Jeanette Heidenberg

Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches

Turku Centre *for* Computer Science

TUCS Dissertations
No 133, January 2011

Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches

Jeanette Heidenberg

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Alpha, the ICT Building on February 11, 2011, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland

2011

Supervisor

Prof. Iván Porres
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

Reviewers

Prof. Casper Lassenius
Department of Computer Science and Engineering
Aalto University
POB 19210, 00076 Aalto
Finland

Prof. Dr. Tom Mens
Software Engineering Lab
Institut d'Informatique, Faculty of Sciences, University of Mons – UMons
Avenue du Champ de Mars 6, 7000 Mons
Belgium

Opponent

Prof. Casper Lassenius
Department of Computer Science and Engineering
Aalto University
POB 19210, 00076 Aalto
Finland

ISBN 978-952-12-2534-5
ISSN 1239-1883

In loving memory of Judit Östman 1924-2010

"I keep on living, breathing though you're far away.
Your gentle eyes smiling at me from far away."
— Camilla Heidenberg 2010

Sammanfattning

Dagens programvaruindustri står inför alltmer komplicerade utmaningar i en värld där programvara är nästan allstädes närvarande i våra dagliga liv. Konsumenten vill ha produkter som är pålitliga, innovativa och rika i funktionalitet, men samtidigt också förmånliga. Utmaningen för oss inom IT-industrin är att skapa mer komplexa, innovativa lösningar till en lägre kostnad.

Detta är en av orsakerna till att processförbättring som forskningsområde inte har minskat i betydelse. IT-proffs ställer sig frågan: "Hur håller vi våra löften till våra kunder, samtidigt som vi minimerar vår risk och ökar vår kvalitet och produktivitet?" Inom processförbättringsområdet finns det olika tillvägagångssätt. Traditionella processförbättringsmetoder för programvara som CMMI och SPICE fokuserar på kvalitets- och riskaspekten hos förbättringsprocessen. Mer lättviktiga metoder som t.ex. lättrörliga metoder (*agile methods*) och Lean-metoder fokuserar på att hålla löften och förbättra produktiviteten genom att minimera slöseri inom utvecklingsprocessen.

Forskningen som presenteras i denna avhandling utfördes med ett specifikt mål framför ögonen: att förbättra kostnadseffektiviteten i arbetsmetoderna utan att kompromissa med kvaliteten. Den utmaningen attackerades från tre olika vinklar. För det första förbättras arbetsmetoderna genom att man introducerar lättrörliga metoder. För det andra bibehålls kvaliteten genom att man använder mätmetoder på produktnivå. För det tredje förbättras kunskapsspridningen inom stora företag genom metoder som sätter samarbete i centrum.

Rörelsen bakom lättrörliga arbetsmetoder växte fram under 90-talet som en reaktion på de orealistiska krav som den tidigare förhärskande vattenfallsmetoden ställde på IT-branschen. Programutveckling är en kreativ process och skiljer sig från annan industri i det att den största delen av det dagliga arbetet går ut på att skapa något nytt som inte har funnits tidigare. Varje programutvecklare måste vara expert på sitt område och använder en stor del av sin arbetsdag till att skapa lösningar på problem som hon aldrig tidigare har löst. Trots att detta har varit ett välkänt faktum redan i många decennier, styrs ändå många programvaruprojekt som om de vore produktionslinjer i fabriker. Ett av målen för rörelsen bakom lättrörliga metoder är att lyfta fram just denna diskrepans mellan programutvecklingens innersta natur och sättet på vilket programvaruprojekt styrs.

Lättrörliga arbetsmetoder har visat sig fungera väl i de sammanhang de skapades för, dvs. små, samlokaliserade team som jobbar i nära samarbete med en engagerad kund. I andra sammanhang, och speciellt i stora, geografiskt utspridda företag, är det mera utmanande att införa lättrörliga metoder. Vi har nalkats utmaningen genom att införa lättrörliga metoder med hjälp av pilotprojekt. Detta har två klara fördelar. För det första kan

man inkrementellt samla kunskap om metoderna och deras samverkan med sammanhanget i fråga. På så sätt kan man lättare utveckla och anpassa metoderna till de specifika krav som sammanhanget ställer. För det andra kan man lättare överbrygga motstånd mot förändring genom att introducera kulturella förändringar varsamt och genom att målgruppen får direkt förstahandskontakt med de nya metoderna.

Relevanta mätmetoder för produkter kan hjälpa programvaruutvecklingsteam att förbättra sina arbetsmetoder. När det gäller team som jobbar med lättrörliga och Lean-metoder kan en bra uppsättning mätmetoder vara avgörande för beslutsfattandet när man prioriterar listan över uppgifter som ska göras. Vårt fokus har legat på att stöda lättrörliga och Lean-team med interna produktmätmetoder för beslutsstöd gällande så kallad omfaktorering, dvs. kontinuerlig kvalitetsförbättring av programmets kod och design. Det kan vara svårt att ta ett beslut att omfaktorera, speciellt för lättrörliga och Lean-team, eftersom de förväntas kunna rättfärdiga sina prioriteter i termer av affärsvärde. Vi föreslår ett sätt att mäta designkvaliteten hos system som har utvecklats med hjälp av det så kallade modelldrivna paradigmet. Vi konstruerar även ett sätt att integrera denna mätmetod i lättrörliga och Lean-arbetsmetoder.

En viktig del av alla processförbättringsinitiativ är att sprida kunskap om den nya programvaruprocessen. Detta gäller oavsett hurdan process man försöker introducera – vare sig processen är plandriven eller lättrörlig. Vi föreslår att metoder som baserar sig på samarbete när processen skapas och vidareutvecklas är ett bra sätt att stöda kunskapsspridning på. Vi ger en översikt över författarverktyg för processer på marknaden med det förslaget i åtanke.

Abstract

The challenges of the software industry get more complex as software permeates more and more of our daily lives. The consumer wants products that are reliable, rich in functionality, innovative but at the same time inexpensive. The industry faces the challenge of creating more complex, innovative solutions to a lesser cost.

For this reason, process improvement is as relevant as ever in the software industry. The question asked by software professionals is: "How do we keep our promises, while minimising our risk, increase our quality and productivity?" Traditional software process improvement (SPI) approaches such as CMMI and SPICE focus on the quality and risk aspect of the improvement process, while more light-weight methods such as agile and lean focus on keeping promises and increasing productivity by reducing waste.

The research presented in this dissertation was performed with that specific goal in mind: improving cost efficiency in the way of working while at the same time not compromising quality. This challenge was attacked from three different angles: the deployment of agile methods for improving the ways of working; the usage of product metrics for quality improvement; and the collaboration on process knowledge for dissemination within a large company.

Agile methods evolved during the nineties as a reaction against the unrealistic expectations placed on software development teams by the then prevailing waterfall method for software project management. Software development is a creative process. It differs from other industries in the fact that the main investment in a software development project is in creating something for the first time. Each developer has to be an expert in her own field and spends large portions of her day creating new solutions and solving problems that she has not solved before. Despite the fact that this has been well known for decades, software development projects are still often run as if they were manufacturing lines. One purpose of the agile movement was to make visible this discrepancy between the nature of software development and the practice of how software projects are run.

Agile methods have been proven to work well in the context for which they were designed: small, collocated teams working towards a committed customer. In other contexts, and specifically in large, geographically distributed settings, the adoption of agile methods is more challenging. Our approach to this challenge has been to use piloting as a deployment method. This has the two-fold benefit of incrementally building knowledge about the methods and their interaction with the context in question while at the same time helping to overcome resistance to change by gently introducing the cultural changes through first-hand experience.

Relevant product metrics can help any software development team im-

prove their way of working. In the case of lean and agile teams, a good set of metrics can be crucial in making the right decisions when prioritising the work items in the backlog. Our focus has been on providing lean and agile teams with internal product metrics for decision support on refactoring. A decision to refactor may be difficult to make, especially for a lean or agile team, that is expected to justify their priorities in terms of business value. In our work, we propose a way of measuring design quality for systems developed using the model-driven paradigm. We also construct a method for incorporating the metrics in the lean and agile way of working.

Software process dissemination is an important part of any software process improvement initiative, regardless of whether the target process is plan-driven or agile. We propose that collaboration on process authoring is a good way to support dissemination and we have surveyed the process authoring tools on the market with this aspect in mind.

Acknowledgements

This dissertation is the result of many years of intense and interesting work – work I could not have accomplished on my own. Many have supported and helped me in different ways, and I owe my debt of gratitude to each and every one of you. The words on this page are a humble recognition of what you have all done for me. Some of you I mention by name, some of you remain anonymous, but the gratitude I feel is for you all.

First I would like to thank my supervisor and friend Professor Iván Porres at Åbo Akademi University not only for supporting and believing in me, but also for initially talking me into using the opportunity that life presented me with to finish my thesis, as well as making it financially feasible for me to do so.

I owe my ernest gratitude to Professor Tom Mens of the University of Mons and Professor Casper Lassenius of Aalto University for reviewing and providing valuable feedback on my dissertation. I am also grateful to professor Lassenius for agreeing to act as opponent at the public defense of my thesis.

All of the papers that serve as a foundation for this thesis where written in cooperation with other people. I am grateful to all my co-authors for the hard and, for the most part, also enjoyable work we carried out together.

I also wish to extend my thanks to Professor Jockum von Wright who initially sparked my interest in research in the area of Computer Science and Engineering.

My gratitude also goes out to the administrative staff of Åbo Akademi and TUCS for their support in all things practical around the completion of this work. I especially wish to thank Tomi Mäntylä, Christel Engblom, Britt-Marie Villstrand, and Irmeli Laine.

All my research was carried out in industry settings. I am grateful for the unique opportunity I was given to learn by immersing myself in the interesting world that is the software industry today. For this and for the support you gave me I wish to, first and foremost, thank the management at Ericsson Finland, including Harri Oikarinen, Heli Hirvo, Sanna Johansson, Hannu Ylinen, and Marko Koskinen. For the opportunity to work with process improvement on a company-wide level, I also wish to thank the management at EB and especially Pertti Korhonen.

Learning is much more effective and fun when done through discourse with your peers. For the many interesting discussions on model driven development I am grateful to the Ericsson SWAN team and the Ericsson IBM Rational team, including Pär Emanuelsson, Olivera Milenkovic, Patrik Nandorff, Tommy Lennhamn, Magnus Antonsson, Diarmuid Corcoran, Dietmar Fiedler, Jussi Katajala, and Andras Vajda.

Hard work, as interesting as it may be, can still be taxing. For keeping my

spirit up by sharing not only my work load, but also both my laughter and frustration, I wish to thank all my good friends and colleagues at Ericsson and EB, including but not limited to Andreas Nåls, Calle Björkell, Terho Närvä, Martin Dusch, Kurt Resch, Peter Kraus, Jari Partanen, and all the past and present members of my family at EB: the EB SEP team. My thanks also go to all my friends at the Software Engineering Lab and the Department of Information Technologies at Åbo Akademi. I also wish to thank my friends outside work for their understanding and patience with my often hectic schedule.

I also owe more gratitude than I can even begin to express to Su Dongyue and all my dear friends at the European Zhineng Qigong Center for supporting me in maintaining both my physical and mental health and for helping me grow as a human being over the last decade.

Last but by no means least, I wish to thank my loving family for their undying support throughout. There is no substitute for the sense of security you get from knowing that, through thick and thin, you are loved. I wish to thank my parents, and especially my mother, for always letting me know they are proud of me. Thank you Camilla, my kindred spirit, I'm so blessed and proud to call you my sister. Last but by no means least: to Chakie, my loving husband, my life companion, my soul mate: thank you for being you and for always being there.

This work was supported by Åbo Akademi, Turku Centre for Computer Science, Stiftelsen för Åbo Akademi and Hans Bangs stiftelse.

Turku 5.1.2011

Jeanette Heidenberg

List of original publications

I Jussi Auvinen, Rasmus Back, Jeanette Heidenberg, Piia Hirkman, and Luka Milovanov. Software process improvement with agile practices in a large telecom company. In *Product-Focused Software Process Improvement, volume 4034 of Lecture Notes in Computer Science*, pages 79-93. Jürgen Münch and Matias Vierimaa, editors. Springer Berlin / Heidelberg. PROFES 2006.

II Jeanette Heidenberg, Andreas Nåls, and Ivan Porres. Statechart features and pre-release maintenance defects. *Journal of Visual Languages and Computing*, 19(4), pages 456-467. Philip Cox and John Hosking, editors. Elsevier Ltd 2008.

III Jeanette Heidenberg, Petter Holmström, and Ivan Porres. Tool support for collaborative software process authoring in large organizations. In *European Systems & Software Process Improvement and Innovation Industrial Proceedings*. Jørn Johansen, Mads Christiansen, editors. DELTA, Denmark. 16th EuroSPI Conference 2009.

IV Jeanette Heidenberg, and Ivan Porres. Metrics functions for kanban guards. In *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. Roy Sterrit, Brandon Eames and Jonathan Sprinkle, editors. IEEE ECBS 2010.

V Jeanette Heidenberg, Piia Hirkman, Mari Matinlassi, Jari Partanen, and Minna Pikkarainen. Systematic piloting of agile methods in the large: Two cases in embedded systems development. In *Product-Focused Software Process Improvement, volume 6156 of Lecture Notes in Computer Science*, pages 47-61. M. Ali Babar, Matias Vierimaa, and Markku Oivo, editors. Springer Berlin / Heidelberg. PROFES 2010.

VI Jeanette Heidenberg, Jussi Katajala, and Ivan Porres. Maintainability index for decision support on refactoring. TUCS Technical Report No 992, Turku Centre for Computer Science, November 2010, ISBN 978-952-12-2517-8.

Contents

Part I

Research Summary

Chapter 1

Introduction

The software development industry tends to the needs of an increasingly computerised world. Today, software can be found in almost every aspect of our life. The technology we rely on, from our telephones and computers to our cars and kitchen appliances, all need software to function.

The challenges of the software industry get more complex as software permeates more and more of our daily lives. The consumer wants products that are reliable, rich in functionality, innovative but at the same time inexpensive. The industry faces the challenge of creating more complex, innovative solutions to a lesser cost.

For this reason, process improvement is as relevant as ever in the software industry [51]. The question asked by software professionals is: "How do we keep our promises, while minimising our risk, increasing our quality and productivity?" Traditional software process improvement (SPI) approaches such as CMMI and SPICE focus on the quality and risk aspect of the improvement process, while more light-weight methods such as agile and lean focus on keeping promises and increasing productivity by, e.g., reducing waste.

Software development is a creative process. It differs from other industries in the fact that the main investment in a software development project is in creating something for the first time. There are few manufacturing types of tasks and these can be easily automated. Each developer has to be an expert in her own field and spends large portions of her day creating new solutions and solving problems that she has not solved before. Despite the fact that this has been well known for decades, software development projects are still often run as if they were manufacturing lines.

One purpose of the agile movement was to make visible this discrepancy between the nature of software development and the practice of how software projects are run. Declarations such as the agile manifesto [2] and the declaration of interdependence [5] highlight the human aspect of software projects. The key to success is, according to the agile movement, motivated, creative people collaborating to deliver what the customer needs.

It comes as no surprise that the agile movement initially met with resistance from more traditional camps of the software industry. This resistance has both technical and cultural aspects. The practices used in many agile methods can be seen as technically challenging, especially in large organisations [51]. Examples include continuous integration and test driven development [64]. The cultural aspects include the management's fear of losing control [20] through, e.g., quantitative project management and measurement and analysis [73].

In this thesis, we suggest that the solution lies in the combination of the acceptance of software development as a creative craft as propagated by agile and lean methods on the one hand with lightweight measurement mechanisms borrowing from the more traditional command and control paradigms on the other hand. Organisations that have reached a certain maturity using plan-driven development methods, should not throw the baby out with the bath-water. A healthy level of measurement can support the agile team in different ways. Good product metrics can help the team make the right decisions regarding the effort spent on, e.g., refactoring and creating unit tests. Good product and process metrics can support the team in prioritising their task list and improving their way of working.

This thesis is supported by six years of industry experience in process improvement through agile deployment and product metrics, starting from a small scale pilot of a selected subset of agile practices [9] initiated in 2004 and ending with a large scale agile deployment initiative, which is still ongoing [34].

This first part provides the background and glue for the individual papers that make up the rest of this thesis. Chapter 2 provides background information on the key concepts: agile methods, lean and kanban methods, collaborative process authoring, software quality, software metrics, and action research. Chapter 3 provides an overview of the research, including a presentation of the research questions, the research methods and data collection methods used and a presentation of each of the papers.

The second part contains all the original publications.

Chapter 2

Background

This chapter introduces the key concepts of the thesis. The first concept introduced is agile methods in Section 2.1, including a brief overview of two of the most popular agile methods: Extreme Programming and Scrum. This is followed by an introduction to lean and kanban methods in Section 2.2, starting with these concepts in the field of manufacturing and followed by their translation into the field of software development. The third concept introduced is collaborative methods used for bottom-up process improvement, which is described in Section 2.3. Section 2.4 briefly summarises the vast field of software quality, focusing on the concept of maintainability, which is a central theme for this thesis. The last section, Section 2.5 gives an overview of action research, which is the overall research framework we used in the research presented in this thesis.

2.1 Agile Methods

The agile methods movement started in the early 1990's with several simultaneous efforts being made to guide the software industry away from the then prevailing waterfall model [67]. The industry saw the emergence of methods such as Scrum [69], eXtreme Programming [12], Chrystal [17], DSDM, Adaptive Software Development, Feature-Driven Development, and Pragmatic Programming. In the year 2001, representatives of these methods gathered and distilled their different methodologies to the now famous Agile Manifesto (see Figure 2.1).

Two of the most widely deployed methods today [3] are Scrum and eXtreme Programming (XP). These are often used together, since Scrum is a project management framework, while XP addresses development practices and methods in software projects.

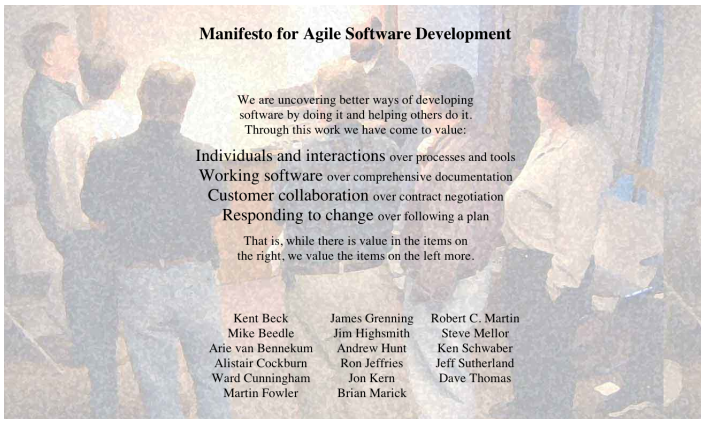


Figure 2.1: The agile manifesto. Fetched from [2].

2.1.1 Extreme Programming

Extreme Programming (XP) [12] is maybe the best known of the agile methods. It is credited to Kent Beck, Ward Cunningham and Ron Jeffries in the mid 1990s. The name comes from the basic principle of taking practices that work well, such as early testing, and using them "to the extreme", as in test driven development where tests are written before any code is written.

The XP process is iterative and incremental with very small increments. An iteration in XP is typically less than four weeks. The input for an iteration is a number of requirements in the form of user stories. These are discussed and prioritised in cooperation with the customer, so that the user stories that provide the most business value are implemented first.

The philosophy of XP is based on five values: communication, simplicity, feedback, courage and respect. XP also prescribes twelve practices to be used when developing software.

XP has evolved over time and some of the concepts have been modified. There used to be only four values. The value of respect is a later addition. The names of the twelve practices have also changed. In the list below, the new name is stated in brackets when applicable.

The Planning Game The team and the customer negotiates the priority of the user stories to be implemented in the next increment.

Small Releases An iteration is no longer than four weeks and every iteration results in a testable increment of the product for the customer to evaluate.

Metaphor The system is described using a simple metaphor. This simplifies communication with the customer and helps internal communication of the architecture.

Simple Design The team should not over-design the system for possible future needs. The design should be based on what is known of the system today. This is sometimes also referred to as the YAGNI principle, an acronym for "You Ain't Gonna Need It".

Early Testing Test early, preferably already before the code is written.

Refactoring (Design Improvement) Continuously improve the design of the system. This way the architecture emerges with the system rather than up front.

Pair Programming Improve quality by working in pairs. Two people share one computer and take turns in acting as the driver (who does the typing) and the navigator (who maintains a higher-level perspective.)

Collective Ownership The whole team owns the produced artifacts. This means in particular that anyone in the team can change any part of the code at any time.

Continuous Integration Ensure that the whole system can be integrated all the time. This is sometimes achieved by means of so called nightly (or daily) builds.

40-Hour Week (Sustainable Pace) Treat the project as a marathon rather than a sprint. Ensure that the working pace is one that can be sustained over time.

On-Site Customer (Whole Team) Ensure that the team has all the members it needs to perform its task. The customer should also be represented in the team.

Coding Standards Enable collective ownership by adhering to shared coding standards.

XP has shown to be both successful [24] and challenging [66, 64, 30]. Critics of this method make the points that it is difficult to scale to large projects, that it is too simplistic and too ad hoc. The individual practices have also been criticised, interestingly enough the criticism here often seems to stem from the practices being too strict. Practices such as pair programming, unit testing and continuous integration require strict discipline both on a personal and an organisational level, which may be difficult to maintain.

2.1.2 Scrum

Scrum [69] is an agile project management framework introduced by Jeff Sutherland and Ken Schwaber in the early 1990s. Scrum is owned by an

organisation called Scrum Alliance, who provides training and certification of Scrum professionals. One of the main merits of Scrum is that the theory is easy to learn. A Scrum Master Certification course is only two days long, and the basics of the method can be summarised in just a few pages. The basic process is depicted in Figure 2.2.

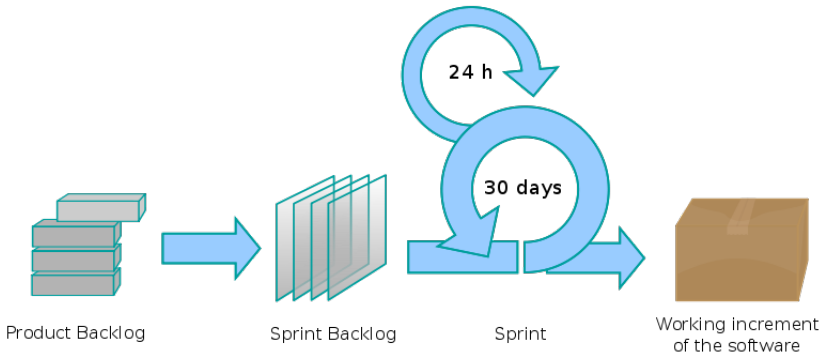


Figure 2.2: The Scrum process, fetched from[46].

Just as with XP, Scrum has evolved over the years and new concepts have been introduced. The basic roles of a Scrum project are the Scrum Team, which is a self-organising, cross-functional team of seven ± two members; the Scrum Master, who is responsible for enforcing the process and acts as a buffer shielding the team from external interruptions; and the Product Owner, who maintains the customer perspective.

The work is focused on transforming the product backlog, which contains all the user stories to be implemented, into working increments of software. This is done through a series of sprints – short iterations of a maximum of four weeks. For each sprint, a portion of the product backlog is selected for implementation in cooperation with the Product Owner. This is called the sprint backlog. The heartbeat of the sprint is the daily scrum meetings. The team gets together for 15 minutes every day in a highly formalised meeting, where progress is tracked. Each member answers three questions:

- What have you done since the last meeting?
- What will you do until the next meeting?
- What are your impediments?

It is common to use a low-tech solution like sticky notes on a whiteboard to visualise the progress. As sprint backlog items get assigned and completed,

they are moved from section to section on the whiteboard. The whiteboard is usually placed in a central place in the team room so that it is easy to immediately see the state of the project. It is also common to use a so called burndown chart to visualise the amount of work effort still remaining before delivery (see Figure 2.3.)

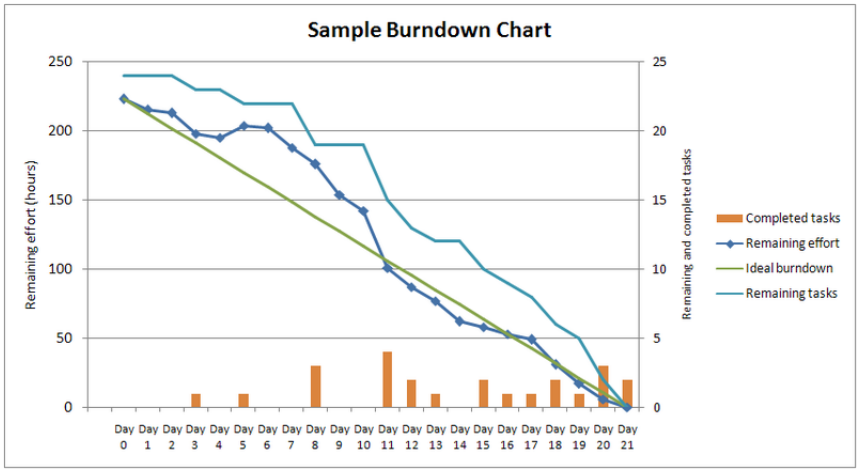


Figure 2.3: A sample burndown chart, fetched from [71].

Sprints are set up and closed by means of meetings. The sprint planning meeting is when the sprint backlog is selected. The sprint review is when the outcome of the sprint is discussed with the product owner. The retrospective is when the team evaluates their performance and tweaks their working methods after a sprint. A later addition to the meeting portfolio of Scrum is the product backlog refactoring meeting, which is organised during an ongoing sprint with the purpose of preparing the product backlog for the next sprint.

One of the great benefits of Scrum is that it is backed by a professional organisation with standardised training and certification. This makes it easier for companies to get started with Scrum. The main criticism against Scrum is that it does not easily scale up to the needs of large projects. The main challenge for larger projects is communication and synchronisation between the larger number of team members and teams, which may even be geographically dispersed. This is a known drawback and the focus of a large body of work. One of the better known is Leffingwell's work on scaling agile [49], which has evolved from his extensive work with companies such as Nokia. Later years have seen the emergence of the lean methodology for supporting the scaling of agile methods to larger contexts [4].

2.2 Lean and Kanban Methods

The lean methodology finds its roots in the Toyota Production System for car manufacturing [59] in the 1980s. During this time it was called just-in-time production or kanban. The term Lean Production was introduced in the book *The Machine That Changed the World* [76], which describes the superiority of the Toyota Production System when compared to the mass-production methods used in North America and Europe.

The Toyota Production System was developed as the answer to a need. The Japanese car industry did not have as large a market as their American counterparts. When American companies reduced cost by mass-producing cars, the Japanese had to find a way of cutting cost while producing fewer units of more varied models, meeting the varying needs of their customers.

The central idea of the Toyota Production system is to minimise waste. In manufacturing, one main source of waste is producing parts too early, since they then become inventory, which constitutes risk. The vision in the Toyota Production System is to manufacture the product when the order has been placed, that is, with no inventory. Two main principles are followed in order to achieve this. One is the principle of not manufacturing anything until it is needed (just-in-time manufacturing). The second is the principle of stopping the line as soon as any problems are detected and immediately rectifying them before work is allowed to continue. This is a clear step away from quality assurance through inspection and testing of the product as it comes off the manufacturing line. In lean manufacturing, quality assurance is built into the system and highly automated, but not fully automated, since a fully automated system is very expensive to build. Instead, some human interaction is required. The term autonomation (Japanese: *jidoko*) was invented for this concept.

The autonomation principle builds a culture where every worker is involved in improving the production line. The Japanese term for this type of continuous improvement is *kaizen*. Operations are optimised by continuous improvement, for instance, by optimising the limits of the queues in the manufacturing chain, but also by autonomation.

In practice, the kanban methodology is quite simple. In a manufacturing plant, the parts needed to assemble, for instance, a car are stored in shelves with token cards (kanbans) on them. When the number of a part runs below a predefined limit, the corresponding card is given to the station upstream that produces that particular part. This station will then proceed to manufacture the part. Any station will not produce new parts until they have an order through a kanban card to do so. This is sometimes called a pull mechanism for production. Instead of focusing on pushing new parts down the production chain, the parts are created on demand.

The lean production system is now widely adopted. The lean way of

thinking has also been successfully deployed in supply chain management and product development. It has proven to reduce the production and development time significantly. The lean methodology also allows for making changes later in the development process, which allows the manufacturer to better react to changing market needs.

2.2.1 Lean in Software Development

The success stories of lean manufacturing and product development have created an interest also in the field of software development. Mary and Tom Poppendieck adapted the lean principles to the software engineering industry in their books on lean software development [62, 63]. The Poppendiecks used lean principles as a means to promote agile principles in software development.

The concept of lean software development is not very clearly defined, and different definitions exist, depending on whether the definition is based on the Poppendiecks' work or on own interpretations of the Toyota Production System. Here, we refer to the Poppendiecks' definition when we talk about lean software development, and we also present an alternate definition in the following section, which we refer to as kanban software development. The Poppendiecks define seven principles of lean software development, listed below.

Eliminate waste. Recognise and minimise any activities that do not add value.

Build quality in. Avoid creating defects in the first place, by means of, for instance, early testing and continuous integration.

Create knowledge. Have a development process that encourages systematic learning, and that is systematically improved.

Defer commitment. Schedule irreversible decisions for the last responsible moment, leaving your options open as long as possible.

Deliver fast. Respond to the customers needs quickly and with high quality.

Respect people. The people doing the work should be empowered to do so. Process improvement should be performed by the people doing the work.

Optimise the whole. Make sure you understand the whole value stream and optimise the whole, not parts of it.

The Poppendiecks' also map the seven manufacturing wastes of the Toyota Production System to corresponding software development wastes. These are listed below together with their corresponding manufacturing wastes in brackets.

Partially done work (In-process inventory) Any work that has not yet resulted in deployable code is partially done work and should be minimised. Examples include documentation not yet translated to code, untested code, and undocumented code.

Extra features (Over-production) Do not develop any features that don't demonstrate a clear and present economic need.

Relearning (Extra processing) Capture knowledge in a way that is easy to reference later, to avoid having to rediscover things you already knew.

Transportation (Handoffs) Tacit knowledge is always lost at handoffs, so the number of handoffs should be reduced.

Task switching (Motion) Let the developers focus on their work. Switching between multiple tasks takes time.

Delays (Waiting) Avoid having people wait for decisions. Ensure that knowledge is available when it is needed.

Defects (Defects) Test driven development decreases the number of defects, and improves the design of the code.

Based on the seven principles and seven wastes, the Poppendiecks provide suggestions on how to improve the value stream of software companies by reducing waste.

2.2.2 Kanban in Software Development

Whereas the Poppendiecks use the lean philosophy to argue for agile development methods, Corey Ladas argues that the kanban methodology is superior to agile methods in general and Scrum [69] in particular in that it provides better support for the early phases of development. He presents experience of using kanban in software development [45].

The basic idea of kanban pull systems in manufacturing translates into the area of software engineering by treating requirements, user stories or tasks as the produced parts of a manufacturing chain. Minimising waste then becomes a matter of not performing tasks that will not end up in production code to be delivered to the customer. For example, the team should not analyse and design more user stories than can be immediately implemented. Chances are that the customer will change their mind or the

market situation will change before the "inventory" tasks can be implemented so that they either are not needed or need to be re-analysed. The initial analysis then becomes waste.

In practice, this can be implemented so that each activity of a software project (analysis, design, implementation, testing, refactoring, documenting, etc...) is considered to be a kanban queue with predefined upper and lower limits on the allowed number of items. When an upper limit is reached, no more items are allowed in that queue until one or more items are ready and moved to the next queue. When a lower limit is reached, items are pulled in from the previous queue.

Organising the work into different queues per activity enables the team to assign members with different skill-sets to different types of tasks. This allows for specialisation, which may be seen as a benefit but may also lead to the waste of transportation (handoffs) in the value chain.

Kanban systems in software development are often visualised using coloured sticky notes on white boards. See Figure 2.4 for an example of this practice. The team gathers around the white board once a day in the daily stand-up meeting [69] to survey and plan the ongoing work. One of the critical steps is the prioritization of new items to be pulled in. It is crucial to make the right prioritization based on the business value of the tasks [62].

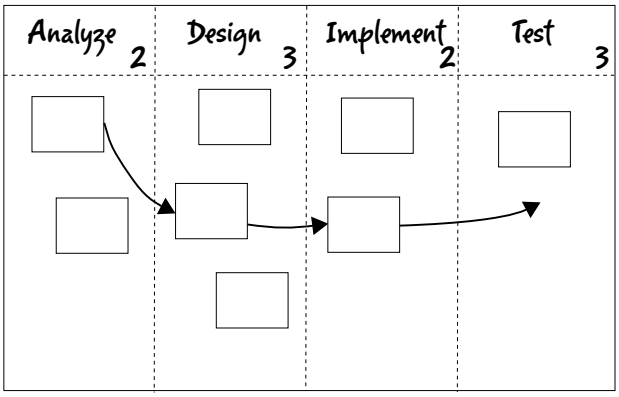


Figure 2.4: Example: a task is pulled into the testing queue.

The lean team is also expected to continuously improve their way of working. If they notice that tasks get backed up in one of the queues, they are expected to analyse the situation and improve the process to avoid that situation from reoccuring. This is the idea of stopping the line and kaizen in software development.

Usually, the team has access to other data besides just the notes on the white board. The sum of all the items on the white board is called the work in progress (WIP). The team may be presented with historical data on their

throughput of WIP and average delivery times for individual user stories. This information can be used to better understand the velocity of the team, to improve the management of the kanban queues and to recognise problems that may need kaizen events.

2.3 Collaborative Methods and Bottom-Up Process Improvement

The way of working in almost any software development project today is collaboration. The power of collaboration as a development method is undeniable with success stories such as the Wikipedia project and numerous examples of open-source projects. There is an ever-growing plethora of collaboration tool-sets striving to support large, geographically distributed projects in their effort to manage knowledge in a collaborative way. Microsoft Sharepoint Workspace [55] is one example of a general purpose collaborative tool-set. MediaWiki [75] – the platform on which Wikipedia is built – is another example. IBM Rational's Jazz platform [40] is an example of a collaboration platform targeted specifically at software development projects.

The technical issue to solve in order to enable collaborative working methods is how to build and share knowledge in a decentralised way. As an example, let's consider our specific case, as described in Paper III. In our case, the knowledge to share is the software engineering process at the company. We wanted to move away from the more traditional "ivory tower" approach to process engineering, where a dedicated team is responsible for defining and disseminating the process. Instead we wanted to involve the entire staff in evolving the process of the company. For this we used a general purpose tool (MediaWiki), which we found to be too generic in nature. We needed a tool that could capture the specific concepts of a process in a more powerful way. Paper III describes our work of collecting specific requirements for this tool and evaluating the tools available on the market.

2.4 Software Quality

One of the important targets for most software projects is to build functioning software with sufficient quality. Software quality is a concept that at first thought seems simple enough. Intuitively, we have a picture of what distinguishes good software from bad software. Software of good quality has few defects and provides a comfortable and reliable user experience. But if we take a second look at the concept of software quality, through the eyes of the software developer, we notice that there is more to the concept than just the user experience.

A developer intuitively knows that the design and code of the software may be of different quality. Well designed software and clearly written code may provide the same user experience and have as few bugs as poorly designed and unclearly written code, but the design and code have an impact on the developer's daily work. Good quality software is easier to work with when correcting defects and adding new features. This is not a pure developer aspect of software quality, however, since it will indirectly affect the user experience through the software's ability to evolve in order to meet future needs of the user. Good quality software may be good also when the next version is released, while bad quality software most likely will deteriorate from one version to the next.

These two main perspectives to software quality: the user experience perspective and the developer perspective, can be further elaborated into a set of standard quality criteria for software quality. The ISO 9126 standard for software engineering product quality [42] does just that. The standard defines software product quality in the terms of six quality criteria: functionality, reliability, efficiency, usability, maintainability and portability. The first four are associated with the user experience, while the last two are associated with the evolution of the software product. In our research we have focused on the evolution aspect of software and more specifically on the maintainability attribute (see Figure 2.5).

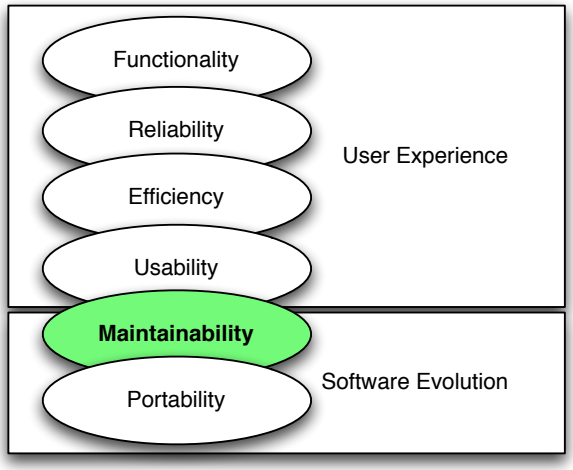


Figure 2.5: The quality criteria of ISO 9126 mapped to the two perspectives of software quality.

2.4.1 Software Metrics

As software quality has an impact on the user experience, it will naturally also impact the business success of the software. If comparable software products are available on the market, the potential customers will look at the software quality as one of the factors impacting the decision to buy. If a piece of software is known to be defective or difficult to use, this will probably be seen as a disadvantage by a potential customer. This is a risk regardless of whether the customer is an end user or another company interested in buying components or services. For this reason, it makes sense to try to measure software quality in order to know how a product will do compared to the competition. Let us take a look at the characteristics defined by the ISO 9126 standard and how they can be measured.

Functionality, reliability and efficiency are most easily measured by testing. Experienced testers can systematically use the software to ensure that the software meets the functional requirements, is reliable and, by looking at the time and resource consumption, is efficient. By observing the software over time, metrics in the form of number of defects, uptime and resource usage can be collected and trends can be analysed.

Usability is more difficult to measure and software companies that have high usability as a priority may use different techniques for assessing usability by, e.g., using satisfaction questionnaires or tracking the eye-movement of test users. Usability metrics may consist of data such as the time spent to complete a task or ratings from usability surveys.

Maintainability and portability can also be measured in retrospect. Maintainability may be measured by looking at defect trends. If we have a growing trend in the number of defects, the maintainability of the software is probably poor. Another interesting maintainability metric is the turn-around time for a defect or a new feature. If defects usually take a long time to correct, then the code is probably not easy to understand and change, and maintainability is probably poor. Portability can be measured in a similar fashion: by measuring how long it takes to port the software from one context to another and how this affects the defect rates.

By collecting metrics such as the examples above, a company can assess its product's quality and compare their product to others on the market. However, this is usually not enough. To have real use for the metrics collected, the company will want to be able to act on them. If the maintainability of a product is poor, the company will want to understand why this is the case and how to improve it. But when the product is ready enough for us to measure its maintainability in terms of defect trends and turn-around times, it is usually too late to act. We want a metric that can be collected and followed up on during the development of the product.

The examples of metrics given above can be characterised [26] as exter-

nal product metrics. They measure characteristics of the product from an external viewpoint, observing the product interacting with its environment. In order to get earlier indications of the quality of the software product, one can collect internal product metrics – looking at the internal workings of the product, more specifically the code and the design itself.

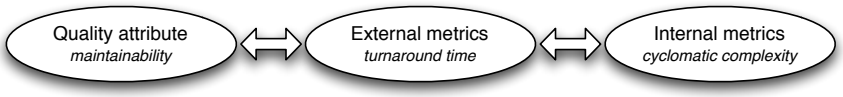


Figure 2.6: Mapping quality attributes to external and internal metrics. Italics denote examples

How can we by looking at the code assess, e.g., how expensive it will be to maintain in the future? As we saw above, we can estimate the cost by looking at external metrics such as defect rates and turn-around times. But what in the code will tell us how many defects will be found in that piece of code, or what the turn-around time for a defect or a new feature will be when extending or correcting that piece of code? The challenge lies in connecting the internal product metrics to the external ones. Figure 2.6 illustrates this mapping from internal metrics to external metrics that quantify the observed quality attribute. The figure also includes examples of each of the concepts. In this example, the quality attribute maintainability is measured through the external attribute turnaround time of defect reports, which in turn is predicted by measuring the cyclomatic complexity of the code.

Deducing the status of a quality attribute based on internal product metrics is a challenge, because the code is the end product of a complex process starting as ideas in the customer's mind. Figure 2.7 illustrates this challenge. The quality of the code depends on at least the following factors:

- How well the developers understood the customer's ideas.
- How well the developers understood the non-functional requirements of the product.
- The developers' proficiency in the used technology (tools, languages, platform.)
- The developers' proficiency in relevant practices, patterns, idioms and standards.
- The developers' proficiency in the problem domain.
- The ability of the developers to predict future needs of the product.

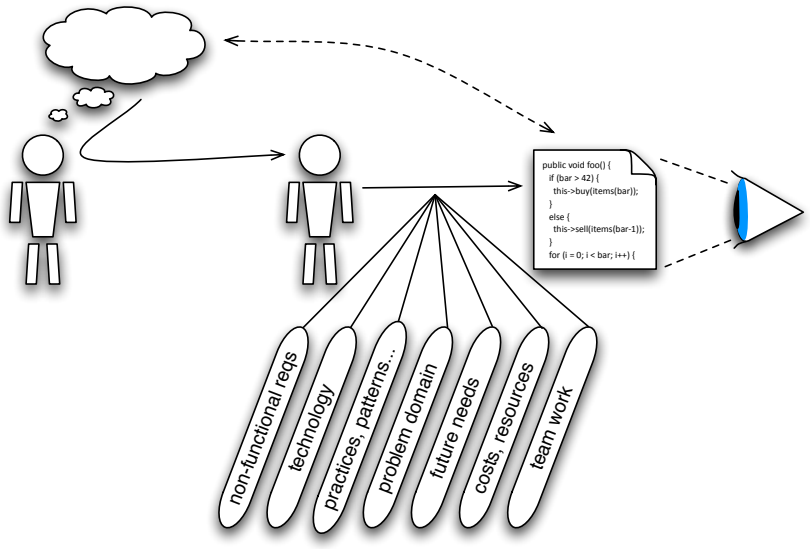


Figure 2.7: Assessing the quality by looking at the code

- Cost and/or resource constraints limiting the effort spent developing the product.
- How well the development team works together to achieve its goals.

The design and code is the end result of this complex process, but a lot of information is lost during the process as to what the reason for possible quality issues are. When we look at the end product, that is, the code and documentation to assess, for instance, the maintainability, we are missing that information, as illustrated by Figure 2.7. Internal product metrics are invariably simplifications and the connection to external metrics is an assumption, based on historical data in the best case, or on intuition in the worst case. For this reason, developers are at risk of distrusting the validity of the metrics collected [74] and simply ignoring them, or even worse: trying to adjust the metrics to make the team look better.

The problem here stems not only from the metrics themselves, but also from the way the metrics are used. If the metrics are used to compare the performance of teams or units, the risk of being penalised for bad performance may overshadow the opportunity of being rewarded for good performance. The team will feel threatened and the metric collection effort will have an adverse effect on the project [37, 74]. This type of metrics is called *performance measurements* by Austin [8] and is contrasted by *information measurements*, which are used to gain transparency into the performance of

the product and process in order to provide the team with feedback that they can use to learn and grow.

It is interesting to note that the two types of measurements, as defined by Austin, correlate nicely with the often stated difference in values between traditional and agile methods. Whereas agile methods value ways of working that support and empower the team (such as information measurements), traditional methods include ways of working that command and control the team (such as performance measurements).

In our research, the quality attribute addressed is mainly the maintainability of large software products. We look into the above mentioned two problem areas of quality metrics: justifying the validity of the internal metrics for measuring specific external metrics; and using the collected metrics in a way that supports the team.

2.5 Action Research

The research work presented in this thesis can be classified as the result of action research. We worked in a specific context not just observing and analysing it, but also trying to change and improve it. Action research is considered to be fairly immature, especially in the field of Software Engineering [25] and there is still no consensus on what constitutes a valid and appropriate methodology for action research.

The term action research was introduced in the 1940s and is credited to Kurt Lewin [50], a psychologist concerned with raising the self-esteem of minority groups in the aftermath of World War II. He defined the action research approach as one where the researcher generates new knowledge about a social system while at the same time trying to change it.

One example of the work Lewin and his students did was an experiment in in a local factory demonstrating that democratic workplaces promote better efficiency and morale than autocratic ones [1]. This illustrates the nature of action research as being highly involved in the practical context of the researched organisation, trying to help it reach certain goals by evaluating, changing and learning from the change.

Figure 2.8 depicts the main steps of the cycle that constitutes action research, as described by Lewin [50]. In the first step, data about the current state is collected and analysed. The situation is diagnosed and an action plan is made. This step is called the unfreezing step. The second step is when the actual change takes place. This step involves action planning, performing the action as well as learning. The third step is called refreezing. This step involves collecting facts about the action through measurement and reflective learning. The process is iterative, with feedback from each of the steps to the previous ones.

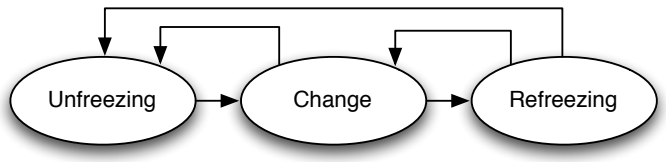


Figure 2.8: Action research.

Action research is a research method especially well suited for research in the field because of the way that it combines theory and practice [10]. The theory is explored in a practical context, thus gathering experience so that the theory can be revised. There is a feedback loop and emphasis on investigating and changing what is really done in the organisation.

It may be interesting to note, however, that there are different schools of

thought in action research today and, as Morten Levin points out [29], there is a difference between action research in the US and in Europe. The main difference is, according to Levin, that action research in the US has "degenerated into positivist experimentation" where observation of the relationship between the researcher and the researched has been left out.

Susman and Evered [72], among others, have refined Lewin's cycle, describing a loop of five steps. Figure 2.9 describes these five steps. Depending on the level of cooperation between the researcher and the organisation, there are different categories of action research, ranging from diagnostic action research, where the researcher only provides the organisation with data collection for diagnosis, to experimental action research, where there is collaboration in all phases.

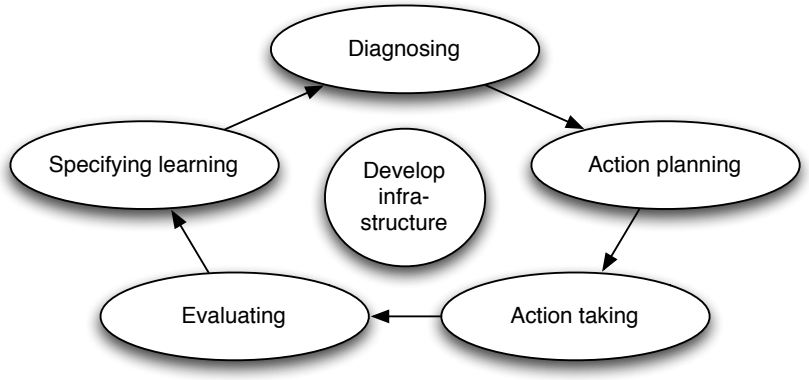


Figure 2.9: Action research, steps according to Susman and Evered.

These four categories of action research are:

1 Diagnostic action research: researcher collects data for diagnosis and feeds data back to the system.

2 Empirical action research: researcher evaluates actions and feeds information back.

3 Participant action research: diagnosing and action planning in cooperation with the organisation.

4 Experimental action research: collaboration in all phases.

According to this definition, our research falls into the fourth category: experimental action research.

Although Susman and Evered [72] point out that action research does not fully comply with the criteria established for positivist science, they still

argue that action research can generate valuable knowledge in organisational science. They even argue that action research has more potential than positivist science when it comes to organisational research, because of its nature of being more involved in the needs and goals of the organisation.

Kock [44] argues that the dichotomy between positivism and action research is a fallacy, since the two are not comparable. He argues that action research is a research approach, whereas positivism is epistemology. In his view, action research can benefit from positivist research approaches.

Kock defines three threats to action research: uncontrollability, contingency and subjectivity [44].

The uncontrollability threat stems from the tight cooperation between research and industry. There will naturally be a division between the research goals and organisational problem solving goals. Thus, there is always a risk that changes in the organisation will result in the research venture being abandoned before it is completed.

The contingency threat is a risk originating in the large amounts of data possibly available to a researcher. The data will typically be broad but shallow and it may be difficult to isolate the individual factors that affect the outcome of the research.

The subjectivity threat occurs because of the involvement of the researchers in the client organisation. Kock provides an example of the researcher being offended by one of the managers, and for this reason inclining to data interpretation that puts this manager's methods in an unfavourable light.

Kock also proposes three antidotes for these three risks. The first antidote is related to the unit of analysis. By identifying the unit of analysis at the beginning of the research project and studying as many instances of the unit of analysis as possible, the external validity can be better ascertained. This helps mitigate all three risks, according to Kock.

The second antidote prescribed by Kock is to use grounded theory for analysis of large bodies of data, using open, axial and selective coding.

The third antidote proposed to alleviate mainly the uncontrollability threat is to use multiple iterations of the improvement loop. This is proposed for the same reason as in agile methods: frequent partial deliveries will make sure that the researcher does not leave completely empty-handed if a project is cut short.

Lau [48] addresses the issue of validity of action research in the field of Information Systems. He lists the following criteria to consider when conducting and evaluating action research:

1 Is the research aim authentic and practical?

2 Is the perspective explicit and authentic according to the adopted stance?

3 Is the adopted action research stream consistently described?

4 Are the theoretical assumptions authentic?

5 Is there sufficient background information?

6 Is the intended change appropriate and adequate?

7 Is the involvement of the research site appropriate and adequate?

8 Are the participants authentic?

9 Are the data credible, dependable and confirmable?

10 Is the study duration adequate for change to occur?

11 Is the degree of openness appropriate and adequate?

12 Is the access/exit point appropriate and adequate?

13 Does the presentation style provide sufficient information?

14 Is the problem practical and authentic?

15 Are the interventions authentic, appropriate and effective?

16 Is the reflective learning trustworthy?

17 Is the iterative process used appropriate for learning?

18 Are there general lessons learned that contribute to new knowledge?

19 Is the researcher role appropriate and effective?

20 Is the participant role appropriate and effective?

21 What competency improvement is planned and has it improved?

22 Are ethical issues addressed satisfactorily?

This concludes the background chapter of this thesis. The key concepts presented here are used in the next chapter, which gives a detailed overview of the research work supporting this thesis. The four first sections of this chapter correspond to the three main research themes of the next chapter: agile and lean deployment (Sections 2.1 and 2.2); product metrics for decision support on maintainability (Section 2.4); and tool support for process knowledge sharing (Section 2.3). Lewin's action research process along with Kock's three threats and Lau's validity criteria (Section 2.5) are used to explain the learnings from each of the research papers.

Chapter 3

Overview of the Research

In this chapter, a detailed overview of the research is presented. Section 3.1 starts by placing the research in context, introducing both the environment in which the research was performed as well as the timeline on which it was executed. Section 3.2 introduces the areas of contribution of the research, detailing the research questions. Section 3.3 gives an overview of the used research and data collection methods. Then follows one section for each of the three contribution areas, where each of the papers in that corresponding area is described in detail (Sections 3.4 – 3.6). Finally the validity of our research within the action research framework is accounted for in Section 3.7.

3.1 Research Setting

The research for this dissertation was mainly executed in an industrial setting. As such, the original research question was a pragmatic one: how do we improve the software development practices so as to improve the performance of the software development organisation and the quality of the software product. The industrial setting is also the main reason behind the selection of the overall research framework: action research. The researcher was part of the studied environment, not only observing but also attempting to change it.

The setting was one software development department of a large telecommunications corporation (Ericsson) and a fairly large, international software company in the areas of wireless telecommunication and automotive solutions (EB).

Ericsson is the largest supplier of mobile systems in the world. At the time, Ericsson's customers included the world's 10 largest mobile operators and some 40% of all mobile calls were made through its systems. This international telecommunications company has been active worldwide since 1876 and is currently present in more than 140 countries.

EB, Elektrobit Corporation, is a large, geographically distributed company specialised in demanding embedded software and hardware solutions for the wireless and automotive industries. The number of employees was just under 2000 at the time, with development activities mainly in Finland, Germany, Austria, Switzerland, China and the USA.

The timeline in Figure 3.1 shows how the main research areas were distributed timewise. The role that the author was appointed in 2003 as Software Design Architect, allowed for some more long term process development work to be initiated. Eventually, this lead to the most of the research presented here.

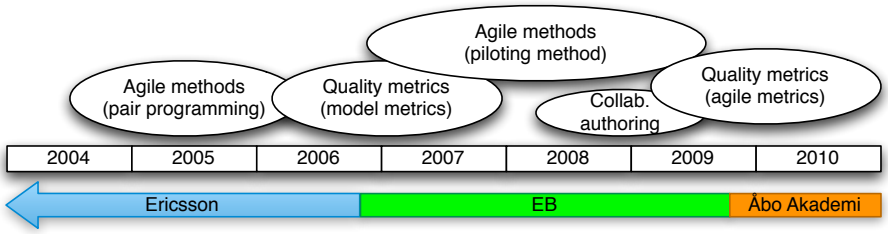


Figure 3.1: Timeline of the research.

Doing research in an industrial setting is in some ways a two-edged sword. On the one hand, you are in the midst of the environment you want to study and affect. There is ample access to real, live data from actual software projects. The problems and questions that industry projects struggle with are at your fingertips. On the other hand, you have to respect the economic reality of the projects. The research questions that you are allowed to study are limited by the business case they represent. You have to be able to give clear, business related arguments for the research in terms of return on investment. In order to get approval for a research project, there has to be management support and clear risk management so that the real targets of the company – delivering high-quality products to the customer on time – are not endangered.

We were fortunate enough to have good management support and, especially in the case of Ericsson, an organisation with high maturity and a process culture where metrics collection was standard protocol. This means we could perform our research and collect data without the need for disrupting the day-to-day business to any large extent.

The first study we did was in the area of agile methods. The business case for this was to improve motivation among the software staff using the XP practice of pair programming, and some other supporting practices. This resulted in a technical report and a conference paper. These are presented

in more detail in Section 3.4.1.

The results of this first study was successful, but had a limited impact on the company. We wrote guidelines based on our findings but there was no champion promoting the way of working. Even though the department management saw clear benefits of the agile way of working, there was no active push from management either. The practices used in the study were allowed and recommended, but not actively promoted.

The result did, however, raise some interest in the corporate level process development department. As a result, we presented our research also in internal conferences and helped sow the first seeds of agile thinking in the company.

The second study was in the area of model metrics. Here, the business case was to reduce the cost and effort of maintenance of the software products. The study covered a large embedded software product, developed using both model driven and more traditional techniques. As the model driven area is less explored in these settings, we focused our efforts on that part of the problem. The findings resulted in a conference paper and a journal paper. These are presented in more detail in Section 3.5.1.

The results of the metrics study resulted in much interest in the company. Model driven development is used in many areas of the company but there was a lack of integrated support for metrics in the tools used. The limiting factor here was a practical one. The Turku unit, where the research had been performed, was sold to another company. So we never had the opportunity to follow up on the research. Some efforts were still done, but not being part of the company any longer raised barriers in the practical work.

In the new company, the author was fortunate enough to have a position in the central process development organisation with the task of deploying agile methods in the company. Even though this provided a unique opportunity to study the impacts of agile methods, the economic reality of a smaller company is even more restricting when performing research. In a smaller company, the margins are smaller and the stakes are higher when interrupting projects with new methods. The company was a collection of smaller companies that had recently merged to a larger company, so the company culture was heterogeneous. The maturity level varied and not all parts of the company were used to collecting metrics.

One of the main challenges we faced when trying to deploy agile methods was disbelief and mistrust. We needed to build trust in the methods. This resulted in two different research areas. The first is related to the setting up and execution of pilots. We needed to create and communicate success stories to win people's trust. As the company was heterogeneous, we needed to be able to run several pilots in different areas, to show that the method works not only in certain areas. This resulted in a conference paper that details a pilot method with two case examples, which is described in Section 3.4.2.

The second resulting research area was concerned with supporting the build-up of a common company culture through knowledge sharing. For this, we did a survey of process authoring tools to see find what tools best support collaborative process authoring. This also resulted in a conference paper that describes the outcome of the survey and gives suggestions on how the currently available tool can be further developed to better meet these needs. This paper is described in Section 3.6.1.

The final research area outlined in Figure 3.1 is the area of agile metrics. This ties together the two main areas of agile methods and model metrics. This resulted in two papers, which were written in an academic setting and is outlined in Sections 3.5.2 and 3.5.3.

3.2 Research Questions

The contribution of this thesis lies in the general area of process improvement. More specifically, we address the issue of deploying agile and lean methods in large, mature organisations with the support of relevant product metrics and collaborative process authoring. For clarity, the contribution can be divided into three different areas: agile deployment, product metrics for decision support and process authoring. In each of the areas, we address a number of research questions. The contributions and the associated research questions are listed below and Table 3.1 shows how each of the papers addresses the questions.

CI Research related to the deployment of agile methods in large companies.

- **Q1** Can corporations with well established and rigid, plan-driven processes use just a few agile methods and still see significant benefits?
- **Q2** How does one deploy agile methods in large, diverse organisations?
- **Q3** How does one systematically implement agile pilots as part of the overall deployment process in a plan-driven environment?

CII Research aimed at supporting teams with relevant product metrics for decision support on maintainability and refactoring.

- **Q4** How do different statechart concepts affect the maintainability of a system?
- **Q5** How does one assess the maintainability of code under development in order to provide decision support for refactoring decisions?

Q6 How does one support decision making in lean and agile teams with code-based metrics in a manner that does not require a large, up-front metrics collection effort from the team?

CIII Research related to tool support for process knowledge sharing.

Q7 How well do existing process authoring tools support the needs of the industry when it comes to collaboratively sharing process knowledge?

3.3 Research Methods and Data Collection

The overall research framework was implicitly chosen by the setting of the research. As the researcher was a part of the researched organisations, the research is most naturally classified as action research. Each of the research projects were performed using the main steps of the action research process, as described in Section 2.5, and each project can be seen as one iteration of the main loop of Figure 2.8 on page 20 (also outlined in more detail in Figure 2.9 on page 21).

In our research, we mainly used empirical research methods. The only exception is the paper that suggests kanban guards as a method for decision support in agile and lean teams, where the approach is clearly constructive (design) research [38]. The methods used are listed below. The terminology used is based on [25]. The abbreviations introduced here are used in Table 3.1.

CCS Case study; confirmatory single case

CEM Case study; exploratory multiple case

MSE Mixed methods: sequential exploratory

CE Controlled experiment

SR Survey research

CR Constructive research

As the research was mainly empirical and conducted in the field, the collection of data was essential. We used a number of data collection methods, as we collected both quantitative and qualitative data. The following data collection methods were used (based on [70], except for the tool survey.) The abbreviations introduced here are used in Table 3.1.

D1 Brainstorming and focus groups

D2 Interviews

D3 Questionnaires

D5 Work diaries

I1 Analysis of electronic databases of work performed

I4 Static and dynamic analysis

TS Tool survey

Table 3.1 gives an overview of the publications. The numbering refers to the numbering in the initial publication list of this dissertation. For each publication, its topic, contribution, research question, research method and data collection methods are listed. The following sections give a more detailed summary of each of the publications.

#	Topic	Question	Method	Data Collection
I	agile practices	CI:Q1	CCS	Quant: D5, I1 Qual: D2, D3
II	maintenance metrics	CII:Q4	MSE & CE	Qual: D1 Quant: I1, I4
III	collaborative authoring	CIII:Q7	SR	Qual: D1, TS
IV	kanban guards	CII:Q6	CR	
V	agile piloting	CI:Q2, Q3	CEM	Qual: D1, D2
VI	maintainability index	CII:Q5	MSE	Qual: D1 Quant: I1

Table 3.1: The Papers at a Glance

We now proceed to describe the papers that make up this thesis. The papers are organised by topic rather than time. Section 3.4 describes the papers in the area of deploying agile methods, while Section 3.5 describes the papers in the area of quality metrics. Section 3.6 describes the work done in collaborative process authoring.

The papers are described using the structured abstract format suggested by [43], but with the Objective section split into two: one for the research question and one for the business goal that the organisation hoped to achieve. By this split, we make it easier for the reader to distinguish between the two, since they are not always the same.

All papers but one present results from action research initiatives. For these papers we also include sections outlining the conformance to the action research framework presented in Section 2.5 along with the lessons learned in the research project. A last addition is a section clarifying the author's role in each of the projects.

3.4 Agile Deployment Papers

3.4.1 Publication I

Title Software process improvement with agile practices in a large telecom company (2006) [9].

Business goal Increase the motivation of the developers, while maintaining the efficiency and quality of the existing way of working. Increase competence within a team.

Background Besides the promise of rapid and efficient software development [41, 65, 68], agile methods are well-appreciated for boosting the communication [19] and motivation [54] of development teams. However, they are not practical without tailoring in large organisations [14, 51], especially because of the well-established, rigid processes in the organisations.

Research question Q1: Can corporations with well established and rigid, plan-driven processes use just a few agile methods and still see significant benefits?

Method We conducted a confirmatory, single case study with a team of four subjects. Data was collected through work diaries of hours spent on different categories of tasks, analysis of electronic databases (defect reports), interviews and questionnaires.

Results There was no measurable change in motivation, but the competence build-up was clearly measurable. Efficiency and quality were not compromised.

Limitations Generalisation of the results is limited by the characteristics of the team: the size (four people), the temporary nature of the team arrangements and the focus on competence build-up. More detailed time monitoring than usual may have lead to changed behaviour in the subjects.

Conclusion We demonstrated that is worthwhile to use pair programming, the planning game and collective code ownership. The used agile methods could be refined to suit the existing settings of a large company.

Conformance to research framework The unfreezing and change steps of Lewin's action research framework [50] were performed in a controlled manner. The refreezing step was not done properly, however. As a result, the findings were not as widely used as we would have wished.

Lessons learned Focus needs to be paid to the refreezing phase. Assign a sponsor that markets the results in the projects and supports wider adoption. Pilots is a good way of building awareness around a new concept, such as agile methods. There was a clear change in the way agile methods were perceived. Before the pilot, agile methods were seen as completely unsuitable for the company, but after the pilot, there was a whole new openness.

Future work One interesting topic to investigate in the future is the formalisation of task allocation and how it can be used to optimise different factors such as competence build-up or lead time. This can take into account attributes of the tasks, such as complexity and estimated completion time, as well as attributes of the developers, such as competence and experience.

Author's role The author, Jussi Auvinen and Rasmus Back constituted the company part of the research team and contributed in the definition of the research question and methods, as well as in the execution of the research project and the concluding analysis and dissemination. The author also acted as the industry supervisor for Back's master's thesis.

3.4.2 Publication V

Title Systematic piloting of agile methods in the large: Two cases in embedded systems development (2010) [34].

Business goal Generate success stories of agile deployment in order to build experience and to overcome the most common challenges of agile deployment, such as resistance to change.

Background Despite reportedly high adoption rates of agile methods [3], deploying agile methods in a large, diverse, geographically distributed setting is still a challenging task [51]. Some attempts have been made to tackle this deployment problem [61, 49, 4] but little is available to provide clear methods and advice with regard to deploying agile methods and principles. Current research reports do not concentrate on what often lies at the very beginning of a software process improvement initiative deploying agile methods: a pilot project.

Research question Q2: How does one deploy agile methods in large, diverse organisations? Q3: How does one systematically implement agile pilots as part of the overall deployment process in a plan-driven environment?

Method We conducted an exploratory multiple-case study on two live projects in a large, geographically distributed, embedded systems company.

Results A method for piloting agile in a large corporation is defined. The method is proven successful in the goals of (1) overcoming resistance to change and (2) ensuring that a pilot project can run agile even if the rest of the organisation is non-agile.

Limitations It is too early to evaluate the long-term implications of the method for company-wide deployment. The generalisation to other companies should be studied further.

Conclusion Piloting agile methods according to the defined method proved a successful way of overcoming resistance to change.

Conformance to research framework We saw the materialisation of the uncontrollability risk, as defined by Kock [44]. A major restructuring of the company and the erstwhile resignation of the main sponsor of the project (the company's CEO) cut the overall project short after the first iteration. We had mitigated this risk by working in short iterations, and this work is the outcome of the first main iteration.

Lessons learned Always work in very short iterations and make sure there is outcome defined for each of the iterations. Again we could see that piloting helps in overcoming fears and changing perceptions.

Future work Future work mainly concerns the limitations of the study and includes further validation and refinement of the method as it is further applied to pilots within the company as well as in other companies struggling with the challenge of agile piloting in the large. Furthermore, the long-term impacts of piloting on the general deployment of agile methods still need to be evaluated.

Author's role The author was the project manager for the research project in her role as project manager for the company-wide SEP (Software Engineering Process) team. As such, the author was involved in all parts of the research undertaking, including planning, execution, analysis and dissemination. The data collection through interviews was done by external researchers (Hirkman and Pikkarainen), though.

3.4.3 Research Contributions

The challenge of deploying agile methods in large companies was addressed in two different studies with different deployment tactics. The first study introduced a small set of agile practices into one team in a non-agile project. The main goal of this first study was to improve motivation amongst the developers. This is in contrast with the second study, where the way of working of two entire projects where changed with the main goal of overcoming resistance to change.

Interestingly enough, the first study did not result in measurable changes in motivation. Instead, we could see improvements in competence build-up and quality. In the second study, we did see an improvement in motivation through a reduction of resistance to change. On the other hand, we could also see that motivational factors external to the project, such as a feeling of insecurity due to the overall financial situation, could not be affected by the changed way of working.

It is an important observation to make that process improvement cannot make up for motivation lost due to a feeling of personal insecurity. An example illustrating this can be taken from the keynote of the SPICE conference in 2009, Riku Granat (Vice President, Nokia Application Software) asked for a process to simplify the knowledge transfer from high-cost countries, such as Finland, to low-cost countries, such as China. This is not something we consider should be expected of a software process improvement initiative. No software process can motivate a developer to work more efficiently at making themselves redundant.

The conclusion that can be drawn from our studies is that piloting is a good way of introducing agile methods into a large company. There is the two-fold benefit of learning by doing while at the same time reducing the resistance to change. The long-term benefits of the first study can be seen in the adoption of agile methods in the company in question [60]. The long-term benefits of the second study remain to be seen.

3.5 Quality Metrics Papers

3.5.1 Publication II

Title Statechart features and pre-release maintenance defects (2008) [35].

Business goal Assess the maintainability of statecharts in order to reduce the cost of maintenance and further development of systems built using this technology.

Background Statecharts is a design notation for reactive systems commonly used in the automotive and telecommunication software industry. Statecharts, according to both the original presentation by Harel [31] and the more recent Unified Modeling Language (UML) specifications [57] comprise a large set of different modelling concepts. These make the statecharts more expressive and presumably easier to understand. We have observed, however, that some of these concepts can conversely make the system more difficult to understand. Understandability and changeability are two important factors of maintainability [42]. Systems that are difficult to understand and change are

also difficult to maintain. Although there is a large body of work addressing ways of measuring the maintainability of software systems [21, 11, 52, 16], it is not clear how this translates to the model driven paradigm using statecharts. Some related work exists [6, 47]. Genero et al. [28] come the closest to our needs in that they evaluate statechart metrics both theoretically and empirically, but they work in an academic environment with no correlation to defect rates of real software systems.

Research question Q4: How do different statechart concepts affect the maintainability of a system?

Method We used a mixed-method sequential exploratory approach, starting with workshops with domain experts and then performing a static analysis and analysis of electronic databases (defect reports, version control system). We concluded with a comparative study involving subjects answering questions on readability and changeability of two example systems designed to display good and bad design idioms respectively.

Results We see correlations over 0.8 between defect rates and these statechart constructs: the number of transitions, choice points and capsule operations, the ratio of choice points per states, the average visual cyclomatic complexity, average number of defer and recall commands, average capsule size. The paired sample t-test of the comparative study points to a clear difference (sig-value less than 0.1) in the points ratio for understandability and the difficulty rating for changeability.

Limitations This study is performed in a specific setting of large, embedded telecommunications systems using model-driven development. Generalisation to other contexts may not follow.

Conformance to research framework We saw the materialisation of the uncontrollability risk, as defined by Kock [44]. We had learned from our previous attempts that the refreezing step of Lewin's framework [50] should be carefully considered. However, part of the company was sold to another company as the project was about to move into the refreezing phase. Unfortunately, all the research staff was then outsourced, and had no influence over the project any longer.

Lessons learned A thorough study such as this helped put the concept of refactoring in a financial context. We could clearly see a change in perception of refactoring from "if it ain't broke, don't fix it" to an efficient and necessary practice for maintaining code quality.

Conclusion We present evidence that suggests that the use of different statechart features during the design of a subsystem may affect the number of defects introduced in posterior maintenance actions. We also conjecture about possible causes for this and we provide recommendations on how to design statecharts in a way that is less likely to result in defects.

Future work In order to generalise the results, a series of controlled experiments with public materials should be carried out and replicated in different sites. It would also be beneficial to factor out different design idioms and statechart features in order to study them individually. Furthermore, some features of the used modelling toolset were intentionally excluded from this study. Inheritance of classes and capsules is one such feature which we would like to study further. Orthogonal regions is another feature which we would like to study, but could not since the toolset did not support it. Another important future work is to develop a complete predictor model for defects in maintenance actions for statecharts.

Author's role The author together with Jussi Auvinen and Andreas Nåls constituted the industrial part of the research team, performing the planning and execution of the research. She also held the main responsibility for the analysis and reporting phase and acted as industry supervisor for Nåls' master's thesis.

3.5.2 Publication IV

Title Metrics functions for kanban guards (2010) [36].

Business goal Improve work prioritization.

Background Agile and lean approaches favour self-organising teams that use low-tech solutions for communicating and negotiating project content and scope in software projects [18]. We consider this approach to have many benefits, but we also recognise that there is information in software projects that does not readily lend itself to low-tech types of visualisation. Different characteristics of the code base is one such example. Good solutions for collecting metrics on the code base exist [58, 39] but they are not integrated with the standard way of working of lean and agile teams.

Research question Q6: How does one support decision making in lean and agile teams with code-based metrics in a manner that does not require a large, up-front metrics collection effort from the team?

Method We use a constructive approach in devising a method for supporting lean teams with metrics.

Results We outline metrics functions, called kanban guards, which consist of three parts and can take advantage of more advanced information of the development artifacts.

Limitations The constructed method has not been validated in real projects.

Conformance to research framework This report is not action research.

Conclusion We demonstrate that kanban guards can provide lean and agile teams with decision support in order to optimise the quality of the software product, while at the same time reducing waste.

Future work We would like to see a series of controlled experiments implementing the kanban guard concept in lean software projects in order to measure its real impact on the quality and performance of the project. This implementation would include the integration of the complete tool chain, but also the adoption of lean working practices and their interaction with the kanban guard concept.

Author's role The author was the main author of the paper.

3.5.3 Publication VI

Title Maintainability index for decision support on refactoring (2010) [33]

Business goal Facilitate communication between technical and business stake-holders regarding refactoring decisions.

Background Maintainability is a software attribute that needs to be continuously addressed during the entire development life-cycle. Failure to do so will result in deteriorating software quality and the build-up of so called technical debt [23]. The state of the practice for fighting technical debt is the agile practice of refactoring [27]. In a large corporation the business expertise and technical expertise is usually represented by different people, often in different organisations, working towards different goals and using different vocabularies. A lack of trust between the two is not uncommon. Furthermore, there may not exist consensus among the technical staff as to what constitutes good software design. For these reasons, reaching a decision to refactor may not be a trivial task. There is a large body of work dealing with the detection of technical debt for the purpose of improving maintainability [15, 53, 52, 11, 21, 56, 7]. The main difference between the existing studies and our needs is the fact that most studies are performed in

a small setting with small example software systems and students as subjects, whereas our study was performed in an industry setting with real systems and experienced software industry professionals as subjects. Coleman et al. [21] and Asthana et al. [7] are two exceptions. The main difference between our work and theirs is that our focus lies more on early diagnostics and actionable product metrics, whereas their approach partly relies on after-the-fact metrics such as effort, in the case of Coleman and quantitative process metrics in the case of Asthana. None of the referenced work addresses the issue of building consensus around maintainability metrics, which is one of our areas of interest.

Research question Q5: How does one assess the maintainability of code under development in order to provide decision support for refactoring decisions?

Method We use a mixed-method sequential exploratory approach starting with workshops collecting the intuition of experts and continuing by static analysis and analysis of electronic databases (defect reports, version control system).

Results The constructed maintainability index corresponds well to the intuition of the experts. It is also successfully validated against a refactoring effort.

Limitations This work was performed in the context of a large software system in the telecommunications industry and with the help of experts from the industry. The studied subsystems were written using IBM Rational Rose RealTime and C++.

Conformance to research framework This report is based on the same research as Publication II, so the risks are the same.

Lessons learned In an industrial setting it may be more difficult for the domain experts to reach consensus on what constitutes a smell and how to measure it by voting, since the most experienced experts are most likely to have some attachment to the system, either by having been involved in the development of the system or at least by knowing they will be involved in it in the future. This type of approach helps build consensus by showing the experts how their intuition corresponds to metrics.

Conclusion The suggested approach to assess the maintainability of software systems and their need for refactoring based on the collection of internal product metrics is validated in the context of a large, mature telecommunications product.

Future work A long-term evaluation of the impact on software maintainability of methods such as the one defined here should be studied, including the evolution of the model itself. The process paradigm in the context of this study was an iterative, incremental one. It would be interesting to see the impact of this method in other contexts.

Author's role See Publication II.

3.5.4 Research Contributions

The challenge of supporting agile teams with relevant product metrics for decision support was addressed in three studies. The first study dealt with the question of how to measure design quality. The second and third studies dealt with different aspects of the question of how to incorporate metrics in the work methods of the team.

The result of the first study showed us that it is possible to use statistical methods in order to find what constitutes good and less good design. The specific idioms identified are quite contextual for the programming paradigm and problem domain in question, though.

In order for metrics to be of support for the team, they need to be integrated in the way of working, the tool sets used and the minds of the developers. The second study constructs a way of integrating metric functions, which we call kanban guards, with the lean and agile daily planning and follow-up procedures. The third study builds on the first by matching the results found with the intuition of experts in the development organisation, and thus aims at convincing developers that the metrics can be of use to them. This is again a question of building motivation.

Software product metrics for decision support is a research topic that is interesting both to the research community and the industry. Since our work was done, there has been some other interesting work published. One in particular is the SQUALE [13] project, which bears similarities to ours. They base their measurements on a bespoke quality model, called the Qualixo model. This is a work in progress which focuses on community building. Participating software houses help create a base of metrics data that is used for continuously improving the quality model. They offer different viewpoints for different roles in the organisation and also attempt cost prediction based on intrinsic software quality.

3.6 Collaborative Methods Papers

3.6.1 Publication III

Title Tool support for collaborative software process authoring in large organisations [32]

Business goal Decide what tool to buy for the company's process authoring needs.

Background Contrary to traditional practices, we consider that software processes should emerge and evolve collaboratively within an organisation. At the time of the study, there was no clear candidate for tool support for collaborative process authoring, nor any good definition of requirements on such a tool. Although wiki-based solutions have been used for this purpose [22], it is our experience that free-form wiki discussions cannot replace a structured software description, even if attempts to remedy this situation have been suggested by Wongboonsin and Limpiyakorn [77].

Research question Q7: How well do existing process authoring tools support the needs of the industry when it comes to collaboratively sharing process knowledge?

Method We conducted a tool survey based on requirements collected from a focus group.

Results None of the commercially available tools fully met our requirements.

Conformance to research framework We saw the materialisation of the uncontrollability risk, as defined by Kock [44] The same financial cutbacks that troubled Publication V also affected this study. The research was finished but the adoption of the tool suggested by the study did not take place, due to tool budget cuts.

Lessons learned Collecting requirements from key users and using them as a basis for comparing tools proved to be an efficient way to build consensus around which tool to buy.

Conclusion Although none of the commercially available evaluated tools completely filled our needs, IRIS Process Author (IPA) stands out due to its innovative way of handling collaborative process development.

Future work It would be interesting to follow up on the usage of the suggested tool in a large, collaborative setting by means of an empirical study investigating the adoption rate and developer satisfaction with the tool.

Author's role The author was the project manager for the research project and as such was responsible for the planning, analysis and dissemination of the research project. The actual execution, i.e., the testing of the tools against the requirements was performed by Petter Holmström

as a part of his master's thesis. The author acted as Holmström's industrial supervisor for his thesis work.

3.7 Validity of Our Research within the Action Research Framework

In Section 2.5, two approaches to assessing the validity of action research were accounted for. Kock defines three threats to action research: uncontrollability, contingency and subjectivity [44]. Lau [48] lists 22 criteria to consider when conducting and evaluating action research.

First, we consider Kock's approach. In our research, the uncontrollability risk was the one that we mainly saw materialised. The software industry of today is rather unstable, and organisations respond to a fluctuating economy with reorganisation, outsourcing and cutbacks. We consider ourselves fortunate to have been able to perform research at all in these challenging times, but the financial situation did clearly exacerbate the uncontrollability risk. In one project, the refreezing suffered because half of the researched department, including the researchers, was sold to another company. As the refreezing was cut short, general adoption of the proposed actions was not enforced and the change did not happen. In a later project, we tried to mitigate the uncontrollability risk by planning for short iterations, but unfortunately, the studied project was cancelled and the development team was dispersed before we even finished the first iteration.

Next, we consider Lau's approach. The numbers in brackets below refer to the criteria listed on page 22. The question of authenticity of research aim (1), perspective (2), theoretical assumptions (4), participants (8), problem (14) and interventions (15) is clear in our research, as it in all cases has been initiated by the studied organisation as an issue to be solved. The theoretical stance (2) is not explicitly stated, but when looking at the chosen methods, it is clear that we adopt a positivist stance. We do not use the term action research stream (3), but in all cases, we aim to change the practice of the organisation. We strive to provide sufficient background information (5). In some cases, the appropriateness and adequacy of the intended change (6) is actually the issue to be studied. The involvement of the research site (7) and the degree of openness (11) from the researcher's point of view has also been good, but we had to respect the subjects' and organisation's need for privacy (22), so some data could not be openly reported to the research community. We have invested a significant effort into collecting good quality data (9). The duration (10) of the study has in general been sufficient, but as mentioned before, some of our studies were cut short. The entry and exit points for the researcher (12) have been clear, as the research projects have been projects with a well-defined project plan, including start date and end

date. Whether the presentation (13) is adequate in all cases can be debated, but the work here has been published in refereed conferences and a journal. We have not explicitly reported on the reflective learning (16) and iterative process (17) in all cases, so this is sometimes a short-coming of our research. We do think that our research contributes (18) also on a general level. The roles of the researcher (19) and participants (20) is not always explicitly stated, but in most cases they are clear from the description of the method used. In the cases where competency improvement (21) was a goal, we have made an effort to measure it.

This concludes the overview part of the dissertation. The first chapter of the overview was an introduction, while the second chapter introduced essential background concepts of the thesis. In this final chapter of the overview, I have given a detailed overview of the research, placing it in context (Section 3.1), detailing the research questions (Section 3.2) as well as the used methods (Section 3.3). Some effort was made to describe each of the research papers in the context of their respective research area (Sections 3.4 – 3.6). This last chapter was closed with a look at the validity of our research within the action research framework (Section 3.7). Part two of the thesis consists of the original publications and begins after the bibliography.

Bibliography

[1] C. Adelman. Kurt Lewin and the origins of action research. In *Educational Action Research*, volume 1, pages 7–24. Routledge, 1003.

[2] Agile Alliance. Agile manifesto. `agilemanifesto.org/`.

[3] S. W. Ambler. Survey Says: Agile Works in Practice. *Dr. Dobb's Portal: Architecture & Design*, September 2006.

[4] S. W. Ambler. Scaling agile software development through lean governance. In *SDG '09: Proceedings of the 2009 ICSE Workshop on Software Development Governance*, pages 1–2. IEEE Computer Society, Washington, DC, USA, 2009.

[5] D. Anderson, S. Augustine, C. Avery, A. Cockburn, M. Cohn, D. DeCarlo, D. Fitzgerald, J. Highsmith, O. Jepsen, L. Lindstrom, T. Little, K. McDonald, P. Pixton, P. Smith, and R. Wysocki. Declaration of interdependence. `pmdoi.org/`.

[6] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32:365–381, 2006.

[7] A. Asthana and J. Olivieri. Quantifying software reliability and readiness. In *Communications Quality and Reliability, 2009. CQR 2009. IEEE International Workshop Technical Committee on*, pages 1–6. May 2009.

[8] R. D. Austin. *Measuring and Managing Performance in Organizations.* Dorset Publishing House, USA, 1996.

[9] J. Auvinen, R. Back, J. Heidenberg, P. Hirkman, and L. Milovanov. Software process improvement with agile practices in a large telecom company. In *Product-Focused Software Process Improvement*, volume 4034 of *Lecture Notes in Computer Science*, pages 79–93. Springer Berlin / Heidelberg, 2006.

[10] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen. Action research. *Commun. ACM*, 42(1):94–97, 1999.

[11] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[12] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[13] A. Bergel, S. Denier, S. Ducasse, J. Laval, F. Bellingard, P. Vaillergues, F. Balmas, and K. Mordal-Manet. SQUALE. *Software Maintenance and Reengineering, European Conference on*, 0:285–288, 2009.

[14] B. Boehm and R. Turner. Using risk to balance agile and plan-driven methods. *Computer*, 36(6):57 – 66, june 2003.

[15] L. C. Briand, C. Bunse, and J. W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Eng.*, 27(6):513–530, 2001.

[16] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245 – 273, 2000.

[17] A. Cockburn. *Crystal clear a human-powered methodology for small teams*. Addison-Wesley Professional, 2004.

[18] A. Cockburn. *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series)*. Addison-Wesley Professional, 2006.

[19] A. Cockburn and J. Highsmith. Agile software development, the people factor. *Computer*, 34(11):131 –133, November 2001.

[20] M. Cohn and D. Ford. Introducing an agile process to an organization. *Computer*, 36(6):74–78, 2003.

[21] D. M. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[22] W. Cunningham. Cunningham & Cunninghamn Wiki. `http://c2.com/`.

[23] W. Cunningham. The WyCash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 29–30. ACM, New York, NY, USA, 1992.

[24] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859, 2008.

[25] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, Jan 2008.

[26] N. Fenton and S. L. Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997.

[27] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[28] M. Genero, D. Miranda, and M. Piattini. Defining metrics for UML statechart diagrams in a methodological way. In *Conceptual Modeling for Novel Application Domains*, volume 2814 of *Lecture Notes in Computer Science*, pages 118–128. Springer Berlin / Heidelberg, 2003.

[29] D. J. Greenwod, editor. *Action Research*. John Benjamins Publishing Company, 1999.

[30] J. Grenning. Launching extreme programming at a process-intensive company. *IEEE Softw.*, 18(6):27–33, 2001.

[31] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[32] J. Heidenberg, P. Holmström, and I. Porres. Tool support for collaborative software process authoring in large organizations. In *European Systems & Software Process Improvement and Innovation Industrial Proceedings*. DELTA, Denmark, 2009.

[33] J. Heidenberg, J. Katajala, and I. Porres. Maintainability index for decision support on refactoring. Technical Report 992, Turku Centre for Computer Science, November 2010.

[34] J. Heidenberg, M. Matinlassi, M. Pikkarainen, P. Hirkman, and J. Partanen. Systematic piloting of agile methods in the large: Two cases in embedded systems development. In *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, pages 47–61. Springer Berlin / Heidelberg, 2010.

[35] J. Heidenberg, A. Nåls, and I. Porres. Statechart features and pre-release maintenance defects. *J. Vis. Lang. Comput.*, 19(4):456–467, 2008.

[36] J. Heidenberg and I. Porres. Metrics functions for kanban guards. In *IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. 2010.

[37] J. D. Herbsleb and R. E. Grinter. Conceptual simplicity meets organizational complexity: case study of a corporate metrics program. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 271–280. IEEE Computer Society, Washington, DC, USA, 1998.

[38] A. Hevner, S. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, Jan 2004.

[39] M. R. Hoffmann. EclEmma Java code coverage for Eclipse. `www.eclemma.org`.

[40] IBM Rational Software. IBM Rational Jazz. `jazz.net`.

[41] S. Ilieva, P. Ivanov, and E. Stefanova. Analyses of an agile methodology implementation. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 326 – 333. Aug.-3 Sept. 2004.

[42] International Standards Organization. ISO 9126: Software engineering – product quality. Technical report, International Standards Organization, 2001.

[43] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting experiments in software engineering. *Guide to Advanced Empirical Software Engineering*, Jan 2008.

[44] N. Kock. The three threats of action research: a discussion of methodological antidotes in the context of an information systems study. *Decis. Support Syst.*, 37(2):265–286, 2004.

[45] C. Ladas. *Scrumban - Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, USA, 2009.

[46] Lakeworks. Scrum process. `http://en.wikipedia.org/wiki/File:Scrum_process.svg/`.

[47] C. F. J. Lange and M. R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 401–411. ACM, New York, NY, USA, 2006.

[48] F. Lau. A review on the use of action research in information systems studies. In *Proceedings of the IFIP TC8 WG 8.2 international conference on Information systems and qualitative research*, pages 31–68. Chapman & Hall, Ltd., London, UK, 1997.

[49] D. Leffingwell. *Scaling Software Agility: Best Practices for Large Enterprises (The Agile Software Development Series)*. Addison-Wesley Professional, 2007.

[50] K. Lewin. Action research and minority problems. In *Resolving Social Conflicts*, pages 201–216. Harper & Row, New York, 1946.

[51] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kähkönen. Agile software development in large organizations. *Computer*, 37:26–34, 2004.

[52] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 297–306. ACM, New York, NY, USA, 2006.

[53] M. V. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Softw. Engg.*, 11(3):395–431, 2006.

[54] G. Melnik and F. Maurer. Comparative analysis of job satisfaction in agile and non-agile software development teams. In *Extreme Programming and Agile Processes in Software Engineering*, volume 4044 of *Lecture Notes in Computer Science*, pages 32–42. Springer Berlin / Heidelberg, 2006.

[55] Microsoft. Sharepoint 2010 - the business collaboration platform for the enterprise and the internet. `sharepoint.microsoft.com`.

[56] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, 2010.

[57] Object Management Group. UML 2.0 Superstructure Specification. Technical report, OMG, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.

[58] Odysseus Software GmbH. Stan structure analysis for java. `www.stan4j.com`.

[59] T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[60] K. Petersen and C. Wohlin. A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. *Journal of System Software*, 82(9):1479–1490, 2009.

[61] M. Pikkarainen, O. Salo, and J. Still. Deploying agile practices in organizations: A case study. In *Software Process Improvement*, volume 3792 of *Lecture Notes in Computer Science*, pages 16–27. Springer Berlin / Heidelberg, 2005.

[62] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[63] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2006.

[64] J. Rasmusson. Introducing XP into greenfield projects: Lessons learned. *IEEE Softw.*, 20(3):21–28, 2003.

[65] D. J. Reifer. How good are agile methods? *Software, IEEE*, 19(4):16 – 18, jul/aug 2002.

[66] L. Rising and N. S. Janoff. The Scrum software development process for small teams. *IEEE Softw.*, 17(4):26–32, 2000.

[67] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, Los Alamitos, CA, USA, 1987.

[68] O. Salo and P. Abrahamsson. Empirical evaluation of agile software development: The controlled case study approach. In *Product Focused Software Process Improvement*, volume 3009 of *Lecture Notes in Computer Science*, pages 408–423. Springer Berlin / Heidelberg, 2004.

[69] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[70] J. Singer, S. E. Sim, and T. C. Lethbridge. Software engineering data collection for field studies. *Guide to Advanced Empirical Software Engineering*, Jan 2008.

[71] P. Straub. Sample burndown chart. `http://en.wikipedia.org/wiki/File:SampleBurndownChart.png/`.

[72] G. I. Susman and R. D. Evered. An assessment of the scientific merits of action research. *Administrative Science Quarterly*, 23(4):582–603, 1978.

[73] R. Turner and A. Jain. Agile meets CMMI: Culture clash or common cause? In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 153–165. Springer-Verlag, London, UK, 2002.

[74] M. Umarji and C. Seaman. Why do programmers avoid metrics? In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 129–138. ACM, New York, NY, USA, 2008.

[75] Wikimedia Foundation. Mediawiki. `www.mediawiki.org`.

[76] J. Womack, D. Jones, and D. Roos. *The Machine That Changed the World*. Rawson Associates, 1990.

[77] J. Wongboonsin and Y. Limpiyakorn. Wikipedia customization for organization. *Advanced Computer Theory and Engineering, International Conference on*, 0:467–471, 2008.

Part II

Original Publications

3.8 Publication I

Software process improvement with agile practices in a large telecom company

Jussi Auvinen, Rasmus Back, Jeanette Heidenberg, Piia Hirkman, and Luka Milovanov

Originally published in *Product-Focused Software Process Improvement, volume 4034 of Lecture Notes in Computer Science*, pages 79-93. Jürgen Münch and Matias Vierimaa, editors. Springer Berlin / Heidelberg. PROFES 2006.

© Springer-Verlag Berlin Heidelberg 2006. Reprinted with kind permission of Springer Science and Business Media.

3.9 Publication II

Statechart features and pre-release maintenance defects

Jeanette Heidenberg, Andreas Nåls, and Ivan Porres

II

Originally published in *Journal of Visual Languages and Computing*, 19(4), pages 456-467. Philip Cox and John Hosking, editors. Elsevier Ltd 2008.

© Elsevier Ltd 2008. Reprinted with permission.

3.10 Publication III

Tool support for collaborative software process authoring in large organizations

Jeanette Heidenberg, Petter Holmström, and Ivan Porres

Originally published in *European Systems & Software Process Improvement and Innovation Industrial Proceedings*. Jørn Johansen, Mads Christiansen, editors. DELTA, Denmark. 16th EuroSPI Conference 2009.

© Delta 2009. Reprinted with permission.

Tool Support for Collaborative Software Process Authoring in Large Organizations

Jeanette Heidenberg (EB, Elektrobit Corporation)
Petter Holmström (Åbo Akademi University)
Ivan Porres (Åbo Akademi University)

Abstract

Contrary to traditional practices, we consider that software processes should emerge and evolve collaboratively within an organization. In this article we present our vision of collaborative process authoring, we evaluate and discuss how existing process authoring tools suit this vision and we suggest a number of improvements to these tools to facilitate the deployment of a collaborative process authoring method in a large and geographically distributed organization.

Keywords

Software process authoring, software process improvement

1 Introduction

As the corporate world is growing increasingly aware of knowledge as an invaluable asset, the software process development follows suit. In large organizations, the software processes may grow into large and complex descriptions of the method know-how of the organization. This holds especially true if the products developed are large and complex, such as telecommunications or automotive systems in our case. A geographically distributed development environment further exacerbates the complexity. Process descriptions may be created and maintained using standard productivity software, but in our experience, specialized software process authoring tools are needed for maintaining complex, dynamic software process descriptions.

Contrary to traditional practices, we consider that software processes should emerge and evolve collaboratively within an organization. The time of the process developer in the ivory tower is coming to an end. The software process of the future should be the result of a collaborative effort of the software professionals of the company, sharing their best practices, methods and learnings.

However, we have found challenges implementing this tenet in practice, especially due to the lack of proper tool support. In this article we present our vision of collaborative process authoring, we evaluate and discuss how existing process authoring tools suit this vision. Finally, we suggest a number of improvements to these tools that would facilitate the deployment of a collaborative software process authoring method in a large and geographically distributed organization.

2 Collaborative Process Authoring

The main purpose of a software process is to support the people involved in the development of software products. There may be other, secondary purposes as well, such as proving the maturity of the company to a potential customer, or producing a paper trail for legal purposes. In the best case, these secondary purposes will automatically follow from a good process description. To the extent that they do place additional requirements on process descriptions, however, we address this issue when necessary.

In this section, we discuss our vision of good software process descriptions and the implication of our vision on the required features of process authoring tools.

We consider that a good software process description performs the following three tasks: communicate, remind and learn. The first task of a process description is simply to communicate the process to the project staff. Whenever a new member joins the team, you can point to the process description and say: “This is how we have agreed to work.” In order to serve this purpose, a process description will not consist of only work instructions, but also include guidance in the form of instructions, templates and examples.

The second task is to serve as a reminder for the project members, allowing them to check the process during the execution of the project. This differs from the first task in the way in which the process is presented. When serving as a reminder, the process description needs to display the specific information the reader needs now, rather than present the reader with the entire process.

The third task is different from the others in the direction of information flow. The process description should be able to learn from the experience of the projects. The process description can then evolve over time based on the experience and feedback provided by the projects using the process. As such, the process description can serve as a means for organizational learning as defined in [9].

In order for a software process description to fulfill its tasks of communicating, reminding and learning, it needs powerful tool support. Although it is possible to edit and maintain short process description documents using standard productivity software such as a word processor, this approach does not scale up to large processes. Process authoring tools such as Eclipse Process Framework Composer [3] and IRIS Process Author [4] simplify greatly the task of creating and maintaining large process descriptions.

In the following, we discuss the three tasks of a software process description separately. We argue for the necessity of each of them and provide a set of requirements we propose for process authoring tools pertaining to support software process descriptions in their task of communicating, reminding and learning.

2.1 Communicate: Disseminate Process to All Interested Parties

The software process of an organization encompasses the method know-how of that organization. This know-how needs to be efficiently disseminated to the project staff in order to be of use.

In our experience, processes are often considered to be an extra burden on top of the "real" work of developing software, so few software developers make an extra effort to observe the software process if it is not easily accessible and clearly communicated. For this reason, we often see traditional process descriptions, consisting of a large set of documents either in paper form or represented, e.g., as a set of PowerPoint slides, failing to be observed by the project staff.

Kellner [11] defines a set of basic requirements on process modeling tools, which we consider to cover the task of communication well. These include requirements on visualization, support for different viewpoints, multiple levels of abstraction, the use of a formal syntax, handling of multiple variants and versions as well as various analysis capabilities. As Kellner formulated his requirements already in 1988, most of them are already well addressed by existing tools. For this reason, we do not specifically list requirements for communicating software process descriptions, but rather focus on the more difficult tasks of learning and reminding. For further details, we refer the reader to Kellner [11].

2.2 Learn: Collaboration and Tailoring

Ideally, the process description embodies the best practices and lessons learned in the organization, in order for future projects to learn from the successes and mistakes of earlier ones [8]. The evolution of the process then becomes a collaborative effort of the company's method experts, i.e., the project staff. A process description can in that way be a dynamic knowledge base encompassing the process experience of the entire organization. As such a knowledge base can grow quite large, it is also important to have the possibility to tailor the process for the specific needs of each project. In this section, we discuss the needs of collaborative process development and process tailoring.

In order for a process description to be collaborative, it should be possible to comment on and annotate it in a distributed manner by anyone using the process. This is quite a large step away from previous paradigms, where a small group of process experts held the exclusive responsibility for updating the process description [14]. This requirement does not only have a significant impact on usability, but also on synchronization mechanisms in order to avoid inconsistencies arising from concurrent updates. Furthermore, there is often a need to make a distinction between parts of the process that are normative and officially released on the one hand and parts of the process that constitute guidance and are fully collaborative on the other hand. One example of such a need is if the software process is required to be appraised with regards to process maturity models or standards such as CMMI [1] or SPICE [5].

We summarize our collaboration issues in the following three requirements.

Learn 1 Comments and annotations possible for any user.

Learn 2 Support synchronized updates.

Learn 3 Differentiate between normative and collaborative versions of the process.

Every software project is unique. Over the years, many process models have been proposed as solutions for the software crisis and the ever increasing need for efficiency and productivity in software development projects. But no one process model has yet provided the perfect solution. In the end, there are no silver bullets that can make every project succeed. In our experience, the specific process needs of a project arise from parameters such as context, criticality, problem domain, size, business model and the experience and personality of the people involved. The agile [12] and lean [16]

communities today emphasize the need for the process to be defined by the team itself. It would be inefficient to always start your process definition from scratch, though. For this reason, process engineering today often consists of tailoring the contents of an existing process repository to the needs of your specific setting [15]. In this way you combine the need for uniqueness with the power of reuse. The success of a process description then comes to depend on how easy it is to tailor.

In order for a process description to be tailorable there has to be a clear mechanism for extending, narrowing and redefining process content. When this is done, the need for version handling of both the original process description and the tailored process description becomes evident. We need to be able to state what version of the normative process the tailored one is based on. Furthermore, if you consider this feature in combination with the collaborative feature, it becomes clear that the connection from a tailored version of the process description to its original version needs to be maintained. If this is not the case, the collaborative work will be in the context of the process instance of the project in question only, and the lessons learned will be limited to the project in question, thus defeating the original purpose of organizational learning. Fig. 1 depicts these dependencies between the normative, the tailored and the collaborative versions of the process description.

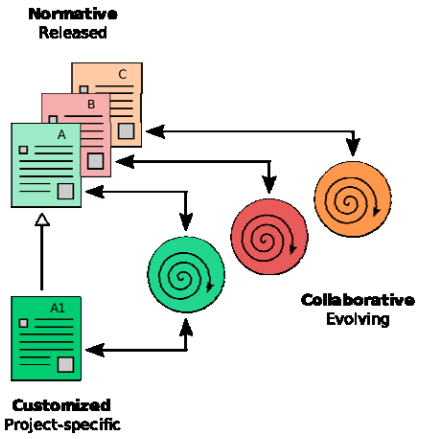


Fig. 1. The relations between normative, collaborative and customized process descriptions.

We summarize these issues with software process tailoring in the following requirements.

- **Learn 4** Support version hand ling of the process description.
- **Learn 5** Maintain traceability between original and tailored process descriptions.

2.3 Remind: Providing the Information the Reader Needs Now

In order for a software process description to serve as an efficient reminder, the content has to be adapted to the specific needs of the reader. A software tester will not want to read 40 pages on requirements engineering and software implementation before finding the parts relevant to testing. So the reader has to be given the possibility to access the process differently depending on his or her current role. On the other hand, the tester may be interested to know what other roles he or she should collaborate with on the activity currently under way or on the artifact under development. So the reader also has to be given the possibility to access the process differently depending on his or her current activity and artifact.

The possibility to access the software process description from different viewpoints was listed as a requirement already by Kellner [11] and has been further developed since. Today, it is a common feature in process authoring tools, due to the fact that many tools implement the standard meta-model for software and systems process engineering (SPEM) [13] as published by the Object Management Group. A meta-model describes the structure of the repository used by a process authoring tool to store process descriptions. We consider this to be a positive development that, however, does bring with it some drawbacks.

The standard meta-model is quite complex, which may lead to situations where the software process description itself gets unnecessarily complex. This in turn may necessitate the creation of company specific tool usage guidelines, which specify what parts of the meta-model should be used. These issues encumber the creation of process descriptions that may serve as efficient reminders, since it becomes difficult for the readers to find the information they need.

For this reason, we consider that process authoring tools should provide a way of easily customizing the meta-model to meet the needs of the users. The GUI itself may also need to be customized in order to better present the customized meta-model. Since usage guidelines may still be needed, it would be beneficial if companies could integrate their usage guidelines closely with the tool.

We summarize these aspects of a software process description serving as an efficient tool for reminding in the following requirements.

- **Remind 1** Enable access to the process through different views based on the need of the user.
- **Remind 2** Support customization of the meta-model.
- **Remind 3** Support integration of company specific usage guidelines in the tool.
- **Remind 4** Support customization of the GUI.

3 Issues with the Current Process Authoring Tools

With the requirements outlined in Section 2 in mind, we have evaluated some of the most popular open source and commercial process authoring tools available today, namely Eclipse Process Framework Composer1 (EPC) [3], Rational Method Composer2 (RMC) [6] and IRIS Process Author (IPA) [4]. In addition, we also included an in-house process authoring tool in the evaluation. The results are summarized in Table 1. We have found that that these tools have some inconveniences that prevent them from fully meeting all our requirements. In the following, we are going to discuss the problems encountered.

Table 1. Tool evaluation results (C=Compliant, PC=Partially compliant, NC=Not Compliant)

Req.	In-house	EPC	RMC	IPA
Learn 1	C	PC	PC	C
Learn 2	PC	PC	PC	C
Learn 3	C	NC	NC	C
Learn 4	C	C	C	C
Learn 5	NC	NC	NC	NC
Remind 1	C	C	C	C

[1] Version 1.2
[2] Version 7.2

Remind 2	C	NC	NC	PC
Remind 3	NC	NC	NC	NC
Remind 4	C	NC	NC	NC

3.1 Learn: Collaboration and Version Handling

In the first requirement related to the task of learning (Learn 1), we suggest that collaboration between process authors and process users demands that any user should be able to comment and annotate the process. We now proceed to present our findings in respect to this requirement.

Many tools are still designed around the traditional process engineering principle, where the processes are designed by a dedicated process engineering team and then exported to the software engineering teams for adoption.

Of the evaluated tools, IPA has the best support for collaboration through its Process Central and Wiki features. Close behind follows the in-house tool that generates links to the company wiki from every process entity. Thus, anyone with access to the company wiki can contribute to the process. However, the changes are not automatically propagated from the wiki to the process authoring tool — they have to be imported manually (as is the case with IPA as well). EPC/RMC can be configured to provide a single link for collecting user feedback. It is not possible to specify different feedback links for different entities within the process.

In Section 2.2 we listed four more requirements which support a process description's task of learning. These are related to various aspects of version handling (Learn 4). When it comes to authoring and documenting process descriptions, a good version control system is important for two reasons: synchronization (Learn 2) and version tracing (Learn 5). Version handling also helps differentiate between normative and collaborative versions of the process (Learn 3). We now proceed to present our findings in respect to these requirements.

Synchronization becomes an issue in all situations where several actors compete for a limited number of resources. In this case, several users must be able to work on the process simultaneously without having to worry about overwriting each other's changes. This issue is normally resolved by using a locking mechanism. Two popular locking mechanisms are "Lock-Modify-Unlock" and "Copy-Modify-Merge" (examples of both mechanisms are presented in Chap. 1 in [7]).

Lock-Modify-Unlock prevents other users from editing a locked entity until the lock is released. This is acceptable if the size of the lock area is small enough. However, if the lock spans large areas of the process description, it effectively blocks collaboration, as only one user can work on the process description at a time.

Copy-Modify-Merge detects conflicts instead of preventing them. When a conflict is detected, the slower author has to merge his or her version with the one currently in the repository before it can be committed. However, if a general revision control system such as Subversion, CVS or ClearCase is used, this becomes a problem as these systems work with ordinary files. The slower author either has to revert his or her changes, update the working copy and re-add the modifications manually, or exit the tool and merge the changes on a raw code level (e.g. if the tool stores its data in XML-files). In either case, it is a time consuming and error prone process.

IPA has tried to address the synchronization issue by using a built-in Lock-Modify-Unlock mechanism that works on the process library level. Thus, it is impossible for two users to work on the same process library simultaneously, causing a possible block to collaboration. This problem can be worked around by making sure the libraries are small enough, which is a good idea in any case as it makes reuse easier as well. IPA handles version tracing by dumping the process descriptions to XML and storing them in external revision control systems. The XML dumps can then be re-imported into IPA as necessary.

The other evaluated tools all rely on external revision control systems for both synchronization and

version tracing. Thus, they all suffer from the problems mentioned previously in this section.

3.2 Remind: User Interface Issues

As we outline in Section 2.3, a process description's function as a reminder relies on how efficiently the user finds the specific information needed at that particular point in time (Remind 1). This in turn depends on the suitability of the user interface on the one hand (Remind 4) and the suitability of the underlying meta-model on the other hand (Remind 2). In situations when the user interface and meta-model are overly complex, the user will need easy access to usage guidelines (Remind 3). We now proceed to present our findings in respect to these requirements.

Most tools use a standardized meta-model such as SPEM [13]. This has the benefit or reducing vendor lock-in and increasing portability between different process authoring tools. However, it has also led to vendors including in their tools all possible concepts that might be needed to model a process, resulting in overly detailed process descriptions and complex user interfaces. Few users will actually need all the functionality, which forces companies to introduce usage guidelines. This in turn makes the tools harder to use and scare off casual users whose feedback and opinions would otherwise have been very valuable.

The meta-model in most tools is fixed, i.e. the users cannot define custom entities or attributes. Even in cases where the meta-model allows customization in principle, the user interface does not support this in practice. Consequently, the users are forced to model their processes using the predefined process elements.

Of the evaluated tools, only the in-house tool supports scaling of the user interface, thanks to its customizable meta-model. IPA supports user-defined attributes, making it possible to make some customizations to the meta-model.

4 A Plea for a New Collaborative Process Authoring Tool

In Section 3 we listed issues with many modern process authoring tools. In this section, we are going to present proposals and ideas to resolve these issues, thereby improving the state of the practice.

4.1 Learn: Integrate Authoring, Publishing and Peer-Review in a Web-Based System

We believe that the key to continuous process improvement is collaborative process review and editing. To simplify this task we propose to integrate process publishing and authoring tools in the same web-based system. A clear precedent for this is the wiki, which has lead to massive collaborative efforts such as the Wikipedia. A wiki allows a user to switch from a reader to a contributor role simply by clicking a button placed next to the contribution to be edited or expanded. There are usually few (if any) limitations on when or by whom the wiki contents can be modified. However, contributions are reviewed afterwards by an editor and they can be edited or even discarded if they are not considered suitable.

In the context of authoring software process descriptions, we propose to use one single web-based application tool to author, communicate, comment and review software processes within an organization. Although it is possible to use a dedicated desktop-based process authoring tool in conjunction with a standard wiki to allow free-form discussions in an organization, we believe that an integrated web-based solution would really foster a collaborative effort where each user of a process description also becomes a contributor.

One of the main challenges when improving a process description collaboratively is reviewing and merging multiple contributions from different sources. Although it is possible to use text-based difference, merge and version control tools designed for source code (such as diff, patch and git) or text documents (such as the revision facilities in Microsoft Word), this may easily lead to inconsistent proc-

ess descriptions. The underlying problem is that a software process has a graph structure, while it is often represented as a text (string). Also, complex process descriptions are often split into different files or pages.

Based on this and in response to the issues presented in Section 3.1, we consider that web-based process authoring and reviewing systems should implement a model aware diff-and-merge component. This component would allow a user to compare the differences between the two graphs representing two versions of the process description the model level rather than two text strings. The component would also allow a user to merge the two versions using a graphical user interface.

4.2 Remind: Scalable User Interface

We propose that every process also comes bundled with an embedded user interface configuration for the tool. The configuration would contain information about the attributes and entities that are to be used for the process, as well as specific usage guidelines as an optional element. That is, instead of having a complex user interface and a separate usage guideline document that instructs the users what to fill in and what to leave blank, the tool would only display the fields that are to be filled in and provide optional corresponding instructions inside the tool, as the user navigates the user interface (see Fig. 2).

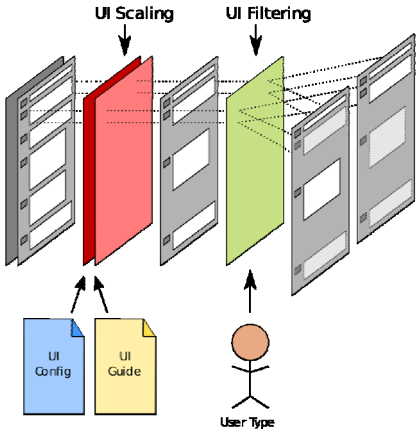


Fig 2. Scaling and filtering of the user interface.

In addition, we propose that the user interface is filterable depending on the needs of the user as described in Section 2.3 (see Fig. 2). A software engineer and a process engineer do not need access to the same information. This feature is already available in many of the tools3, but only in the exported process description. We propose that this feature is applied to the user interface as well.

The tool could come with a set of predefined user interface configurations for different types and sizes of projects. However, it should also be possible to customize these configurations in an easy way, e.g. similar to the content variability mechanisms of SPEM (see Sect. 6.3.4 in [13]).

[3] This feature is called *Views* in EPC/RMC.

5 Conclusions and Related Work

In this article we have discussed the importance of a collaborative effort in process authoring and what we consider are the main challenges in deploying this in practice and how these challenges could be overcome with proper tool support.

In order to meet the challenges of an industry that requires increased efficiency, agility and streamlining of the software development operations, we consider collaborative process authoring to be of vital importance.

We have presented three tasks for a software process: communicate, learn and remind. We consider that the existing authoring tools perform the communicate task well. However, when we consider the learn and remind tasks, they fall short. The reasons for this are that first, they have been designed with the traditional process engineering principle in mind, where the processes are designed by a dedicated process engineering team and then exported to the software engineering teams for adoption; and second, their vendors have included all possible process modeling concepts into them without proper filtering and scaling capabilities. We suggest that the tool vendors shift focus and concentrate on making their tools more collaborative, customizable and scalable to different process sizes.

Among the commercially available evaluated tools, IRIS Process Author (IPA) [4] stands out due to its innovative way of handling collaborative process development. Its Process Central and Wiki features come close to the needs we have outlined. The original assumption, however, still seems to be that process development is driven by a team of process engineers, whereas we propose a completely decentralized way of working where everyone is a potential process engineer. The linkage between the normative version and the Wiki version is limited in IPA to a top level link. We would prefer a link for each process entity. We have been informed, however, that this feature is included in the roadmap of IPA. It is still unclear how well this feature will interact with process tailoring, as outlined in Section 2.2. The feedback loop from the collaborative version to the normative one is also still unclear.

We can consider that the Cunningham & Cunningham wiki [2] has been used as a collaborative process authoring tool. Many software development approaches have been discussed and criticized in that web site, and this has been highly influential in the current thinking in agile software development. However, we consider that free wiki discussions do not replace a structured software process description, although attempts to remedy this situation have been suggested by Wongboonsin and Limpiyakorn [17]. Also, there are no distinctions between the normative version of the processes and the collaborative versions under discussion and review.

In their article “Enough Process - Let’s do Practices” [10], Jacobson et al. present similar concerns to those presented in this article. They suggest “a shift of focus from the definition of complete processes to the capture of reusable practices” and that “teams should be able to mix-and-match practices and ideas from many different sources to create effective ways of working”. In their article, they list some common problems that many software processes struggle with and provide some interesting ideas on how to solve them. We think that in addition to the ideas presented in this article, the ideas of Jacobson et al. should be taken into account when the next generation of process authoring tools is designed.

In conclusion, we believe that software process authoring in large organizations is moving in a collaborative direction. In order to support this positive development, we propose that the tool development effort needs to focus on collaborative authoring, on integrated process presentation and authoring as well as on more flexible tailoring of the process presentation.

6 Acknowledgements

This work was partly funded by the EB Flexi project and the Flexi EU-project of ITEA framework. The authors wish to thank Jari Partanen for his support and Martin Dusch and the members of the process authoring tool user group at EB for their input and support.

7 Literature

1. Carnegie Mellon Software Engineering Institute. http://www.sei.cmu.edu/cmmi/.
2. Cunningham & Cunningham Wiki. http://c2.com.
3. Eclipse Process Framework Web site. http://www.eclipse.org/epf.
4. IRIS Process Author Web site. http://www.osellus.com/IRIS-PA.
5. ISO/IEC 15504 (SPICE). http://www.isospice.com/categories/ISO%7B47%7DIEC-15504-Standard/.
6. Rational Method Composer Web site. http://www.ibm.com/software/awdtools/rmc/index.html.
7. B. Collins-Sussman, B.W. Fitzpatrick and C.M. Pilato. Version Control with Subversion. Published online, http://chestofbooks.com/computers/revision-control/subversion-svn/index.html, 2008.
8. Victor R. Basili and Gianluigi Caldiera. The experience factory: strategy and practice. Technical report, College Park, MD, USA, 1995.
9. Nancy M. Dixon. Organizational learning cycle : How we can learn collectively. Gower Publishing Limited, 1999.
10. I. Jacobson, Pan Wei Ng and I. Spence. Enough Process - Let's Do Practices. Journal of Object Technology, 6(6):41–66, July-August 2007.
11. M.I. Kellner. Representation formalisms for software process modelling. In Proceedings of the 4th international Software Process Workshop on Representing and Enacting the Software Process, pages 93–96, Devon, United Kingdom, 1988.
12. Robert C. Martin. Agile Software Development: Principles, Patterns and Practices. Prentice Hall, 2003.
13. OMG. Software & Systems Process Engineering Meta-Model Specification Version 2.0. Specification, Object Management Group, Inc., April 2008. OMG Document Number formal/2008-04-01.
14. P. Fowler, S. Rifkin. Software Engineering Process Group Guide. Technical report, SEI, September 1990. CMU/SEI-90-TR-024.
15. Oscar Pedreira, Mario Piattini, Miguel R. Luaces, and Nieves R. Brisaboa. A systematic review of software process tailoring. SIGSOFT Softw. Eng. Notes, 32(3):1–6, 2007.
16. Mary Poppendieck and Tom Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
17. Jenjira Wongboonsin and Yachai Limpiyakorn. Wikipedia Customization for Organization's Process Asset Management. International Conference on Advanced Computer Theory and Engineering, pages 467–471, 2008.

8 Author CVs

Jeanette Heidenberg

Jeanette Heidenberg received her M.Sc. degree in Computer Science in 1999 at Åbo Akademi University. She has since worked as Software Designer and Architect for Ericsson (2000-2006) and Software Specialist for EB (2006-2009) in Finland. Since 2003, her main task has been process development, focusing on model driven and agile methods. She is currently pursuing her Ph.D. in Computer Science at Åbo Akademi University.

Petter Holmström

Petter Holmström is a master's student in Computer Engineering who will receive his degree in 2009. In his master's thesis, he investigated how well current software process authoring tools are able to conform to the requirements of a large organization. He currently works as a research assistant at Åbo Akademi University.

Ivan Porres

Ivan Porres received his M.Sc. degree in Computer Science in 1997 at the Polytechnic University of Valencia, Spain and in 2001 his Ph.D. in Computer Engineering at Åbo Akademi University in Turku, Finland. He is currently professor of Software Engineering at Åbo Akademi University where he performs his is research on software design languages and supporting tools, engineering software services and software process improvement.

3.11 Publication IV

Metrics functions for kanban guards

Jeanette Heidenberg, and Ivan Porres

Originally published in *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. Roy Sterrit, Brandon Eames and Jonathan Sprinkle, editors. IEEE ECBS 2010.

© IEEE 2010. Reprinted with permission.

3.12 Publication V

Systematic piloting of agile methods in the large: Two cases in embedded systems development

Jeanette Heidenberg, Piia Hirkman, Mari Matinlassi, Jari Partanen, and Minna Pikkarainen

Originally published in *Product-Focused Software Process Improvement, volume 6156 of Lecture Notes in Computer Science*, pages 47-61. M. Ali Babar, Matias Vierimaa, and Markku Oivo, editors. Springer Berlin / Heidelberg. PROFES 2010.

© Springer-Verlag Berlin Heidelberg 2010. Reprinted with kind permission of Springer Science and Business Media.

3.13 Publication VI

Maintainability Index for Decision Support on Refactoring

Jeanette Heidenberg, Jussi Katajala, and Ivan Porres

©Turku Centre for Computer Science 2010

Maintainability Index for Decision Support on Refactoring

Jeanette Heidenberg
TUCS Turku Centre for Computer Science
Åbo Akademi University
Joukahaisenkatu 3-5, FI-20520 Turku, Finland
jeanette.heidenberg@abo.fi
Jussi Katajala
OY LM Ericsson Ab
Hirsalantie 11, FI-02420 Jorvas, Finland
jussi.katajala@ericsson.com
Iván Porres
TUCS Turku Centre for Computer Science
Åbo Akademi University
Joukahaisenkatu 3-5, FI-20520 Turku, Finland
ivan.porres@abo.fi

TUCS Technical Report
No 992, November 2010

Abstract

Maintainability is a software attribute that needs to be continuously addressed during the entire development life-cycle. But the decision to refactor in order to keep the software product maintainable is not always an easy one. In this paper, we present a metrics-based approach for assessing the maintainability of code under development with the purpose of providing decision support for refactoring decisions. Our approach is developed and validated in the context of a large, mature telecommunications product.

Keywords: software maintenance, software quality

TUCS Laboratory
Software Engineering Lab

1 Introduction

According to the IEEE standard glossary [1] maintenance is "the process of modifying a software system or component after delivery." As such, maintenance is usually associated with the last stages of the development cycle. This may not necessarily be the case in an iterative development process such as the Rational Unified Process [16] or Scrum [22], where the software is effectively under maintenance already after the first iteration. The cost of maintaining and further developing a poorly designed system can be high. This cost increases the longer the technological shortcomings go uncorrected, until the code becomes unmanageable and essentially has to be completely rewritten. This escalating problem is often called technical debt, or design debt [7]. In order to ensure that the technical debt does not escalate out of control, the maintainability of the product under development should be addressed in all stages of the software process.

Refactoring [11] is the state of the practice for fighting the build-up of technical debt. The design and architecture of the product should be under constant improvement in order to accommodate new constraints and requirements. Looking for code and design "smells" (code or design patterns indicating bad design) and amending these should be an integral part of the development work.

In reality the development team often has a difficult choice to make. Whereas smaller refactoring efforts can usually be easily included in the implementation effort, larger changes may require a greater effort and have to be planned for. The project will face situations where they have to weigh the cost of refactoring against the cost of cutting features from the delivery. This decision is essentially a business decision, but it cannot be made without a clear insight into its technical merits. In a large corporation the business expertise and technical expertise is usually represented by different people, often in different organizations, working towards different goals and using different vocabularies. A lack of trust between the two is not uncommon. For these reasons, reaching a decision to refactor may not be a trivial task.

In this paper, we present a metrics-based approach for assessing the maintainability of code under development with the purpose of providing decision support for refactoring decisions. Our goal is to provide development teams with a method for assessing and visualizing the technical debt in a way that supports decision making and facilitates prioritization of refactoring efforts.

Our approach is developed in the context of a large, mature telecommunications product. The product in question has been under development for several years and has seen a development effort comprising hundreds of person years, using several programming languages. Metrics from a number of subsystems developed using IBM Rational Rose RealTime are collected. A model for quantifying the maintainability of the subsystems is created and

calibrated against the technical experts' intuition of the maintainability of the subsystems in question. We also validate the metrics against the before and after states of a large refactoring effort (of one person year) known to have had a positive effect of maintainability.

2 Background

2.1 Software Metrics

Software metrics are usually classified [9] as either product metrics or process metrics. A *software product metric* is a function that quantifies a property of the measured software. Lines of code (LOC) is an example of a software product metric. In contrast, a *software process metric* is a function that quantifies a property of the process used to develop software. Average LOC per person-month is an example of a process metric.

In this paper we suggest the usage of metrics in order to assess the maintainability of code under development. The purpose of this assessment is to provide the project with decision support for refactoring. We believe that a good collection of trusted metrics can facilitate communication between business stake-holders and technical experts to this end. We issue a strong warning against the temptation to use these metrics to draw conclusions regarding the skill or professionalism of the developers. It is essential to understand that a subsystem's maintainability is not necessarily a reflection of the competence of the development team. Many other factors influence this attribute. These include the complexity of the implemented features, the maturity of the product itself, the maturity of the used interfaces, clarity of the requirements, and time pressure to mention but a few. For this reason, the metrics proposed here should be used as information measurements only and never for comparing the "goodness" of teams or individual developers.

2.2 Measuring Maintainability

We strive to measure the maintainability of software. Intuitively, we define maintainability as the ease of which a software system or component can be modified to add or remove features, correct defects, improve quality attributes such as performance, or adapt to changes in its environment.

We base our definition on standards such as the ISO 9126 [14]. Maintainability is one of the six quality attributes listed in the ISO 9126 standard, alongside functionality, reliability, usability, efficiency, and portability. As we are addressing maintainability from the point of view of the developer, we are interested in the internal aspects of this attribute, as opposed to the external ones observed by, e.g., the customer. More specifically, we are interested in the sub-attributes analyzability and changeability.

In [12] we demonstrated by means of analysis of historical data as well as a controlled experiment, that the use of certain design constructs can indicate low maintainability in Rational Rose RT models. Briand et. al. have reached similar conclusions [4] for object oriented software. Studies provide sets of OO metrics for Java [17] and C++ [5] that can be used for decision support for refactoring. Mäntylä and Lassenius show that such metrics can indeed be used to predict refactoring needs [18], especially if complemented by manual reviews of critical parts of the code [19]. It is even possible to estimate the cost of maintenance based on complexity metrics, as demonstrated by Fioravanti [10]. In contrast, Yu et. al. argue that it is not possible to measure maintainability. However, their results make use of process metrics only and are restricted to open-source contexts.

Coleman et.al. [6] present a number of methods for deriving a maintainability index based on internal product metrics. We loosely follow their approach, especially with regards to taking advantage of the intuition of the developers in order to calibrate their index.

Moha et.al. [20] present a complete method for identifying code and design smells, starting from the identification and definition of smells, continuing with processing the definition into executable algorithms, actual detection of smells, and ending with the manual validation of the found smells.

2.3 M-MGw Software

This study was performed in the context of the Ericsson Media Gateway for Mobile Networks (M-MGw) [8], which is a part of the softswitch solution for Ericsson's Mobile Core Network. The M-MGw was first commercially launched in early 2003 and has since then been under development and maintenance with more than 100 people working actively in the projects at any one time. The M-MGw is a mature and industrialized product, with an install base of several hundred units in customer networks around the world.

The M-MGw is developed using 3 different languages and environments: C, Java, and C++ in IBM Rational Rose RealTime. C is used to implement low-level code running on digital signal processors (DSP). The C code includes strict resource and real-time constraints. Java is used to implement monitoring and configuration facilities, including their user interfaces. The Java code has no special resource or performance constraints. IBM Rational Rose RealTime is used to describe reactive software using statecharts and C++. The Rational Rose RealTime models contain performance constraints, but no real-time constraints. The models contain action code in C++ and are compiled to executable software with no separate step where the generated code would be modified. This study was performed in the context of the subsystems developed using C++ in IBM Rational Rose RealTime.

2.4 IBM Rational Rose RealTime

IBM Rational Rose RealTime is a modelling tool for reactive systems, used for model-driven development with a subset of the UML 2.0 [21] standard called UM RealTime.

The Rational Rose RealTime modeling approach is based on the concept of a capsule. A capsule is an active component with its own thread of control. It can communicate with other capsules via ports. A port may require or realize an interface and an interface is defined as a set of signals. Capsule communication is asynchronous. The main behavior of a capsule is specified using a statechart diagram. In the M-MGw product, the guards and actions of a statechart are defined using the C++ programming language. A capsule may contain other capsules and passive objects, which are instances of C++ classes.

The execution of a transition is triggered by the reception of a signal on one of the capsule's ports. The triggered transition decides whether it deals with the signal immediately or defers it until later. A deferred signal may be recalled at any time during execution. When a transition is triggered, the currently active state may change. If the previously active state contained an exit action, the code within this action will be executed before the transition code is executed. If the target state contains an entry action, the code within this action will be executed after the transition code is executed. Transitions may be composed of multiple transitions, either through nesting of states or by choice points. A choice point contains a boolean expression, and depending on the value of this expression the next transition in the chain is selected.

2.5 The Challenges of Technical Debt

This work is part of a larger body of work instigated by the studied organisation as the answer to a specific need. As the model driven approach of IBM Rational Rose RealTime was rather new to the industry at the point in time of the study, there was a lack of consensus among the developers as to what constitutes good statechart design heuristics.

Partly as a consequence of this lack of consensus, the organisation did not properly address technical debt. The business stakeholders and technical staff did not have a good way of discussing technical debt. The business stakeholders suspected the developers of being overzealous in perfecting the technical details of the system, while the technical staff felt they were not given enough time to restructure and refactor code that had deteriorated over time.

The technical debt of one subsystem eventually reached a limit where it could not be properly maintained any longer. A major restructuring effort

was carried out with good results, and the organisation decided to learn more about maintainability of IBM Rational Rose RealTime models and how to, properly and in a timely fashion, address technical debt.

The question of what constitutes good statechart design was studied using statistical methods and a controlled experiment and was reported in [12]. The question of how to address technical debt is discussed in this paper.

3 Towards a Maintainability Index

This study was performed in a specific context with a specific goal of improving the practices of the researched organization. This mainly affects the study in the fact that the approach is pragmatic. The metrics need to be easy to collect, in order for the organization to be able to deploy this practice without encumbering the daily work of the developers. The metrics presentation should be easy to understand, in order for it to serve as a good decision support system for both technical and business staff. The metrics should be trustworthy, in order for it to be effective as a negotiation tool.

In order to serve the pragmatic needs of the organization, the metrics collected are summarized in a *maintainability index (MI)*. The purpose of the maintainability index is to give an estimate of the technical debt of the system in order to provide the projects with decision support on when and where to intensify the refactoring efforts.

The M-MGw product is implemented in an incremental, iterative manner. Small, internal deliveries are built and delivered for internal quality assurance following a regular cadence. A number of such internal deliveries constitute a formal delivery of the product. Quality assurance is a continuous effort on different integration levels and a significant effort is invested in ensuring that the formal delivery is of high quality.

In building a model of the maintainability of the IBM Rational Rose RealTime models of the system under study, we collected metrics from seven subsystems of one delivery of the M-MGw product. This we calibrate against the intuition of subsystem experts regarding the maintainability of the subsystems in question. In the following, we account for the way these data were collected and how they were used to evolve a model. This model was finally validated by checking it against three deliveries of the same subsystem: a version known to have poor maintainability, a restructured version known to have significantly improved maintainability, and the version which was current at the time of the study, and which was known to have deteriorated somewhat since the restructuring effort.

	Sys1	Sys2	Sys3	Sys4	Sys5	Sys6	Sys7
Rating:	M	H	L	H	H	M	L

Table 1: Maintainability Rating per Subsystem

3.1 Expert Intuition

For model calibration, we used the intuitive perception of a group of experts regarding the maintainability of seven subsystems of the embedded system developed by the organization in question. These experts were seasoned developers and architects with extensive experience in working with the specific subsystems. They were either the technical owner of the subsystem in question or held the role of software design architect or system architect.

This group of experts had formed with the objective to evaluate the quality of the architecture of one specifically problematic subsystem. During this effort, one of the conclusions they reached was that they did not always agree on what constitutes good design and whether a certain solution would be a smell or an acceptable solution. This is an issue that is somewhat overlooked in approaches such as the one proposed by Moha et. al. [20], but that in our experience is not uncommon in the industry.

What the experts did agree on, however, was the relative maintainablitiy of the subsystems, when compared to each other. This was based on their vast experience in working with the different subsystems, systematic architectural reviews of the systems as well as experience of the defect trends in the subsystems over serveral years. Table 1 depicts this intuitive evaluation of the maintainability of the subsystems on a a three-point ordinal scale, where one (L) signifies low maintainability and a clear need for refactoring, two (M) signifies medium maintainability and possible need for refactoring and three (H) signifies high maintainability and no need for refactoring.

It is worth noting that the evaluation was not explicitly collected for this study, but was rather a by-product of the architectural evaluation effort. Given more time and resources, we would have asked the experts to make a more detailed rating. As we were limited in this respect, we chose to use three-point scale, since a more detailed scale would have required a larger evaluation effort by the experts.

3.2 Metrics Collection

The data points were gathered from three internal deliveries of the Rational Rose RealTime subsystems of the product. First, we looked at the (then) current internal delivery which would best match the experts' intuition. We refer to this delivery as d_n. We also looked at historical data for one of the subsystems, $Sys6$. This subsystem underwent major restructuring at an earlier point in the lifetime of the product, since it had been determined that

its technical debt had lead to an increasing maintenance effort. We look at the delivery before (d_{r-1}) and just after (d_r) the refactoring of this subsystem.

Ericsson as a company values process maturity and as such has a high standard of adherence to good software development practices. Good tools and practices for configuration management are used. The software repository contains ample information on what version of the code was delivered in which delivery, thus enabling us to easily extract the relevant data from the repository.

As there were no commercial tools available for collecting metrics on systems designed using IBM Rational Rose RealTime, we collected metrics using an in-house tool designed specifically for the purpose.

As a part of the earlier effort to clarify what constitutes good design of statecharts [12] we had a studied a vast collection of metrics, and ended up with a list of problematic design idioms to look for. Based on this study, we also have a set of threshold values (or "trigger points" according to the terminology used by Coleman et.al. [6]) for when these design idioms should raise a warning flag. Furthermore, we used simple size and complexity metrics to evaluate the maintainability of the action code in transitions, methods and choice points.

Below follows a list of the selected metrics. Some of the selected metrics are related purely to the visual aspects (V) of the state machines, while others are related to the action code (C) in C++. The thresholds values were selected based on the experience of our expert group, and were appropriate for the context in question at the time of the study. The intention is for the thresholds to be evaluated and updated periodically to reflect the current state of the system as it evolves.

(V) Ratio Choice Points / State This metric calculates the ratio of choice points per state in the statechart. A large number indicates that there are many different paths through the statechart. This may indicate the antipattern commonly known as "flag jungle" among the organisation's developers. In this antipattern, the statechart depends on too many attributes for decision making and some form of abstraction should be considered, either by splitting the offending capsule or by turning some of the attributes into states. The suggested threshold for this metric is 1.

(V) Visual Cyclomatic Complexity This metric calculates the cyclomatic complexity of the statechart as the number of transitions plus two minus the sum of the number of states and choice points: $\sum transitions - (\sum states + \sum choicepoints) + 2$. The suggested threshold for this metric is 100.

(V) Visual Size This metric calculates the number of visual elements in

the statechart (ports, operations, attributes, states, choice points, transitions.) The suggested threshold for this metric is 300.

(*C*) **Method and Transition LOC** This metric calculates the number of efficient lines of code in methods and transitions. The suggested threshold for this metric is 80.

(*C*) **Method and Transition Cyclomatic Complexity** This metric calculates McCabe's cyclomatic complexity for action code in methods and transitions. The suggested threshold for this metric is 10.

(*C*) **Choice Point LOC** This metric calculates the number of efficient lines of code in choice points. The suggested threshold for this metric is 20. Note that this is much lower than the threshold for methods and transitions. This is due to the findings in our previous study, where choice points seem to have a higher impact on the defect rate.

(*C*) **Choice Point Cyclomatic Complexity** This metric calculates McCabe's cyclomatic complexity for action code in choice points. The suggested threshold fro this metric is 5. Please note that this is much lower than the threshold for methods and transitions, for the same reason as the LOC metric.

(*C*) **Entry / Exit Actions** This metric indicates whether there are entry or exit actions in the statechart. (Warning)

(*C*) **Defer / Recall** This metric indicates whether there are calls to defer or recall in the action code of the statechart. (Warning)

We use the term *design flaw* to denote a location in a subsystem where the measurement for one of the metrics listed above falls outside of the trigger. We do not attempt to measure the degree of fit, i.e., much the design flaw deviates from the trigger point range. Please note that we do not consider a design flaw to be a defect. Although a design flaw may make the subsystem more difficult to maintain, it does not necessarily degrade the functionality of the subsystem. We also make this distinction for the reason that, although the aim usually is to deliver a product without defects, it is not always economically feasible to deliver a flawless product.

3.3 Definition of a Maintainability Index (MI)

Based on the metrics collected and with the calibration the intuition of the developers we proceed to construct a model for assessing the maintainability. We call this model a maintainability index (MI). We divide our MI into two parts: the MI for action code (MI_c)and the MI for visual elements (MI_v).

Our MI is based on the total number of design flaws, both visual and code based. The count of design flaws found for the visual part of the measured subsystem is denoted fl_v. The count of design flaws found for the action code is denoted fl_c.

The maintainability index is not normalized over size, as the intention is first and foremost to help the developers understand the size of the technical debt by finding and flagging indications of problematic design. Normalization over size would hide the true extent of the technical debt in large subsystems.

It may be difficult to understand the maintainability of a subsystem given just on a number, though. If a subsystem has the a visual MI of 46 and code MI of 2, should that subsystem be improved or not? In order to simplify evaluation, the maintainability index is translated to a number between 0 and 1, where 1 indicates very good maintainability and 0 indicates very poor maintainability.

The first step in this translation is to consider the relative weight of the two parts: the code flaws (fl_c) and visual flaws (fl_v). Looking at the collected data from delivery d_n, we observe that the difference is in an order of magnitude of 10 (see Figure 1). Further evaluation places the median for the ratio $\mathrm{fl}_c/\mathrm{fl}_v$ at about 20 (see Figure 2. The outlier is the reworked $Sys6$, which had exceptionally few visual flaws due to the improvement work it had seen some deliveries earlier.

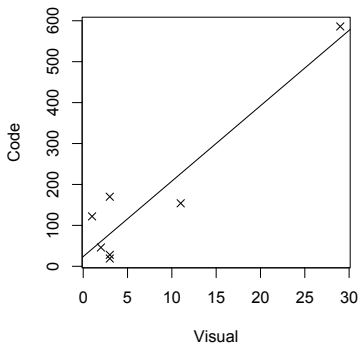


Figure 1: The number of code flaws and visual flaws for each subsystem.

We continue to plotting the subsystems of delivery d_n together with the expert rating of the subsystems. The rating is translated to numeric values between 0 and 1, so that low maintainability (L) corresponds to value 0, medium (M) to value 0.5 and high (H) to value 1. When looking at the

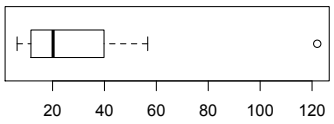


Figure 2: Boxplot of code flaws divided by visual flaws.

resulting graph (see Figure 3) , we notice that $Sys3$ seems to be an outlier. This was known to be an especially problematic subsystem, and the experience was that the maintainability of this subsystem was much lower than that of the others. The figure also shows a linear fit of the subsystems, excluding outlier $Sys3$.

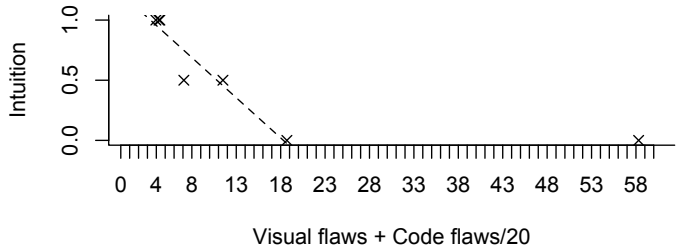


Figure 3: A plot of $\mathrm{fl}_v + \mathrm{fl}_v/20$ of the sybsystems of delivery d_n.

Using this linear fit, we define the overall maintainability index for our IBM Rational Rose RealTime subsystems. The index is limited to the interval [0..1] and defined a follows:

$$\mathrm{MI}(\mathrm{fl}_c, \mathrm{fl}_v) = min(max(\frac{19}{16} - \frac{\mathrm{fl}_c/20 + \mathrm{fl}_v}{16}, 0), 1)$$

Figure 4 depicts this function. In order to make the maintainability index more easily accessible for non-technical staff, we define five color coded levels ranging from red (for very poor maintainability) to green (for very good maintainability). The thresholds for these five levels can be seen in the figure

and were derived by dividing the interval [3..19] (representing the minimum and maximum of the function) in equally sized sections.

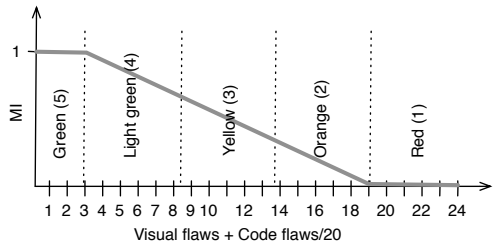


Figure 4: Maintainability function for IBM Rational Rose RealTime Models

For transparency reasons, MI can be split up into its components (MI_c and MI_c). This way it is easier to see whether a subsystem needs code improvements or improvements in the visual elements. Using the fact that we have constructed our MI as $\mathrm{MI}_v + \mathrm{MI}_c/20$ we deduce the following two functions for MI_c and MI_v, where fl_v and fl_c are the number of design flaws for visual elements and action code respectively.

$$\mathrm{MI}_c(\mathrm{fl}_c) = min(max(\frac{19}{16} - \frac{\mathrm{fl}_c}{160}, 0), 1)$$

$$\mathrm{MI}_v(\mathrm{fl}_v) = min(max(\frac{19}{16} - \frac{\mathrm{fl}_v}{8}, 0), 1)$$

Figures 5 and 6 depict these functions.

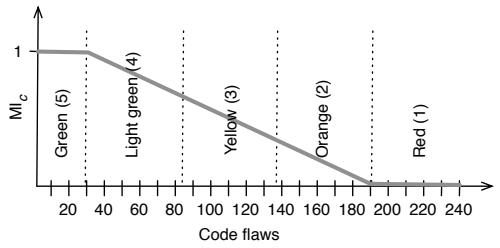


Figure 5: Maintainability function for code flaws.

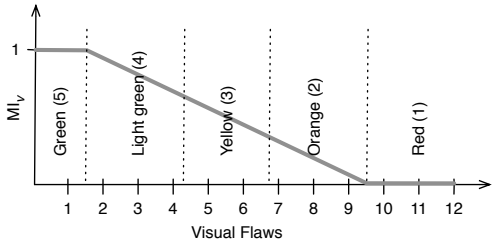


Figure 6: Maintainability function for visual flaws

3.4 Data Presentation and Evaluation

Table 2 lists the metrics gathered for delivery d_n of each of the subsystems, and also for deliveries d_{r-1} and d_r in the case of subsystem $Sys6$. The table also shows the calculated values for MI_c, MI_v and MI_{rrt} together with their corresponding color levels, as well as the expert rating for each of the subsystems.

3.5 Validation against a Refactoring Effort

Figure 7 highlights the different values for MI for the three deliveries of $Sys6$. The first (d_{r-1}) is the last delivery before the refactoring; the second (d_r) is the first delivery after the refactoring; the third (d_n) is the current delivery. As we can see, the maintainability index increases substantially after the refactoring. During this refactoring, both the code and the visual design were improved. As a result, the project could measure a significant decrease in defect reports and many of the defects that were found in earlier deliveries and mapped to the current delivery could be discarded, since the fault just simply did not exist any more due to the refactoring. The size of this refactoring effort was approximately one person year.

It is also interesting to note that $Sys6$ has seen a slight degradation in maintainability between d_r and d_n, when no refactoring has been done. The experts have noticed this degradation, and we can also measure a lower maintainability index.

4 Suggestions for Deployment Practices

In this section we give some suggestions on how to deploy a software maintainability index in an organization.

We suggest that the MI metrics should be collected at least at delivery and the resulting list should be stored. For every warning, an action should

	Sys1 d_n	Sys2 d_n	Sys3 d_n	Sys4 d_n	Sys5 d_n	Sys6 d_{r-1}	Sys6 d_r	Sys6 d_n	Sys7 d_n
Visual size warnings:	0	0	3	0	2	0	0	0	0
Visual cyclomatic complexity warnings:	1	2	4	0	1	2	0	1	2
Visual choice points/states warnings:	2	1	4	0	0	0	0	0	5
Number of entry actions (warnings):	0	0	11	1	1	3	2	0	4
Number of exit actions (warnings):	0	0	7	2	0	3	0	0	0
Class operation amount warnings:	3	4	2	0	1	2	2	2	6
Number of defers (warnings):	15	0	48	8	3	45	19	30	28
Number of recalls (warnings):	4	0	19	7	4	32	14	20	23
Class operation LOC warnings:	27	1	49	0	5	2	5	14	16
Capsule operation LOC warnings:	11	1	44	2	0	0	0	1	5
Transition LOC warnings:	19	6	28	2	11	2	2	6	3
Choice point LOC warnings:	0	3	36	1	0	0	0	0	1
Class operation CC warnings:	38	0	160	0	12	5	17	29	42
Capsule operation CC warnings:	26	1	96	4	0	11	1	9	18
Transition CC warnings:	27	1	60	3	10	11	5	11	11
Choice point CC warnings:	0	2	44	1	0	0	0	0	1
SUMMARY:									
Total number of code warnings:	170	19	586	28	46	110	65	122	154
Total number of visual warnings:	3	3	29	3	2	10	2	1	11
MI_c:	0.13	1.00	0.00	1.00	0.90	0.50	0.78	0.43	0.22
Color level:	(2)	(5)	(1)	(5)	(4)	(3)	(4)	(3)	(2)
MI_v:	0.81	0.81	0.00	0.81	0.94	0.00	0.94	1.00	0.00
Color level:	(4)	(4)	(1)	(4)	(4)	(1)	(4)	(5)	(1)
MI:	0.47	0.94	0.00	0.91	0.92	0.21	0.85	0.74	0.02
Color level:	(3)	(4)	(1)	(4)	(4)	(1)	(4)	(4)	(2)
Expert rating:	M	H	L	H	H	L	H	M	L

Table 2: Metrics per Subsystem

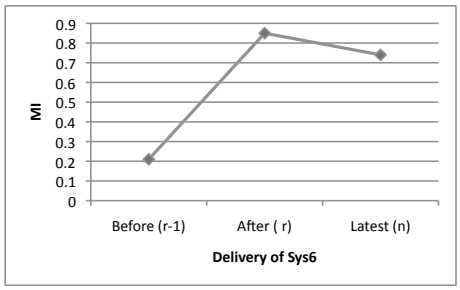


Figure 7: MI for Sys6 Before and After Refactoring

be proposed. This action may be modified by the designer to describe the solution in more detail. Each warning should have a status field, where it can be noted whether it has been fixed or not. Figure 8 is an example of how this could be documented.

As we would rather have the tool find false positives than miss real problems, all the findings that are not easily amended should be manually checked. If a finding is a false positive it should be possible to exclude it from the index. To avoid discarding real findings, a system expert should always approve the discarding of a finding. We consider human intervention to be important in this analysis, as there may be special circumstances that require exceptional coding practices.

Class/capsule	Warning	Action	Status
MyCapsule	Method size	Split method	Corrected
YourCapsule	Cyclomatic complexity	Split capsule, use active class	Planned for delivery 65
HisCapsule	Choice points per state	Split capsule, use active class	Planned for delivery 67
HerCapsule	Uses defer/ recall	No action, used safely	No action (approved by JK)

Figure 8: Maintainability Index Actions

We also strongly suggest that the organization defines a process for maintaining the model, including the definition of the MI function as well as its thresholds. The model should be evaluated regularly and adjusted when necessary. Both the overall maturity of the product and the business goals of the project may be taken into account when defining the model. When a product reaches high maturity, it may be more appropriate with stricter thresholds than when the first versions of the product are created and the business priority may be to hit a market window. The type of product and

the estimated time it needs to be under maintenance also affect this decision. A safety critical system with a total life time of several decades needs to display better maintainability than a mobile phone application, which is in use for only a couple of years. The model should be adjusted to reflect these circumstances.

4.1 MI Deployment Level (MDL)

As with any process improvement initiative, the deployment of the new method may take time and need gradual introduction efforts. In order to ensure that the level to which the maintainability index has been analyzed and acted upon is clear to both the team and the project, we suggest a maintainability index deployment level (MDL) to be measured. These are the suggested levels.

Lvl 1: the metrics have been collected.

Lvl 2: the metrics have been collected. Actions for the findings have been suggested.

Lvl 3: the metrics have been collected. Actions for the findings have been suggested and the minor ones implemented.

Lvl 4: the metrics have been collected. Actions for the findings have been implemented or planned for later implementation, including project approval.

Lvl 5: the metrics have been collected. Actions for all the findings have been taken.

Targets for when the subsystems should have reached the certain levels should be agreed upon.

4.2 Visualization

The MI and MDL can be visualized using a graphical visualization method familiar from, e.g., the balanced score card approach [15]. An example for two subsystems is displayed in Figure 9. For each subsystem, a gauge and a progress bar shows the current status of the subsystem. The cost to improve the MI by one step can be estimated, e.g., by looking at how much the measurements deviate from the threshold and based on historical data of the cost of similar changes. Initially, this number can be based on cost estimates given by the developers.

It should be possible to drill down into the subsystem to see exactly where the problem is located. In the example in Figure 10, we have drilled down

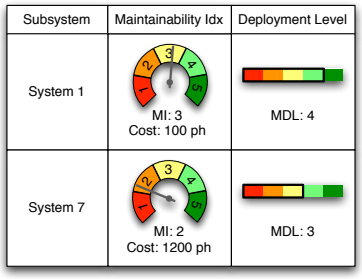


Figure 9: Maintainability Index Visualization

into the System 1 subsystem and are looking at the visual and code indexes and the number of warnings that are the cause for the index.

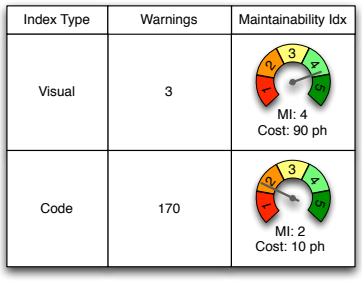


Figure 10: Maintainability Index Drilldown

We should be able to further drill down into the two different index types to see the individual capsules and warnings. For an example of this, see Figure 11.

5 Related Work

Different methods for measuring maintainability have been proposed and validated through empirical methods before. Many use an approach similar to ours, including a qualitative element, where subjects evaluate the design or code based on their experience and knowledge of good design [4, 19, 18, 3, 6]. The main difference is the fact that most studies are performed in a small setting with small example software systems (or "toy code application[s]" to quote Mäntylä and Lassenius [18]) and students as subjects, whereas

Warning	Value	Thrshld
Visual Warnings	**3**	
- Visual Size Warnings:	0	
- Visual Cyclomatic Complexity Warnings:	**1**	
- Capsule1C	**111**	**100**
- Choice Points/States Warnings:	**2**	
- Capsule2C	**1.125**	**1**
- Capsule3C	**1.083**	**1**
- Number of Entry Actions (Warning):	0	
- Number of Exit Actions (Warning):	0	

Figure 11: Maintainability Index Drilldown

our study was performed in an industry setting with real systems and experienced software industry professionals as subjects. Although academic studies are invaluable for furthering the state of the art, there are issues that arise specifically when the trying to apply the results in an industrial setting.

Coleman et.al. [6] provide methods for deriving a maintainability index for the purpose of decision support. They also calibrate their models using expert intuition. One difference between their approach and ours is the fact that our focus is more on early diagnostics and actionable metrics, whereas their approach partly relies on after-the-fact metrics such as effort. Although this can be estimated for diagnostic purposes, we chose to focus on a purely internal product metric for our purposes. As an effect our approach provides specific detailed improvement proposal support.

Moha et.al. [20] present a very complete method for specification and detection of smells. They rely on domain experts to identify and specify smells and how they are mesured in the specific context. In academic settings this can be rather easily acieved by, e.g., majority rule. It is our experience, however, that in an industrial setting it may be more difficult for the domain experts to reach consensus by voting, since the most experienced experts are most likely to have some attachment to the system, either by having been involved in the development of the system or at least by knowing they will be involved in it in the future.

Asthana and Olivieri [2] have done similar work in an industry context. Their "Software Readiness Index" has a very different purpose than ours, though. They look at the entire life-cycle of the software system and measure whether it is ready for release. Their focus is more on process metrics, whereas we are more interested in internal product metrics. Furthermore, they use quantitative data only and do not evaluate their results formally. There are many similarities between our approaches, especially due to our wish to provide an organization with decision support.

Our recommendation is to use an approach such as ours to help the organization understand what constitutes maintainability in their domain, capturing the experience of the domain experts and providing the organization

with decision support on where to focus their refactoring efforts.

6 Conclusions and Future Work

In this paper we present an approach to assess the maintainability of software systems and their need for refactoring. It is based on the collection of internal product metrics. This work was performed in the context of a large software system in the telecommunications industry and with the help of experts from the industry. The studied subsystems were written using IBM Rational Rose RealTime and C++.

The historic data on one subsystem gave us data points just before and after a major refactoring. It was clear that the maintainability of the subsystem had improved substantially due to this refactoring. This observation was confirmed by our model. Through our model, we could also observe that this subsystem had later seen a slight degradation in maintainability. This confirms the accepted fact that continuous refactoring is needed in order to preserve the maintainability of a software system.

Future work includes long-term evaluation of the impact on software maintainability of methods such as the one defined here, including the evolution of the model itself. The process paradigm in the context of this study was an iterative, incremental one. It would be interesting to see the impact of this method in other contexts. Heidenberg and Porres have presented a method for adopting this approach in an agile and lean context [13] as well.

Acknowledgments

Besides Jeanette Heidenberg and Iván Porres, the project team also consisted of Jussi Auvinen and Andreas Nåls. Marko Toivonen assisted the team with his enthusiastic CLISP coding, which provided us with the tool that we used to collect metrics for our study. We wish to thank the following people in M-MGw management: Stefan Blomqvist, Harri Oikarinen, Seppo Korhonen, Heli Hirvo, Marko Koskinen, Teppo Salminen, Hannu Ylinen.

References

[1] ANSI/IEEE. Std-729-1991: Standard glossary of software engineering terminology. Technical report, IEEE, 1991.

[2] A. Asthana and J. Olivieri. Quantifying software reliability and readiness. In *Communications Quality and Reliability, 2009. CQR 2009. IEEE International Workshop Technical Committee on*, pages 1–6, May 2009.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[4] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Eng.*, 27(6):513–530, 2001.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[6] Don M. Coleman, Dan Ash, Bruce Lowther, and Paul W. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[7] Ward Cunningham. The wycash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 29–30, New York, NY, USA, 1992. ACM.

[8] Ericsson. Ericsson media gateway for mobile networks (m-mgw). Available at `http://www.ericsson.com/products/hp/Ericsson_Media_Gateway_for_Mobile_Networks__M_MGw__pa.shtml`, 2006.

[9] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach.* PWS Publishing Co., Boston, MA, USA, 1997.

[10] F. Fioravanti and P. Nesi. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans. Softw. Eng.*, 27(12):1062–1084, 2001.

[11] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Jeanette Heidenberg, Andreas Nåls, and Ivan Porres. Statechart features and pre-release maintenance defects. *J. Vis. Lang. Comput.*, 19(4):456–467, 2008.

[13] Jeanette Heidenberg and Iván Porres. Metrics functions for kanban guards. In *IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2010.

[14] International Standards Organization. Iso 9126: Software engineering – product quality. Technical report, International Standards Organization, 2001.

[15] R. S. Kaplan and D. P. Norton. The balanced scorecard-measures that drive performance. *Harvard Business Review*, pages 71–79, Jan-Feb 1993.

[16] P. Kruchten. *Rational Unified Process*. Addison-Wesley, 1998.

[17] Martin Kunz, Reiner R. Dumke, and Andreas Schmietendorf. How to measure agile software development. pages 95–101, 2008.

[18] Mika V. Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 297–306, New York, NY, USA, 2006. ACM.

[19] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Softw. Engg.*, 11(3):395–431, 2006.

[20] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, 2010.

[21] Object Management Group. UML 2.0 Superstructure Specification. Technical report, OMG, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.

[22] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

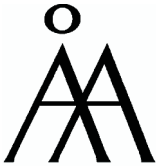
Turku Centre for Computer Science
TUCS Dissertations

101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming
128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2534-5
ISSN 1239-1883

Jeanette Heidenberg

Towards Increased Productivity and Quality in Software Development
Using Agile, Lean and Collaborative Approaches