page.

- [1] K. Beck. *Embracing Change with Extreme Programming*. IEEE Computer, 1999.
- [2] C3 Team. *Chrysler Goes to “Extremes”*. Distributed Computing, pp. 24-28, October 1998.
- [3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [4] M. Fowler. *The New Methodology*. ThoughtWorks Library. November 2001.
- [5] AgileAlliance. *Manifesto for Agile Software Development*. Agile Alliance, 2001.
- [6] J. Highsmith, A. Cockburn. *Agile Software Development: The Business of Innovation*. IEEE Computer, September 2001.
- [7] A. Cockburn , J. Highsmith. *Agile Software Development: The People Factor*. IEEE Computer, November 2001.
- [8] B. Boehm. *Get Ready for Agile Methods, with Care*. IEEE Computer, January 2002.
- [9] M. Fowler. *Variations on a Theme of XP*. Martinowler.com. Accessed February 21, 2002.
- [10] N. L. Kerth. *Project Retrospectives: A handbook for team reviews*. Dorset House Publishing, 2001.
- [11] T. Dybå et. al. *SPIQ Metodehåndbok V3.0* (in Norwegian). SINTEF / NTNU / UiO / Telenor Geomatikk, January 2000.
- [12] T. Dybå. *Enabling Software Process Improvement: An Investigation of the Importance of Organizational Issues*. Doctoral Dissertation. Norwegian University of Science and Technology, November 2001.
- [13] V. R. Basili, G. Caldiera, H. D. Rombach. *The Goal Question Metric Approach*. Encyclopedia of Software Engineering, Vol. 1, pp.528-532, John Wiley & Sons, 1994.
- [14] A. Birk, T. Dingsøy, T. Stålhane. *Postmortem: Never leave a project without it*. IEEE Software, Special issue on knowledge management in software engineering, 2002.
- [15] R. Scupin. *The KJ Method: A Technique for Analysing Data Derived from Japanese ethnology*. Human organization, Vol. 56, pp. 233-237, No 2, 1997.
- [16] K. J. Wedde. *TKL – 7 ledelsesteknikker* (in Norwegian). SPIQ, Norway, 2000.
- [17] M. C. Paulk, C. V. Weber, B. Curtis, M. B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

-
- [18] C. Shelley, I. Seward. *Rapid Process Improvement*. Oxford Software Engineering. International Conference on Software Process Improvement (SPI2000), Gothenburg, Sweden, 4-7 December 2000.
- [19] P. Halloran. *Organisational Learning from the Perspective of a Software Process Assessment & Improvement Program*. Proceedings of the 32nd Hawaii International Conference on System Sciences 1999. IEEE 1999.
- [20] C. Argyris, D. A. Schön. *Organizational Learning II: Theory, Method, and Practice*. Addison Wesley, 1996.
- [21] C. Rhodes. *Researching Organisational Change and Learning: A Narrative Approach*. The Qualitative Report, Vol. 2, Number 4, December 1996.
- [22] M. Dodgson. *Organisational learning: A review of some literatures*. Organisation Studies, 14/3: 375-394, 1993.
- [23] G. P. Huber. *Organisational Learning: The Contributing Process and the Literatures*. Organisational Sciences, Vol. 2, No 1, February 1991.
- [24] J. S. Brown, P. Duguid. *Organizational Learning and communities of Practise: toward a Unified view of Working, Learning, and Innovation*. Organization Science, Vol. 2, No 1, pp. 40-57, February 1991.
- [25] C. Leverentz, H. Rust, F. Simon. *A Model for Analyzing Measurement Based Feedback Loops in Software Development Projects*. 3rd International Workshop on Learning Software Organizations (LSO'01), 2001. Lecture Notes in Computer Science, No 2176, pp. 135-149, 2001.
- [26] IEEE Std 730-1998. IEEE Standard for Software Quality Assurance Plans. IEEE, 1998.
- [27] ISO/IEC 14598-1. *Information Technology – Software Product Evaluation. Part 1: General Overview*. Joint Technical Committee ISO/IEC JTC1 Information Technology, May 1998.
- [28] ISO/IEC 9126-1. *Information Technology – Software quality characteristics and metrics. Part 1: Quality characteristics and sub-characteristics*. Joint Technical Committee ISO/IEC JTC1 Information Technology, July 2001.
- [29] G. M. Weinberg. *Quality Software Management, Volume 1 Systems Thinking*. Dorset House Publishing, 1992.
- [30] A. Aune. *Kvalitetsdrevet ledelse – kvalitetsstyrte bedrifter* (in Norwegian). Gyldendal Akademisk, 2000.
- [31] J. Bøegh. *Quality Evaluation of Software Products*. Software Quality Professional. Volume 1, number 2, pp. 26-37, 1999.
- [32] International Standard Organization. ISO Online, 2002.
- [33] W. Strigel. *Reports from the Field: Using Extreme Programming and other Experiences*. IEEE Software, November / December 2001.

-
- [34] A. Elssamadsy, G. Schalliol. *Recognizing and Responding to “Bad Smells” in Extreme Programming*. ACM, 2000.
- [35] A. Elssamadisy. *XP On A Large Project – A Developer’s View*. Presented at XP Universe Conference, 2001
- [36] A. M. Hassan, A. Elssamadisy. *Extreme Programming And Database Administration: Problems, Solutions, and Issues*. ThoughtWorks Library, 2002.
- [37] C. Taber, M. Fowler. *An Iteration in the Life of an XP Project*. Cutter IT Journal. Vol.13, No. 11, November 2000.
- [38] C. Poole, J. W. Huisman. *Using Extreme Programming in a Maintenance Environment*. IEEE Software, November / December 2001.
- [39] C. J. Poole, T. Murphy, J. W. Huisman, A. Higgins. *Extreme Maintenance*. XP Universe 2001, July 2001.
- [40] J. Grenning. *Launching Extreme Programming at a Process-Intensive Company*. IEEE Software, November / December 2001.
- [41] K. Dunsmore, C. Wiemann, G. Wolosin. *A Practical Application of XP*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [42] P. Schuh. *Recovery, Redemption, and Extreme Programming*. IEEE Software, November / December 2001.
- [43] M. Lippert, S. Roock, H. Wolf, H. Züllighoven. *XP in Complex Project Settings: Some Extensions*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [44] F. A. Adrian. *micro-eXtreme Programming (μ XP): Embedding XP Within Standard Projects*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [45] L. Williams, R. Upchurch. *Extreme Programming for Software Engineering Education? 31st ASEE/IEEE Frontiers in Education Conference*, Reno, NV. October 2001.
- [46] J. Kivi, D. Haydon, J. Hayes, R. Schneider, G. Succi. *Extreme Programming: A University Team Design Experience*. IEEE. Electrical and Computer Engineering, 2000 Canadian Conference on, Vol. 2, 2000, pp. 816.820.
- [47] M. M. Müller, W. F. Ticky. *Case Study: Extreme Programming in a University Environment*. IEEE, 2001.
- [48] J. Haungs. *Pair Programming on the C3 Project*. IEEE Computer, February 2001.
- [49] L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries. *Strengthening the Case for Pair Programming*. IEEE Software, July/August 2000.
- [50] A. Cockburn, L. Williams. *The Costs and Benefits of Pair Programming*. International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000), June 2000.

-
- [51] G. Succi, M. Stefanovic, W. Pedrycz. *Quantitative Assessment of Extreme Programming*. Electrical and Computer Engineering, 2001. Canadian Conference Electrical and Computer Engineering, Volume: 1, 2001.
- [52] T. Johansen. *Ekstrem Programmering, Praktiske erfaringer* (in Norwegian). Proxycor AS. February 2002.
- [53] Ø. Mollan, O. J. Lefstad. *En undersøkelse av Ekstrem Programmering for å se på effekten i et utviklingsprosjekt* (in Norwegian). Diploma thesis at the Software Engineering Group, Department of Computer and Information Science, Norwegian University of Science and Technology, June 2002.
- [54] K. Beck, M. Fowler. *Planning Extreme Programming*. Addison-Wesley, 2001.
- [55] JUnit, *Testing Resources for Extreme Programming*. JUnit, 2002.
- [56] Mercury Interactive. *Astra QuickTest 5.5*. Mercury Interactive, 2002.
- [57] S. Stornes, J. B. Sandnes. *Development of a prototype of a universal profile system using a personal secretary*. Diploma thesis at the Information System Group, Department of Computer and Information Science, Norwegian University of Science and Technology, June 2002.
- [58] R. Jeffries, A. Anderson, C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley 2001.
- [59] L. A. Griffin. *Managing Problem People in XP Implementation*. Yahoo Newsgroup on Extreme Programming, Files. 2001.
- [60] C. Collins, R. Miller. *Adaptation: XP Style*. RoleModel Software. Presented at the XP2001 Conference, 2001.
- [61] C. Hendrickson. *When is it not XP?* Xprogramming, XP Magazine, May 12, 2000.
- [62] T. Dybå. *Improvisation in Small Software Organizations*. IEEE Software, September / October 2000.
- [63] J. Kerievsky. *Continuous Learning*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [64] Cunningham & Cunningham. Wiki, Extreme Programming Roadmap. Cunningham & Cunningham Inc, 2002.
- [65] A. Cockburn. *Agile Software Development*. Addison-Wesley, 2002.
- [66] Norwegian University of Science and Technology and SINTEF. *PROFIT Newsletter No. 6*, February 2002. PROFIT, Research Council of Norway project 137901/221.
- [67] M. Jørgensen, D. Sjøberg. *The Importance of NOT Learning from Experience*. EuroSPI'2000 -European Software Process Improvement, 7 - 9 Nov. 2000, Copenhagen, Denmark, pp. 2.2 - 2.8. November 2000.
- [68] M. Marchesi, W. Ambu. *XP and ISO 9000*. Tutorial at XP2002. Italy, May 2002.

- [69] K. Frühaof, P. Gassman. *Agile way to quality*. Workshop at the Quality Connection 2002. 7th European Conference on Software Quality. Helsinki, Finland. June, 2002.
- [70] G. Keefer. *Extreme Programming Considered Harmful for Reliable Software Development*. AVOCA, February 2002.
- [71] R. Jeffries. *How does the conventional QA function fit into XP?* Xprogramming.com, Q and A.
- [72] S. Smith, G. Meszaros. *Increasing the Effectiveness of Automated Testing*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [73] P. Schuh, S. Punke. *ObjectMother, Easing Test Object Creation in XP*. XPUniverse Conference 2001 / ThoughtWorks Library.
- [74] L. Crispin, T. House, C. Wade. *The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [75] M. Finsterwalder. *Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment*. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
- [76] L. Crispin. *Extreme Rules of the Road: How a tester can steer an Extreme Programming project toward success*. The Software Testing & Engineering Magazine, July/August 2001.

9 Appendix A: Experiences with XP in the industry

This appendix includes the experiences with the XP practices in the industry. Chapter 3 in this report cover issue. The experiences are summarised in section 3.1.2 in this report. This appendix is used as a reference for that summary. The projects where these experiences come from are described in section 3.1.1.

The experiences are order under the twelve practices defined in XP: The planning Game, Small releases, Metaphor, Simple design, Testing, Refactoring, Pair programming, Collective ownership, Continuous integration, 40-hours week, On-site customer, and Coding Standards.

The Planning Game

(1) The planning meeting should be kept as short as possible [35]. When the development team is big (50 in this example), a full team meeting during the iteration-planning meeting did not work. The customers and the developers should rather split in groups to discuss the current story cards and estimate them. This implies that the customer is looked at as a group of individuals, representing different business knowledge. Afterwards everyone regroup for presentation of estimations and findings and then developer signup. The regroups shouldn't last longer than 30-45 minutes.

(2) To make it easier to track and suggest more accurate estimations, the iteration to be planned should not be long [35]. Two weeks is experienced to be more suitable than one month.

(3) The way of defining stories and how the story card should be divided and estimated will not be the same throughout the project [34, c.2]. The procedures used to define and estimate stories should be rethought in each iteration. The sort of functional unit that was simple and easily estimated in earlier iterations can later become more complex and harder to complete within a single iteration. E.g. it may take longer time to finish a customer-usable feature as the complexity grows.

(4) The customer that takes decisions in the planning game may not be trained or not representative to the real customers [34, c.3]. The customer must be "coached" sufficiently to provide honest and substantial feedback from the very beginning of the development process. If an XP customer does not find something to complain or change after early deliveries in the development process, the customer is, more than likely not as engaged in the process as he or she should be. The customer must also represent the real users from the beginning. Otherwise the surprises after delivery will be expensive.

(5) In [43] they take the problem with untrained and unfitted customer further. It is suggested to split the customer role into a user and a client role when the project settings are complex. The user is the source of application knowledge including functional requirements, while the client defines the project goals and supplies the money for the project. Both parties should be included in the planning game.

(6) The in-house customer service and sales organisation acted as customer proxies [38]. The project was in a maintenance environment and these two departments forwarded and prioritised bugs to be fixed in the next iteration. They also ran test cases as acceptance tests for each customer story against each patch delivered to a customer. They experienced some interruptions and mid-cycle reprioritising, but generally things worked well.

(7) In a company delivering safety-critical software [40], they used use cases rather than user stories. The use cases were written based on the existing requirements and stored in a database. The use of a database was done to make sure the use cases didn't disappear. Otherwise the team practiced scope limiting, task breakdown, and task sign-up techniques.

(8) All parties that will be directly involved by the decisions taken in the planning meeting should be present [36]. If the project is cooperating with a separate Database Administrator (DBA) of a DBA team, they should be represented by at least one person. Then, bad database decisions that could result in bad design throughout the project can be avoided.

(9) In [41] they found that it was possible gather the requirements without the customer onsite. Instead a small requirements team was stationed at the customer site, gathering requirements and delivering the prioritised user stories as use cases to the developers. Estimates were developed during short group sessions. Changes and impacts of changes are communicated between the developers and the customer by the requirement team. The requirements team was always one week ahead of the developers in gathering requirements and changes to make this work efficiently.

Small releases

(1) The releases should be kept as small as possible. In [35] two week iterations worked fine. If larger pieces need several iterations to be completed, one could be a little flexible with the release cycle, but the progress must be reviewed after each iteration.

(2) The release cycle was moved from months down to two weeks with great success [38]. Since the project was mostly bug fixing and few changes to functionality, the releases were delivered after every iteration. Without automated regression testing and unit testing this could not have been possible.

(3) To make the required reviews as efficient as possible, the iterations lasted for one month each [40].

(4) In [41] three-week cycles provided the ideal balance between the administration costs of frequent software releases and development time.

Metaphor

(1) The customer already had a story, which they described the system by, so this was used as the system metaphor [38].

(2) Because of the already existing UML-diagrams of the high-level design, these were used instead of a metaphor [40].

(3) In large projects like [35] metaphors seems to be unrealistic. The system turns out to be too complex.

Simple design

(1) Simple designs have helped to release a working product to the customer regularly, but design refactoring is needed in parallel to use simple design [35]. To avoid incompatible solutions in larger teams, good communication is needed. Frequent and informal design

meetings, for instance lunch meetings, are recommended especially during stages of intense new functionality.

(2) Doing the simplest thing that could possibly work isn't always the best solution [34, c.6]. When you know you have to generalise an implementation in the end, you should do so immediately. Some look-ahead design must be performed. Using the factory pattern is suggested for many situations. This will also result in less refactoring.

(3) This experience was similar in [40]. Since the team knew that things would change later, they preferred to do a little design. They called in a Little Design Up Front (LDUF), not unlike the Quick Design Session described in [58, c.10]. This gave a broader vision of how things should be designed. However, caused by required reviews, the LDUF was later replaced by these design reviews after each iteration. Here the senior people could monitor the design decisions and make sure that the design was satisfying.

(4) This was not fully performed in the maintenance project [38]. The product was complex and it was necessary to continue to use high to midlevel design documents to quickly be able understand the system. However, the development tended to focus more on implementing and passing the acceptance and regression tests, rather than generating design documents.

Testing

(1) Test first was adopted from start, and is a naturally part of a maintenance project [38]. A test case was always developed before the work with a bug fix started. All test cases were automated and ran nightly.

(2) Both test-first design and functional tests were used [40]. CppUnit [55] was used for test-first design and a custom scripting language was used for functional tests. However, the team got behind making automated acceptance tests. They afterwards regret this tendency and suggest doing something about it next time.

(3) There is too much overhead in setting up unit test [4, c.9]. A test will often require an object being in a particular state. When the object is complex such set-up is being done often, this may be very extensive. Setting up general fixtures that created wanted objects solved this problem partly, but for complex objects it is suggested to use ObjectMother [73]. This is a pattern that looks like the Factory pattern, but is extended to create objects in different states.

(4) Unit tests are important and their coverage must be as complete as possible [35]. An automated test suite of functional test is recommended to keep the test coverage at an acceptable rate.

(5) It is a repeating problem that, even though the unit tests are passing, the system is still broken [34, c.7]. Automated functional test must be used as far as it's possible to solve this problem. This is especially important when changes of the system causes conflicts with earlier functionality and corresponding functional tests. If they aren't automated, they will probably be performed seldom, and when they are performed it may be too late.

(6) When the system is growing more and more complex, it will be difficult for everyone to keep the overview [34, c.4]. The responsible for a story card may not be sure that all functionality has been accounted for in the functional tests developed for that card. A graphical picture of the system must be made, posted visible for everyone, and constantly updated.

(7) In [41] the developers were responsible for testing their own code, while the complete functionality and responsibility of acceptable quality were given to a full-time Quality Engineer (QE). This relieved much of the administrative burden of the quality process from the rest of the team.

Refactoring

(1) Refactoring is the only way to be able to have simple designs [35]. Refactoring of design is just as important as refactoring of code. It will always be tempting not to refactor and to just patch a solution, but if it is patched too much, the team will be forced to make major refactorings later on.

(2) Large refactorings are not desirable [34, c.7]. Initially bad design can be costly to refactor at late stages of the project. It is important to do small refactorings often and not saving problem solutions till later.

(3) Since the refactoring was performed regularly, the design evolved smoothly [40]. The design reviews after each iteration also avoided huge refactorings in the end.

(4) Refactoring was used as the cornerstone in the efforts in improving code maintainability and stability [38]. It was important to make refactoring a part of an engineer's personal practices. Sometimes refactoring resulted in a wholesale reengineering, but for the most part it gave fewer problems in the releases, a significant decrease in code entropy, and reduced code complexity.

Pair programming

(1) This practice was difficult to introduce [38]. It was difficult for experienced engineers to change their programming habits, but since the team did see some positive sides of pair programming (PP), they will focus more on this the next time. To make this easier, they suggest using PP only a few hours a day as a passing stage to a fully implementation. When PP was used, they experienced more effectively work, higher morale, and the senior staff enjoyed the mentoring when pairing with the junior members of the team.

(2) In [40] it was important to concentrate the usage of PP only to the creation of production code. Tests, interfaces, and simulations were implemented by each person alone.

(3) Similar experiences are described in [35]. PP should be religiously followed when new functionality is added, and should be skipped when fixing bugs or doing repetitive tasks that have already been 'solved' by a pair of developers.

(4) When the solutions involve more than system changes, e.g. database changes, the pair should include someone that is familiar and responsible for the database [36]. The real gain from this is increased communication, melding of the teams into one to write one application, and avoidance of inefficient database design and decisions.

(5) The use of field experts is also highlighted in [37]. Here it is stated that PP is positive when experts in different fields shall solve a common problem and knowledge of both fields are needed. On the other hand PP is not necessarily the solution when the problem is to increase performance. Testing is more important to solve such tasks.

Collective ownership

- (1) In [38] they had people changing code everywhere. Some had a stronger knowledge of certain areas, but with fewer people maintaining the code, it was necessary to let people work on all parts of the code.
- (2) If one programmer owned a module, the development slowed down in that part of the system [40]. The experience was that collectively owned code was more efficient.
- (3) Collective ownership goes hand in hand with communication [35]. The team must figure out a way to communicate effectively. Informal discussions or regular stand-up meetings are good ways to disseminate information.

Continuous integration

- (1) Even though it took long time to build an automated testing system necessary for continuous integration, the resulting merge process was strong and followed quite well [38]. Mutex files, requiring check in and check out, was used as source control system, to ensure that no one else was integrating while the merge into the mainline was performed.
- (2) With only one pair in the beginning, there were obviously no problems [40]. However, as soon as more pairs were added to the project, the team had integration problems. The avoidance of collisions and how to perform successfully merges was, however, quickly learned.
- (3) When estimating a story, the time to integrate it with the rest of the system must be considered [34, c.5]. This may be a problem when the system is complex and everything can't be integrated until a release. Even though all the stories are considered finished, there may still be many problems that are discovered when subsystems are integrated. This must be taken into consideration when the estimation is being done, especially when continuous integration isn't possible.

40-hour week

- (1) This was not adopted in [38]. To quote them directly; "We haven't felt courageous enough to tackle this".
- (2) With only few exceptions before some iteration finishes, the team worked at a suitable pace [40].
- (3) A 40-hour week has never been an issue in [35]. 40 hours is the minimum and they have not been adversely affected by working more than 40 hours. The reason may be that they're not pair programming 100% of the time.

On-site customer

- (1) In a project working in a maintenance environment [38], the internal customer service team and sales force was used to act the customer role. They sat priorities and generated acceptance tests. The reason for this solution was that the users who continuously are reporting bugs and wanted changes were widely spread and it was impossible to make them

speak with 'one voice'. To prevent all the interruptions from these users, the sales and service team was used as a 'proxy' between the users and the developers.

(2) Since the project was a pilot project, a systems engineer acted as the on-site customer [40], and seems to be a suitable solution when the goal is to learn more about XP.

(3) The on-site customer was a team of several business analysts [35]. This was a large project, and the actual client was offsite and the analysts communicated with them.

(4) The client may be unwilling to spare key individuals to be an on-site customer [41]. To keep up good communication in such a situation, an efficient solution was to station a requirements team at the customer site.

(5) In small projects it is often not feasible to have an on-site customer [44]. In its place, a high-available customer was required. Answers to a question should be returned within a couple of hours, without the customer necessarily being physically present.

Coding standards

(1) This was already partly used before XP was introduced [38].

(2) The main coding standard was "make the code look like the code that is already there" [40]. The coding standard was already in place, and was followed up with the pair programming to make sure that it was followed.

(3) Coding standards have been informal in [35]. The importance of a strict standard is tuned down. Furthermore, the code is not enough to get the see big picture of large systems. Ongoing presentations are more useful to communicate the ongoing work.

10 Appendix B: XP resources

Here we have gathered various resources that are related to XP. The resources are listed in alphabetical order. All links were accessible June 8, 2002.

Resource	Link
Agile Alliance	http://www.agilealliance.com
AVOCA	http://www.avoca-vsm.com
Extreme Programming	http://www.extremeprogramming.org
Fowler, Martin	http://martinfowler.com
IEEE Explore	http://ieeexplore.ieee.org
ISO Online	http://www.iso.ch/
Jera XP FAQ	http://www.jera.com/techinfo/xpfaq.html
Mercury Interactive	http://www.mercuryinteractive.com
Object Mentor	http://www.objectmentor.com/xp/xp.html
Proxycom AS	http://www.proxycom.no
Ratio Group	http://www.ratio.co.uk
Riehle, Dirk	http://www.riehle.org/
RoleModel Software	http://www.rolemodelsoft.com/xp
SPIQ Norway	http://www.geomatikk.no/spiq/
Testing Foundations	http://testing.com/agile/
ThoughtWorks	http://www.thoughtworks.com/index.html
Together Community	http://www.togethercommunity.com
Wiki, XP Roadmap	http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
Williams, Laurie	http://collaboration.csc.ncsu.edu/laurie
XP2001 Papers	http://www.xp2001.org/xp2001/conference/papers
XP2002	http://www.xp2002.org
XProgramming.com	http://www.xprogramming.com
XP Universe	http://www.xpuniverse.com
XP Newsgroup	news:comp.software.extreme-programming
Yahoo XP Newsgroup	http://groups.yahoo.com/group/extremeprogramming