Quantitative Approach for Lightweight Agile Process Assessment

Master's Thesis
UNIVERSITY OF TURKU
Department of Information Technology

24.07.2008
Johan Paul

Tarkastajat:
Timo Knuutila
Antero Järvi

TURUN YLIOPISTO
Informaatioteknologian laitos

JOHAN PAUL: Quantitative Approach for Lightweight Agile Process Assessment

Diplomityö, 81 sivua, 0 liitesivua.

24.07.2008

Tämän hetken haasteita ohjelmistokehityksessä ovat korkean laadun saavuttaminen tuotantoprosessin ketteryydestä tinkimättä. Ketteryydellä tarkoitataan kykyä reagoida muuttuviin asiakasvaatimuksiin ja toteuttaa ohjelmistokokonaisuus pienissä laadukkaissa inkrementeissä. Haasteeseen vastaaminen vaatii optimoituja ja jatkuvasti kehittyviä ohjelmistotuotantoprosesseja. Prosessin toimivuus voidaan todeta ulkoisella arvioinnilla, mutta yleensä arviointi edellyttää raskasta ja kallista auditointiprosessia. Sisäisen kehitysprosessin toimivuuden toteamiseen riittää kevyempi arviointiprosessi.

Tässä diplomityössäni tarkastelen aluksi Software Process Improvement (SPI) paradigmaa referenssiprosessimallien ja Experience Factory (EF) näkökulmasta. SPI:n tavoitteena on parantaa ohjelmistotuotantoprosessin laatua ja arviointimenetelmät ovat tärkeä osa SPI:tä, joilla mitataan prosessin laadun kehitystä. Tämän jälkeen esitän kvantitatiivisen ja kevyen arviointimenetelmän ketterille prosesseille. Kokemuksen mukaan tällä menetelmällä voidaan prosessi arvioida kahdessa tunnissa. Menetelmä antaa numeerisen tuloksen prosessin ketteryydestä. Lopuksi totean, että tässä diplomityössä tehty työ on ehdotus ja ensiaskel tämän kaltaiseen ketterien menetelmien kevyeen arviointiin, joka kuitenkin vaatii lisätutkimusta. Tässä diplomityössä esitetyt tulokset liittyy Plenware Oy:n ajankohtaisiin kehityshankkeisiin.

Asiasanat: quantitative, process assessment, agile, GQM, QIP, SPI

UNIVERSITY OF TURKU
Department of Information Technology

JOHAN PAUL: Quantitative Approach for Lightweight Agile Process Assessment

Master's Thesis, 81 p., 0 app. p.
24.07.2008

One focus in software development is to achieve high quality without loss of agility in the development process. In software engineering an agile development process is able to react to frequently changing customer requirements and also able to develop the software in small well tested increments. This is not an easy thing to do and will not emerge on its own, but requires well optimized and continuously improved software development processes. The quality of a process can be assessed by some external audits but they usually are heavyweight and costly processes. To measure the quality and follow the improvements in a process a lightweight assessment method is desired.

In this thesis I will look at Software Process Improvement (SPI) paradigm from the perspective of both Reference Process Models and the Experience Factory (EF) infrastructure. Then, I will present a quantitative lightweight process assessment method for agile projects. This method allows the assessment of a process in less than two hours. The method yields numeric results of the agility of a process in a lightweight fashion. I will conclude by stating that the work done in this thesis is only indicative of this kind of assessment for agile processes there is a need for further research. The results presented here result from the current need in Plenware Oy related to SPI evolvement.

Keywords: quantitative, process assessment, agile, GQM, QIP, SPI

Contents

List of Figures

List of Tables

Chapter 1

Acknowledgements

All the research work presented in this Master's thesis has been conducted at the Turku office of Plenware Oy.

First of all, I express my deepest sense of gratitude to my supervisor PhD Luka Milovanov at Plenware Oy for his patient guidance, encouragement and excellent advice throughout this study. Without his inspiration this Master's thesis would not have been possible.

I also express my gratitude to Mr. Lassi Korjonen and Mr. Pertti Hannelin at Plenware Oy for sponsoring the writing of this thesis.

My deepest gratitude is also due to the my supervisory Prof. PhD Timo Knuutila and Ph.D. student Antero Järvi at the University of Turku for assisting in writing this thesis.

I direct my very special thank to PhD Minna Pikkarainen and Prof. PhD Pekka Abrahamsson, for helping me to get valuable references in lightweight process asssessment.

I also thank my parents, Riitta and Robert, for all the support they have given me, not only during my studies, but throughout my life. A special thank you goes to my father for being an inspiration for me in science and letting me use his Mikro Mikko 1 CP/M computer and also for buying me my first computer, a Commodore 64, on my 7th birthday and thus sparking my interest in computers.

Finally, I express my deepest love and gratitude to my wife to be, Oksana, who belived

in me and stood by me when preparation for this thesis faced its challenges.

Turku, 24th of July 2008

Johan Paul

Chapter 2

Abbreviations and definitions

CMMI	Capability Maturity Model Integration is a process improvement approach that provides organizations with the essential elements of effective processes.
GQM	Goal/Question/Metric software metric paradigm developed by Victor Basili.
GIP	Quality Improvement Paradigm. A model that focuses on continuous process improvement and engineering of the development processes developed by Victor Basili.
ITIL	The Information Technology Infrastructure Library (ITIL) is a set of concepts and techniques for managing information technology infrastructure, development, and operations.
ISO	The International Organization for Standardization is an international-standard-setting body composed of representatives from various national standards organizations.
RUP	The Rational Unified Process is an iterative software development process framework created by the Rational Software Corporation.

Scrum	Scrum is an iterative incremental process of software development commonly used with agile software development.
SEI	The Carnegie Mellon Software Engineering Institute (SEI) is a federally funded research and development center headquartered on the campus of Carnegie Mellon University in Pittsburgh, Pennsylvania, United States.
SPI	Software Process Improvement initiatives improves the methods by which software is developed in an organization.
SPICE	SPICE, Software Process Improvement and Capability dEtermination, also known as ISO/IEC 15504 is a "framework for the assessment of processes" developed by the Joint Technical Subcommittee between ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission).
XP	Extreme Programming is an agile software engineering methodology prescribing a set of daily stakeholder practices that embody and encourage particular XP values.

Chapter 3

Introduction

Software quality is difficult to define exactly. Everyone feels they understand what software quality is and know what is meant by it. A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but is nevertheless desirable and enhances the quality of the software program. Some obvious software quality factors are understandability, completeness, reliability and usability. Software quality is something that tends to be a given when a new software project is started. While everyone is excited when a new project is started software quality tends to become forgotten and the question of quality emerges only later, even at the end of the development process.

Software quality needs to be built into the development process and cannot be examined as an external or separate entity. From the point of view of the organization, this requires well defined development processes that, almost automatically, yield high quality in the end. But this does not come for free. A well working and optimized development process requires understanding of the domain in which it is working, a well trained staff and suitable tools. This is an ongoing improvement process which has to be built into the development process naturally. There is a term for this in software engineering; Software Process Improvement (SPI).

Being a midsize software development company with about 500 employees, Plenware

Oy is continuously seeking ways to improve the quality of its services through well defined and optimized software development processes. As an employee of Plenware Oy, I was presented with the opportunity to work on one such quality improvement effort. The task was to assess software development processes for their agility. At the outset, the need at that time was not to assess the process for external auditing or certification, but to apply metrics and to continuously improve and understand the process for ultimate packaging of experience to be used in later projects. A further requirement was to keep this assessment lightweight and easy to deploy with the intent to minimize costs, hence making it more attractive for the project to self-monitor the development process.

In this thesis I will first present a set of widely used software development process models focusing more on them through the Software Process Improvement (SPI) perspective. I will describe two approaches to the SPI paradigm; the approches are termed Process Reference Models (chapter 6.1, p. 37) and Experience Factory (chapter 6.2.1, p. 48). This will be the background for a lightweight quantitative process assessment method for agile processes that will combine the reference model approach and software metrics. The work done in this thesis is an introduction to this kind of assessment of agile processes. Further research is needed to refine and develop the model and the assessment. The results reflect the current needs of Plenware Oy for easy and fast assessment of the agility of the process. I will conclude with a discussion on the limitations of the method presented in this thesis.

Chapter 4

Background

The term software engineering first appeared in the literature in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering and debated what engineering might mean for software development. Software engineering was spurred by the so-called software crisis [4] of the 1960s, 1970s, and 1980s, which identified many of the problems of software development. The crisis was due to several circumstances. Many software projects ran over budget and schedule. Some projects caused property damage. A few projects even caused loss of life. The software crisis was originally defined in terms of productivity, but evolved to emphasize quality.

In 1969 a NATO sponsored conference was held to address the problems of software engineering [5]. Current problems were addressed and some improvement methods proposed. Already then the problems of estimating large software projects and software portability issues were acknowledged. Solutions that today sound so familiar were proposed, e.g. computer aided system design and structured programming languages. Some of the problems have been solved. We have good compilers, programming APIs provided by the operating system that make portability easier, integrated development environments with syntax checking and etc. Still the management of large software projects is a challenge. We are going towards standard development processes and verification methods, but there is still a lot of ground to cover.

4.1 Challenges in software engineering

Why is software engineering so difficult? Not much has changed in 20 years since Brooks [1] wrote about the problems in software engineering. According to the International Software Benchmarking Standards Group (ISBSG), 57% of the projects they studied underestimated the effort needed and, according to the same database, 44% projects delivered late [2]. Since the data collected by the ISBSG probably represents the projects that were more successful, these figures may perhaps still be too positive.

The crucial part in developing a software is that it does what the user or customer wants. This is also the most challenging part. Before any code can be written, a specification needs to be documented to lay out the tasks at hand. It is equally important to verify that software (or a piece of it) does what was specified. In his paper, Brooks[1] highlights four important properties in software projects that make them difficult to manage; complexity, conformity, changeability, and invisibility. These unwanted properties are still today the core issues that make software projects fail.

Today's software systems are enormously complex. Nobody can fully understand every detail of a modern software system. Because we have to trust closed source modules and interfaces written by other members of the team, the importance of correct specifications is even more emphasized. All this adds up to a very large number of logical states which make unwanted behavior of the software likely.

In many cases software is the last piece putting components together to create a working unit. Or there are many different people working on a software project and all the distinct pieces must conform with each other. Any embedded device, mobile phone or PDA is a good example of hardware and software working together to create one well working unit with software conforming to many different interfaces. Also, a piece of software must maybe conform to some existing system because it is seen as easy for the software component to conform to existing hardware. The complexity is in the software itself and cannot be simplified out by redesign.

Software changes all the time. It is more common to integrate reused code into new use domains than to rewrite a system from scratch. The system requirements may also change over time: support for new operating systems is needed, the system needs to be scaled up to handle more users, the system needs work with new peripherals etc.

The most notable difficulty of software development is probably the fact that software is invisible. It is much easier for humans to understand the structure of a building or even of an electric circuit. One can try to visualize the structure or the work-flow of the software, but one can only get a two-dimensional representation of a restricted part of the complete system. On one hand we cannot visualize the system in enough detail and on the other hand we miss the complete picture. This makes system design, communication between people and specification validation, despite its importance, very difficult to manage.

It is fascinating to notice that back in 1987, when agile development was not a buzzword among marketing people, Brooks proposed[1] a similar approach to deal with problems in software development. He called it prototype software system. Brooks acknowledged the importance of prototyping, the iterative refinement of requirements and the need to be able to test early and to communicate with the customer to assess if the result is what was planned.

Software needs to be grown - not built. SPI helps us to understand how this happens.

Chapter 5

Software Process Models

For software development, a statement is needed that defines what is to be accomplished, and why the software development process was started in the first place. Also, a package of methods and tools and a plan are needed. The plan is known as a process model. A process model has two distinct uses:

- to predict what will be done and it what order.
- to analyze what is happening during the development. The aim is not only to follow up schedules and budgets but also to measure and collect data to improve the process for future projects.

Developing software is always a challenging task, let alone estimating the cost and the schedule for a software project. The competition between companies offering software engineering is getting tougher all the time. Subcontractors have to cut costs to win contracts, but on the same time they must maintain high quality, reduce risks and improve delivery times. This is not an easy equation to solve. Effective, stable and mature process models are the keys to any successful software project.

A software development process is a set of tasks or activities imposed in a given order on the development of the software product. There are several models that describe such software development processes. The process models help stakeholders to understand the

current state of the project, to speak a common language and help ensure stable, capable, and mature processes.

I will introduce four common software development models. I start with what is called the traditional development models: the Waterfall Model and the Spiral Model. Then I will introduce the iterative development methods: the Rational Unified Process and agile process models (eXtreme Programming and Scrum). These more recent models attempt to solve some of the problems of the traditional models but are no "silver bullets" on their own.

Reference process models (also called process methodologies) are sets of best practices for development process models. CMMI and SPICE are reference process models I will cover. When a process model is combined with a reference process model, it gives tools to measure and compare the maturity of development processes between companies or projects. A CMMI or a SPICE model is not a process by itself, but allows the project to implement a model that best suits the needs of the project or company standards.

5.1 Traditional development

Software needs to be built in an organized fashion. The traditional development model is to build software like any other engineering effort: first we define what we want to build, then we build it, test it and ship it. But because constructing software is far from constructing other engineering accomplishments, say buildings or vehicles, these methods impose difficulties into the process. Nevertheless, traditional development models are still widely used, especially in smaller software development companies.

5.1.1 Waterfall model

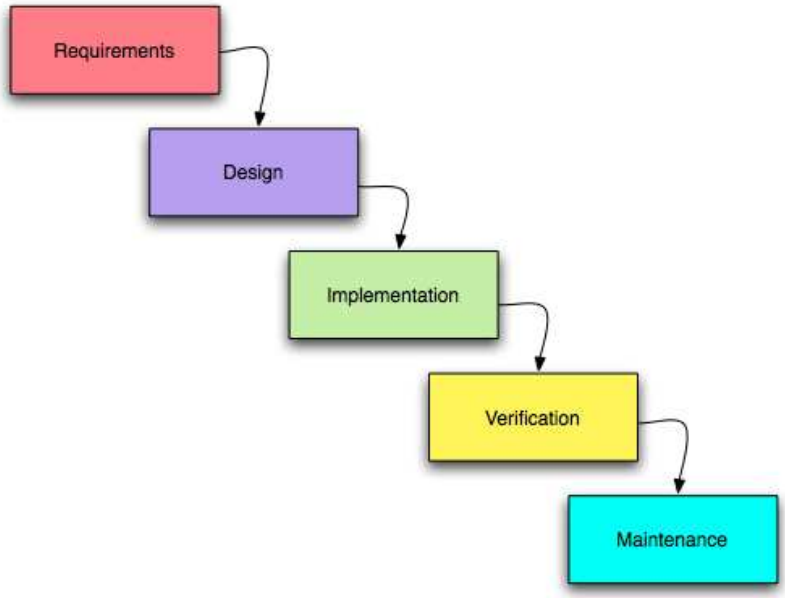
The waterfall model is the most straightforward approach to tackle *ad hoc* development. The waterfall model was first presented by Winston W. Royce [3] although he never used

the term "waterfall model". Later Royce proposed a final version of a software development model, the spiral model.

Waterfall model - phases

In the waterfall model the development process is strictly sequential. The development is split up into a number of independent steps where each step produces a product that is the input for the next step, as shown in figure 5.1.1.

Figure 5.1: Waterfall development model



- *Requirements* - During the first step of the process, all the requirements are collected from the customer and analyzed for complete understanding of the project and its subtasks. The format of how the requirements are described is open.

- *Design* - Once the requirements have been analyzed and accepted by the customer one or more design specifications are written. These documents form the basis for implementation of the system: The customer can understand the implementation and is also the reference for testing.

- *Implementation* - In this phase the project takes shape. All the code is written and tested by the developers. After this phase the project should be fully functional and an implementation document is written describing the interaction between components.

- *Verification* - The testers prepare their test cases, run them against the implemented system and verify the functionality specified in the Requirements phase. Also the customer should analyze the end product and verify that it meets the requirements laid down in the first phase. A test document is written describing the test cases and results. After this step, the system is fully functional and running in the intended environment.

- *Maintenance* - If any faults are detected during the usage of the system, they are corrected.

At first glance, this model is attractive. It is definitely an improvement over ad hoc development. The waterfall model splits complex tasks into smaller, easily manageable sub-projects that deliver an outcome that can be inspected. The product of the previous step needs to be inspected and verified and each step must be flawless. Unfortunately, this model is too naive. In practice steps overlap each other: during the design phase, problems of specification are identified, during the implementation phase problems of design are identified and thus the waterfall model is not as streamlined as one would wish. Usually this model works acceptable well in small and short projects, although even large projects, employed by for example US Department of Defense and NASA, use the waterfall model or some variation of it [6].

The waterfall model is still widely used today, but mostly because it is easy to understand and has become a legacy model for software development within many companies. Also for very short projects where the requirements are well understood and not likely to change, the waterfall model is the best approach for developing software.

Waterfall model - operational roles

One of the problems of the waterfall model can be identified through the operational roles that inherit to the model. Each phase requires a different role and because the model is sequential, only one set of roles (and people) are active at a time. This is, especially for larger organizations, a huge waste of resources.

The roles withing the waterfall model are the "traditional" software development roles; the *Requirements* phase is owned by Business Process Analysts and by Systems Analysts who write a requirement specification based on the business goals of the system or project.

Systems Architects and Designers create a design documentation in the *Design* phase. This documentation is based on the requirements gathered earlier.

Software Engineers write the code and an implementation document in the *Implementation* phase. The implementation and the documentation is based on the design document from the previous phase.

In the *Verification* phase the code written in the Implementation phase is tested by Test Engineers. They also create the test cases which they document together with their results.

Technical Writers, Deployment Managers and, possibly, the Project Manager are involved in the *Maintenance* phase. They deliver additional project material and maintain the functional system. The Project Manager is involved in this phase, because he is responsible for correcting and managing any faults in the system. This function entails, in fact, an iteration of the whole waterfall model.

Waterfall model - pros

The waterfall model is easy to adopt by an organization without a history of software development. The process is also easy to understand and visualize by less computer savvy people and by people who are in managing positions but lack the knowledge of software engineering. Thus, the waterfall model is tantalizing because it gives a false impression of

predictability. It is easy for project managers to delegate non-overlapping responsibilities to people.

The phases in the waterfall model fit also well with other, non-software, processes. For this reason the waterfall model is easilly seen as a logical model if the project developes, in co-operation with the software, non-software componets such as circuit boards or other hardware or machinery.

The Waterfall model enforces system documentation. Documentation that describes the design, implementation and testing of the new system and other source code comments are important artifacts that developers might otherwise neglect unless they use the waterfall model.

Waterfall model - cons

Unfortunately, the waterfall model does not work [7] in practice. Even Royce[3] identifies the problems of the waterfall model and claims that the waterfall model should not be implemented as such. This is why it is ironic that the waterfall model is so widely used today. The fundamental problem of the model is that before each next step can be taken the previous step needs to be completed. If this is to succeed, perfect knowledge of the client's needs is required for requirement analysis and designers have to design a flawless system from the very beginning to the very end which is, of course, utterly seldom possible, because upcoming problems of implementation cannot be predicted. Because the client does not have any opportunity to test any of the software before implementation, it is only as the model is implemented that one may judge if the first step, requirement analysis, was successful or not. If changes are to made at this stage, the whole process must start all over again. Consequently the waterfall model can only be used when the requirements are fully understood and unlikely to change during the development process.

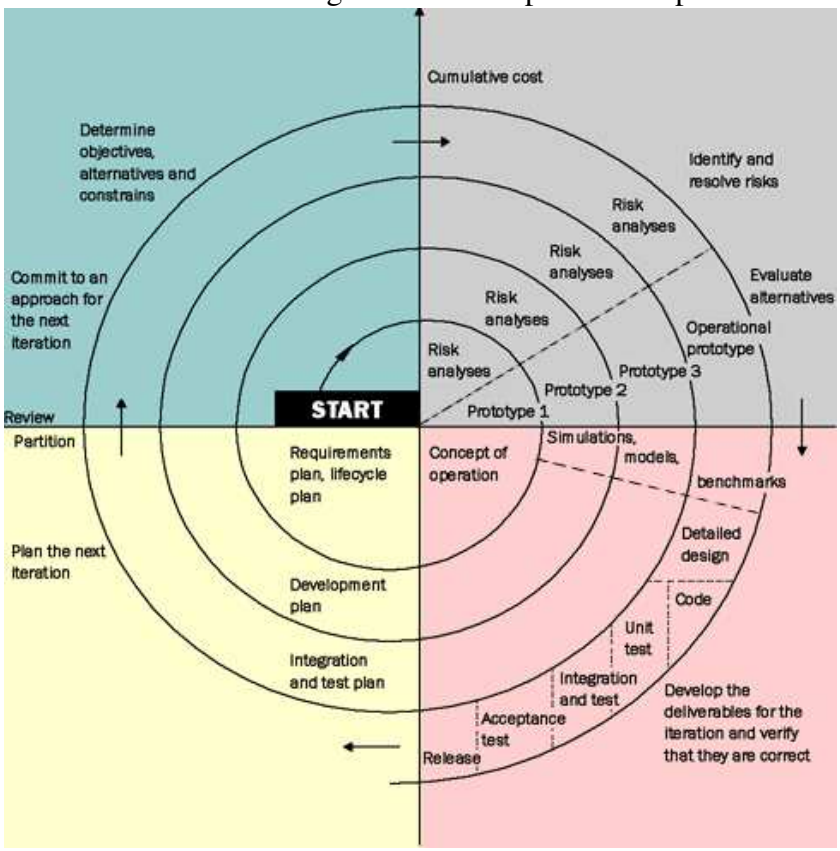
As Royce mentions in his paper[3], the waterfall model is risky and invites failures. Also, a reliable cost estimate is very difficult to present prior to project initiation, since it

is unclear how long each step will take and if changes are needed after one or several steps. Very specific skill sets are required for each phase, and thus multiple projects should run in sequence if resource use is to be optimized, and if all project members collaborated throughout the entire course of a given project.

5.1.2 Spiral model

The problems of the waterfall model have been known for a long time, especially in long, expensive and high-risk projects. As late as in 1988 Barry Boehm explained why the so called spiral development model would be a superior to the Waterfall model [8]. In his paper he addresses the risks involved in the development process and the changes intrinsic to software development. These concerns are not very well covered by the waterfall process model. Figure 5.2 is a graphical representation of the spiral model.

Figure 5.2: The Spiral development model



Development is an iterative process, where each phase has a duration of 6 month to 2 years. Each phase starts with requirement definition and ends with the client reviewing the progress. An important factor is that the progress to be reviewed may, in essence, be an early prototype of the system. The client has a much easier job to review the progress in terms of strengths, weaknesses, and risks from scaled down prototype very early in the development. The process continues by defining new and refining excisting requirements, planning a second prototype and finally by constructing and testing the second prototype - all from the input based on the first version of the prototype. The client can abort the project whenever the client considers that it carries an unacceptable risk. This cycle continues until the client is satisfied with the ultimate version of the prototype which can now be considered the final version.

Roles

Because the development model follows somewhat the same paths as the waterfall model, the roles in the process are same the same as in the waterfall model.

Spiral model - pros

The major breakthrough of the spiral model was the introduction of prototyping. This is an important factor also in current agile software development processes. Early releases and working prototypes allow a simpler review and analysis of the risks involved in the software development compared to the waterfall model. In the waterfall model the first working version of the software is by implication the final version. When the spiral model is applied, time and budget estimates become more realistic as the process evolves and adjustments to the development may be done before it is too late. There are also more or less inevitable changes to the requirements, and as they become known they can be taken better into account.

An iteration cycle of 6 months to 2 years may be too long for many projects. One

might argue, however, that very large projects for large customers could benefit from such iteration cycles. For example, the US Defence Force might benefit from long cycles that require complicated tasks, not only by software engineers but by fields of engineering, as well.

Spiral model - cons

Today it is quite obvious that software has to be constructed in an evolutionary fashion. But with the spiral model, the iterations are far too long. During the period between the first prototype or the first milestone, huge amounts of resources have been spent and it might be too late to discontinue the project. In the worst case, the model evolves into the waterfall model. Nor does the spiral model specify any clear artifacts or define roles for project members.

5.2 Iterative development

Iterative development methods are developed in response to the weaknesses of the classic waterfall model. The spiral model is an early example of an iterative development model. Today the spiral model model has been refined further and its basic principles are currently essential parts of the Rational Unified Process (RUP)[9], Extreme Programming[15] and generally the agile software development frameworks.

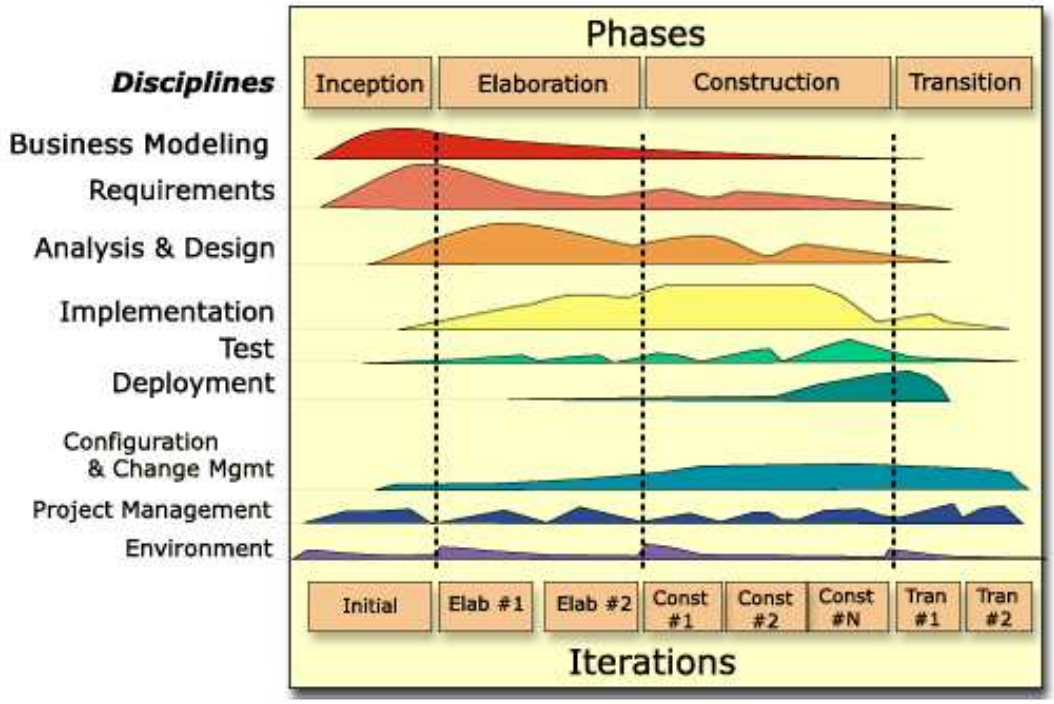
Historically iterative development may also mean incremental development. But to avoid confusion these two terms were merged into practical use in the mid-1990s. The authors of the Unified Process (UP)[10] and the RUP[9] selected the term "iterative development" to generally mean any combination of incremental and iterative development.

5.2.1 Rational Unified Process

The Rational Unified Process (RUP)[9] is an iterative software development process framework and, at the same time a software process product developed by Rational Software, a division of IBM since 2003. RUP is based on the spiral model by Barry Boehm[8] but is highly modified from the original model. RUP further evolves and defines the principles for iterative development and use of prototypes. It embeds object-oriented techniques and uses the UML as the principal notation for the several models that are built during the development.

The software lifecycle is broken down into subcycles, each subcycle working on a new generation of the product. RUP divides each development cycle into four consecutive phases: inception phase, elaboration phase, construction phase and transition phase. The process can be described in two dimensions as shown in figure 5.3.

Figure 5.3: RUP (Rational Unified Process) development model



The horizontal axis represents time and shows the dynamic aspect of the process as it is enacted, and it is expressed in terms of cycles, phases, iterations and milestones. The

vertical axis represents the static aspect of the process: how it is described in terms of activities, artifacts, workers and workflows.

RUP phases

Each phase has a specific purpose and is concluded with a well-defined milestone at which a critical decision must be made: whether to continue with the project or not. Before this decision can be made, the key goals must have been achieved. Each phase may contain several internal development cycles.

During the *inception phase* the business goals and success factors of the system are established. This entails in software engineering terms, to find all pertinent actors and to define the use cases for the system based on the requirements. The outcome of this phase should be a clear high-level vision of the system and the requirements, risks, resources and budget should be understood and estimated. One or several prototypes may also be constructed to improve the understanding of the nature of the system.

In the *elaboration phase*, an executable architecture prototype is built in one or more iterations depending on the scope, size, risk and novelty of the project. The elaboration phase is where the project starts to take shape. The problem domain analysis is made and the architecture of the project is conceived. Clearly, the elaboration phase is the most critical of the four phases, because it is often after this phase that the decision is made whether to go forward with the project or not.

During the *construction phase*, all remaining components and application features are developed and integrated into the product, and all features are thoroughly tested. Here, most of the coding takes place.

The purpose of the *transition phase* is to carry the software system to the end user. All all use cases are now functional and internal integration tests are successfully passed. All the documentation stands completed. Typically, this phase includes several iterations, including beta releases, general availability releases and bug-fix and enhancement releases.

The quality level set in the inception phase is tested and documented.

It must be noted that RUP is not a process model but a process framework. RUP is not to be used "out of the box" and unless this distinction is very clear, RUP could be very costly and weighty to adopt. RUP needs to be modified to fit a project or at least be tailored to the company development process. For example, projects do not have to produce all artifacts that are specified by RUP: selecting and tailoring artifacts is a necessary part of the process. Producing all of the some 100 artifacts RUP describes would be very cumbersome. Also, the document templates described in RUP may be modified, unless all of the fields are relevant to the project. RUP provides guidelines on what can be omitted and what can be tailored.

RUP disciplines

The disciplines that RUP defines describe the workflows that take place during the iterative development process and they consist of the following building blocks: roles, work products (or artifact) and tasks. There are six core engineering disciplines (business modeling, requirements, analysis and design, implementation, test and deployment) and three supporting disciplines (configuration and change management, project management and environment).

RUP has been designed together with UML and thus the disciplines of RUP are oriented around associated UML models.

Business modeling aims at modeling the business process using business use cases. Understanding the business means that software engineers must understand the structure and the dynamics of the target organization (the client), the current problems and improvement possibilities in the organization.

The *requirements* discipline aims at describing the use cases; actors who interact with the system are identified and use cases help the customer and developers to identify the business problem which the system or project should solve.

The *analysis and design* discipline will create a design model, or blueprint, of the system with UML models, e.g. architectural models, component models, object models and sequence models. This discipline also contains descriptions of how objects of these design classes interact to perform use cases.

The components are implemented in the *implementation* discipline based on the design model created in the analysis and design discipline.

The *testing* discipline works in close interaction with the implementation discipline and verifies the interaction between objects, verifies the proper integration of all components of the software and verifies that all requirements have been correctly implemented. The RUP proposes an iterative approach, which means that testing is performed throughout the project.

The goal of the *deployment* discipline is to create a plan for the delivery of the system, execute the plan and to make the system available to end users. This means to create a product release, to package it and create installation packages, to distribute it to the end users, to train end users and, if necessary, to deploy the system in the production environment for the customer.

The *configuration and change management* is a supporting discipline that manages changes to the system. This entails for example maintainance the version control system and different build settings for necessary configurations.

Project management is a supporting discipline, which manages the overall system development, e.g. risk management. It also monitors progress of an iterative project and manages people, budgets and contracts.

Environment is a supporting discipline that provides the software development organization with the software development environment (both tools and processes) that will support the development team.

RUP roles

The roles in RUP are defined from the RUP disciplines. Each discipline requires a certain skill set by which a role can be named for the person carrying out the activity in the discipline. One person can, of course, perform several roles in different disciplines and all roles mentioned below are seldom required in all RUP projects.

- *Business Modeling* - Business process analyst and business designer. They should discover the business use cases and specify a detailed set of use cases for each iteration.
- *Requirements* - The systems analyst is responsible of identifying all the requirement use cases.
- *Analysis and Design* - Software architect and designer. They should decide on the technologies for the whole solution and specify the detailed design model.
- *Implementation* - An integrator is needed to manage the long run integration of new features into the system, and implementers write code for the set of classes required for the current use cases.
- *Testing* - A test manager and test analyst select, specify and ensure that the correct tests are run. The test designer selects which tests should be automated and which manual, and writes automated test cases as needed. A tester conducts specific tests.
- *Deployment* - The deployment manager is the manager who oversees the deployment. The tech writer, course developer and graphic artist create the material needed for a successful launch.
- *Project Management* - The project manager has the responsibility to plan, track and manage risk for a single iteration. Also, the project manager is the one that makes the business decisions prior to iterations.

- *Configuration and change management* - The configuration manager and the change control manager. The configuration manager is responsible for maintaining and writing scripts needed to build the system for different configurations, while the change control manager is responsible for managing version control and bug tracking systems.

- *Environment* - The process engineer owns the process for the project and ensures that the contructed RUP process is suitable for the project and makes changes to it as needed.

RUP - pros

The Rational Unified Process is developed by Rational Software, currently a part of IBM. The fact that a large actor, like IBM, supports RUP is an obvious strength of RUP. When an organization is to adopt RUP, IBM can provide support and an efficient tool to support their process model, the IBM Rational Method Composer (RMC)[11].

RUP has detailed instructions on what to do, what the process phases are, what their inputs and outputs are and how to delegate tasks to people. This makes following the progress of the process transparent for the management.

RUP - cons

RUP is a heavy process model and poorly suited for lightweight prototyping or small organizations. This is true, since RUP is based on use cases and requires keeping track of the project with artifacts. Althought RUP-based processes can be customized and made more lightweight, they still require a fair amount of documentation.

5.3 Agile methods

Agile development is not a formally specified software development process model as the ones described previously. There is no one correct way to do agile development, but rather it is a set of development practices and a different way of organizing the development team that have naturally emerged as a solution to the difficulties in software development. There are several development models implementing agile practises.

Agile methods focus on developing working software delivered in short increments. It welcomes and acknowledges the fact that software requirements will change from the initial requirement analysis once the functionality becomes clearer for the customer following early prototyping. This risk of changes is managed through shorter iterations, which reduces the risk of large software integrations. Short iterations also help to keep quality under control by driving to a releasable state frequently, which prevents a project from collecting a large backlog of defect correction work. On the management side, the frequent iterations provide frequent evidence of progress, which tends to lead to good status visibility, good customer relations and good team morale. Agile development emphasizes every team members' role in the development process. Previously, for example, testing and integrations were separate development phases after the implementation, but with agile development these two phases have become important parts of the complete implementation and provide valuable feedback into iteration cycles.

Agile development evolved as a reaction to the heavyweight, non-flexible development methods characterized by poor track records. Software development needed to be about building working software with a minimum amount of micromanagement but still retaining the visibility of the project status. As I previously mentioned, the waterfall model suffers from, among other things, lacking visibility of the process status. As a matter of fact, agile development strives to do things in a different way compared to the waterfall model as much as possible. Agile development is also a lightweight alternative to RUP, but it incorporates the importance of short iterations.

A famous, almost cliché, manifesto summarizes the core values of agile software development:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

According to a survey [14], agile development has increased the success rate of some 1700 companies significantly. The survey reported accelerated time to market, increased productivity (10% or higher improvements reported by 83% of respondents), reduced software defects (25% or higher improvements reported by 54% of respondents) and, more modestly, reduced cost (25% or higher improvements reported by 28% of respondents). Other important improvements were recorded, as well, e.g. improved team moral and enhanced ability to manage changing priorities.

Clearly agile software development leads us to the right track of building software in a more efficient and customer oriented way. Nevertheless, the development model needs to be adopted correctly and needs often be tailored to the needs of the organization. And finally, it is not the silver bullet that makes software projects work "by itself", but it will increase the probability to carry out a software development project to a succesful end.

Again, agile development is only a conceptual framework of software engineering. Extreme Programming, Scrum, Dynamic Systems Development Method (DSDM), Adaptive Software Development, Crystal, Feature Driven Development and Pragmatic Programming are some of the process models implementing agile development principles. In this thesis I will cover the two most frequently used models - eXtreme Programming (XP) and Scrum.

5.3.1 Extreme Programming

eXtreme Programming (XP) is probably the best known agile software development methodology. It was introduced in 1999 by Kent Beck [15] as an answer to problems faced by the long development cycles in traditional development. XP emphasizes the facts that systems have vague user requirements, and acknowledges that rapid changes are inevitable. These starting points matched well with the current time of software development where short development cycles, introduction of new technology and emphasized focus on speed-to-market were considered competitive business factors as a result of the rise of the Internet and the "dot-com boom".

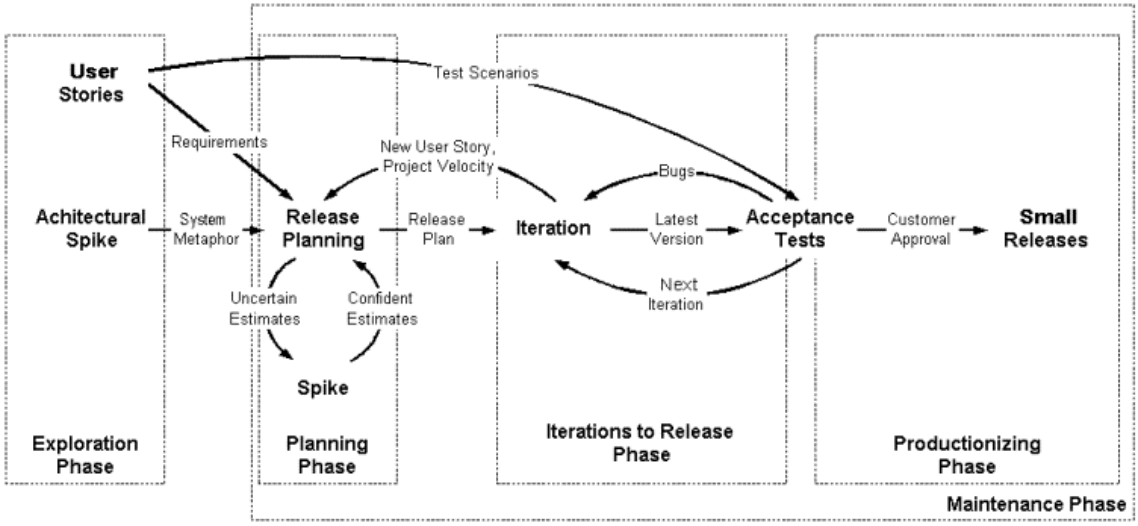
The ideas behind XP are not that "extreme" or new. Rather, XP takes traditional development practices "to the extreme", for example by writing all unit test cases before any code is implemented. It also takes to the extreme the fact that the end customer wants working software over documentation. In XP, the amount of documentation is limited to what is absolutely necessary; for example, all implementation documentation should be automatically generated from the code and its comments. The code should be the documentation itself, because eventually when someone has to make changes to the code, it is more important to know what the actual code does over what a possible outdated document says. Coding standards and shared code ownership aid in this matter.

XP phases

A process lifecycle of XP can be broken down to the phases shown in figure 5.4.

- *Exploration* - During the exploration phase the user requirements of the system are written on story cards. Enough story cards are produced to make the customer satisfied that the material will make a good first release. At the same time, the development team familiarizes itself with the technology to be used during the actual development. Architectural ideas are also be considered and evaluated. This is a phase where the system gets its form and the development team uses the technol-

Figure 5.4: eXtreme Programming phases



ogy for practice. The exploration phase can have a duration from a few weeks to a few months, depending on how well the team knows the problem domain and the technology to be used.

- *Planning* - During the planning phase a deadline for the first production release is set with the smallest, most valuable set of stories done. The schedule for the first release should be set at two to six months - any shorter than this will probably not deliver any real solutions to the business and any longer than that carries too much risk. The planning phase should take no more than a few days.

- *Iteration* - The implementation of the system starts with the iteration phase. Implementation is broken down into one- to four week iterations, each of which will produce a set of functional test cases for each of the stories scheduled for the respective iteration. The first iteration should include stories that structure the whole architecture for the system - the "skeleton". The customer picks the forthcoming stories for each iteration. At the end of an iteration the customer should have a working system with added features.

- *Productionizing* - In the productionizing phase everything from the iteration phase gets even more tightened. Iteration cycles may be only one week long, new test cases are introduced to make sure that the system is ready for production and the performance is fine-tuned, if needed. Any stories left unimplemented from other postponed ideas, because of deadline pressure, are listed and may be added later, in the maintenance phase.

- *Maintenance* - After the initial release the XP project goes into the maintenance phase. This is where the live system is supported, while new iterations produce simultaneously new user stories. This typically means customer support tasks and may require new staff for maintenance tasks. Development time decelerates.

- *Death* - The death phase means that all the user stories have been implemented and that the customer doens not want nor need any new ones. In this phase all necessary documentation is written which means no changes in the architecture or the implementation. Naturally, if the customer no longer wants to maintain the system or for other reasons abandons, it the system is in the death phase.

Values

XP identifies the following five values as important during the development: communication, simplicity, feedback, courage and respect. At closer counting, all of the above factors are important in any software development projects. XP just emphasizes them explicitly.

Transparent communication between the customer, and also inside the development team, simplifies the understanding of the requirements as well as the current state of the project. In traditional development this task is accomplished by documentation but because XP emphasizes working code over written documentation no time is sacrificed on writing documents.

In XP the developer should ask "what is the simplest solution to get this working?". The developer should not think of any future designs over the current functionality. It

is about not constructing a framework that is never used, but rather about solving the problem *ad hoc*. Simplicity also supports communication; a simple design with very simple code is easily understood by most programmers throughout the team.

XP believes in feedback from the system and also from the customer as early as possible. To get feedback from the system, a user writes unit tests to test every user story (requirement). The customer writes functional tests (acceptance tests). Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break.

It takes courage to solve a problem today instead of designing for the future. Courage means reviewing the existing system and modifying it so that future changes can be implemented more easily. Another example of courage is to know when to abandon code: it takes courage to remove source code that is obsolete, no matter how much effort was used to create that source code.

Respect implies that no one in the team should commit such code to the source repository that will break existing unit tests. Members respect their work by always striving for high quality and seeking for the best design for the solution at hand through refactoring. This also ultimately implies that members of an XP team should not be novices but they should stand above standard programmers meaning that every member of the team is able to on his own create and implement a good design and spot problem early on.

5.3.2 Scrum

Scrum[1] is an agile software project management method in contrast to XP which is an agile software development method. It was introduced by Ken Schwaber in 1996 [16]. Scrum will not define in what way the software is developed, what documents are to be produced or how requirements are defined or gathered. Rather, Scrum is a guide on

[1]From rugby - "a tight formation of forwards who bind together in specific positions when a scrumdown is called"

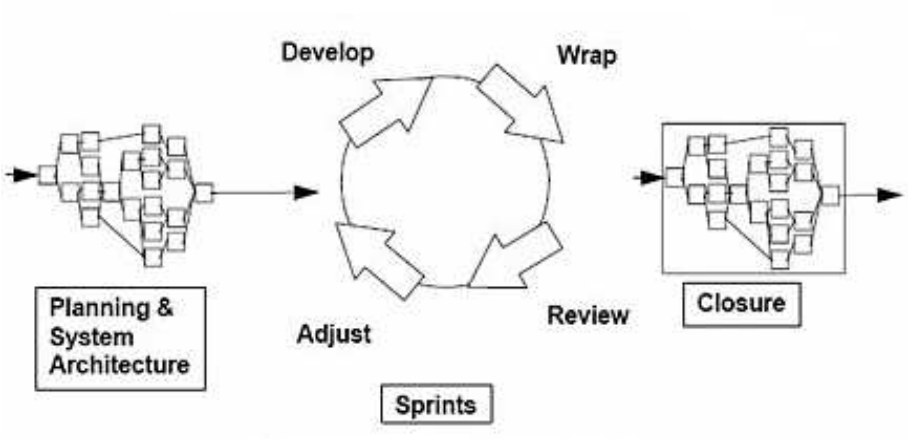
how an agile implementation team should be managed. Schwaber [16] observes that Scrum is probably most successful when it is used for prototyping new technology or for implementing a completely new system with a number of uncertainties. Obviously, Scrum and XP work very well together.

As any agile approach, Scrum notices that the development phase is under constant pressure of change and involves several environmental and technical variables (e.g., requirements, time frame, resources and technology) that are likely to change during the process. Scrum goes as far as calling the development environment as being a set of "chaotic circumstances".

Scrum phases

Scrum consists of three phases: Pre-game phase (planning and system architecture), development phase (sprints) and post-game phase (closure) shown in figure 5.5.

Figure 5.5: Scrum phases



- *Pre-game phase* - In the pre-game phase, all the requirements and system features are collected into a Product backlog. The requirements are prioritized and the effort needed for their implementation is estimated. The product backlog is kept up-to-date and reviewed by the Scrum team at every iteration. Scrum does not on comment how and when the product backlog came into existence, but only that it

must exist before implementation when it needs to be updated. In the pre-game phase a system architecture is created which is based on the product backlog. If any problems related to the architecture arise as a result of new backlog items, the problems and their solutions are included in the product backlog. Again, Scrum does not say anything about the process architecture in detail.

- *Implementation phase* - The Implementation phase (or the game phase) is the agile part of Scrum. Scrum aims at controlling the uncertain and changing variables of the development (such as time frame, quality, requirements, resources, implementation technologies and tools, and even development) rather then specifying them at the beginning. Scrum controls the development in agile iterations called sprints. Sprints are iterative development cycles during which the development team implements the selected items from the product backlog. Each sprint includes the traditional phases of software development: requirements, analysis, design, evolution and delivery. The architecture and the design of the system evolve during sprint development. A sprint may have a duration from one to four weeks. Again, Scrum does not comment on how the development team works internally.

- *Post-game phase* - In the post-game phase all items in the product backlog are implemented. The post-game phase includes also documentation, integration and system testing. The system is now ready to be released.

Scrum roles

Scrum defines the roles and tasks for different people inside the Scrum team. The team consists of: Scrum master, product owner, the Scrum team, customer, user and management.

The *Scrum master* is the boss for the Scrum team. His roles it to see that the development is done according to Scrum practices and rules but acts also as a buffer between the customer, the management and the implementing team.

The *product owner* is the person responsible of the product backlog and therefor of control, management and visibility. He makes the final decisions of the tasks related to product backlog, participates in estimating the development effort for backlog items and turns the requirements in the backlog into features to be developed.

The *Scrum team* is the group of people implementing the product backlog. Scrum only says that the team should organize itself to be able to implement the product backlog. In reality, this means using some existing software development methodology; for example, XP works very well together with Scrum.

The *customer* participates in the tasks related to product backlog items for the system being developed or enhanced. The customer is the stakeholder ending up using the final system.

Management is in charge of final decision making, along with the charters, standards and conventions to be followed in the project. Management deals with backlog, risk and release content.

5.3.3 Agile methods - pros and cons

While agile development is a fairly new concept it has become obvious that it is not the silver bullet to solve all the problem in software engineering. But in a short time it has proven to help in many cases where the more traditional development methods have failed. Agile development is maybe the development method getting most attention in larger companies and leading them to explore on how much and which parts from agile development could benefit them. Next I will discuss some of the pros and cons related to agile development.

Pros

The obvious strength of agile development is that it provides the ability to react easily to change requests while the development is ongoing. If we had perfect knowledge of the

system requirements in the beginning of the project, a waterfall model would be suitable. Usually, however, this is not the case. The customer can after each iteration choose the tasks that he consideres important for the next release. This would not be possible with the waterfall model. Agile methods also stress the importance of a tight relationship between development and continuous testing. Each iteration must provide the customer with a fully functional, well tested solution with known features. If some features are not fully tested (or implemented) they should be left out of the current iteration. Such omission could be due to time constraints or wrong time estimates during sprint planning. This should give onfidence in the maturity of the software throughout the development process.

Agile methods focus on functionality - something that in the end matters for the customer. A prototype is often constructed as early as possible to support the understanding between the customer and the development team. A prototype also highlights the most important factors of the implementation, since it disregards all extra, possibly useless code, that is written into projects "just in case".

Only a minimum of documentation is done in advance. Also, agile development often recognizes that source code comments are valuable documentation as such, whereas more traditional approaches (for example RUP) do not tend to recognize code as a resource for documentation.

Cons

Agile methods usually suit better small, dynamical teams where all individuals work well as team players and as individuals experts as well. This is, of course, not always possible. Because the development environment of the project is so dynamic, the project members need to be able to adapt and invent new solutions in a rapid pace. Also multisite projects are a challenge to agile development due to the tight nature of testing and implementation.

Agile methods tend also to neglet some parts of the process that may be eventually

critical for it. For example, every project needs to estimate its duration, cost, some overall requirements and above all - if the project is worth spending on. Agile methods, in general, give the impression that this should be done quickly, indeed very quickly. Planning and exploring are important parts of scheduling any project and this takes time. Agile methods tend to overlook this by setting features aside into a backlog for later revisiting as necessary. Of course, not every project will work like that. It may be very difficult to express all requirements in the form of user stories or use cases.

XP believes in refactoring the code, if the design requires it as a result of a failed unit test or if a newly implemented feature requires it. However, Beck states [15] only that refactoring requires *courage* and nothing more detailed on how this should be done. Also, refactoring does not address any problems in the overall framework such as scalability issues.

XP explicitly requires a very intimate relationship between customer and development team, it requires that the customer on-site full-time, and that customer writes the stories and defines the releases. This kind of relationship is more frequent in in-house developments than in formal development contracts with external clients.

Chapter 6

Software Process Improvement

In the previous chapter I have described some software process models for developing software. A software process model is an abstraction of a software process, which, in turn, is an abstraction of a set of real-life activities. Because of the abstracted, or simplified, nature of software process models, the activities have to be customized and optimized to the actual development environment (business goals, work methods, system requirements, resources etc.).

In this chapter I will describe the methodology for improving implemented software development process models, i.e., *Software Process Improvement* (SPI). SPI is the activity where an implemented software development process is being improved to either meet a reference model (chapter 6.1) or a set of business goals (chapter 6.2.1) external to the development process. These two approaches have a common goal: to make the implemented software development process better for the organization, but they look at it from two different perspectives. *Software metrics* are closely related to SPI, since they provide a measure of the process improvement and guide to the actions necessary with regard to the goal of the SPI.

In the next chapter I will apply the GQM model in a lightweight project assessment method used in Plenware Oy for internal audits. The assessment method is used in conjunction with SPI to characterize and improve the agility of processes.

6.1 Process Reference Models

A process reference model is not a process model that is implementable by itself. Instead, it proposes an ideal configuration of processes based on well known patterns that are proven to be good. This approach has several advantages. First, it sets a goal: What would be an ideal way to run the process? It gives explicit direction towards improvement. Second, this approach allows comparision between achievements before and after software process improvements and between organizations. A capability assessment against a reference process model results into a measurement called *capability level*. This indicates how well the software process is working compared to the reference process model. A high capability level makes a clear advantage over competitors when competing for subcontractor deals.

I will present two well known process reference and assessment models that are used in software engineering; CMMI and SPICE. I will also briefly mention ITIL and ISO 9001 which are process models not strictly related to software development.

6.1.1 CMMI

Capability Maturity Model Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes on developing software. It helps making business management decisions on how to develope software as a whole in an organization. In contrast to a software development model, that only describes for programmers how to code, test, deploy and build on their software, CMMI describes how the software management model should work as a whole - a way for software project managers to plan, organize and identify what needs to be done to run a software development project succesfully. CMMI includes processes that are required in software implementation, configuration and testing, but as we see later on in this chapter, it includes also many other components that are vital for managing a software develop-

ment process.

The history of CMMI begins in the 1980s. At that time, most projects failed to deliver on time and on budget, and often they did not deliver at all. To tackle these problems, and to lower the price of software devopment projects, the United States Department of Defense funded the Software Engineering Institute (SEI) at Carnegie Mellon University to find ways to help defense contractors build software more economically. They focused on defining what a successful project is, how to determine the quality of software, and then to find projects that met the criteria and analyze what these successful projects had in common. The result was the Capability Maturity Model for software, or CMM. Subsequent maturity models for other aspects of managing technical projects were created. In 2001, the CMMI was released which integrated all the models.

The current version (1.2) of CMMI has two framework components: CMMI for Development (CMMI-DEV) and CMMI for Acquisition (CMMI-ACQ). CMMI-DEV addresses product and service development processes while CMMI-ACQ addresses supply chain management, acquisition and outsourcing processes in government and industry. In this thesis I will only describe CMMI-DEV.

CMMI Process Areas

CMMI-DEV (CMMI version 1.2) [19] defines 22 Process Areas (PA). A process area is a cluster of related practices in an area that, when implemented collectively, satisfies a set of goals that are important for making improvement in that area. Every PA consists of at least one specific goal (SG) and 1-5 generic goals (GG). SGs are made of at least two specific practices (SPs) and GGs are made of generic practices (GPs). Each SG is unique to a PA whereas GG are common among several PAs. Both SGs and GGs of every PA are required model components and are used in appraisals to help determine whether a process area is satisfied.

For example, a specific goal in the Configuration Management process area is "in-

tegrity of baselines is established and maintained" and an example of a generic goal, also found in the Configuration Management PA, is "the process is institutionalized as a defined process".

The Process Areas for CMMI-DEV 1.2 are:

- Causal Analysis and Resolution (CAR)
- Configuration Management (CM)
- Decision Analysis and Resolution (DAR)
- Integrated Project Management + IPPD (IPM+IPPD)
- Measurement and Analysis (MA)
- Organizational Innovation and Deployment (OID)
- Organizational Process Definition + IPPD (OPD+IPPD)
- Organizational Process Focus (OPF)
- Organizational Process Performance (OPP)
- Organizational Training (OT)
- Product Integration (PI)
- Project Monitoring and Control (PMC)
- Project Planning (PP)
- Process and Product Quality Assurance (PPQA)
- Quantitative Project Management (QPM)
- Requirements Development (RD)
- Requirements Management (REQM)
- Risk Management (RSKM)
- Supplier Agreement Management (SAM)
- Technical Solution (TS)
- Validation (VAL)
- Verification (VER)

Staged and continuous representation of CMMI

When an organization is rated for a maturity, certain process areas are "staged" together with the expectation that the groupings makes sense as building blocks. Since the latter blocks depended on the prior blocks, the groupings resembles stair-steps, or "levels". This is called the staged representation of CMMI and is the original approach of CMM. In the staged representation, the summary components are maturity levels.

Continuous representation is an alternative approach to implement CMMI. In the original approach of CMM the organization could not improve and master a certain set of process areas, probably more important to them, without needing the staged representation. Hence, in the staged representation the ability to mature a capability in any one process area does not exist, so in CMMI, the idea of a continuous representation was implemented. Here an organization can choose to optimize its processes at any number of PAs without having to put forth efforts to implement low-value or unused PAs within their organization. This becomes especially meaningful to organizations that need to benchmark themselves (or to be formally rated) in only areas that matter to them. The continuous representation of the model allows organizations to pick any number of PAs and also to pick the level of capability in the pertinent process areas. The key determinant in such a capability lies in the generic goals and an organization's process improvement achievement is measured in capability levels.

The generic goals are parallel with the capability levels. Generic Goal 1 (GG1) aligns with Capability Level 1 (CL1). GG2 with CL2, GG3 with CL3 and etc. Hence, the PAs of an organization are performing at Capability Level 3 the PAs should be at Generic Goal 3 level. The generic goals are cumulative, i.e., if a process area is CL3 (or GG3) GG1 and GG2 are achieved as well. The generic goals are met by successfully supporting the generic practices included in a specific generic goal. Hence, every generic goal (capability level) is made of a set of certain generic practices.

The generic goals that determine the capability level of an organization in a chosen

set of PAs, are:

- *Generic Goal 1* - The process supports and enables achievement of the specific goals of the process area by transforming identifiable input work products to produce identifiable output work products.
- *Generic Goal 2* - The process is institutionalized as a managed process.
- *Generic Goal 3* - The process is institutionalized as a defined process.
- *Generic Goal 4* - The process is institutionalized as a quantitatively managed process.
- *Generic Goal 5* - The process is institutionalized as an optimizing process.

Maturity Levels

An organization is assessed by a maturity level which is rated from 0 to 5. A maturity level consists of related specific and generic practices for a predefined set of process areas that improve the overall performance of the organization. In other words, the maturity levels are measured by the achievement of the specific and generic goals associated with each predefined set of process areas.

The maturity level of an organization provides a way to predict the future performance of an organization within a given discipline or set of disciplines, for example CMMI-DEV defined disciplines for (software) engineering. High maturity levels are of a great value for organizations competing in, for example, subcontractor deals.

The process areas (PAs) are associated with maturity levels. When an organization is to be assessed for a certain maturity level, it needs to achieve the PAs required for the respective maturity level and also for any consecutive PAs in previous maturity levels. This approach to CMMI is called the staged representation of CMMI. The organization gets *appraised to a maturity level*. A team lead by a SEI certified lead appraiser scrutinizes at the evidence produced by projects. The number of projects required depends on the organization. There are three types of evidence: direct artifacts, indirect artifacts and

affirmations. At least one Direct Artifact plus either an Indirect Artifact or an Affirmation is required for each practice in the scope of the appraisal. The appraisal process follows a guide called Method Definition Document (MDD).

An overview of the maturity levels is provided next.

- *Level 0: Incomplete.* The PA (for example requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.

- *Level 1: Performed.* All of the specific goals of the PA (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

- *Level 2: Managed.* All level 1 criteria have been satisfied. In addition, all work associated with the PA conforms with an organizationally defined policy: all people doing the work have access to adequate resources to get the job done, stakeholders are actively involved in the process area as required, all work tasks and work products are monitored, controlled and reviewed and are evaluated for adherence to the process description

- *Level 3: Defined.* All level 2 criteria have been achieved. In addition, the process is tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process improvement information to the organizational process assets.

- *Level 4: Quantitatively managed.* All level 3 criteria have been achieved. In addition, the PA is controlled and improved using measurement and quantitative assessment. Quantitative objectives for quality and process performance are established and used as criteria in managing the process.

- *Lever 5: Optimized.* All capability level 4 criteria have been achieved. In addi-

tion, the PA is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration."

6.1.2 SPICE

SPICE (Software Process Improvement and Capability dEtermination, also known as ISO/IEC 15504 standard) [21] is a software process assessment framework developed by the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC). The goal is to provide an international standard for software process assessment and to allow comparisons among different assessment methods. The first version of the SPICE standard was released in 1995 and the current version was revised in 2004.

The SPICE standard has nine parts, which include guidance material, a reference process model and an example model.

The purpose of the reference model is to act as a common basis for software process assessment and to facilitate the comparison of assessment results. The reference model has two dimensions: a *process dimension* and a *capability dimension*. The process dimension includes the processes to be assessed and the capability dimensions includes the scale on which the processes are assessed.

The process dimension of the SPICE reference model defines five categories of processes: software development, maintenance, acquisition, supply and operation. Part 2 of the SPICE specification defines these process categories. Each process category description includes a characterization of the processes it contains, followed by a list of process names. The purpose of each process is first described, followed by additional detailed descriptions to clarify the meaning of the process, inputs and outputs for the process and a clarification of when the process is invoked. The following example is taken verbatim from the SPICE specification for process management [22]:

> 6.2 Engineering process category (ENG)
>
> The Engineering process category consists of processes that directly specify, implement or maintain a system and software product and its user documentation.
>
> In some circumstances, there is no "system" so the scope of the engineering processes is reduced to only software and user documentation, and processes ENG.1 and ENG.6 become "not applicable."
>
> While the processes listed below appear in "waterfall model" sequence, the intent is not to preclude either their concurrent or iterative execution. (The sequence is determined and documented by base practice, "Determine release strategy" ENG.1.4 and by process, "Plan project life cycle" PRO.1.)
>
> Inputs to the "Engineering" process category possibly include a contract or agreement describing what work is to be done, and a plan(s) on how that is to be accomplished (see processes, "Establish contract" CUS.2, and "Establish project plan" PRO.2.)

This discussion followed by a description of the processes that are included in the engineering process category. These are: ENG.1 Develop system requirements and design, ENG.2 Develop software requirements, ENG.3 Develop software design, ENG.4 Implement software design, ENG.5 Integrate and test software, ENG.6 Integrate and test system and ENG.7 Maintain system and software.

The capability dimension of SPICE defines the following maturity levels for the processes:

- 0 - Incomplete process
- 1 - Performed process
- 2 - Managed process
- 3 - Established process
- 4 - Predictable process

- 5 - Optimizing process

These maturity levels are compatible with the maturity levels of CMMI. The capability of a process is defined using *process attributes*. Each process attribute defines a level of maturity for the process. The SPICE standard defines the following process attributes and their corresponding maturity levels, where the number prior to the dot is the corresponding maturity level and the following number the number for the process attribute within the maturity level:

- 1.1 Process Performance
- 2.1 Performance Management
- 2.2 Work Product Management
- 3.1 Process Definition
- 3.2 Process Deployment
- 4.1 Process Measurement
- 4.2 Process Control
- 5.1 Process Innovation
- 5.2 Process Optimization

Each process is assessed by the evidence of to what extent the process satisfies the attributes above. The attributes are rated as *not satisfied* (N), *partially satisfied* (P), *largely satisfied* (L) and *fully satisfied* (F). The process is assessed at a particular capability level if it fully satisfies all lower capability levels, and at least largely satisfies the process attributes at the assessed capability level.

6.1.3 ITIL

The Information Technology Infrastructure Library (ITIL) [23] is a framework of best practices for IT Service Management (ITSM). ITSM focuses on the fact that the customer of an IT project does not want to concentrate on the technology of their organization nor

the project, but rather on quality of the services they provide and on customer relationship. ITIL is a useful framework for people in an organization providing IT services such, as IT directors and managers and Business Managers. ITIL does not provide guidance or best practices on project or program management, but it does, however, recognize that they are key elements for a successful ITIL implementation.

ITIL was originally created by the CCTA (Central Computer and Telecommunications Agency, now called the Office of Government Commerce (OGC)), a UK Government agency. It is now being adopted and used worldwide as the de facto standard for best practices in the provision of IT services. The latest version of ITIL, called ITIL v3, was released on May 30th, 2007. ITIL v3 consists of five core volumes (books), each focusing on a distinct part of the IT Service Management. The core volumes are:

- *Service Strategy*. The Service Strategy book provides a view of ITIL that aligns business and information technology. It specifies that each stage of the service lifecycle must stay focused on the business case with defined business goals, requirements and service management principles.

- *Service Design*. The Service Design book provides guidance on the production and maintenance of IT policies, architectures and documents.

- *Service Transition*. The Service Transition book focuses on change management and release practices, providing guidance and process activities for the transition of services into the business environment.

- *Service Operation*. This book focuses on delivery and control process activities based on a selection of service support and service delivery control points.

- *Continual Service Improvement*. This book focuses on the process elements involved in identifying and introducing service management improvements and on issues surrounding service retirement.

6.1.4 ISO 9001

ISO 9001:2000 [17] is a well known process reference model for quality management systems that an organization can be certified against. The ISO 9001 process model requires, for example, that the organization has documented all key processes in the business, monitors the processes to ensure they are effective and checks output for defects, with appropriate corrective action, where necessary. ISO 9001 lays a foundation for the organization to get a certification that it operates at a reasonable level of quality assurance. Certification to an ISO 9000 standard does not guarantee the quality of end products and services. Rather, it only certifies that consistent business processes are being adhered to.

An organization, from any field of business, can become ISO 9001 certified, and it is not software development specific. A "product", in ISO vocabulary, can mean a physical object, services or software. Since ISO 9001 is not closely linked to software development (although it could be [18]), I will not cover ISO 9001 in any more depth in my thesis.

6.2 SPI Based on Experience and Business Goals

Software engineering offers a framework called *Quality Improvement Paradigm* (QIP) to improve the quality of the software development process. This paradigm works in strong cooperation with an other paradigm, the *Goal/Question/Metrics Paradigm* (GQM), which supports the establishment of project and business goals and a mechanism for measuring against those goals. These two paradigms are usually used inside an infrastructure called Experience Factory (EF) which defines a set of practises to create packages of experience collected from past projects and reuse them in an organization. I will next discuss all of these paradigms. Together, these three paradigms (QIP, GQM and EF) provide a unified framework for software process improvement based on experience and business goals.

6.2.1 Experience Factory

An important asset of any company is the business knowledge that has accumulated during years of experience. Higher quality at lower cost is usually achieved by reusing processes, knowledge and experience from similar projects that have been succesful in the past. In his paper Victor Basili presents [24] an infrastructure called *Experience Factory* (EF) for improving the quality of software processes by systematically saving and reusing experience from previous projects. It is important to distinguish the experience factory infrastructure from process reference models: the former improves the development process by analyzing business goals, while the latter assesses the process against a given predefined process model that needs to be evaluated against the business needs of an organization.

The experience factory infrastructure defines two distinct organizations: the development organization and the experience factory. The experience factory is a logical organization that supports project development by collecting and analyzing experiences from previous projects, by acting as a repository for such a knowledge and by packaging experience into reusable knowledge packages. The development organization represents the R&D part of the main organization, the ones that actually uses and works by the processes. They also provide the experience factory with all project and environment characteristics, development data, resource usage information, quality records and general feedback from the performance of the models and tools in use.

The experience factory produces what is called *experience packages* that are stored in the knowledge repository. The experience packages may constist of:

- Equations describing the process effort by measured software metrics used to achieve a goal (see GQM (6.2.3))
- Product Packages - a collection of architectures or designs together with reuse documentation to use in future projects

- Process Packages - processes and methods proven to work in certain environments and projects
- Tool Packages - tools that work well together with certain projects in the organization, for example code generators, configuration management tools or testers
- Management Packages - reference information for project management.
- Data Packages - historical information of software metrics from previous projects

By collecting experience packages the organization can (depending on the goal) characterize and understand better the projects and the choices of processes and methods, evaluate and analyze the process to understand what kind of activities improve the outcome, predict and control the expected cost, time and reliability of the process and end product and motivate and improve the techniques used for example in testing and reducing certain kinds of errors. When the concept of EF was introduced in the late 1980s, it had already been researched and implemented by NASA/GSFC Software Engineering Laboratory (SEL) since 1976. During the decade of research that had passed, the increased understanding of project data collection, validation and on how to use the data to benefit the learning organization, had given very encouraging results. Thus, as an example flight dynamics software benefited from the EF model by reducing the number of defects by 75%, reducing costs by 55% and improving reuse by 300% in a period of 4 years [25].

Basili [25] also noted that the key to success of EF is a combination of many factors. He presents the following observations:

- *Establishing a baseline of an organization's products, processes and goals is critical to any improvement program.* First of all, the current situation and the (business) goals that are set must be understood if the exact improvement goal is to be defined.
- *Data collection requires a rigorous process and professional staff.* Data collection (software metrics) is not a part-time activity. It has to be a natural part of the

organization and its processes and needs sufficient resources. Equally important is the continuous motivation and training of the development staff to data collection.

- *The organization attempting to improve their process has to take ownership of the improvement process.* The process management in the organization has to be unified and committed to improving the processes in a unified fashion.

- *The management must be committed to software process improvement*, not only to justify the overhead costs of this activity but to also give a clear signal for focusing on improvement in a controlled fashion.

The organization needs to introduce an experience factory usually because of business needs: the organization needs to build better software in less time. This goal does not, however, come for free. Vasili noted [25] that the introduction of experience factory into the development process added a 10% overhead to the costs. But as mentioned earlier, the organization can save in the future in costs and still deliver software of higher quality. The 10% overhead is an investment for the future if the experience factory paradigm is used properly.

6.2.2 Quality Improvement Paradigm (QIP)

The basic tool for successful implementation of experience factory is the methodology called Quality Improvement Paradigm (QIP) [26]. There are several other process improvement paradigms ([27], [28]) but since QIP evolved from the lessons learned in the SEL project [25] at the same time as EF, I will discuss only QIP here.

QIP consists of six fundamental steps:

- *Characterize.* We need to understand the environment and establish a baseline with the existing business processes in the organization. The characterization builds models of various factors and studies the interactions between them to increase the understanding of the context. of the respective domain. We can characterize the

project by a number of criteria, e.g., the number of people in the project, level of experience, problem and process experience, problem factors (application domain, novelty in relation to the state of the art), programming language and budget.

- *Set goals.* A realistic quantifiable definition of goals correlates with the characterization of the environment. We need to establish goals for the process and these goals should be measurable and model based. Goals can be set with a number of different techniques, but QIP uses the Goal/Question/Metric Paradigm (GQM) described in chapter 6.2.3.

- *Choose process.* Construct a product or implement a project with a chosen software development process model (see chapter 5). The most appropriate software model should be used based on the environment characteristics.

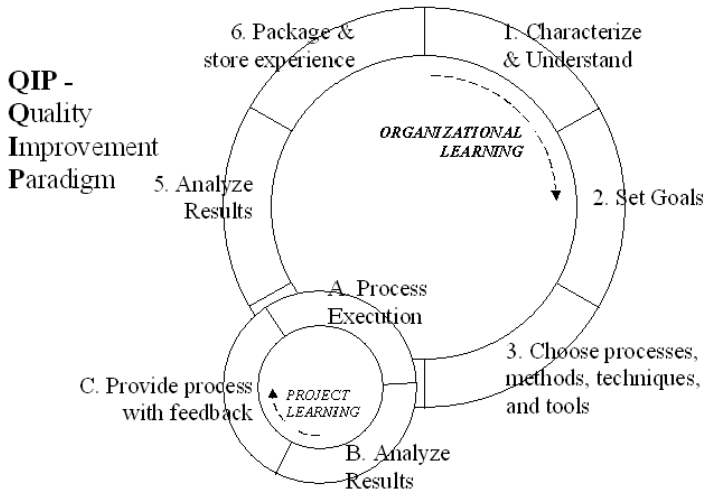
- *Execute.* Execute the processes, construct the products and give feedback based upon the data on the goal achievement that are being collected.

- *Analyze.* After each process, gather data, analyze and evaluate it based on the current practices, determine problems and make recommendations for project improvement. The feedback obtained by analyzing the measurable goals and process effectiveness will support continuous improvement of both business and methodology.

- *Package.* Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base for future use.

The QIP provides two iterative feedback loops (see figure 6.1. The organizational (larger) and the project (smaller) loop. The project-specific feedback loop provides feedback to the project during the execution phase to prevent and solve problems, to monitor and support the project and to realign chosen processes with defined goals. The organizational feedback loop provides feedback to the organization after completion of the project

Figure 6.1: QIP iterative feedback loops.



and adds to the accumulated experience packages described in the experience factory paradigm.

QIP is an iterative process, and data collection and analysis must be an integrated part of the development process, not an add-on.

6.2.3 Goal/Question/Metric paradigm (GQM)

Feedback is an essential part of any improvement, and software process improvement does not make any exceptions. Software metrics makes up for the feedback needed for software process improvement. Only with correctly chosen metrics and valid data can a process be assessed with regards to its progress and to help us support project planning in upcoming projects. But more importantly, metrics helps us to determine the strengths and weaknesses of the current processes and products (to form a baseline) and it provides a rationale for adopting and refining the techniques needed to determine if a process has improved or not.

All measurements of software process improvement must be done in a top-down fashion, since there are very many metrics to measure in a software process and since process

improvement and business goals must determine which ones are relevant. In his paper [29] Victor Basili describes the *Goals/Question/Metric Paradigm* (GQM) as a method for defining and interpreting operational and measurable software. In the Experience Factory infrastructure, GQM is the method for defining the business goals and the data to measure.

GQM defines a measurement model on three levels:

- *Conceptual level (Goal).* A goal is defined for an object with respect to various models of quality, from various points of view, relative to a particular environment. The object of measurement are products (specifications, designs, programs, test suites), processes (specifying, designing, testing, interviewing) and resources (personnel, hardware, software, office space).

- *Operational level (Question).* A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model. Questions are asked to characterize the object of measurement (product, process, resource) with respect to a selected quality issue and to determine its quality from the selected viewpoint.

- *Quantitative level (Metric).* A set of metrics, based on the models, is associated with every question. This set is needed to answer the questions in a measurable way.

The three levels create a hierarchical structure. On the top is the goal that specifies the purpose of the measurement, e.g. "Improve the error correction response time". The goal is refined by subdividing it into several questions that usually break down the issue into its major components, for example "What is the current error correction response time?". Each question is then refined into metrics, for example "Average correction response time". The metrics can be subjective or objective, but they have to be measurable.

GQM is described as a six-step process where the first three steps are about using business goals to drive the identification of the right metrics and the three ultimate steps

are about gathering the measurement data and making effective use of the measurement results to drive decision making and improvements. Basili [29] described his six-step GQM process as follows:

1. Develop a set of corporate-level, division-level and project-level business goals and associated measurement goals for productivity and quality.

2. Generate questions (based on models) that define those goals as completely as possible in a quantifiable way.

3. Specify the measures needed for answering the questions and track the process and product conformance to the goals.

4. Develop mechanisms for data collection.

5. Collect, validate and analyze the data in real time to provide feedback to projects for corrective actions.

6. Analyze the data postmortem to assess conformance to the goals and to make recommendations for future improvements.

Once a GQM model is defined, appropriate data collection techniques, tools and procedures are used to gather, store and analyze the data. The data will then be used in the organization as defined - e.g. in the QIP paradigm as a part of the Experience Factory framework.

Chapter 7

Quantitative Process Assessment

A need for quantitative process assessment emerged in Plenware Oy from internal assessments of a part of the regular internal quality initiatives. The goal of the assessment approach in to assess the agility of the processes, but in a lightweight fashion to minimize project overhead expenses and also to collect quantitative metrics for SPI of those processes. The process is kept lighweight by formulating a set of questions with quantitative metrics as answers. The metrics should be easy to obtain from a process management tool. The development process is broken down into three groups with questions in each of them; *project and requirement management*, *development* and *testing*. Each question has a point scale that the answers can be compared against; the overall agility is reflected by the sum of the points. The metrics in each of the groups give indications for the SPI initiatives to spot bottlenecks.

7.1 Motivation

For internal audits several assessment methods are available. The ISO 9001 [17] standard could have been used to assess the quality of the process. Alternatively, SCAMPI[30] class C is sufficient to assess the maturity by the CMMI-DEV[19] rating. In our case, however, the goal is not to obtain official certification of external audits, but to use process

assessment as a tool for QIP and Software Process Improvement initiatives to assess the agility of a process and to identify improvement items in that process. Thus, instead of doing long preparations and passing through cumbersome formalities required by official assessments we only needed a lightweight method that conformed to our needs.

At the moment, no serious academic research is apparently focusing on assessing the agility of a process. Related work has been published by Pikkarainen [31], [32] on combining SPI with lightweight agile process assessment using CMMI. However, our goal is not to conform to any reference process model including CMMI. Abrahamsson's work [33] is related to achieving CMMI maturity levels with XP. Again, our goal is not to use CMMI, but to improve the development process from business goals rather than being CMMI compliant. Hence, we needed to develop our own lightweight assessment approach that focuses on assessing the agility of a process and collects quantitative data that could be used for SPI.

Because the goal of the assessment was to be a part of SPI, we also wanted to be able to compare previous audit results of a given process, so that improvement could be measured and compared. Thus, the assessment method needed to be quantitative in the sense that some of the assessment results could be presented as metrics of the process.

7.2 Background

We define the following goals for our lightweight quantitative agile process assessment:

1. *Lightweight.* Traditionally, an audit is a big event that needs to be prepared days and months in advance. Also, the audit process itself may take several days. For example, if the organization is to perform a SCAMPI class A audit to get a staged CMMI-DEV maturity level, four people needed to prepare and perform the audit on the organization for several weeks[34]. Our goal was to be able to perform the audit in a matter of hours that would not carry an extra burden (in terms of labor

hours or cost) on the process or the organization.

2. *Quantitative.* We needed to rely on quantitative data rather than qualitative data as much as possible to be able to measure any improvement. An additional benefit was that gathering quantitative metrics can be automated. We decided to collect metrics using GQM. Besides metrics that are driven by business goals, we needed to establish the relevant metrics to support SPI for our process.

3. *Complete.* The assessment needed to be able to collect the right data to be able to compare the development process against a given reference model. The reference model could be CMMI, ITIL, an organization specific model or, in our case, the agility in general. A requirement was that our assessment process should give as its output the status of the improvement in the process after a few audit iterations.

While developing the assessment process we defined a set of *components* which in our opinion are necessary, to achieve the stated goals of the assessment process: *the assessment checklist*, *assessment metrics and points* and *metric boundaries*. I will discuss each of these components in the following sections.

I will also present the findings and suggestions for the components that were collected during a lightweight quantitative process assessment that was conducted for two agile projects in Plenware Oy. These findings should be considered as a set of starting points for further research.

The ideas and findings mentioned in this thesis can be applied to create a lightweight assessment process for any given assessment, for example RUP-based processes or even usability assessment. However, since these assessment areas are out of the scope of this presentation, I will focus on assessing the agility of processes only.

7.3 Implementation of the Quantitative Agile Assessment Method

The quantitative assessment method presented in this thesis consists of three parts which I will discuss in more detail; *The assessment checklist*, *metrics and points* and *metric thresholds*.

In section *7.3.1 The Assessment Checklist* I will present a set of questions that were used for assessing the process. The questions were divided into three groups; *project and requirement management*, *development* and *testing*. The grouping was done based on the activities found in the projects to be assessed in Plenware Oy and was considered to represent the different agile parts of the projects. Each group contained questions to reflect the specific area of the process. Mostly of the questions were quantitative, but there were also some binary yes-or-no questions and qualitative questions for non-quantifiable variables. The decision to select the questions presented here was influenced by the IEEE draft for the Recommended Practice for the Customer-Supplier Relationship in Agile Software Projects [35], together with knowledge of agile development in general and the experience from assessments in Plenware Oy of Scrum-based processes.

In section *7.3.2. Assessment Metrics and Points* I will present the metric scale to calculate an *assessment point* for each question. Each question will be given a score from 0 to 3, where a higher number means "better". A score of 3 is considered optimal for an agile process, while a score of 0 indicates that the question reflects a serious problem in agility. In the last section (*7.3.3. Metric Thresholds*) I will discuss some threshold values for the metrics. These threshold values degrade the overall agility of the process and therefore each question with a result over a threshold value scores -5. We use these values to address "improvement possibilities" as a part of SPI.

Each question can be given an adjustable weight, depending on the importance of the question. However, I have not used a weighted sum in my thesis but this is mentioned

here as a reference for future studies in the field; which weights should be used for any given question to get the best possible representation of agility from that question? By calculating the weighted sum of the assessment points, a quantitative result is obtained that will yield the assessment result. In our case, the result tells how agile the process is.

7.3.1 The Assessment Checklist

Below are the questions to identify agility in the process. The decision to select these specific questions was influenced by the IEEE draft for the Recommended Practice for the Customer-Supplier Relationship in Agile Software Projects [35], together with knowledge of agile development in general and the experience from assessments in Plenware Oy of Scrum-based processes. However, one should not take the presented list of questions as given, but rather as a starting point for this kind of quantitative assessment of agile processes and for future research.

An agile development process may be split into three assessment areas: *project and requirement management*, *development* and *testing*. The questions are grouped into these areas. The areas were chosen mainly based on the experience from the assessment conducted in Plenware Oy, because, in our case, we realized that the process was naturally split into these entities. The questions are based on experience from assessments in Plenware Oy of Scrum-based processes and the literature on agile processes.

These questions are relevant from the SPI point of view to improve the agility of a process. Any business oriented questions (for example those based on GQM) should be added to make the assessment relevant to a given organization. It is also worth to note that this assessment looks at the project as an agile process and that there are no "right" or "wrong" answers, only answers that reflect the agility of a process. Naturally, other aspects will influence the success of the project and how successful the process is, but these are not relevant to agile development and assessing the agility of the process.

Project and Requirement Management

Table 7.1 presents the questions that search for evidence for best practices in agile project management. Good agile project management, according to [35] and based on the experience in assessment at Plenware Oy, has the following characteristics:

- The customer should be actively involved in the development process. The features and bug fixes needed for implementation during a sprint are selected by the customer from the product backlog. The backlog, in turn, is based on the priorities set up by the customer. To understand these requirements, the customer should be available all the time to explain and create/refine use cases. The customer should also be a part of the process when creating test cases - or at least understand and approve them.

- Agile development works in short iterations (so called sprints) which should be 2-8 weeks in duration.

- Fluent communication between the customer and within the agile team is essential. Therefore internal daily "stand-up meetings" should be held by the project manager to discuss with the development team the current status of the project, to bring up any problems and uncertainties that need to be resolved and to help understand the overall picture of the project. The daily meetings should be short, maybe some 15 minutes in maximum.

Table 7.3.2 presents the answers to these questions with a corresponding point scale to reflect the characteristics above.

Development

Table 7.2 presents questions that identify evidence for best practices in agile development. A well working agile development process has the following characteristics:

Table 7.1: Agile project and requirement management assessment questions

	Assessment Question for Project and Requirement Management
PRM.1	How long are your release cycles / sprints?
PRM.2	How many sprints do you have for this project?
PRM.3	How many developers do you have in your agile team?
PRM.4	How many test engineers do you have in your agile team?
PRM.5	How often do you have meetings with the developers and testers?
PRM.6	How long are the meetings?
PRM.7	How many issues are discussed during the meetings?
PRM.8	What proportion of developers and testers are present at internal sprint start-up meetings?
PRM.9	How many contact persons do you have from the part of the customer?
PRM.10	How many contact persons does the customer have towards the development team?
PRM.11	How many milestones or releases are defined for this project?
PRM.12	How often is the scope of the project re-planned and updated?
PRM.13	Who updates/owns the product and the sprint backlog?
PRM.14	Do you use any project management tools in this project?

- The features and bug fixes that are selected from the backlog are split into smaller manageable subtasks. A subtask is an activity that takes about 6-18 hours to do and supports the use cases that are selected for implementation in the current sprint.
- The development process should consist of frequent, time-boxed iterations that deliver code that is ready to deploy to the customer at the end of each iteration.
- The co-operation between the development and the testing teams should be close. Code should be committed to the version control system at least nightly and a new

build of the software should be available for testing daily.

Table 7.3.2 presents the answers to the questions with a corresponding point scale to reflect the characteristics discussed above.

Table 7.2: Agile development assessment questions

	Assessment Question for Development
DEV.1	How many features do you have for this sprint?
DEV.2	How many subtasks is each feature split into?
DEV.3	In general, how many features must be pushed back to the product backlog at the end of each sprint?
DEV.4	How many open bugs do you have for this sprint?
DEV.5	How many bugs do you have in this sprint from the product backlog?
DEV.6	How much "extra time" is given for cleaning up and re-factoring the source code?
DEV.7	How often is source code committed to the version control system?
DEV.8	Who creates time estimates for the features?
DEV.9	Who splits features into subtasks?

Testing

The questions in table 7.3 identify evidence for best practices for agile testing. A well working agile test process has the following characteristics:

- The system is proven to work by passing repeatable tests that the customer helps to specify.
- Tests should be updated and run constantly on daily builds of the system. The tests should preferably be automated.

- Communication between the testers and developers should be seamless, but the testing process itself should be autonomous - development should not interfere with how the tests are run on a feature.

- All features must pass the test cases before they are delivered to the customer. If a feature fails a test, the feature is not shipped and it is pushed back to the product backlog.

Table 7.6 presents the answers to the questions with a corresponding point scale to reflect the characteristics above.

Table 7.3: Agile testing assessment questions

	Assessment question for testing
TST.1	How many times will a test engineer usually test a feature during a sprint?
TST.2	How often is a build made for testing?
TST.3	How many unit tests do you have?
TST.4	How many automated unit tests do you have?
TST.5	Who writes the unit tests for the features?

7.3.2 Assessment Metrics and Points

In this section I shall present the assessment points for the corresponding questions presented in section 7.3.1. The answer to each question will get points in relation to how well it suits agile development characteristics. As mentioned earlier, the scale and the optimal characteristics presented here are based on knowledge of agile methods from the literature and experience from assessments in Plenware Oy and it will require further research to establish more relevant characteristics and corresponding quantitative assessment questions.

A scale of 0 to 3 is used, where 3 is considered optimal while 0 is considered not to be agile. The grading points are presented along with each question. Not all questions use the whole scale. For the qualitative questions, the assessor needs to adopt a scale depending on the answer. In section *Characteristics of the optimal qualitative answers* I will discuss the characteristics the assessor should look for in the qualitative questions.

Characteristics of the optimal qualitative answers

Here I will present the reasoning behind the qualitative questions. There is not one correct answer to the qualitative questions but instead the assessor should get a feeling of the general agility of the process based on the answers. Below I will present one interpretation behind the reasoning to the questions and what the answers should reflect.

The answer to question PRM.13 *"Who updates/owns the product and sprint backlog?"* should reflect the fact that the customer owns the product backlog and selects features or use cases to the sprint backlog based on his or hers needs and priorities. If the customer does not care about the sprint development model but wants agile development for other reasons (that is, the customer expects a product after a fixed budget or time table), then the project manager should own the product backlog and select the features for each sprint from the product backlog.

The answer to question DEV.8 *"Who creates time estimates for the features?"* should take into considiration that according to agile principles it is the implementing team that provides with time estimates and in that sense commits to the time schedule. The implementing team is also naturally the best source of knowledge in technical matters for the project and they should be able to give realistic time estimates. Depending on the nature of the answers, fewer points can be given if only one the lead developer or architect makes the time tables.

The same goes for question DEV.9 *"Who splits features into subtasks?"*. The implementing team gets the features as input to the sprint. The features should be split

Table 7.4: Agile Project and Requirement Management assessment metrics and points

	Assessment Question for Project and Requirement Management	Assessment points
PRM.1	How long are your release cycles / sprints?	3p: 4-6 weeks 2p: 3,7 weeks 1p: 2,8 weeks
PRM.2	How many developers do you have in your agile team?	3p: 1-3 developers 2p: 4-5 developers 1p: 6 developers
PRM.3	How many test engineers do you have in your agile team?	3p: 2 testers 2p: 1 tester
PRM.4	How often do you have meetings with the developers and testers?	3p: Every day 1p: Every other day
PRM.5	How long are the meetings?	3p: 15 minutes 2p: <15 minutes 1p: 15-30 minutes
PRM.6	How many items are discussed during the meetings?	3p: 3 items 2p: 4 items 1p: <3 items
PRM.7	What proportion of developers and testers are present at internal sprint start-up meetings?	3p: All 2p: 2/3 of them 1p: 1/3 of them
PRM.8	How many contact persons do you have from the customer?	3p: 1 contact
PRM.9	How many contact persons does the customer have towards the development team?	3p: 1 contact
PRM.10	How many milestones or releases are defined for this project?	3p: Release for every sprint 1p: Release every other sprint
PRM.11	How often is the scope of the project re-planned and updated?	3p: After each sprint 1p: After about every other sprint
PRM.12	Do you use any project management tool in this project?	3p: Yes
PRM.13	Who updates/owns the product and sprint backlog?	

into manageable pieces by the developer implementing the feature. As stated in question DEV.2 *"Into how many subtasks is each feature split into?"* the subtask should have a duration no longer than 18 hours. On the other, the duration must be long enough as not too small to create unnecessary overhead to project management. The subtasks should be created for easy follow-up of time usage.

In question TST.5 *"Who writes the unit tests for the features?"* the optimal situation is that the customer writes the unit test because the customer knows exactly what is needed from the use cases that are the basis for the features. But this is usually too much to ask. For successful agile, testing the customer should at least be aware of what tests will be run, understand them and be committed to the tests.

Agile testing should be as automated as possible. Tests are the foundation for a successful release at the end of each iteration and it is only the test results that can guarantee the quality of the release. At then end of each iteration, a new working increment of the software should be available to the customer to use in a production environment and it is only by extensive tests that this can be guaranteed with. Hence, the customer's role in specifying the tests is vital.

7.3.3 Metric Thresholds

I have presented a set of questions to assess the agility of a process. These questions have yielded points in three different process areas for agile software development (*project and requirement management*, *development* and *testing*). A specific question was given 0 points if the answer was not inside the scale that is presented next to the question and means that the characteristics that the question reflects did not improve the agility in that area.

However, there are metrics based on the questions that in fact will *degrade* the overall agility of a process area. Below I will present some relevant metrics and their threshold values that may be considered harmful to agility and thus will score negative points (-5)

Table 7.5: Agile development assessment metrics and points

	Assessment Question for Development	Assessment Points
DEV.1	How many features are to be implemented for this sprint?	3p: 1 feature 1p: 2 features
DEV.2	Into how many subtasks is each feature split into?	$time = \frac{PRM.1}{DEV.1 \cdot DEV.2} \cdot \frac{1}{PRM.3}$ 3p: 7.5h $\geq time$ <18h 2p: 3.5h $\geq time$ <7.5h 1p: 18h $\geq time$ <22h
DEV.3	In general, how many features must be pushed back to the product backlog at the end of each sprint?	3p: None 1p: One feature
DEV.4	How many open bugs do you have for this sprint?	3p: 0 bugs 1p: $bugs < \frac{DEV.2}{2}$
DEV.5	How many bugs do you have in this sprint from the product backlog?	3p: 0 bugs 1p: $bugs < \frac{DEV.1}{3}$
DEV.6	How much "extra time" is given for cleaning up and re-factoring the source code?	3p: > 15% of the time 2p: 10%-15% 1p: 5%-10%
DEV.7	How often is source code committed to the version control system?	3p: > Daily 2p: Daily
DEV.8	Who creates time estimates for the features?	
DEV.9	Who splits features into subtasks?	

Table 7.6: Agile testing assessment metrics and points

	Assessment Question for Testing	Assessment points
TST.1	How many times will a test engineer usually test a feature during a sprint?	3p: 3 times 2p: 4-5 times 1p: 1-3 times
TST.2	How often is a build made for testing?	3p: Nightly 1p: Every other night
TST.3	How many unit tests do you have?	3p: DEV.2 1p: 1-DEV.2
TST.4	How many automated unit tests do you have?	3p: DEV.2 2p: 1-DEV.2
TST.5	Who writes the unit tests for the features?	

to the process area.

Project and Requirements Management Threshold Metrics

Below are the threshold values for metrics that are especially harmful for agile development in a project. If the questions below fit the threshold description, the question should be scored -5.

- PRM.1 *"How long are your release cycles / sprints?"*: The iterations should not be longer than 8 weeks, but also not 1 week. Any iteration longer than 8 weeks will degrade the process into the waterfall development model.

- PRM.5 *"How long are the meetings?"*: Transparent communication inside the project is essential for any project, but probably even more important for agile development. If the project manager does not have meetings at least every other day with the development team, the project work is harmed.

- PRM.12 *"Do you use any project management tool in this project?"*: The scope of the overall project should be revised after every sprint. No long term design decisions or specifications should be made for forthcoming iterations because, like above, the development process will be based on the waterfall model if the project is designed ahead.

- PRM.13 *"Who updates/owns the product and sprint backlog?"*: If it has not been made clear to the customer that the customer owns the product backlog and if the project manager (or similar) makes decisions instead of the customer for forthcoming iterations, then the project will degrade into the waterfall model.

- PRM.14 *"Do you use any project management tools in this project?"*: If the testing and the implementing team are not located in the same room, some sort of project management, bug tracking and/or project tool is essential for good communication

between testers and developers. With a tool like this, the tester will, for example, get notified immediately of new implementation or fix to test - and the developer will get noticed of new faults in the features.

Development Threshold Metrics

The threshold values for agile development answers that score -5 for the agile development area are:

- DEV.1 *"How many features are to be implemented for this sprint?"* and DEV.2 *"Into how many subtasks is each feature split into?"*: The iteration should consist of a manageable amount of features split into subtasks. The subtasks should be of about 4-18 hours in duration. If the number of subtasks per developer per features in one iteration exceeds 5, then the feature should be split into new subtasks on a higher level.

- DEV.3 *"In general, how many features must be pushed back to the product backlog at the end of each sprint?"* and DEV.5 *"How many bugs do you have in this sprint from the product backlog?"*: A feature has not been scheduled appropriately if it is pushed back to the product backlog at the end of an iteration. It is, of course, natural that not every estimate is right, but, still, if every iteration has more than 1 feature pushed back, then the features should be split into subfeatures and be implemented in separate iterations.

- DEV.6 *"How much "extra time" is given for cleaning up and re-factoring the source code?"*: An essential part of agile development is to refactor the code instead of planning ahead. That is why some extra time must be scheduled into every iteration to account for clean-up and refactoring. If less than 5% of the project time is scheduled for refactoring, the development process is not agile.

- DEV.7 *"How often is source code committed to the version control system?"*: Source

code must be committed daily to the source code repository. If source code is not committed daily, development is probably not agile.

- DEV.8 *"Who creates time estimates for the features?"* and DEV.9 *"Who splits features into subtasks?"*: If someone else than the implementing team (or a leading team member) splits features into subtasks and creates timetables for them, the development process is not agile.

Testing Threshold Metrics

The threshold values for agile testing answers that score -5 for the testing development area are:

- TST.2 *"How often is a build made for testing?"*: Nightly builds should be made of the software that the testers can run tests against. If builds are not done automatically on a regular basis, the testing cannot be agile.

- TST.5 *"Who writes the unit tests for the features?"*: If the implementing team can influence the test results by, for example, explaining how some test should be run, testing is not agile.

7.4 Process Improvement

Now that we have categorized the assessment questions into three project areas, it is easy to analyze how well the assessed processes succeeded in the respective areas by summing all the points based on the answers. The maximum points based on these questions and with the scale presented here are:

- Project and Requirement management: 39p

- Development: 24p

- Testing: 15p.

It is easy to see from the checklist which areas that need to be improved. Some improvement in agility can be done if a process area did not score maximum points. This assessment method can easily and in a lightweight fashion spot fundamental problems of agility of a process and because this method is lightweight, it is well suited for SPI initiatives.

When examining QIP (6.2.2), we can determine that we have fulfilled some of its principles: we have *characterized* the current process, we know what our *goals* are to achieve more agility and we can *analyze* the process to measure the improvements. QIP tells us that once we have a tool to analyze the process and its improvement, we need to iterate over a sequence that consists of choosing an improved process, execute the process, measure any improvements with the altered process and package experience on how we achieved the improvement.

The qualitative assessment method presented in this thesis could be integrated as a part of QIP and GQM in an organization. The assessment method would be a part of the QIP process when evaluating for any improvement in the process. Using GQM the organization could find more relevant assessment goals that are based on their business goals which would yield in different questions. Also the points scale and weights could be assigned differently based on an evaluation done with GQM. However, this should be the focus on a different research and is left out from this thesis.

Chapter 8

Discussion

In this thesis I have examined why software engineering is so difficult and have presented development processes to reduce the complexity of software development. Also, the complexity of today's software is alone a reason why so many software projects fail. Failure is further augmented by the lack of well-defined development processes and poor communication not only between contractor and client but also within the development team.

To address these issues, I have examined software development processes. I started with the most straightforward one, the waterfall model, where tasks are done sequentially one after another. To deal with some of the issues in the waterfall model, I presented an evolved version of it, the spiral model, which adds basic iterative principals into the waterfall model. Next I presented the RUP model, which defines a rich set of practices and documentation according to which the organization needs to tailor suitable processes. Then two agile development models, XP and Scrum, were described as a response to changing customer needs and requirement prioritizing. They work in short iterations where the customer needs and working, tested, software is valued over anything else.

Then I described SPI that introduces methodologies for improving development models set up by organizations. I looked at two approaches to SPI; Process Reference Models and Experience Factory. In the first, a development process was assessed against a given

reference model (CMMI, ITIL or SPICE) that contains best practices for the process, while the latter uses business goals and metrics to create experience packages of best practices best suitable for the given organization (QIP and GQM).

After looking at both of these development models and how to improve them, I suggest a lightweight assessment method for agile processes. The goal of this method is to assess the agility of a process using quantitative metrics and a checklist with predefined questions that are divided into three project areas (project and requirement management, development and testing). Each question has a point scale that the answers can be compared against; the overall agility is reflected by the sum of the points. Also, some threshold values for these metrics were discussed to address major problems of the agility of a process that could degrade the overall agility. The use of predefined questions with quantitative answers to assess a specific aspect of the process makes the assessment very lightweight compared to traditional assessments..

It is, nevertheless, worth observing that this assessment method has not been tested for its validity in this thesis. As mentioned in the previous chapter the assessment method is a result of the needs at Plenware Oy at the time and the presented assessment method is a result of assessments of two internal projects at Plenware Oy. The questions, together with the corresponding point scale, were chosen from the goals that were set those two specific projects. Thus, the goal for this assessment was only to find the level of agility of these processes and consequently the presented questions do not give general answers on how well the process works or how successful the project will be in other aspects, even if the assessment would give good scores. To be agile just for the sake of agility is no silver bullet to success in software engineering, although agile methods are well suited to handle the changing requirements that software development projects generally face.

Also, because the validity of the assessment method has not been tested outside Plenware and with only two projects, the set of questions may need revision. Such a revisioning would require, in my opinion, some 3-5 processes to be assessed using the presented

method. Is a score of -5 enough for the agility assessment area enough to reflect how the overall agility of a process is influenced by some threshold values, or should there be a variation to this negative value depending on the question and the agility area? Should the score scale be more fine tuned? As of today, there is no standard method for assessing the agility of a process so this also requires more research.

8.1 Limitations and Future Work

The assessment method presented in this thesis has been developed to assess a single small agile team with one project manager, maybe one software architect and a few developers and testers. It is certainly possible to refine the method by collecting assessment metrics results from several projects to measure the overall agility of the development in an organization, but the variability of agile development processes should be then taken into account since there is no official method of doing agile development. The presented assessment method is based on the assumption that the development process uses Scrum and XP to introduce agility into development. Probably a different set of questions should be developed for projects using Crystal Clear or Agile Unified Process (AUP). Also the same methodology could be used for assessing different kinds of processes and a completely different set of questions and points should be considered for assessing a process based on RUP. Similary, it could also be possible to quantify the usability of an user interface of an application. Then it would be interesting to examine how to integrate the results of the assessment of two such different aspects of a software engineering project into an overall result.

The assessment method presented in this thesis is a good starting point for future research and work. The demand for lightweight process assessment used internally by companies as a part of SPI is huge. We have seen that this method fits well with Experience Factory and SPI methods, but more work is required to adopt this assessment method

on a corporate level and also to combine GQM into this method to measure business goals in cooperation with agile process models.

References

[1] Brooks Jr., F. P. : *No Silver Bullet - Essence and Accidents of Software Engineering*, Computer Magazine, Volume 20 , Issue 4 (April 1987)

[2] International Software Benchmarking Standards Group: *Practical Project Estimation 2nd Edition*, http://www.isbsg.org (2005)

[3] Royce, Winston : *Managing the Development of Large Software Systems*, Proceedings of IEEE WESCON 26 (August): 1-9, http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

[4] P. Naur and B. Randell, (Eds.). *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969) 231pp., http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF - checked December 2007.

[5] B. Randell and J.N. Buxton, (Eds.). *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO (1970) 164pp. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF - checked December 2007.

[6] http://en.wikipedia.org/wiki/Waterfall_model#Usage - checked December 2007.

[7] http://www.it-director.com/technology/productivity/content.php?cid=7865 - checked December 2007.

[8] Boehm, Barry W. : *A Spiral Model of Software Development and Enhancement*, IEEE-CS Computer, Volume 21, Number 5 (May 1988)

[9] *IBM Rational Unified Process*, http://www-306.ibm.com/software/awdtools/rup/?s_tact=105agy59&s_cmp=wiki&ca=dtl-08rupsite - checked december 2007.

[10] Ivar Jacobson, Grady Booch and James Rumbaugh: *The Unified Software Development Process*, ISBN-13: 978-0201571691, Addison-Wesley Professional, February 4, 1999

[11] *IBM Rational Process Composer*, http://www-306.ibm.com/software/awdtools/rmc/features/index.html - checked December 2007.

[12] *Eclipse Process Framework Project (EPF)*, http://www.eclipse.org/epf/ - checked December 2007.

[13] Smith, John : *A Comparison of RUP and XP*, Rational Software White Paper, 2001.

[14] Barnett, Liz : *Agile Survey Results: Widespread Adoption, Emphasis on Productivity and Quality*, Agile Journal and VersionOne, August 2007, http://www.agilejournal.com/articles/from-the-editor/agile-survey-results

[15] Beck, Kent : *Extreme Programming Explained*, Addison-Wesley, 2000

[16] Schwaber, Ken : *SCRUM Development Process*, Advanced Development Methods, 1996.

[17] ISO 9001:2000, http://www.iso.org/iso/iso_catalogue/management_standards.htm, checked March 2008.

[18] Mutafelija, Boris and Stromberg, Harvey: *Mappings of ISO 9001:2000 and CMMI Version 1.1*, Software Engineering Institute, July 2003, http://www.sei.cmu.edu/cmmi/adoption/pdf/iso-mapping.pdf, checked March 2008

[19] Software Engineering Institute (SEI): *CMMI for Development, Version 1.2*, August 2006.

[20] *The CMMI website*, http://www.sei.cmu.edu/cmmi/ - checked January 2008.

[21] Gerard O'Regan, *A Practical Approach to Software Quality*, Springer-Verlag, New York, 2002.

[22] *SPICE specification - Part 2 : A model for process management, Version 1.00*, ISO/IEC, July 1995.

[23] *IT Infrastructure Library (ITIL)*, http://www.itil-officialsite.com/home/home.asp, checked March 2008.

[24] V. Basili, G. Caldiera and D. Rombach: *The Experience Factory*, Encyclopedia of Software Engineering. Wiley 1994.

[25] V. Basili, F. McGarry, R. Pajerski, M. Zelkowitz: *Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory*, Proceedings of the 24th International Conference on Software Engineering, May 19-25, 2002, Orlando, Florida

[26] V. R. Basili, *Quantitative Evaluation of Software Engineering Methodology*, Proc. of the First Pan Pacific Computer Conference, Melbourne, Australia, September 1985 [also available as Technical Report, TR-1519, Dept. of Computer Science, University of Maryland, College Park, July 1985].

[27] W. Edwards Deming, *Out of the Crisis,* MIT Center for Advanced Engineering Study, MIT Press, Cambridge MA, 1986.

[28] Armand V. Feigenbaum, *Total Quality Control*, fortieth anniversary edition, McGraw Hill, NY, 1991.

[29] V. Basili, G. Caldiera and D. Rombach: *The Goal Question Metric Approach*, Encyclopedia of Software Engineering. Wiley 1994.

[30] Software Engineering Institute: *Standard CMMI Appraisal Method for Process Improvement (SCAMPI), Version 1.1: Method Definition Document*, http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01hb001.pdf

[31] M. Pikkarainen, A. Mantyniemi: *An Approach for Using CMMI in Agile Software Development Assessments: Experience from Three Case Studies*, SPICE 2006 Conference, Luxemburg, 4.5-5.5.2006

[32] M. Pikkarainen, F. Mc Caffery, I. Richardson: *AHAA - Agile, Hybrid Assessment Method for Automotive, Safety Critical SMEs*, International Conference on Software Engineering, Leipzig, Germany, 2008

[33] P. Abrahamsson, T. Kahkonen: *Achieving CMMI Level 2 with Enhanced Extreme Programming Approach*, 5th International Conference on Product Focused Software Process Improvement (PROFES 2004), 5-8 April 2004, Japan, LNCS 3009, Springer

[34] M. Hoggerl, B. Sehorz: *An Introduction to CMMI and its Assessment Procedure*, Seminar paper for Seminar for Computer Science, Department of Computer Science, University of Salzburg, February 2006

[35] Software and Systems Engineering Standards Committee, IEEE Computer Society: *Draft Recommended Practice for the Customer-Supplier Relationship in Agile Software Projects*, IEEE P1648/D5, New York, February 2007