TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Timo Lehtonen

Metrics and Visualizations for Managing Value Creation in Continuous Software Engineering

Julkaisu 1453 • Publication 1453

Tampere 2017

Tampereen teknillinen yliopisto. Julkaisu 1453
Tampere University of Technology. Publication 1453

Timo Lehtonen

Metrics and Visualizations for Managing Value Creation in Continuous Software Engineering

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 3rd of February 2017, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2017

ISBN 978-952-15-3899-5 (printed)
ISBN 978-952-15-3905-3 (PDF)
ISSN 1459-2045

Metrics and Visualizations for Managing Value Creation in Continuous Software Engineering

Doctoral Dissertation

Timo Lehtonen

January 12, 2017

Abstract

Digitalized society is built on top of software. The supplier of a software system delivers valuable new features to the users of the system in small increments in a continuous manner. To achieve continuous delivery of new features, new versions of software are delivered in rapid cycles. The goal is to get timely feedback from the stakeholders of the system in order to deliver business value.

The development team needs timely information of the process to be able to improve it. A demonstrative overview of the process helps to get better understanding about the development process. Moreover, the development team is often willing to get retrospective information of the process in order to improve it and to maintain the flow of continuous value creation.

The team uses various tools in the daily software engineering activities. The tools generate vast amount of data concerning the development process. For instance, issue management and version control systems hold detailed information on the actual development process. Mining software repositories provides a data-driven view to the development process.

In this thesis, novel metrics and visualizations were built on top of the data. The developed artifacts help to understand and manage the value creation process. With this novel, demonstrative information, lean continuous improvement of the development process is made possible. With the novel metrics and visualizations, the development organization can get such new information on the process which is not easily available otherwise.

The new information the metrics and visualizations provide help to different stakeholders of the project to get insight of the development process. The automatically generated data reflects the actual events in the development. The novel metrics and visualizations provide a practical tool for management purposes and continuous software process improvement.

Keywords: software visualization, software metrics, mining software repositories, value creation, software process improvement, continuous delivery

Preface

When I started this journey over ten years ago, I had very little knowledge on how science actually produces new information and how useful the mindset of scientific reasoning really is. There is no science without people. Luckily, I have had the opportunity to work with great colleagues in both academia and industry during all these years.

The most valuable guidance I have received from my two supervisors, professors Hannu-Matti Järvinen and Tommi Mikkonen from the Department of Pervasive Computing in Tampere University of Technology (TUT). Also all the colleagues in paper related communication channels have been a key factor for the advancements of this work. First of all, Timo Aho (Yleisradio, Finnish Broadcasting Company) and Timo Aaltonen (TUT) have guided this work especially from the data science point of view. Solita ltd. has been an essential enabler for this work. From Solita, Timo Raitalaakso and Mikko Puonti have been pioneers in Solita Science program and have supported this work by defining the constraints and deadlines for advancements. Timo Honko (Solita) has been in key role for supporting this work from the industrial side by creating the opportunities to advance academic cooperation. I would also like to thank Petri Sirkkala, Ville Marjusaari and Janne Rintanen for their great input to support empirical feedback iterations of the work.

Moreover, the research colleagues at TUT have provided their help to put this work forward. Sampo Suonsyrjä has provided a lot of support for the research methods of this work. Terhi Kilamo, Kati Kuusinen and Laura Hokkanen have brought in their great knowledge on how to actually advance in the process towards dissertation. Anna-Liisa Mattila, Essi Isohanni and Petri Ihantola have helped in defining the scope for this work. Paavo Toivanen has given a lot of useful hints for writing in English.

Through the national Tekes-funded Need for Speed (N4S) research program, I have had the chance to meet many skilled researcher groups. Pasi Kuvaja and Lucy Lwakatare from University of Oulu have supported this thesis with their strong argumentation about the topic. Emilia Mendez and Ville Leppänen gave great feedback with top expertise during the pre-examination

phase. Moreover, Juha Itkonen, Raoul Udd and Casper Lassenius from Aalto University, have brought in their advanced knowledge on academic work related to the research on long-term changes in software engineering. Great discussions with Jürgen Münch about the value creation processes has created great insight to the topic. Moreover, many other colleagues have been important for this work to be done.

My family has supported this work in many ways. Thanks to my parents for their support. They showed me the way to the field of software engineering. My wife Heidi has been a great help in supporting the decisions in order to finish this work in time. She has guided the work towards the final published version in an agile manner.

Contents

List of Figures

List of included publications

[P1] T. Lehtonen, S. Suonsyrjä, T. Kilamo, T. Mikkonen. Defining Metrics for Continuous Delivery and Deployment Pipeline. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST)*, 2015.

[P2] P. Tyrväinen, M. Saarikallio, T. Aho, T. Lehtonen and R. Paukkeri. Metrics Framework for Cycle-Time Reduction in Software Value Creation. In *The Tenth International Conference on Software Engineering Advances (ICSEA)*, 2015.

[P3] T. Lehtonen, S. Suonsyrjä, T. Kilamo, T. Mikkonen. Continuous, Lean, and Wasteless: Minimizing Lead Time from Development Done to Production Use. In *Euromicro Conference series on Software Engineering and Advanced Applications (SEAA)*, 2016.

[P4] T. Lehtonen, V. Eloranta, M. Leppänen, and E. Lahtinen. Visualizations as a Basis for Agile Software Process Improvement. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, 2013.

[P5] A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen, and K. Systä. Mashing Up Software Issue Management, Development, and Usage Data. In *Proceedings of the 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE)*, 2015.

[P6] T. Lehtonen, T. Aho, T. Mikkonen, and K. Kuusinen. Visualizations for Software Development Process Management. In *the 26th International Conference on Information Modelling and Knowledge Bases (EJC)*, 2016

Author's contribution to the publications

In Publication P1, the candidate was the first author and the key researcher to collect the data available for the metrics from an industrial case project which was then used to develop new metrics in cooperation with the other authors.

In Publication P2, the candidate was the fourth author in role of data collection and analysis together in a cooperation with the main authors to connect the industrial data to a wider value oriented framework of metrics.

In Publication P3, the candidate was the first author and conducted the data collection and analysis. Moreover, the candidate applied the existing metrics presented in the paper to software engineering context.

In Publication P4, the candidate planned and conducted the study and interviewed the customer project management personnel in cooperation with the researchers from Tampere University of Technology.

In Publication P5, the candidate was the second author and the role was to collect and analyze the data, create the visualizations and explain their meaning in an industrial context.

In Publication P6, the candidate was the first author that planned, conducted and collected the data for the research. The visualization artifact was developed further by the candidate.

Chapter 1

Introduction

We are using digital devices all the time. The software in them is continuously updated. New versions of software with new functionalities and fixes are continuously delivered to the end-users.

The software development process is a value creation process [140]. In *continuous software engineering*, the release frequency has gone up [16]. Value is created iteratively in small increments by delivering new versions of software continuously. Techniques and practices of continuous delivery, continuous integration (CI) and continuous deployment produce rapid cycle feedback to the organization which continuously develops new features to software.

Agile methods and practices in software development have been widely adopted [48]. The goal of a software development process is to produce *business value* to the stakeholders of the software system. However, term *business value* has no rigorous definition [140]. In feature-driven development [128], the delivery of new features is considered to create value. Furthermore, the actual usage of the features or value-in-use [61] is a tangible mechanism for value creation.

Delivery of new features is often achieved with a deployment pipeline, which consists of computing resources that, among other purposes, perform continuous, automatic testing to the change sets committed to the software [73]. The developers of the system continuously integrate their work and deliver changes to the numerous environments of the pipeline. The purpose of the several environments of the pipeline is to provide timely feedback for stakeholders who participate in the development of the system.

The development team utilizes several tools in the development work. When the developers use the tools in their daily work, a large data set concerning the actual software development process is generated as a side effect. For instance, a version control system and an issue management system are often used. This data can then be mined and analyzed. Moreover, a logging

tool produces the data of the actual usage of the features. Mining software repositories [68] provides a data-driven view to the development process. The analysis produces new information about the deployment pipeline and the underlying software development process. The information can then be utilized for software process improvement (SPI) [159] purposes.

Lean software development, which is tightly connected with agile software development [38], puts emphasis on continuous improvement. The development process is continuously improved in order to adapted to any external changes. Any actions that do not create value, i.e. actions that are *waste*, are constantly eliminated. The analysis of data set generated by the development tools helps to recognize sources of waste. The analysis provides information which helps to improve the development process.

Loss of management control is one of the greatest concerns when adopting lean software development methods [153]. Novel tools for measuring and demonstrating progress may help in software process management. The characteristics of the development process can be understood based on the traces the development tools leave.

Humble and Farley [73] define cycle time or the time between two subsequent releases as the most important metric in software delivery. They refer to Poppendiecks' question [136] "How long would it take your organization to deploy a change that involves just one single line of code?" They state that cycle time tells more about the process than any other metric. Moreover, the importance of cycle time has been presented in numerous white papers and blogs [145]. In this work, the metrics and visualizations focus on enabling the improvement of cycle time.

1.1 Aims and scope

The main goal of this thesis is to develop automatic, data-driven metrics and visualizations for managing value creation in continuous software engineering. Value creation is a lively, difficult and richly articulated research field in the software engineering community [141]. The definition for *value creation* in this context is based on three points of view. First, the software development process is seen as a value creation process since a key characteristic of any software development process is its explicit focus on value creation [140]. By managing the development process, value creation is managed. Second, value-in-use [61] emphasizes the customers' perspective for value creation. In this context, it means the actual usage of the features developed. Value is created by delivering features that are used by the users. Third, the decisions related to selecting which features to include in the system being developed

are not taken into consideration in the context of this thesis. It is assumed that the most valuable features have already been selected. The focus is in the development process and the actual usage of the selected features. Figure 1.1 demonstrates the scope of this work in more detail.

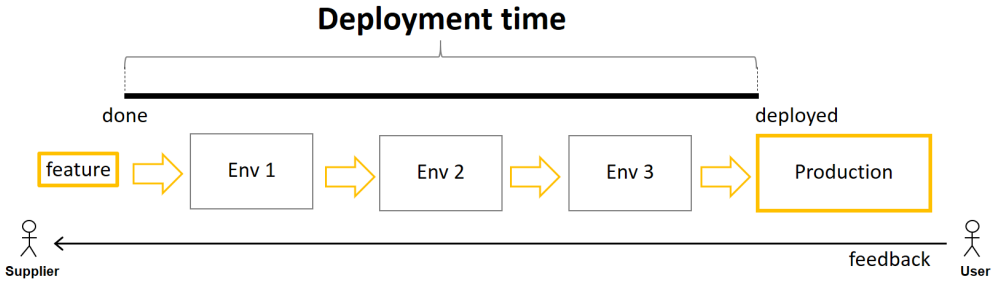

Figure 1.1: The supplier releases new features continuously to get fast feedback from the users.

The diagram depicts the flow of a single new feature from development to production usage. On the left, the supplier implements a new feature to software. The process for choosing the features to be implemented is outside of the scope of this work. Apparently, features with high business value have been chosen and this decision has been made by some stakeholder of the system, for instance, by the customer, the supplier, or the agile Product Owner role [149]. When the development of a feature is *done*, the feature is then continuously integrated with other features in the numerous execution environments of the system (Env 1, Env 2 and Env 3 in the diagram). At this stage, in case of continuous software engineering, the supplier gets rapid cycle feedback from the CI system. Moreover, acceptance testing may be performed by the customer, for instance. Finally, on the right, the new feature is deployed to the production environment. *Deployment time*, which measures the time from development done till the production deployment, is over. This novel metric acts as a key metric in this work. Then, after a while, a user may use the feature. The supplier then gets the feedback of the actual usage of the feature. Some feedback can be acquired from the production logs even without contacting the users directly. For instance, if there are bugs in the implementation, information on them is acquired through the logs. Moreover, information on the actual usage frequency of the features can be acquired by mining the logs.

The goal of this work is to develop novel data-driven metrics and visualizations which characterize the software development process. The purpose of the metrics and visualizations is to create a basis for improvement. The

proposed metrics and visualizations help to reduce cycle time [73] in order to get rapid cycle feedback from the users to the development phase. Moreover, the metrics developed help the development organization to guide their work towards value creation. With the developed artifacts, the development team is able to get information on the process. They get a novel basis for improvement of the process by the information provided by the visualizations and the metrics.

1.2 Research questions

A key constraint for the developed artifacts in this context is that the data for them is generated automatically during the software development process. No extra work is needed to produce the data. The research questions are:

- RQ1: What metrics help to eliminate waste in continuous software engineering?
- RQ2: How to construct visualizations to demonstrate value creation in continuous software engineering?
- RQ3: How to manage value creation with metrics and visualizations based on automatically generated data?
- RQ4: Which data for metrics and visualizations is automatically generated by the tools used in software development?

The research questions are addressed with empirical evidence from an industrial context by applying a methodology consisting of several quantitative and qualitative methods. The research has been conducted in a mid-sized Finnish software company, Solita ltd., which provides digital business consulting and services to its customers. The main research methods applied are Action Research [4] and Design Science Research [129] with a data-driven approach [31] and support of qualitative methods, for instance, thematic analysis [169]. The focus in this work is in quantitative methods because of objectivity. A data-driven approach is used in order to produce objective information on the target of the research. Novel quantitative metrics provide objective new information on the target of analysis. Moreover, information visualization is a very powerful tool that extends the cognitive capabilities of the human mind. Visual representations automatically support a large number of perceptual inferences that are extremely easy for humans [98]. By presenting the data visually, a high bandwidth channel from the computer to

the human brain is opened [98]. The combination of methodology consisting of quantitative and qualitative approaches with the mindset of Design Science targeting to *utility*, not truth [168], creates a solid methodological basis for the work. The key artifacts of the work have been developed in an iterative manner in an industrial context where the results have been constantly validated both in the industrial context among practitioners and among the research oriented audiences on the academic side.

There are existing solutions related to applying both information visualization and metrics to software engineering data in order to improve the process. For instance, a cumulative flow diagram (CFD) provides similar kind of information on the development process as the visualizations presented in the publications of this compilation. However, the artifacts developed in this thesis, contain more information related to continuous software engineering. For instance, cycle time is included as extra information in the visualizations developed. Moreover, the relationship between cycle time, batch size and feedback speed related to a parallel development process is highlighted in a novel way in the visualizations of this work. The developed metrics and visualizations construct a novel holistic basis for software process improvement.

1.3 Results and contributions

The thesis contributes towards novel metrics and visualizations for continuous software engineering. The main results are threefold.

First, the contribution consists of a metrics framework developed. The framework constructs a basis for continuous improvement of a software development process. The metrics framework presented in this work helps to manage continuous value creation in software projects. The key metric, *deployment time*, can be used as a tangible tool for software process improvement. Moreover, when the *deployment time* of thousands of features during several years is shown in the visualizations, new insight to the evolution of the process can be constructed.

Second, the visualization artifacts developed in this work construct a demonstrative basis for understanding and managing a software development process. The acquired new information can be utilized to improve the process. For instance, feedback speed, batch size and cycle time can be evaluated based on the visualization. For instance, a retrospective meeting of an agile project might benefit from the novel information. Figure 1.2 is introduced in more detail in Chapter 5. The visualization consists of a huge amount of data concerning the *deployment time* of features. This kind of visual image could be utilized for software process improvement (SPI) purposes.

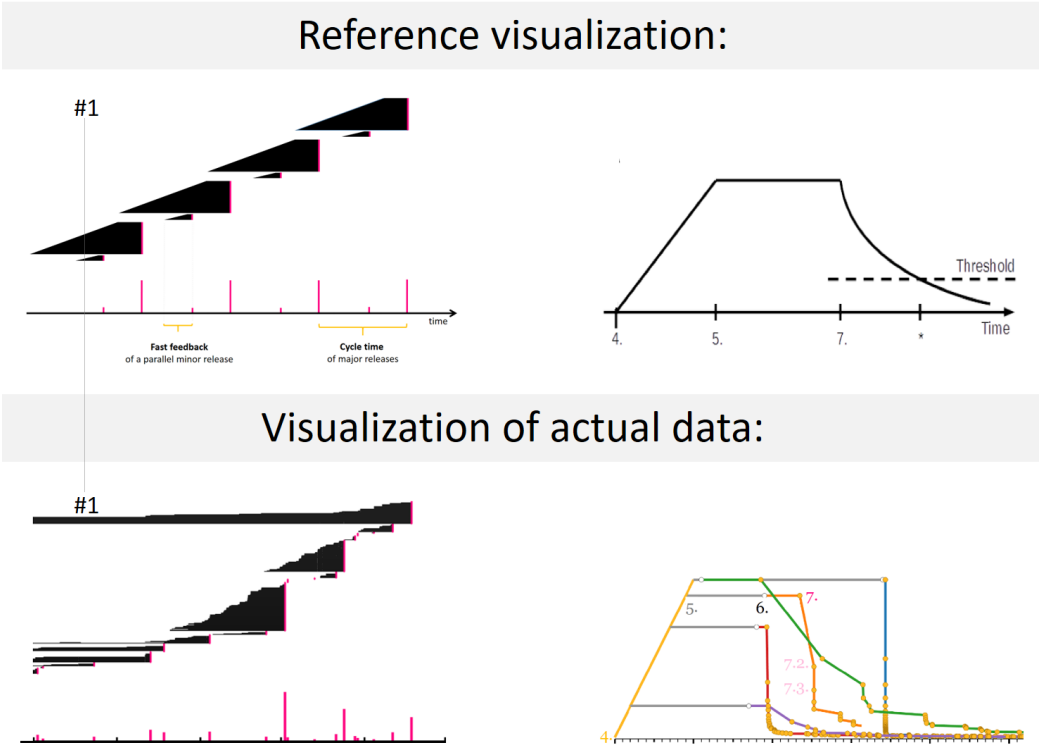

Figure 1.2: Visualization of the reference process and the actual process.

Finally, the contribution consists of demonstrating the existence of novel software engineering phenomena in an industrial context. For instance, continuous delivery and continuous deployment [73] have been demonstrated visually. The empirical evidence of the existence phenomena of this kind in an industrial context is a result itself. Information visualization provides a high bandwidth channel from the computer to the human [98]. Visual imagery in general is a powerful cognitive system with parallel processing capabilities while the verbal system processes sequentially [127]. For instance, in Figure 1.2, at spot `#1`, it is easy to point out a visual difference between the reference software development process and the actual process. The semantics of the long tail among other visual indicators for the characteristics of the process, is presented in detail in Chapter 5.

1.4 Structure of thesis

The thesis is structured as follows. Chapter 2 introduces the research approach, which consists of the industrial context and the research methodol-

ogy applied. Chapter 3 presents the relevant background of the research field. The definitions and existing knowledge in the field of software engineering related to value creation, software analytics, mining software repositories and metrics are presented. Related work is presented in Chapter 4. The results are presented in Chapter 5 and discussed further in Chapter 6 with possible scenarios for future work. Chapter 7 presents the concluding remarks. Finally, the publications that construct this compilation, are presented.

Chapter 2

Research approach

In this chapter, the research approach is introduced. First, the industrial context in which this research was conducted, is introduced. Then, the research methods used in this work are presented. First, we introduce Action Research, which was used as a research approach in the publications of this work. Then, we introduce Design Science, which targets to utility of the results. Then, we reflect the work to different categories of theories. Finally, qualitative methods applied are presented.

2.1 Industrial context

The research has been conducted in a mid-sized Finnish software company, Solita ltd.[1]. The company provides digital business consulting and services to its customers both in public and private sectors. During its nearly 20-year long journey, the company has completed over 1000 projects. Some of the projects were used as case projects in this research.

The high-tech case company has provided many state-of-the-art technological solutions available for research. The company is eagerly adopting the newest useful technologies into use. For instance, *continuous integration* introduced by Fowler in 2006 [52] was adopted to the organization in 2007. Term *deployment pipeline* introduced by Humble and Farley in 2010 [73] was introduced in the company in 2011. Since then, *continuous delivery* has been a standard practice in the customer projects. Nowadays the company is going beyond *DevOps* [6] and continuously adopts new concepts. Academic co-operation in a national Tekes[2] funded Need for Speed[3] research program

[1] http://www.solita.fi/en/
[2] http://www.tekes.fi/en/
[3] http://www.n4s.fi/en/

is continuously importing new knowledge from the academia to the company and vice versa.

The role of the researcher in the case projects has been two-fold. Firstly, the researcher has acted as a team member in some of the case projects. The role has been to design and develop software with skilled colleagues in customer projects. This kind of position has given good possibilities for designing novel and relevant research settings. Secondly, the role has been to conduct research in the projects. The combination of two separate roles has provided a chance to observe the case projects extensively. Industrial data available has provided good circumstances for acquiring the empirical evidence of contemporary software engineering phenomena. A suitable research methodology for this kind of setting is presented in the following.

2.2 Action research

Action Research is a collaborative method that can be applied to the cooperation of researchers and practitioners and adapt to the process [4]. Action Research is a suitable approach for working in a complex research environment.

The key idea in Action Research is to make academic research relevant as the researchers should try out their theories with practitioners in real situations and real organizations [4]. Action Research consists of the following three steps [4]: diagnosis of the problem, action intervention and reflective learning. The first step in Action Research of diagnosing the problem consists of creating an overall picture of the status quo. This is followed by an action intervention and then followed by reflective learning. Action Research is continuous and iterative in nature and stops when a satisfactory result has been achieved.

In Action Research, the emphasis is on what practitioners do rather than on what they say they do [4]. In software engineering, the tools the practitioners use in their daily work produce a vast amount of data that can be analyzed. The data analysis produces detailed information on what has been actually done. In this sense, Action Research is a suitable method for the research conducted.

In article "Action research is similar to design science" [79], the similarities between Action Research and Design Science are discussed. A conclusion is drawn that Action Research and Design Science are similar. Much advice can be taken from Design Science in the ways how to validate the Action Research study and what to include to the study report.

2.3 Design science research

Science, research and design are related to each other in multiple ways [121]. One of the modern methodologies in Information Systems (IS) research is Design Science, which has been widely adopted to the IS research community [129]. During recent years, several researchers have succeeded in bringing Design Science research into the IS research community, making Design Science a promising IS research paradigm [129].

A number of researchers have provided guidance to define Design Science [129]. Peffers et al. [129] define Design Science Research Methodology for the production and presentation of Design Science research in IS. Peffers et al. [129] define the key activities of applying Design Science to a research problem by taking into account the boundary conditions, for instance the requirement stated in [168] of addressing only important and relevant problems or the system objectives or meta requiremenents in [170].

The activities in Design Science presented in Figure 2.1 [129] are briefly described in the following. In the case of this research, the iterative creation of the artifact occurred in several design and development activities together with active demonstration, evaluation and communication. The results have been continuously demonstrated, evaluated and communicated to several professional audiences both in the industry and the academy.

Activity 1: Problem identification and motivation. In this activity, the problem is defined and the value of a solution is justified.

Activity 2: Define the objectives for a solution. The objectives, which can be either quantitative or qualitative, are specified. The objectives should be inferred from the problem specification.

Activity 3: Design and development. In this activity, the actual artifact is created. The architecture and the design of the artifact are developed.

Activity 4: Demonstration. The artifact is demonstrated in a context where the goal is to solve one or more instances of the problem specified.

Activity 5: Evaluation. Based on the demonstration, the artifact is observed and measured in order to evaluate how well it supports the solution to the problem.

Activity 6. Communication. The problem, its importance and effectiveness are actively communicated to researchers and other professional audiences.

In the following, the iterations related to design science approach are presented in more detail. The activities in the list are presented step by step.

2.4 Design science iterations

The definition of the problem occurred early in the research process during the first publications of this compilation. The problem to be solved was to demonstrate a software development process visually in an efficient manner that would enable process improvement. The need emerged from the customer project presented in Publication P4. In general, agile methodologies put emphasis on retrospective reflection and making problems visible in order to learn from the past. This acted as a starting point for the development of the visualization artifact. In the first publication, the visualization artifact was developed from a first draft to visualization I presented in Figure 2.1.

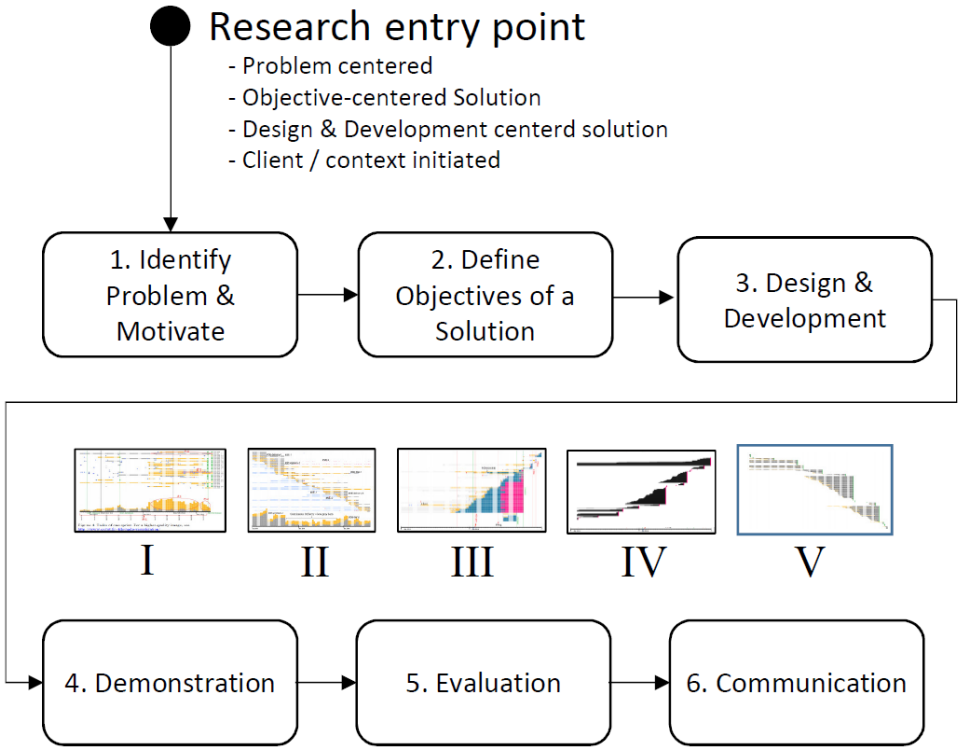

Figure 2.1: Design science Process Model applied from [129] to the iterative development of the visualization artifact.

The design process of the visualization artifact presented in this work is presented in Figure 2.1. The series of visualizations I, II, III, IV and V are the result of applying the design science research methodology cycles to the data in industrial context. The work done in Publication P4 was developed further in Publication P5. The feedback from the previous customer case

project was taken as a basis for further development. In visualization II in Figure 2.1, there were multiple data sources for the visualization. In addition to issue management data, two more data sources were utilized. The visualization packs a huge amount of data from a version control system and production logs to a single visualization. At this point, the personnel of another case project were utilized for getting feedback from professional software engineers. In this case, there was no customer involved since the case project was an internal product developed at the software company. Then, in the next publication the visualization artifact was again applied to another context. The architect of the development team gave an idea of using a triangular shape for the visualization. This occurred after multiple feedback sessions with the team, the project manager and the architect. Then, in visualization IV in Figure 2.1, the existing visualization was simplified to contain only key information. This visualization was then shown to an agile coach in an interview with a thematic analysis. By utilizing the feedback, visualization V with layout of a Gantt diagram could be introduced. However, more feedback from the actual users of the visualization should be acquired in order to develop the artifact to right direction. A suitable methodology for future improvement could be found in the field of human computer interaction (HCI) [35] or user experience (UX) [69].

The case projects have been a target of both demonstration and evaluation of the visualizations. The communication through both customer interaction and scientific conferences have provided valuable feedback to the further design and development work of the visualizations. Moreover, communication with other stakeholders in the industrial context has provide valuable feedback. In all the phases shown in Figure 2.1, the feedback loop from a single phase to the development of the artifact has been constant. During the iterations, the developed artifact has successfully managed to solve the problem of demonstrating a software development process visually. The visualization has brought new insight the target processes of the customer cases.

The work done related to metrics in Publication P1, P2 and P3 has supported the development of the visualization artifact in two ways. Firstly, the metrics developed provide the same information as the visualization from a single feature point of view. The visualization artifact actually combines multiple values of the metrics into a single figure. For instance, the horizontal lines depicted in visualization IV in Figure 2.1 tell the value of metrics *development time* presented in publication P1. The visualization packs a huge amount of data into a single presentation which makes new inference possible. Moreover, the key metrics of the framework presented in Publication P2 are depicted in the visualization II. For instance, the value of metric *core*

cycle (from development done to first usage of the feature) can be easily observed from the visualization in Publication P5.

2.5 Qualitative methods applied

The methodological approaches presented in this chapter were supported by a qualitative approach. An interview with a thematic analysis was conducted. A thematic analysis is an approach often used for identifying, analyzing, and reporting patterns within data in primary qualitative research [29]. Dybå et. al [29] present the concept of thematic synthesis applied to the field of software engineering. They provide a step by step guide for applying the method. They define the steps of extracting data, coding data, translating the codes into themes, creating a model of higher-order themes and assessing the trustworthiness of the synthesis.

In this work, the method of thematic analysis was applied to interpret how the interviewed subject understands the visualizations. The results and discussion related to the tangible findings with this method are presented in Chapter 6.

2.6 Categories of theories

Information systems are implemented within an organization in order to improve the effectiveness and efficiency of the organization [168]. Characteristics of the information system in the organization and its work systems and people together determine the extent to which that goal is achieved. Acquiring such knowledge involves two complement, distinct paradigms, namely Behavioral Science and design science [108], where the Behavioral Science has its roots in natural science research concerning, for instance, principles and laws that explain or predict organizational and human phenomena. This is related to the classification presented by Gregor [60] of theory types. According to Gregor, there are five categories of theories:

Type I – a theory for analysis and description. The question is: "what is". A theory in this category describes and classifies the features or properties of individuals, groups, situations or events. The findings of single cases are summed up to a more general case.

Type II – a theory for understanding. The questions are: "how" and "why". There are two subtypes of theories in this category. Firstly, there are theories that can be used to find surprising observations of phenomena. Secondly, there are theories that contain conjectures about the reasons for

the events of real-world issues. Methods suitable are for instance, case study, phenomenological and ethnographical overviews.

Type III – a theory for prediction. The question is: "what will be". The causalities between the input and output may not be totally understood. Research methods that are suitable for this kind of theories are: statistical analysis, for example correlation and regression analysis.

Type IV – a theory for explaining and prediction. The questions are all the questions from types I, II and III combined. Many researchers understand this as a traditional way of a theory. The suitable research methods are: grounded theory [22] in addition to the combination of the research methods of type I, II and III.

Type V – a theory for planning and acting. The questions for this category are not provided by Gregor. A theory in this class has two types of aspects. Firstly, the methods and tools used. Secondly, the design principles including design information and design decisions, where the latter are meant to be included in the built artefact, method, process or system.

This work contributes towards Gregor's categories *Type I* and *Type II* theories. The artifacts and principles developed in this work can be used for analysis, description and understanding about a software development process. The patterns of the process can be revealed by the visualizations created in this work. When the patterns and properties of the process are visible, the process is easier to be analyzed and described. The visualizations help to write descriptions of the process details, because they work as a graphical reference for the verbal narratives describing the process. Furthermore, the process can be understood better. Based on the demonstrative information the metrics and visualizations provide, the human mind is able to create a holistic conception of the information.

To combine the point of view presented by Gregor [60] with the design science research presented in [168], the Type IV category of theories corresponds to the behavioral-science paradigm part of explaining and predicting organizational and human phenomena. The design science paradigm having its roots in engineering is fundamentally a problem solving paradigm, which focuses in creating innovations, practices and technical capabilities, and products through which the analysis, design, implementation, management, and use of information systems is effectively and efficiently accomplished [33]. The creation of such artifacts relies on existing kernel theories applied by the researcher that solves the problem.

Bock [13] presents four categories of knowledge: speculative, presumptive, stipulative and conclusive knowledge. The knowledge the visualizations create is often speculative. The observations made from the holistic visualizations are for the most part opinions formulated by individual people. On

the other hand, the visualizations reveal many facts about the process, since many of the data sources provide the very accurate data of the software engineering events. Moreover, particularly the metrics provide stipulative knowledge as numerical facts describing the actual low-level software engineering events.

The complexity of creation of new artifacts due to the growth of knowledge [13, 168] forms an ecosystem where the applications of new technologies are built on top of the existing yet novel applications. The resultant artifacts extend the boundaries of human problem solving capabilities by providing new intellectual and computational tools.

Design science research has an emphasis on *utility* while traditional scientific research methods focus on *truth* [168]. Moreover, truth and utility are inseparable and an artifact may have utility because of some yet unknown truth. The research conducted in this thesis related to information visualization is based on visual representations of information in order to create a basis for communication. Moreover, visualizations help to understand the data available and for acquiring new knowledge. In Design science, representations have a profound impact on design work [168]. For instance, the field of mathematics was revolutionized with the constructs defined by Arabic numbers, zero, and place notation. Furthermore, the search for an effective problem representation is crucial to finding a solution based on design [173]. On the other hand, theory on information visualization [21] treats visual representations as a process for defining new languages and possibilities for other human beings to learn the symbols and conventions of the language, and the better we learn them, the clearer that language will be [21]. Diagrams are effective in the same way as the written words on this page are effective – the human brain uses its high bandwidth capabilities [98] to acquire the knowledge produced with data ink [157] or the ink that represents data – text or figures.

2.7 Summary

The research has been conducted in an industrial context. The case company has provided a fruitful environment for empirical research and for applying various research methods. Action Research and Design science have been effective, iterative research methods for a complex industrial environment. Qualitative methods have supported the other methods applied. The goal has been to develop the utility of the designed artifacts with the chosen methodology in an iterative and continuous manner. To get feedback, the artifacts have been continuously demonstrated to several audiences in the

industry and the academy. Several feedback cycles to develop the artifacts further have been conducted. Following the Design science methodology, the target has been in utility of the results. The developed visualization artifact has been tested with several stakeholders consisting of experts in the field of software engineering. According to their feedback, the goal of utility has been reached. The contributions of the research approach are related to Gregor's *Type I* and *Type II* categories of analyzing and understanding phenomena in an industrial context. With the support of qualitative methods, the methodology applied in this work constructs a solid basis for the contributions presented in this work.

Chapter 3

Background

This chapter introduces the background. First, we present continuous value creation from software engineering point of view. We start by introducing the topic of continuous software engineering and advance towards lean continuous improvement. Second, software analytics is presented. We cover the topics of data analytics, information visualization and mining software repositories.

3.1 Continuous value creation

In this section, we introduce the topic of value creation in feature-driven software development. We continue by introducing the concepts of continuous software engineering, continuous integration and continuous delivery. Then, the topics of software process management and improvement are introduced. Finally, metrics for supporting continuous improvement are introduced.

3.1.1 Value creation in software engineering

Value creation is a richly articulated research field in the software engineering community [141]. Many organizations base their software development method on agile and lean principles [38]. Lean software development is tightly connected with agile software development [38]. In their first book "Lean Software Development – An Agile Toolkit" [135], Mary and Tom Poppendieck present the Agile manifesto [8] as a shift of perception of value. They state it as a shift from process to people, from documentation to code, from contracts to collaboration and, from plans to action. In agile software development [37], delivering business value is the heartbeat that drives, for instance, XP projects [7]. Moreover, the key goal of the widely applied Scrum method [149] is to deliver business value. However, while term business value

is used in the software intensive industry extensively, it has no rigorous definition [140]. Either value creation in software engineering does not have a single rigorous definition.

Marketing literature and practice present the idea that, especially when it comes to services, customers play foundational roles in value creation mechanisms [126]. According to service dominant logic (SDL), the customer is not the target of value [126] but an active stakeholder in value creation and a co-creator of value [166]. From this point of view, the supplier designs, develops and delivers *potential value* and exchanges it with another stakeholder [64]. The production process of potential value overlaps with the customer's co-creation participation [64]. Then, through actual usage of the service, *value actualization* [63] takes place. In this sense, supplier's value facilitation is seen as a foundation for customers' value creation [64]. The production of resources by the supplier generates only potential value for the customer [61]. The role of the firm is to facilitate the value creation process by providing supporting resources for the customer's use [155].

According to Grönroos and Voima [61], value creation refers to two points of view, namely *value-in-exchange* and *value-in-use*. Value-in-exchange is the value observed from the provider point of view while value-in-use emphasizes the customers' perspective and the actual usage of the service. Figure 3.1 presents two spheres for value creation by Grönroos and Voima [61].

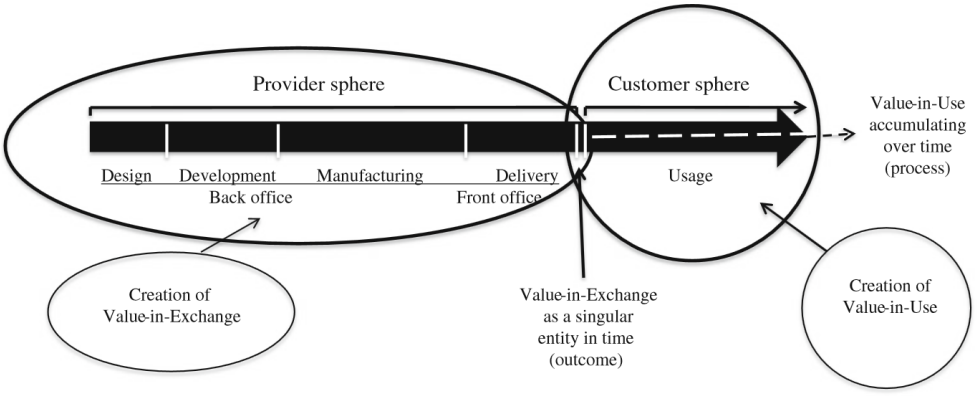

Figure 3.1: Provider and customer spheres where value-in-use and value-in-exchange occur. Source: [61].

Firstly, the provider sphere consists of steps design, development, manufacturing and delivery. The provider produces value that can be exchanged with another stakeholder. Secondly, the customer sphere consists of value-in-use i.e. usage of the service by the customer. In software engineering,

feature-driven development [128] is one approach to design and deliver valuable changes to software. Boehm [11] presents Value-Based Software Engineering (VBSE) where the emphasis is in considering the value propositions of implemented software components to various stakeholders. Boehm even mentions visualization techniques as an approach for stakeholder value proposition reconciliation.

A feature is a piece of functionality, something that delivers value to the user [75]. Features are new functionalities or bug fixes, for instance [75]. Features are often managed in an issue management system, for instance Jira[1]. Each new feature to be implemented can be presented as a single task in the system.

In practice, features are often implemented as source code changes committed to a version control system (VCS). Earlier, centralized version control systems, for instance RCS [156] and Subversion [26] were used. Nowadays, distributed version control systems (DVCS), for instance Git [104], are widely used. The impact of distributed version control systems compared with centralized in terms of committed software changes is impressive. Even 30% higher productivity has been reported [18] by using a distributed version control system compared with a central approach. This is achieved due to the possibility to commit changes locally, which leads to a more fluent flow in programming.

When the development team uses a version control system, they need a suitable branching model. For instance, the flow branching model presented by Driessen [36], can be chosen. In the Driessen model, new features are implemented as separate feature branches that are merged to the develop branch and then to a release and the master branch. By applying this kind of a branching model, a clear separation between the commits related to different features is achieved.

When a feature has been implemented, the changes are delivered to the users of the system in order to actually produce value-in-use. To achieve this, several tools and techniques presented in the following are often needed.

3.1.2 Continuous software engineering

In *continuous software engineering*, the release frequency has gone up [16]. Continuous software engineering resembles the concept of flow found in lean manufacturing [49]. Adopting a continuous approach to software engineering enables the development organizations to move towards continuous value creation with a continuous experimentation approach [45], for instance. Fitzger-

[1]https://www.atlassian.com/software/jira

ald et al. [49] state that a useful concept from the lean approach, namely that of 'flow', is useful in considering continuous software engineering. In continuous software engineering, software development is not a sequence of discrete activities [49]. Rather, development is a set of actions which mimic the concept of lean thinking [176]. In lean thinking, value is defined by the ultimate customer, and it is created by the producer [176]. The product is constructed with a flow from raw material to the customer [176] as a continuous movement [49].

Continuous integration

Continuous integration (CI) is a set of tools and practices that automatically give feedback to the developers of each change committed and pushed to the version control system (VCS) [18]. A CI system can be seen as a feedback system for the committed change sets. Automatic commit stage testing [73] provides the developers a short feedback cycle and a quality assurance system for supporting the development work. The length of the feedback cycle on the commit stage is often minutes, which makes continuous improvement possible. In case of, for instance, a compilation error in the committed change set, the report is available in minutes and the fix is often committed in minutes [52]. The fundamental practices related to continuous integration among other agile software development practices, have a strong effect on the motivation of the development team [175]. The use of physical artifacts to present the status quo, for example, such as interactive wall charts is a key factor in the development team coordination and motivation [175]. Moreover, an information radiator or a screen in the team workspace showing the information on project status, provides a practical continuous feedback loop from the CI system to the developers [46, 133]. In this sense, the use of physical artifacts strengthens the impact of CI.

Continuous delivery and deployment

Continuous delivery builds on top of CI [51]. Continuous delivery (CD) is a set of tools and practices to implement software in such a way that the software can be released to production at any time [51]. Fowler [51] introduces four criteria for continuous delivery. Firstly, software is deployable throughout its life cycle. Secondly, team prioritizes keeping the software deployable over working on new features. Thirdly, an automated rapid cycle feedback on the production readiness of after any changes to the software is present. Finally, anyone can perform a single click deployment of any version of the software to any environment. In their experience report, Neely et

al. [122] accompany the criteria defined by Fowler. They define continuous delivery as "the ability to release software whenever we want". Moreover, they point out that frequency is not the key factor – it is the ability to deploy at will. They see continuous delivery as a requirement for continuous deployment i.e. the software is in such a condition that it can be deployed at any time.

Humble and Farley [73] present *deployment pipeline* as an automated manifestation of the process of getting software from the VCS into the hands of the users. They define it as a holistic, end-to-end approach that holds the build, deploy, test and release processes for delivering software. They end up in a lean pull system where different stakeholders can deploy builds into various environments at the push of a button. According to Leppänen et al. [100], achieving continuous delivery comes from establishing a productized pipeline with adequate tool support and short setup time. A suitable infrastructure enables small batches [143].

Continuous deployment, as Humble and Farley [73] put it, is a practice where every change to the source code of a system is delivered immediately to the hands of users. However, according to a recent mapping study by Rodriguez et al. [145], most of the scientific literature uses the terms continuous deployment and continuous delivery interchangeably. They build the concept of CD to three major themes of deployment, continuity and speed. Firstly, deployment means the ability of bringing valuable product features to the customer. Secondly, continuity can be seen as the series or patterns of deployments that aim at achieving a continuous flow. Finally, speed is about shorter lead times. However, Fitzgerald et al. [49] argue that speed is not everything. They refer to Taiichi Ohno's point of view [125]: a more consistent flow of slower continuous changes is better than a speedy race which occasionally stops to doze. Continuous delivery aims at delivering value continuously in smaller batches.

Figure 3.2 illustrates the difference between small and large batch sizes presented by Reinertsen [143]. *Cadence* or the regular intervals when the flow items leave the queue is different for the two processes. The visual illustration presented is rather similarly to the visualizations presented in this work. Reinertsen states that reducing batch size reduces cycle time and accelerates feedback [143]. Moreover, Reinertsen considers large batch sizes problematic: reduced efficiency, lower motivation, and exponential cost and schedule growth. In continuous software engineering, release frequency is higher [16], which leads to more frequent smaller batch sizes compared with infrequent large batch sizes.

According to Reinertsen [143], we must get a deeper understanding how queues affect development processes. As queue size increases, more capacity

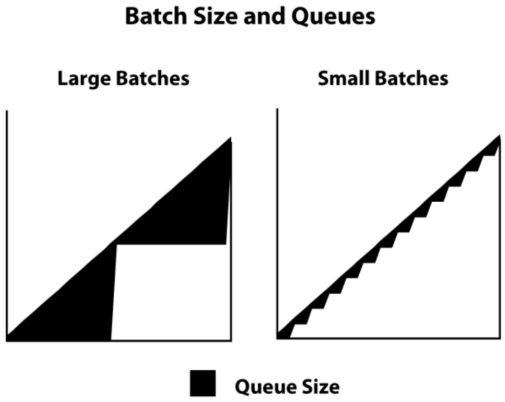

Figure 3.2: Larger versus smaller batch size according to Reinertsen [143].

is needed to process the flow units. Reducing batch sizes has four benefits in software development [143]. Firstly, smaller changes lead to easier debugging. Secondly, fewer open bugs lead to less non value-added work and fewer status reports, for instance. Thirdly, faster cycle time causes less refactoring. Finally, early feedback produces faster learning and lower cost changes. Based on the feedback, the developers can implement new features that the customers or end users need [88].

As a consequence of continuous delivery and reduced cycle times or smaller batches, users do not experience significantly more post-release bugs and the bugs are fixed faster [87]. With continuous delivery, faster feedback cycles, increased productivity and improved communication are achieved [73, 114]. Rapid releasing implies less time for testing and bug fixing which allows faster time-to-market and timely user feedback [107]. In this sense, the frequency of deployments is a key factor in software engineering. Reducing the cycle time i.e. the time between two subsequent releases has been widely presented in organizations in numerous white papers and blogs [145].

3.1.3 Continuous process improvement

Software process management is about successfully managing the work associated with developing, maintaining and supporting software systems [50]. The goal is often to improve the process in order to find out if the software development organization is meeting the business objectives in an efficient way. *Software process improvement* (SPI) [50] relies on understanding, planning and assessing a software development process. The process can be observed from several points of view – performance, stability and capability, for instance [50]. Rico et al. [144] present SPI as the act of changing the software

engineering process, which usually leads to improved cycle time, better quality and happier customers. They point out that processes are often changed without clear knowledge of the current status of the process. Process performance is rarely measured and analyzed as a basis for improvement.

Resolution of the process improvement issues raises a need for the measurement and analysis of the process [50]. Measurements can be used to manage and improve a process. Florac et al. [50] present a framework for improving the process consisting of six steps: clarify the business goal, identify and prioritize issues, select and define measures, collect data, analyze process behavior and evaluate process behavior. They emphasize the importance of the first step, business goals. The goals should be related to cost or time to market or quality, for instance. The goals can then be used to prioritize the issues and selecting the measures. Data collection is an important step where the data can be used to visualize the process including patterns and trends [50]. The gathered information can then be used to analyze and evaluate process performance. Florac et al. present several measurable attributes of software process entities. For instance, processing time, throughput rates, delays, length of queues and number of development hours can be measured.

According to Unterkalmsteiner et al. [159], "Pre-post comparison" is very common practice in software process improvement. In it, the process is evaluated before the SPI initiatives have been applied and after it. They state that it is necessary to setup a baseline from which the improvements can be measured. As Rozum et al. [147] put it: "What quantifying measures can be used to determine the progress of software process improvement efforts, and what effect have those efforts had on the organization?". They state that one measure will typically not be able to show the overall change and benefit of the software process improvement activities.

3.1.4 Continuous improvement in Lean Software Development

Lean Software Development refers to applying the principles of lean manufacturing into the context of development of software systems [135]. As Kiichiro Toyoda, the founder of Toyota Motor Company in 1930s puts it [125]: "I plan to cut down on the slack time within work processes and in the shipping of parts and materials as much as possible. As the basic principle in realizing this plan, I will uphold the just-in-time approach. The guiding rule is not to have goods shipped too early or too late.". Applying the same principles to continuous software engineering context highlights the importance of timing of the development work and delivery.

The Poppendiecks present continuous improvement as a key strategy in lean manufacturing practices [135]. Production workers are expected to stop the line when things are not perfect, then find the root cause and fix it before continuing manufacturing. They state that Toyota Production System [125] started with few practices which were continuously improved over decades. Furthermore, they state that in a similar way, in the context of software systems, the developers should improve the system and the development process continuously.

The Poppendiecks present waste as a key concept in lean thinking [135]. According to them, eliminating waste is a necessity. Anything that does not add value to a product is waste [135]. For instance, if developers code features that are not immediately needed, that is waste. They present that the ideal is to find out what a customer wants and then develop and deliver it immediately. They list the seven wastes of software development [135]. In this context, three of them are presented in detail from the deployment pipeline point of view.

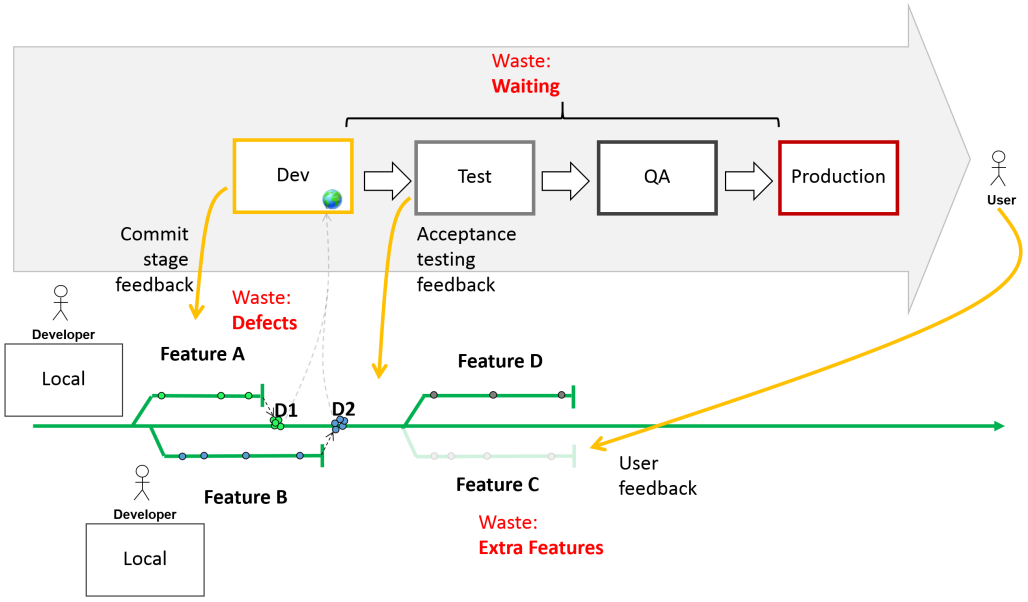

Figure 3.3: Three types of waste in the deployment pipeline.

Defects are waste. In this context, this type of waste is relevant from the continuous integration point of view. The goal of CI tools and practices is to automatically test the change sets in order to maintain high quality. A CI system provides short cycle feedback to the developers of the possible

quality problems in the pipeline. Defects are effectively eliminated with the help of automatic resources provided by the pipeline.

Waiting is waste. In this context, waiting can be considered as relevant type of waste since the goal of continuous delivery is to deliver changes to the software system with a short cycle. Any extra waiting in the inventories of the deployment pipeline can be considered waste.

Extra features are waste. Fowler presents the biggest risk to any software effort to build something that is not useful [51]. Fowler presents user feedback as one of the most important principal benefits achieved by applying continuous delivery. The earlier and more frequent feedback from the real users helps evaluating how valuable the implemented features are.

Figure 3.3 presents a typical deployment pipeline and the three types of waste which can occur. The pipeline consists of five environments: Local (each developer has an own local environment), Dev (the common development environment), Test (for acceptance testing), QA (Quality assurance) and the Production environment. There could be more environments, for instance, a demonstration environment for demonstration purposes. In Figure 3.3, a developer is implementing feature A into the system. The sample feature consists of three commits which are shown on the timeline of the version control system branch. In the mean time, another developer is implementing feature B. When the development of feature A is done, deployment D1 to the Dev environment is triggered automatically. When the new version is deployed, the CI system executes automatic tests and gives commit stage feedback to the developer. Accordingly, when feature B is done and deployed, the CI system gives commit stage feedback of deployment D2. When the CI system integrates the changes continuously, the waste of type *defects* is effectively eliminated. Without the CI system, the defects would not come into prominence in this phase and they would enter the other pipeline environments.

Next, the features are deployed to Test and QA environments. In this phase, manual acceptance testing may occur. This is a possible source of waste in terms of waiting. For instance, the features may have to wait for deployment or acceptance testing. When there are no extra idle steps at this stage, waste of type *waiting* is eliminated.

Finally, in Figure 3.3, the features are deployed to the production environment. In this phase, it is possible to get feedback from the users of the system. This may help to prevent the implementation of extra features which are not needed. Thus, waste of type *extra features* can be eliminated. For instance, feature C may not be implemented based on the user feedback. Effort can be invested to implement feature D instead, for instance.

3.2 Software analytics

In this section, background for software analytics is presented. Firstly, the concepts of data analytics, software analytics, and information visualization are introduced. Secondly, the topic of software engineering data is explored. Then, research related to mining software repositories is introduced. Finally, the role of metrics and software visualization is presented.

3.2.1 Data analytics

Today's society is driven by data [160]. Tremendous amount of data is available for analysis purposes, which has led to growth of analytics in many domains [3]. Data analytics is a widely used term which is often defined by the intent of the activity [160], namely descriptive analytics, predictive analytics and prescriptive analytics. Descriptive analysis [28] is a basis for any analysis. It helps to understand the data set and phenomena related to the history. Predictive analytics then focuses on the future, i.e. predictions of what will happen in the future. Once the past is understood and predictions can be made, prescriptive analytics, if applicable, then helps to propose the optimal actions in order to increase the chances of achieving the finest outcome [28].

Davenport and Harris [31] present the concept of analytics as "extensive use of data, statistical and quantitative analysis, explanatory and predictive models, and fact-based management to drive decisions and actions". The key goal of data analytics is to support decision making.

Especially in descriptive analytics, presenting data visually is a common practice. In the literature, there are two major disciplines of visualization [34]. *Scientific visualization* refers to processing of physical data while *information visualization* refers to processing of abstract data. The distinction is not obvious and the two disciplines overlap [34]. The origins of data visualization [177] are in the statistical and scientific disciplines. Furthermore, according to Keim et al. [84], *visual analytics* is defined as the science of analytical reasoning facilitated by interactive visual interfaces in [27]. They present visual analytics as an integration of scientific and information visualization with adjacent disciplines related to data mining and human computer interaction among others [84]. The topic of information visualization as a tool for data analytics is presented in more detail in the following.

3.2.2 Information visualization

Nobody has ever seen an atom, but most of us think of it as a core surrounded by small spheres or orbital clouds [34]. Visual images help the human mind to process and remember complex phenomena. Paivio [127] presents visual imagery as an important cognitive system, not the language related mechanisms alone. Paivio over-simplifies as follows: visual imagery is a parallel processing system while the verbal system is specialized in sequential processing. As an example, he states "Occasionally, when I have been required to list the names of my colleagues from memory, I have found myself visualizing the hallways in which their offices are located, systematically moving past these offices, then picturing and naming the occupants." Paivio presents visual imagery as an important tool for reasoning.

The early work of Edward Tufte [157] paved the road for information visualization as an efficient tool for constructing understanding about the target of the visualization. Today computers are playing an important role in information visualization and as a consequence visualization has become a discipline of its own [34]. Card et al. [21] define information visualization as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition". This definition emphasizes the importance of computers in the visualization process. However, the definition uses term abstract data. Which data is abstract and which is not? Is data always an abstraction of the target entity it represents? Moreover – what is the cognition that is amplified? Card et al. [21] use term external cognition as the use of the external world to accomplish cognition, which is then amplified. They state that visualizations amplify the capabilities of the human brain. The most important ways information visualization empowers the human cognition are: increase processing and memory resources, reduce searches, enable pattern detection, and strengthen perceptual inference operations.

Visualization may pack a huge amount of data into a small space. The human brain, which has huge cognitive processing capabilities can make such inferences easy which are not easy otherwise [21]. In their classical study, Larkin and Simon [98] illustrated three basic ways why diagrams help. Firstly, diagrams can group information that is used together, thus large amount of search is avoided. Secondly, diagrams often use location to group information about an element so that matching of symbolic labels is needed. Finally, diagrams support a large amount of perceptual inferences which are extremely easy for humans.

Lately, vision researchers have found out that the retina of the eye already processes the visual information, not the brain alone [56]. The retina solves a diverse set of tasks and then sends the results explicitly to downstream brain

areas. Information visualization can be considered as a powerful tool that takes the most efficient cognitive capacities of the human brain to effective use.

The information seeking mantra presented by Schneidermann et al. [150] presents data exploration as follows: overview first, zoom and filter, and then details-on-demand. This way, the user of the visualization can advance from an overview to a detailed view in an interactive manner. Moreover, Ware et al. [172] present the ultimate goal of an interactive visualization design so that they help us perform cognitive work more efficiently. They present visualization as an ability to comprehend huge amounts of data. Moreover, visualizations allow both perception of emergent properties and exploration of both large-scale and small-scale features [172]. In addition to this, in the work of Wijk et al. [163], visualization is seen even as a scientific discipline.

By following certain rules, for instance, the principles of ink space presented by Tufte [157], the data available can be put to an effective visual form for allowing patterns in the data to reveal themselves. Tufte claims that good graphical representations maximize data-ink, or the ink on a graph that represents data, and erase as much non-data-ink as possible. This maximizes the amount of informations the visualization contains, which can then be used for efficient reasoning.

3.2.3 Exploring software engineering data

Software engineering is a data rich activity [19]. The tools used in a software development process generate vast amount of data to be available for analytic purposes. Software analytics is analytics on software data for managers and software developers [112]. The aim is to get insight on the data in order to make better decisions. An important part of software analytics is that they provide actionable advice for different stakeholders [112]. Software analytics aims to obtain actionable information from software artifacts that help practitioners to accomplish tasks related to software development, systems, and users [179]. The audience for analytic results can be e.g. developers, managers and researches [112]. In general, software analytics employs the following technologies [179]:

- large-scale computing to handle large scale datasets
- machine-learning-based and data-mining-enabled analysis algorithms
- information visualization to help with data analysis and presenting insights.

The large-scale computing and machine-learning algorithms tackle the problem of large amount of information available. Information visualization is a tool that can be used to understand the data and to communicate the data efficiently.

According to Keim et al., visual data mining in general is an efficient approach to explore large datasets [118]. They state that visual data mining techniques have proven to be of high value in exploratory data analysis (EDA). According to [9] EDA is a tradition based primarily on the philosophical and methodological work of John Tukey [158]. Exploratory data analysis is an approach to learning from data in order to create understanding [9]. Natrella et al. [120] present exploratory data analysis as an approach or philosophy for data analysis that employs a variety of techniques, mostly graphical. However, they state that EDA is not identical to statistical graphics although these two terms are used interchangeably.

3.2.4 Mining software repositories

Mining Software Repositories (MSR) research focuses on analyzing various data sources related to software engineering [68]. The aim of MSR is to provide actionable information about software systems and projects. Hasan et al. [68] present source control repositories, bug repositories and archived communication as some examples of repositories for data analysis. They state that in the mining software repositories field several repositories can be mined to create insight on software systems and projects.

Kagdi et al. [82] state that empirical and systematic investigations in the field of MSR uncover pertinent information, relationships and trends about evolutionary characteristics of a system. In their literature survey on MSR, they present a taxonomy consisting of four dimensions – the type of software repositories mined (what), the purpose (why), the adopted methodology used (how) and the evaluation method (quality). Such a taxonomy can assist in the continued advancement of the field of MSR. In their literature review, Hemmati et al. [70] extracted categorized comments which define best practices from the field since 2004. They identified several common recommendations. Three of them are presented in the following. Firstly, source code management (SCM) repositories contain a variety of noise making the validation of heuristics and assumptions essential. Secondly, when working with issue trackers, it is best to only consider closed and fixed issues. Finally, they mention visualization as a useful tool. For instance, they state that visualizing SCMs, bug tracking systems and mailing lists may be helpful.

There has been a wide range of topics in MSR research [91]. For instance, the development process of an open source project has been analyzed based

on version control system and communication data [116]. Another example is defect analysis and prediction based on the version control system and bug tracking data [47]. Hassan [68] suggests that research related to MSR should show and demonstrate the value of data in software repositories. Field of MSR analyzes and cross-links the rich data available in software repositories [68]. The data can be utilized to uncover interesting and actionable information about software systems and projects. Hassan states that metrics related to actual time and money savings could help practitioners in their daily activities. The need for measurement in software and systems is ubiquitous [130]. Measurement practices are integral to basic management activities regardless of discipline.

3.2.5 Metrics in software engineering

In their systematic literature review "Why are industrial agile teams using metrics and how do they use them?", Kupiainen et al. [95] present reasons for usage of metrics. The reasons are related to iteration planning and tracking, motivating, and improvement and identifying process problems among others. Metrics for iteration planning offered help into prioritization work by introducing estimation metrics for measuring the size of features and the customer value (revenue) the customer is willing to pay for the features [95]. They conclude that academia has given a lot of emphasis to code metrics yet they found a little evidence of their use in the industry. Moreover, planning and tracking metrics were often used and a research gap in this area was recognized. However, in their systematic literature review, Jaitly et al. [78] point out that many metrics have been developed and utilized resulting in remarkable success.

Another approach to categorizing metrics is presented in [115], where the authors define the core agile metrics to include product, resource, process, and project centered metrics. The product metrics deal with size, architectural, structure, quality, and complexity metrics. The resource metrics are concerned with personnel (effort metrics, etc.), software, hardware, and performance metrics. The process metrics deal with maturity, management, and life cycle metrics, and project metrics with earned business value, cost, time, quality, risk, and so on. Each of these submetrics can define a range of additional metrics such as velocity, running tested feature, story points, scope creep, function points, earned business value, return on investment, effort estimates, and downtime. The researchers conclude that there is no hard and fast rule on how to select metrics. Furthermore, they point out that the teams should invent new metrics and not use a metric simply because it is commonly used.

Humble and Farley [73] put emphasis on cycle time as the most important metric related to continuous delivery by referring to Poppendiecks question: "How long would it take your organization to deploy a change that involves just one single line of code?" [136]. Humble and Farley claim that this metric tells more about the process than any other metric [73].

3.2.6 Software visualization

Software visualization is a concept for applying information visualization to the domain of software engineering [86]. Diehl et al. [34] define software visualization as the visualization of artifacts related to the software development process and the software itself. A wide variety of artifacts is covered from program code and documentation to bug reporting and visualizing the structure and behavior of the software. Software evolves over time through program code changes to extend the functionality of the system or simply to remove bugs [34]. The overall goal in software visualization is to improve the productivity of the software development process [34].

In its narrower meaning, software visualization is often used interchangeably with *program visualization* which means the visualization of the software as an executable program [154]. From this point of view, software visualization is related to computer programs and algorithms. Moreover, according to Petre et al. [132], software visualization uses visual representations to make software visible. They refer to loosely distinguished themes within visualization, among them information visualization, software visualization and program or algorithm visualization.

In addition to software visualization, many researchers have been proposing software evolution visualization [123]. A systematic mapping study of software evolution visualization by Novais et al. [123] highlights software evolution as one of the most important topics in software engineering. They found out that authors present their visualization tools in diverse manners. Moreover, the formal validation and collaboration in the area is missing. They list different types of data that can be used to visualize and analyze evolution, for instance source code management systems, bug or issue tracking systems, project documentation and different versions of source code itself.

The real-world applications of visual data analysis often access information from a number of information sources ending up in problems with data quality [86]. Problem of integrating several data sources into a single visual representation for visual analysis is related to many fundamental problems in decision theory, information theory, statistics, and machine learning [86]. The challenges for visual analytics are many. For instance, user acceptabil-

ity, where the actual applications of developed visualizations have not taken place due to the users' refusal to change their working routines.

Quality problems related e.g. to data capture errors, noise, outliers, low precision, missing values, coverage errors, clones etc. can already be contained in the raw data which jeopardize the conducted visualizations [86]. Moreover, [146] point out similar flaws for data quality in software engineering research in general. However, even noisy and incomplete data can be used as a basis for visualizations when the interpreters of the visualization are knowledgeable of the data incompleteness.

3.2.7 Ambient visualizations

Ambient information visualization presented in [151] is information visualization application that does not reside on the screen of a desktop computer, but in the environment or periphery of the users of the visualization. In a broader meaning, using the physical environment to present information has been explored previously, in particular in ambient media [76], where information displays are designed to present information in the same context where the users are located. In Toyota Production System [125] under the name Kanban or just-in-time (JIT), visualizations have been under a debate for decades with an objective of improving production efficiency in terms of continuously and thoroughly eliminating waste. Kanban is a Japanese word for "card", "ticket" or "sign" [101] used for managing the flow and production of materials and value in a Toyta-Style "pull" production system [101] where a visual control device in the production area alerts the workers on defects, equipment abnormalities, or any other problems by using signals such as lights or audible alarms. In a similar manner, information radiators [113], which are placed to the team workspace, can show information to the development team. For instance, the status of CI system can be showed constantly to the team.

3.2.8 Categorizing visualizations

Lam et al. [97] conducted an extensive literature review of over 800 visualization publications and present seven distinguish categories with different study goals and types of research questions – evaluating visual data analysis and reasoning, evaluating user performance, evaluating user experience, evaluating environments and work practices, evaluating communication through visualization, evaluating visualization algorithms, and evaluating collaborative data analysis. The scenarios can be used to choose appropriate research questions and goals for a research. Johnson et al. [80] list top scientific visu-

alization problems. They mention that creators of visualization technology do not spend enough time endeavoring to understand the underlying data they are trying to represent.

In their mapping study, Keim et al. [85] present several classifications for visualizations. They present a classification based on the data type to be visualized, the visualization technique, and the interaction and distortion technique. Moreover, they present visual data exploration process as a hypothesis generation process – visualizations allow the user to gain insight into the data and come up with new hypotheses. Visualization is seen as an iterative exploring method.

Lengler et al. [99] developed a categorization method for visualizations. They constructed a periodic table of visualization methods to group similar approaches. They define a visualization method as follows: "A visualization method is a systematic, rule-based, external, permanent, and graphic representation that depicts information in a way that is conducive to acquiring insights, developing an elaborate understanding, or communicating experiences.". The periodic table they developed, consists of six categories: data, information, concept, strategy, metaphor and compound visualization. The visualization methods in the categories are then ordered by represented information type (process or structure), cognitive process (convergent or divergent thinking) among other criteria. With the categorization system developed, it is possible to group for instance Gantt [111] charts and Pert [42] charts to the same group. Both chart types have been widely used in management activities.

3.3 Summary

Continuous value creation in contemporary software engineering relies on rapid cycle delivery of new features to the actual users of the system. With continuous delivery of new features, the supplier can get the feedback of implemented features with rapid cycle. This helps to gain understanding of value-in-use, i.e., how do the users of the system adapt the new features. A deployment pipeline with effective continuous integration and continuous delivery tools and practices provides rapid feedback to the development organization. The numerous environments of the deployment pipeline are potential sources of waste. By continuously recognizing waste in the process, the team can improve its process. High frequency of deployments implies smaller batch sizes.

The tools used in software development generate a huge data set which holds detailed information about the events related to development. Min-

ing software repositories research focuses on providing actionable information about software systems and projects. Metrics based on the software engineering data can provide new insight of the development process. Information visualization is a powerful tool for exploring data sets. Visualizations enable the effective use of cognitive capabilities of the human brain. By analyzing and presenting the data related to software engineering visually, new information concerning the development process can be acquired. Such information is possible to be shown as an ambient visualization in the team workspace.

Chapter 4

Related work

We start by introducing approaches to value creation management in software engineering. Then, metrics in software engineering are introduced. Finally, we discover approaches and examples related to information visualization. Some empirical work published in the Internet is included in order to construct a comprehensive picture of related work in the field.

4.1 Value creation management

There are different approaches for understanding and managing value creation in software engineering. The concepts of value creation and business value are closely related. The objective of agile work is to deliver value [67]. A systematic literature review by Racheva et al. [140] presents how business value is created in agile projects. They found that most published studies take the concept of business value for granted. Subjective observations of value are common. The selection of people affects to value perceived. They found statement "different people consider different things valuable" a common observation. According to them, value creation should be studied further with empirical research methods.

In their article "Using Metrics and Diagnostics to Deliver Business Value" [67], Hartmann et al. collect existing information measurement for agile delivery of customer value. They state that agile software development continually measures both the product and the process used to create it. Value creation management is built in to agile software development. They state that software is simply inventory until its value is realized. The authors recommend choosing one key metric that drives towards value creation. The metric should be closely tied to the economics of investment. As concrete examples, they present two key metrics. *Velocity* is a very useful metric for

the team, and should be used during the project. The second key metric, *business value delivered*, measures net cash flow per iteration. The metric can be used in multiple phases of a project, for instance in planning and prioritizing. With the guidance of metrics, more value can be created.

Lindgren et al. [102] studied software development as an experiment system. They state that an experiment-driven approach to development is gaining increasing attention as a way to channel limited resources to the efficient creation of value. They found that experimentation is rarely systematic and continuous. The study found that there was a wish to focus on customer value creation in the case companies. Agile development practices together with continuous integration and short release cycles had been adopted. However, while the current development practices supported experimentation targeting to value creation, the state of practice was not yet matured.

Fabijan et al. [44] present an iterative approach for quantifying feature value. They present a technique for validating the value of features early in the development process. The goal is to estimate what impact a feature will have when fully developed. If value is perceived early in the process, the feature is further developed. The development is stopped if value creation early in the process does not reach the goals set. With the proposed technique, the organization can improve its value creation process.

Even the first principle of the agile manifesto focuses to value creation: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." [8]. However, finding what is valuable software is not an easy a straightforward task. Some approaches use, for instance, web based question and answer forum to collect information on valuable features the customers or end-users need [105]. A literature review by Racheva et al. [139] presents prioritization techniques in agile development. Some of the methods used the term importance instead of value when features are prioritized. They observe that knowledge of value in terms of business goals is crucial for any prioritization process. They propose that business value is calculated based on initial business value and current cost estimate. Moreover, they put emphasis on customer collaboration.

4.2 Metrics and measurement in software process improvement

In their systematic literature review related to the evaluation and measurement of software process improvement, Unterkalmsteiner et al. [159] found seven distinct evaluation strategies. The most common one "Pre-post com-

parison" was applied in approximately half of the reviewed papers. Statistical analysis, where descriptive analytics is involved, was the second one most commonly used. Analyzing the variation of the process was a common use case. The most measured attributes were quality, cost and schedule. According to Unterkalmsteiner et al., measurements most commonly assess the short-term impact. Moreover, they state that CMMI [2] and ISO standards [41] propose numerous measurements and metrics, however, often without showing how the measurements are applied in practice.

In their book "Best Practices in Software Measurement: How to use metrics to improve project and process performance", Ebert et.al [39] present metrics as a useful tool from the practitioner's point of view. They state that metrics have to be available timely on a push-button approach. Metrics must be sustainable, meaningful and goal-oriented. With them, concrete, understandable objectives can be reached. They refer to Goal-Question-Metric (GQM) approach created by Basili [5]. Basili states that measurement is a mechanism for creating a corporate memory for answering a variety of questions related to software development. The GQM approach has three levels: the conceptual level (goal), the operational level (question) and quantitative level (metric) [15]. As a concrete metric, for instance, hours per function point can be used to improve efficiency [15]. By using the GQM approach, Mahiko et al. [109] build descriptive models of the software process. They introduce a framework to measure and analyze the development process and conclude that some results suggest that the architecture of the product influences the process. The goal of measurement in software development is often to improve the process [5]. The measurement information fed back to all stakeholders, e.g. developers, managers, customers and the corporation, helps to understand and control the process [5]. However, measuring the success of software process improvement is not a simple and straightforward task but a multi-dimensional entity of multiple viewpoints [1].

Mishra et al. [115] state that process and project metrics are core agile metrics, among others. Process metrics refer to management and life cycle metrics. Project metrics describe earned business value, cost, time and quality. They conclude that there exists a large number of metrics and that there is no exact rules on how to select metrics for a particular project. They define a developer centric point of view: the developer defines what he really needs to make his project and team successful. Johnson et al. [81] present detailed metrics for characterizing development work, among others. For instance, *DevTime* describes how much time each developer spends in working on each file associated with the project. They state that developers and researchers must put emphasis on easily obtained, richer analytics, but in the mean time, they should have privacy and overhead concerns in mind.

In general, software development process is a value creation process [140]. Therefore, metrics related to the development process measure the value creation process. In their article, Hartmann et al. created a compilation from various sources to suggest characteristics of good agile metrics to deliver business value [67]. They state the first tenet of Lean Thinking [176] as: "Define value in the eyes of the customer". They collected 11 principles for good metrics. Four of them are presented in the following:

- is easy to collect
- measures outcome, not output
- follows trends, not numbers.
- provides fuel for meaningful conversation

The principle of easy collection is a necessity. Johnson [81] mentions a significant overhead cost in a manual collection process of even 500 distinct values that developers must manually calculate. Ktata et al. [94] investigated what agile developers need and want to measure. They mention increased overhead activities to support data collection as one major risk of failure. Hartmann et al. state ideal as "one button" automation where data is drawn directly from operational tools with low overhead. The second principle in the list above of measuring the outcome, not output, is of "maximizing the amount of work not done" where the highest outcome might be achieved by reducing planned output while maximizing delivered value. The third principle of following trends, not numbers, is of measuring "one level up" where aggregated information is measured instead of optimized parts of a whole. To summarize, metrics should measure the trend of actual outcomes in a way that is easy to collect and is also understood and iteratively developed with the customer. Finally, a good metric provides fuel for communication. A good metric reveals meaningful characteristics of the target which can then be discussed about.

From the mining software repositories (MSR) point of view, there is a wide set of literature of using repository data for process analysis and improvement. Kim et al. [90] mined repository data to analyze lead times of bugs. They state that this metric is important for analysis. Sliwerski et al. [152] mined version control system data and combined it to a bug database. With this information, they are able to identify the change sets that caused the defects. They use the size of a fix as a metric to predict the existence of bugs. VanHilst et al. [164] mine object process metrics from repository data. The metrics they present use data from an issue management system and

from a configuration management event log. They apply Little's law [103] to interpret the results. According to them, it is good to minimize the amount of work in process at any given time. They conclude that objective process metrics can be mined for analyzing the process itself. In their later work, VanHilst et al. [165] analyze a waterfall project using repository data. They propose a new approach to process improvement based on empirical data and analysis. They focused in finding waste in process practices and present various indicators of various types of waste. With their metrics and process improvements, it is possible to reduce time to market [25] by several weeks.

In their systematic literature review of industrial studies, Kupiainen et al. [96] studied using metrics in agile and lean software development. They found a total number of 102 metrics. The metrics are focusing on the following areas: sprint planning, progress tracking, software quality measurement, fixing software process problems, and motivating people. Some examples of the metrics are: fix time of failed build, open defects and throughput. For instance, burndown, check-ins per day or number of automated passing test steps were used to manage risks and to provide progress monitoring. In some cases, metrics were used to check if project goals were achieved. According to them, metrics were also used to motivate people. For instance, the number of defects was shown to motivate developers to fix the reported defects. Moreover, employees' behaviour was also changed with the metrics. In some cases, measuring the work in progress (WIP) of developers, the focus was put to implement only a single feature at a time. However, using metrics may also have negative effects. Using a velocity metric had negative effect such as cutting corners in the implementation of features. Finally, they conclude that quality needs to be measured and problems in the process need to be identified and fixed.

Modig et al. [117] divide the efficiency of a lean process into two different points of view: *flow efficiency* and *resource efficiency*. Flow efficiency is the proportion of value-added activities to all activities concerning an entity. If flow efficiency of a flow unit is high, the flow unit flows rapidly through the process. Resource efficiency targets to maximize the usage of resources. In general, this means maximizing the time that resources, for instance people or machines, spend executing their work. When resource efficiency is high, the lead time of the flow unit through the process may be low.

The Poppendiecks [137] emphasize the importance of customer-centricity in metrics. They present examples of these including time-to-market (in product development) and end-to-end response time (for customer requests). Furthermore, they present the following attributes of metrics important – the success of a product in the marketplace, business benefits attributable to a new system, customer time-to-market, and impact of escaped defects.

4.3 Usage data mining

Software vendors are unaware of how their software is performing in the field [161]. According to Schuur et al. [161], the research community has developed metrics to measure and quantify the quality of service based on information on end-user usage. Schuur et al. [93] present the concept of Software Operation Knowledge (SOK) to contain four types: performance, quality, usage and feedback. In the context of this thesis, usage is the most relevant point of view. It consists of knowledge how a software solution is used by its end-users and how they use it. In their reference framework [162], Schuur et al. state that it remains unclear how and to which extent, end-user feedback can be used to improve vendor's practices, processes and products. Data on software usage [162], in this context, expressed with term *usage data*, describes how software is used by the end-users and how it responds to end-user behavior. They conclude that although software vendors consider SOK valuable, supporting integrations and infrastructure is missing.

Data analysis is becoming a common practice among software development teams [32] as data scientists are working alongside developers. The purpose of data analysis is often to get insight on some aspect of the software. Improving software user experience through A/B testing, for instance, is a promising direction for research [32]. Moreover, testing a hypothesis about feature usage is possible [32]. In general, data analysis and data scientists have an emerging role in software development teams [89].

Applying data mining to software usage data reveals valuable information on software usage and its users [40]. The data contains information on quality of software and dynamics of software development. Guzdial implemented a tool for deriving software usage patterns from log files based on Markov chain analysis [65]. Log files have been analyzed to answer a wide range of questions, for instance, usability measures and usage patterns [66]. The authors conclude that use of visualization provides a lot of potential for working with log files. Moreover, Zhang et al. [180] analyzed users of a mobile Internet application based on a large data set concerning the usage. They managed to display several usage patterns of the service.

4.4 Information visualization of software engineering data

In their systematic literature review, Mattila et al. [110] state that the most studied topics during the past six years (2016) in the field of software visualization are related to software structure, behavior and evolution. According

to them, there is a research gap in applying information visualization to data related to software development process or software usage data. However, there are multiple examples of applying visualizations to the domain of software engineering in general. Chuah et al. [23] use glyphs for demonstrating software project management data in an efficient way. They present a visual approach to highlight interesting patterns and anomalies in the data set. Gall et al. [54] apply information visualization for demonstrating the release history of a software system. According to them, information visualization technologies can be applied to the analysis of software evolution for uncovering valuable information. Ohira et al. [124] collected data for software process improvement from configuration management systems, mailing lists and issue management systems. Then, they presented the data visually. They mention that real-time visualizations motivated developers to fix bugs, since people were aware that unresolved issues still exist. As a problem, they report that visualizations can be too complicated to understand. Voinea et al. [167] make a similar conclusion: the most critical requirement for visual analysis tools on software repositories is simplicity. Visual techniques and tools need to be simple to be understood.

In [174], the authors mine version control system data and examine how developers work together. With the visualizations, they are able to find interesting phases during the evolution. Moreover, Yasutaka et al. [178] propose a tool that calculates metrics from a variety of software repositories for personal process improvement. Their retrospective tool provides information on version control system activities and contribution.

In [30] the authors propose an approach to support the analysis of a bug database with two visualizations. They provide two views to the data. System radiography shows the bug database in large and helps understanding the system products and components over time. Figure 4.1 presents the visual approach they introduce.

The goal of the visualization they present is to understand where and when the open bugs are concentrated. In addition to this, the bug watch view shows the characteristics of a bug visually. Approaches to showing the number of defects in general, and visualizing the build status in monitors were found to lead into faster build and fix times [95].

According to [148], software development processes are often chaotic. They propose a framework for mining the process data. With their technique, they are able to obtain process models for realistic software projects. Fischer et al. [47] propose a solution that combines version control data and bug system data for populating a release history database which can then be used for querying information on release history of a project. By combining

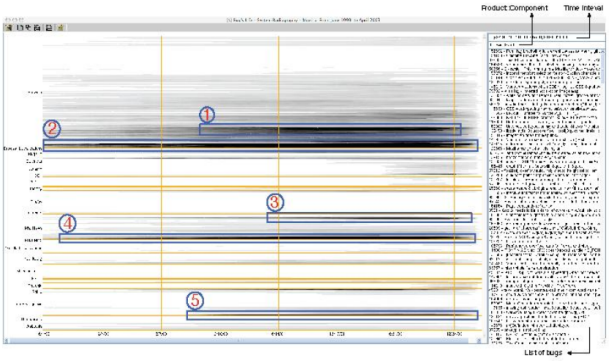

Figure 4.1: System radiography view of a bug database in [30].

data describing the software development process or its release history, new insight of the process can be constructed.

The latest advancements in the field of mining software repositories introduce use of more advanced analytic methods to analyze the data. Bodo et al. [14] apply machine learning algorithms with semi-supervised learning together with information visualization. They propose the use of visualization techniques of multidimensional information to support the labeling process.

In agile software development, visualizations are often applied to show progress. Scrum [149] presents the burndown chart which can be used as a tool for evaluating progress [106]. It shows the amount of work remaining in, for instance, story points [24]. Extrapolating lines on the burndown chart may reveal the status of project completion early in an early phase the project [83]. With the information the burndown chart provides, the organization is able to get the insight of velocity [67], i.e. how many features can a team deliver per iteration. Moreover, a burndown chart helps to understand the workload and it indicates effectively if work is added to a sprint [10].

Cumulative flow diagram (CFD) is a method for tracking the progress of an agile software project building upon the burndown charts [20]. It allows further detailed understanding of the process and enables the early detection and correction of problems [20]. Reinertsen [143] suggests the use of CFD to monitor queues. According to Reinertsen, CFD is a useful tool for tracking queues and process progress since it shows exactly which of the factors is the problem in the process.

Figure 4.2 presents a CFD [143]. Reinertsen uses the diagram to demonstrate several dimensions of the target process. First, the basis of the diagram is the dimension of time flowing from left to right. Second, the diagram shows the arrival of the flow units. Third, the time in queue is demonstrated. Finally, the diagram demonstrates the quantity of flow units in a queue, i.e.,

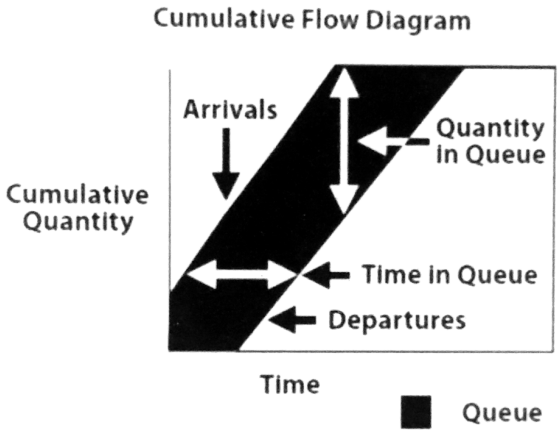

Figure 4.2: A cumulative flow diagram by Reinertsen [143].

the batch size. Cabri et al. [20] apply cumulative flow diagrams among other diagram types to show earned value. According to them, earned value is a project management technique to measure the progress and performance of a project against the plan. Moreover, future performance can be estimated. They state that CFDs can show the benefits of earned value concepts.

In their experience report, Greaves et al. [59] applied CFDs to an agile software project. Figure 4.3 depicts the CFD they used as a simple mechanism to measure the impact of process changes. The cumulative flow diagram in Figure 4.3 helped the organization to drive quality improvements into the development process. Moreover, it helped in measuring cycle time and make predictions of defect fix rate.

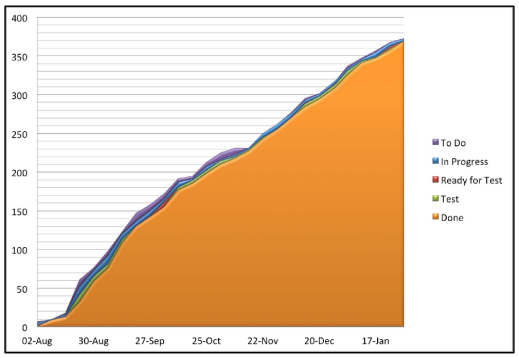

Figure 4.3: A cumulative flow diagram applied in [59].

The cumulative flow diagram is also present in issue management software

tools commonly used. For instance, Jira[1] issue management system has a cumulative flow chart plugin. The author of the plugin [142] states that such diagrams help in getting a conception of work in progress. Moreover, the diagram helps to reach the goals set.

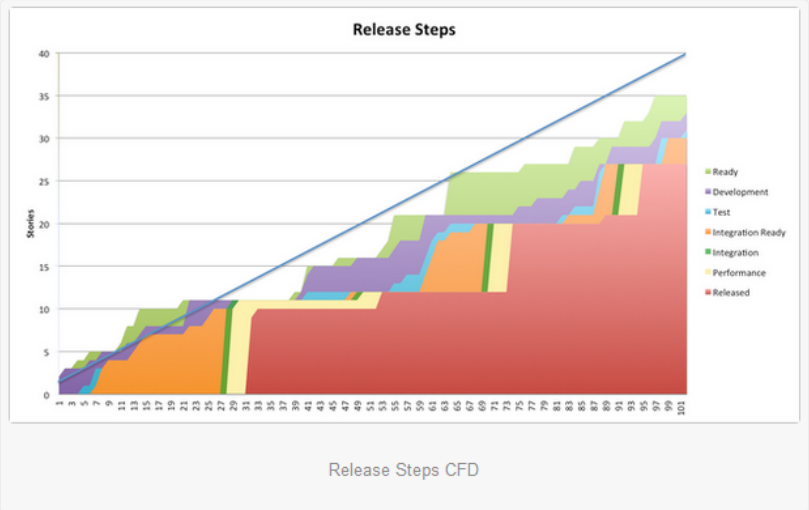

Figure 4.4: A cumulative flow diagram by Evans [43].

In his blog posting, Evans [43] applies a CFD to show the true cost of holding software inventory. Figure 4.4 presents a "Release steps CFD" by Evans. According to Evans, the diagram can be used to convince an organization of the benefit of delivering small batches frequently. The diagram contains information on a flow unit. The states of the flow unit are presented with colors. The states are: ready, development, test, integration ready, integration, performance and released. The usage context of the diagram is a team with very long release cycles, even months or years. Evans even presents a calculation of cost savings for releasing small batches more often. However, the calculation for cost savings is not a peer reviewed scientific publication.

In their experience report, Wang et al. [171] list visual tools and techniques which were used in 30 experience reports published in agile software conferences. In them, lean approaches were applied to agile software development. In the report, CFD is mentioned as one approach to visualize the process. Moreover, according to the report, a Kanban board which is seen as a central communication hub among the members of the development team [119], is a tool often used. The practitioners have used Kanban boards successfully even in implementation of very large systems [92, 131]. Princi-

[1]https://www.atlassian.com/software/jira

ple of continuous improvement or Kaizen [72] is often tightly connected with visual indicators [171]. Moreover, visual indicators on a timeline have been found useful in evidence-based timelines presented by Bjarnason et al. [12]. They propose using historical data as input to project retrospectives.

4.5 Summary

There is a wide set of literature available for value creation management and for applying metrics and visualizations to the context of software engineering. The existing work on useful practical metrics and visualization for continuous software engineering leaves a research gap to be filled. A novel set of metrics and visualizations that are targeted to the context of continuous delivery and deployment provide new meaningful tools for the research community. Novel metrics and visualizations for filling the gap are presented in the following chapter.

Chapter 5

Results

In this Chapter, the main results of this thesis are presented. We start by summarizing the contributions per publication. Then, we present a synthesis of the results. We reflect the quantitative results to the empirical evidence gathered with qualitative methods.

5.1 Summary of contributions per publication

The thesis contributes towards novel metrics and visualizations in the field of software engineering. Contemporary continuous software engineering relies on tools and techniques which enable continuous delivery of valuable features to the users of the system. The usage of development tools generates vast amount of data available for analysis. This data provides information on the development process. The novel metrics and visualizations based on the tool data create new knowledge of the underlying development process. They construct a basis for lean continuous improvement.

The empirical data from industrial context provides a solid basis for the contributions of the publications. The data sets and industrial feedback have enabled construction of novel metrics and visualization artifacts. The industrial data collected and demonstrated is a contribution itself. Empirical evidence related to characteristics of contemporary software engineering phenomena is a contribution from the practitioners to the research community.

A key constraint for the artifacts developed is that all data for them is automatically generated by the tools used in the development. No extra data collection work is needed from the development team to utilize the metrics and visualizations. All publications rely on data-driven metrics and visualiza-

tions. Information visualization has been a suitable method for exploratory data analysis which the designed artifacts are based on.

Publication P1 presents novel metrics for characterizing continuous delivery. Research output is definitions for three fundamental metrics: *development time*, *deployment time* and *activation time*. The data for the metrics is automatically generated by the various development tools used in contemporary software engineering. The publication paves road for applying lean principles based on the information the proposed new metrics provide.

Publication P2 defines a metrics framework which measures the process from the initial development effort up to the point of customer use and feedback. The framework can be used to drive cycle time reduction and improve value creation activities and decision making. The publication presents metrics together with a *value capture visualization* which constructs a basis for value creation management. By taking into account the usage of the information the metrics provide, cycle time reduction and more efficient value creation is enabled.

Publication P3 applies an existing lean metric, *flow efficiency*, to the context of continuous software engineering. The flow efficiency of features flowing through the deployment pipeline was measured. The metrics can be applied to analyze the performance and the present status of the pipeline. The metrics provides valuable information for the team to improve the processes and the pipeline. Applying the metrics in a continuous delivery project setting can help to achieve a continuous flow of value delivery. Moreover, we introduced the concept of the first usage related to a new feature delivered through the deployment pipeline in terms of metrics.

Publication P4 uses an Action Research (AR) approach to develop visualizations to be used as a basis for software process improvement. The actual data mined from the issue management system of a case project can be used to evaluate the characteristics of the development process. For instance, the first version of the *process visualization* revealed a new fact in a project setting where the sprint length was reported to be four weeks but in the visualization the actual timeline was even 10 weeks. The visualization can be used as a basis for software process improvement.

Publication P5 develops the *process visualization* further. A wider range of software engineering development data sources are combined into a mash-up visualization which demonstrates the software development process. The current status of Continuous Delivery [73] can actually be seen in the visualizations. The combination of mashing up issue management, development and usage data makes it possible to measure the lead time from issue creation till the actual usage of the feature. The novel visualization presented in the publication helps to understand the value creation processes.

Publication P6 utilizes the *process visualization* artifact developed in Publications P4 and P5. The visualization artifact is iterated one step further to show the issue management data in a way that can be used as a basis for the visual analysis of the process. The verbal description provided by the team corresponds to the visual reference shape of the process. Moreover, the transformation during a longer period of time can be visually demonstrated with the *process visualization*. The visualization method developed provides a basis for understanding and improving continuous value creation in an efficient way.

5.2 Designed artifacts and their dependencies

Table 5.1 describes a summary of the key artifacts of this work including their dependencies. The metrics are closely related to the visualizations and vice versa. All the artifacts rely on a data source, for instance, an issue management system, a version control system or production logs. The data for them is generated automatically during software engineering activities. The tools the team uses produces a large data set which can be mined for interpreting the phenomena related to the software development process.

The list of nine artifacts in Table 5.1 consists of metrics and visualizations. All the five metrics are related to time elapsed between certain software engineering events with exact timestamps. Therefore, all metrics can be drawn to a timeline. The four visualization artifacts are timeline visualizations based on the metrics. When a large amount of exact values for the metrics are drawn into a single timeline visualization, the characteristics of the underlying process are revealed. Visual representations amplify human cognition [21]. Visualizations enable pattern detection and perceptual inference operations, for instance. The visualization artifacts in Table 5.1 enable evaluation of characteristics of the process. For instance, *cycle time*, *feedback speed*, *batch size* and the amount of simultaneous *work in progress* can be evaluated. Parallel development processes are revealed. Such information is not easily available otherwise. Furthermore, the visualizations work as a basis for communication related to the process. Experts working in the field of software engineering are able to understand the visualizations and construct new artifacts based on the visualizations. For instance, listed in Table 5.1, artifact *rectangle of unexploited potential* reveals the amount of waste in the process. The artifact was designed by an expert working in the field of

software engineering during the feedback cycle of the applied Design Science method.

Figure 5.1 presents a reference visualization based on a narrative and a visualization based on actual data from industrial context. The visualization artifacts developed in this work help to pack huge amount of information into a small space. The visualization artifacts developed reveal the characteristics of the underlying software development process efficiently. The details of the visualizations are described in detail in the following sections.

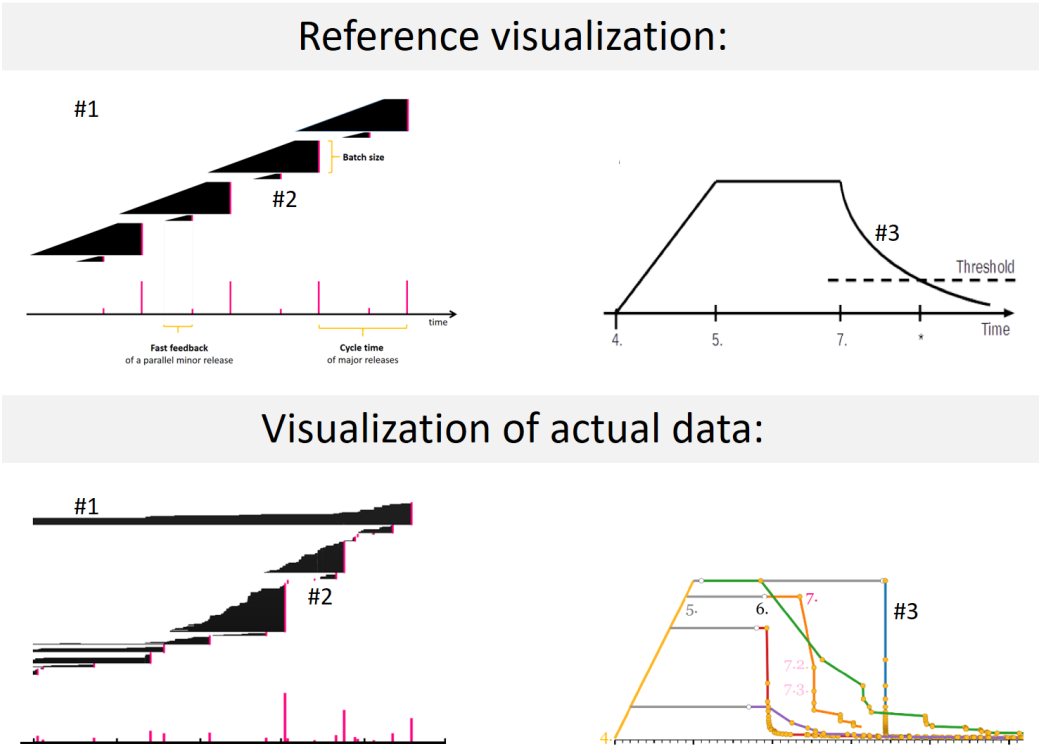

Figure 5.1: The reference process based on a narrative and the actual process based on data.

The benefits of applying information visualization to software engineering data are visible in Figure 5.1. Visual perceptual inferences are extremely easy for humans. Searching similarities and differences between the reference and actual shapes in Figure 5.1 is almost automatic. For instance, on the left at spot `#1` there seems to be a difference: there is a long tail in the actual data. At spot `#2` there seems to be a similarity: shape of a triangle with a smaller triangle below. At spot `#3`, the actual curve differs from the reference curve. The semantics of the designed artifacts and their *utility* for SPI, are presented in the following.

Artifact	**Formula**	**Questions related to value creation**	**P/S**
Development time	DEVT	How long does it take to implement a new feature?	P1
Deployment time	DT	Suppose a feature has been done. How long does it take until the feature has been deployed to the production environment?	P1
Activation time	AT	A feature has been deployed. How long does it take until first user uses the feature?	P1
Flow efficiency, an existing metrics from [117]	proportion of value-adding-activities to all activities	What is the proportion of value-adding activities during the life-cycle of the flow unit?	P3
Development done to first usage	*D2FU* = DT + AT	Development is done. How long does it take until the feature is actually used?	P2
Development done to value capture	*D2VC* = DT + AT + time-to-threshold	How long does it take until the invested DEVT has been paid back?	P2
Process visualization	Draw DT of features to a timeline visualization, order by release date and done date.	Shape of the triangle demonstrates the characteristics of the process. (cycle time, feedback speed, batch size, work in progress)	P4, P5, P6
Rectangle of quality assurance	Draw a *process visualization*. Then, observe flatness.	Is there a separate quality assurance period in the process?	5.4.1
Value capture visualization	Draw upwards y = DEVT, then x = DT + AT and y = DEVT * 1/n, where n = number of usages	When is the invested DEVT paid back? Enables visual comparison of feature usage profiles. Which features are used frequently?	P2
Rectangle of unexploited potential	Draw a *value capture visualization*. Then, observe the area beneath the curve.	What is the amount of waste in the process?	5.4.2

Table 5.1: Designed artifacts and their relationships.

5.3 Synthesis

In this section, a synthesis of the results is presented. The synthesis consists of metrics and visualizations presented in the publications of this compilation. Moreover, following the design science research method principles presented in Chapter 2, the artifacts have been developed one iteration further in Section 5.4. We reflect the questions in Table 5.1 to recognize the value creation mechanism related to the artifact. Then, measures to eliminate waste in the process are presented.

5.3.1 Metrics for the process visualization

In the following sub sections, we present the metrics needed to construct the *process visualization*.

Development time

The first simple metrics introduced in Table 5.1 is *development time*. It answers to value creation related question *How long does it take to implement a new feature?* Development time consists of activities related to implementation of features. For instance, programming, testing, refactoring, reviewing, daily practices of the project, communication, specifications, checking CI status, fixing broken unit tests etc. To put it short, development time in this context covers activities that are mandatory to actually implement a new feature to software with adequate quality. Figure 5.2 presents *development time* with yellow color on the left of the timeline.

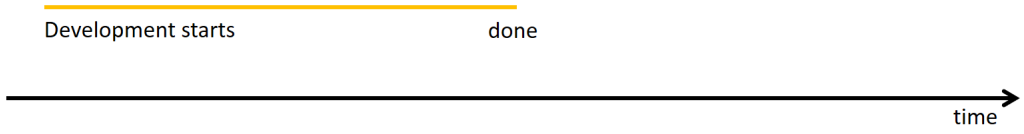

Figure 5.2: Synthesis – development time.

When development is done, development time ends. Definition of done (DoD) often consists of automatic testing and pair reviewing activities. In the context of this thesis, DoD contains an assumption that the feature is considered to be ready for production use. Naturally, there is a risk of defects.

Measures for eliminating waste: in the context of this thesis, there are no known measures for eliminating waste related to *development time*. It is assumed that the development team does its best in close co-operation with

the customer in order to develop valuable new features that have been chosen to be implemented to software. Therefore, development time is not the most relevant metrics related to value creation in this context and it is left out from the final *process visualization* we are targeting to in this narrative. However, development time is used as a basis for other metrics.

Deployment time

Metrics *deployment time* measures the time elapsed from development done till the feature has been deployed to the production environment. Figure 5.3 presents metrics *deployment time* visually.

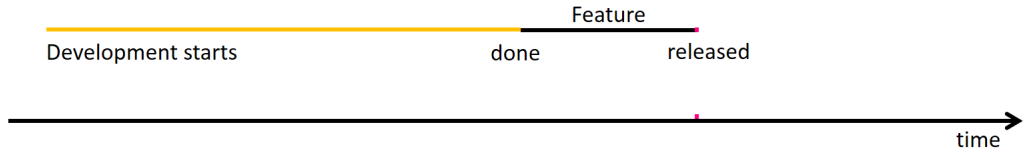

Figure 5.3: Synthesis – development time and deployment time.

The line begins from the moment when development was done. Then, after some time has passed, often days or even weeks, the feature is released to the production environment, which stops the deployment time. During the deployment time, quality assurance work is often performed. For instance, the customer may perform acceptance testing to the features to be released in the next version. Moreover, the customer is often willing to be knowledgeable of new features implemented. Acceptance testing often has a communication purpose. *Deployment time* is a key metrics to construct the visualizations presented in this work.

Measures for eliminating waste: to shorten *deployment time*, perform production deployments frequently. Plan the release schedule in co-operation with the customer to be, for instance, once per week. Plan the customer acceptance testing in advance and perform it efficiently. The supplier can make decisions towards reducing deployment time in co-opeartion with the customer.

Process visualization

Next, we advance towards artifact *process visualization* with a single metrics, *deployment time*. At the top of Figure 5.4, there are five lines depicting metrics *deployment time* of features released in two separate releases.

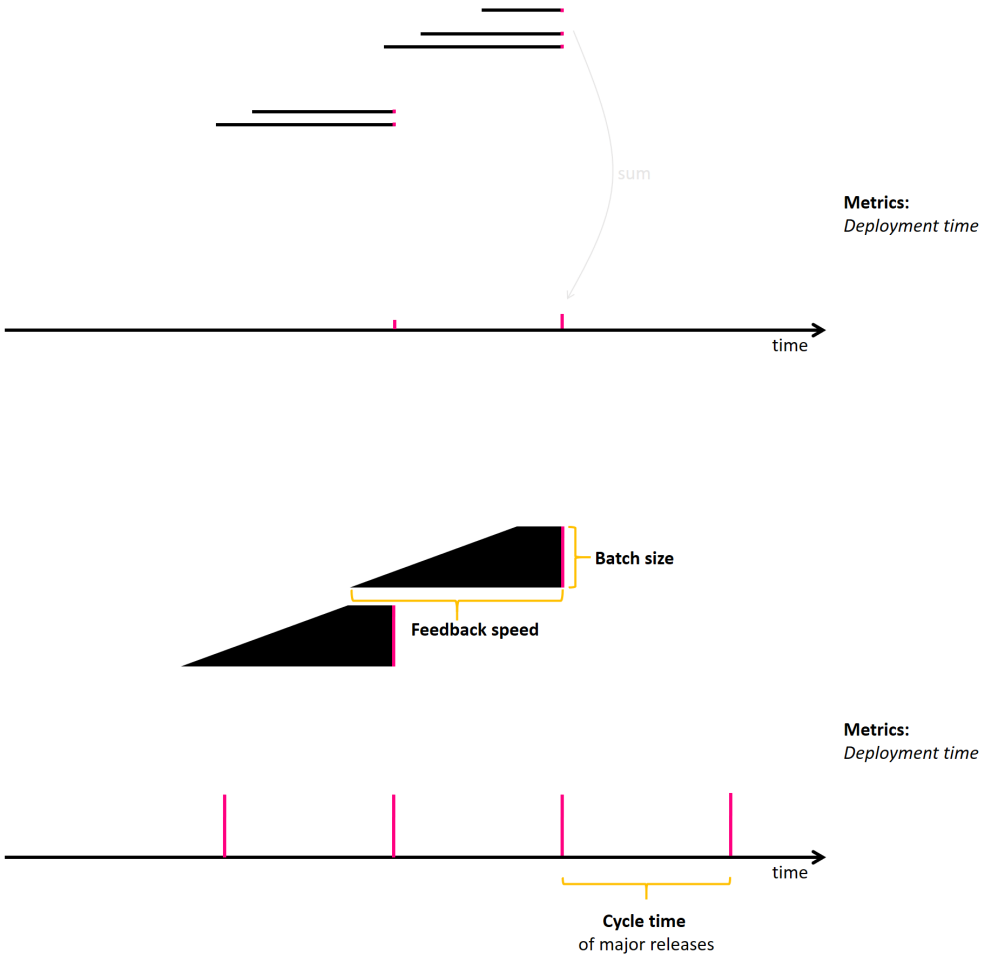

Figure 5.4: Synthesis – from metrics to process visualization.

According to the drawing rules of the *process visualization*, features are sorted from bottom to top by release date and done date ascending. Therefore, multiple features in a single release start to resemble a shape of a triangle. Moreover, the number of features released in a single version is shown as a summary at the bottom of the diagram near the timeline. At the bottom of Figure 5.4, a larger amount of features is included into the reference *process visualization*. The resulting triangles reveal the characteristics of the development process. *Batch size* equals to the number of features in a single released version. *Feedback speed* equals to the deployment time of a feature in the release. According to the definition presented in Chapter 3, *cycle time* is the time between two subsequent releases, which is presented at the bottom

of the visualization with vertical bars, which depict the *cadence* [143] or the regular intervals of the process.

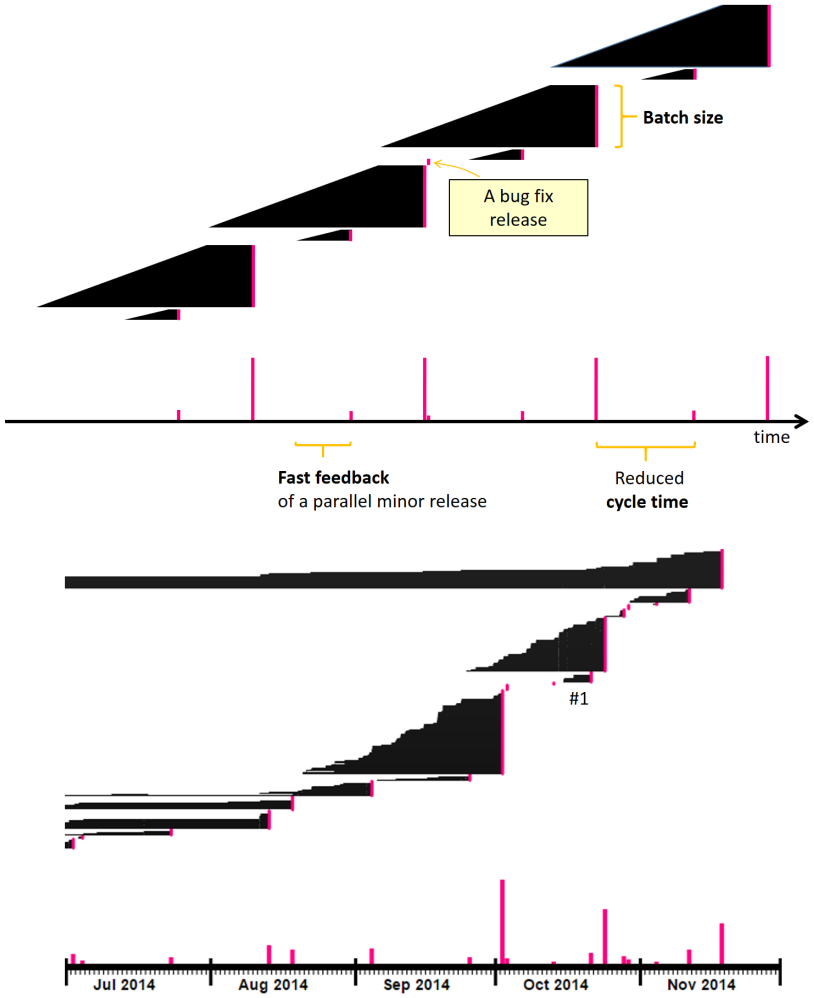

Figure 5.5: Synthesis – major version releases, parallel minor development process and a separate bug fix release.

Figure 5.5 demonstrates the reference visualization and an actual visualization based on the data set from a case project. The visualizations resemble each other. For instance, the triangular shapes are visible in both visualizations. Therefore, cycle time, batch size and feedback speed of the released versions can be evaluated. Moreover, at spot `#1`, a minor development version in the actual data can be seen. Because of the drawing rules of the *process visualization*, the triangular shape of the parallel minor development process is drawn beneath the triangular shape of the major release. A separate minor development process enables faster feedback. According to the definition the

development team used in the case project, parallel minor development does not consist of bug fixing. Instead, it consists of rapid cycle implementation of new features to satisfy urgent needs of the customer.

In the case project, the release cycle of major releases was approximately one month. In addition to this, the team published minor releases in a continuous manner. According to another study [77] described in more detail in Section 6.3, the customer was very satisfied to this kind of dual process solution. With the help of parallel minor versions, the end-users got their features in a rapid manner. The communication was focused to instant value creation. Figure 5.5 shows the reduced cycle time when a separate minor development process is used. Moreover, feedback is faster when software is implemented in small batches with frequent releases. In continuous software engineering, the release frequency has gone up [16]. A consistent flow of slow, continuous changes is better than a speedy race which occasionally stops to doze [125].

Measures for eliminating waste with the process visualization: when the team strives to reduce the *deployment time* continuously, feedback is faster. Moreover, the process can be improved by releasing smaller versions, i.e., versions with small batch size. Furthermore, a separate minor development process helps to reduce the cycle time of the process. With the visualization, the team is knowledgeable of the effect of changes to the development process. The team can make such decisions in, for instance, retrospective meetings.

Figure 5.6 demonstrates a *rectangle of quality assurance* suggested by a testing specialist in a focus group meeting. The shape of a rectangle inside the triangle occurs because of quality assurance work, e.g., customer acceptance testing. When the quality assurance phase begins the developers switch context to the implementation of features of the next release. The context switch is presented with a small blue arrow at the top of Figure 5.6. Moreover, the visualization shows the risk of incomplete quality assurance which is realized as bug fixes after the major release.

The *process visualization* can be used as a basis for software process improvement. In the interview, an agile coach (described in detail in Section 5.4.2) emphasized the importance of the usefulness of the *process visualization* for retrospective usage. The visualization shows how the team has managed to improve the process in a tangbile way. Figure 5.7 shows evolution that has happened in the case project in Publication P6 during the years.

On the left of the diagram, there is a process visualization of the development process which the case project used in 2011. This process is referred to as "discontinuous delivery" since long cycle times can be observed at the bottom of the diagram. On the right, the process has evolved towards continuous delivery. During years 2012 and 2013, the team improved the de-

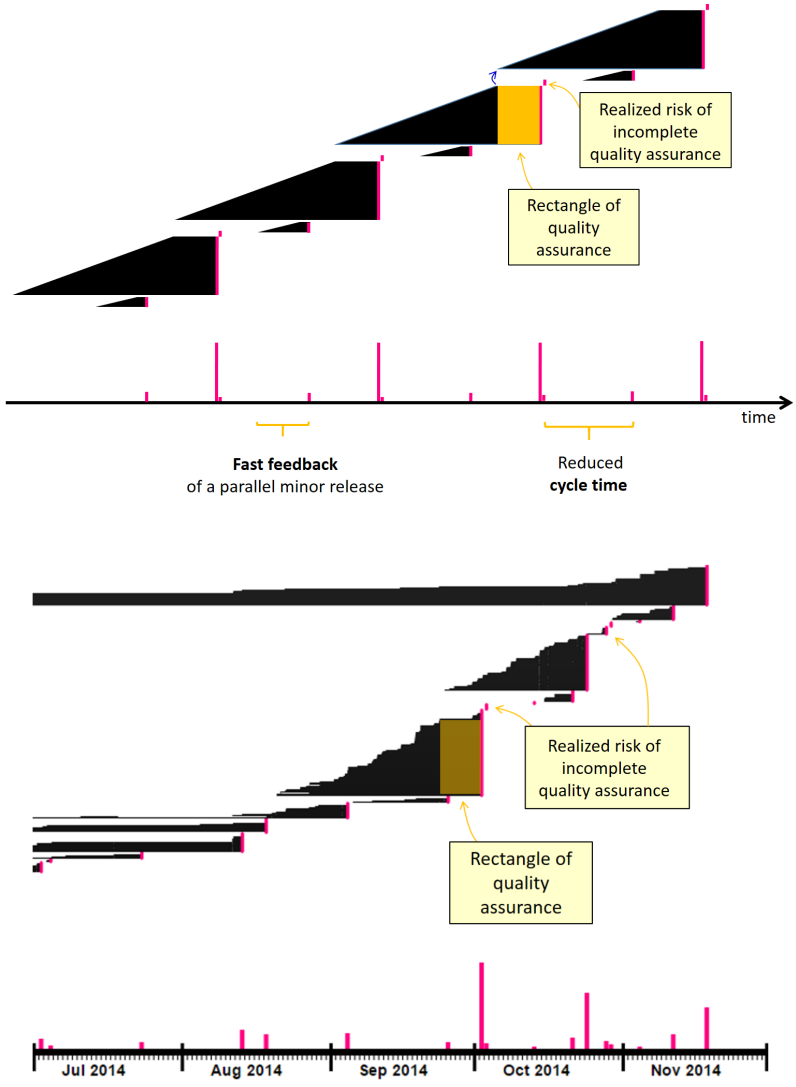

Figure 5.6: Synthesis – Rectangle of quality assurance.

velopment process in two important ways. First, database management of the data intensive system was improved to be automatic. Second, the team implemented single-click deployments to the CI system, which makes the deployments easy and fluent. The cycle time of the process has shortened due to the frequent deliveries of smaller batches with faster feedback. Earlier there was no separate parallel minor development process. The minor releases are visible beneath the major releases. Faster feedback from the actual usage of the features is acquired with rapid cycles.

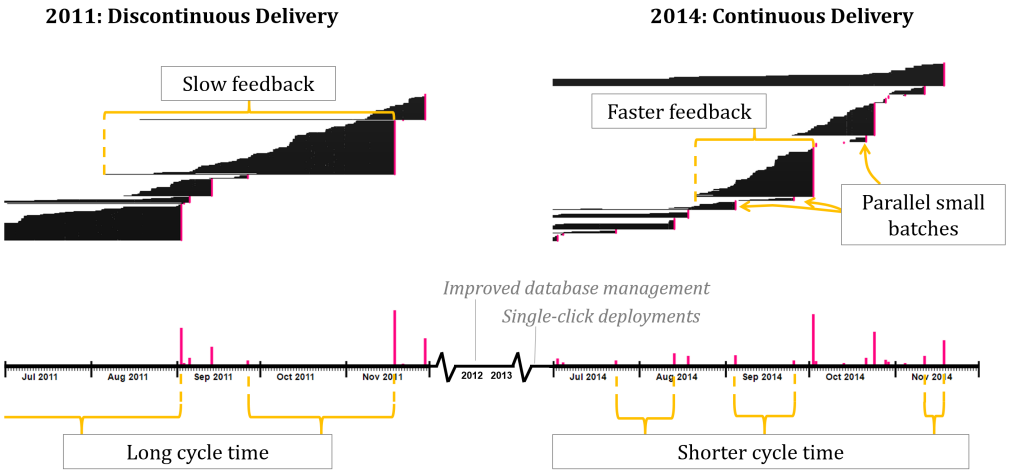

Figure 5.7: Evolution of batch size, cycle time and feedback speed of the case project.

Figure 5.8 shows a tangible direction for improvement of the current process towards the target process.

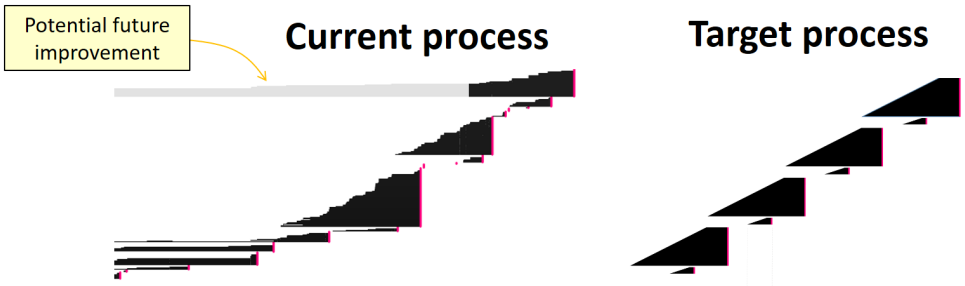

Figure 5.8: Potential future improvement of the software development process.

In this case, the team could improve the feedback speed of the process towards a defined target process with fast feedback and short cycle time. The suggested improvement in Figure 5.8 could be achieved by postponing the implementation schedule of certain features which were implemented into an inventory. The *process visualization* artifact could be useful as a basis for software process improvement when applying SPI guidelines of, for instance, [57] or [74].

5.3.2 Metrics for the value capture visualization

When the development process has been improved with the *process visualization*, value creation management can be continued by taking use of the *value capture visualization*. The metrics the visualization depends on are presented in the following.

Activation time

Activation time is the time it takes until the feature is actually used by a user in the production environment. As put in Table 5.1: a feature has been deployed. *How long does it take until the first user uses the feature?* In Publication P3, the latency measured was typically days. Figure 5.9 presents the activation time of a single feature.

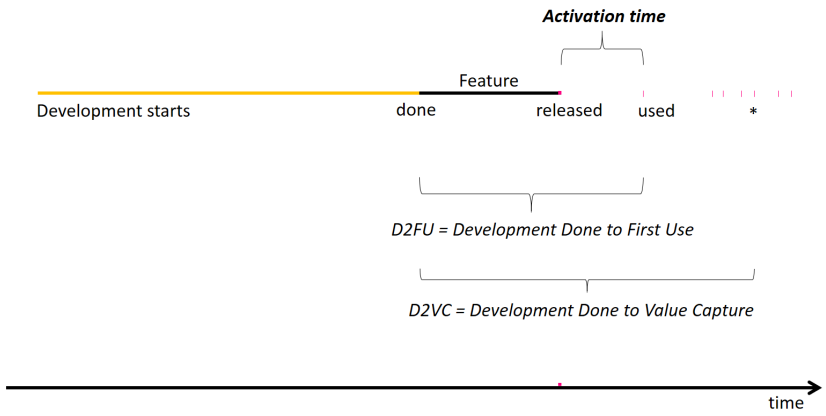

Figure 5.9: Synthesis – Metrics activation time, D2FU and D2VC.

Measures for eliminating waste: The activation time can be reduced, for instance, by contacting the end-users directly and by asking them to use the new features available. Moreover, informing the users of the new features, e.g., by emailing announcements or with a popup in the user interface, may reduce activation time. This improves the feedback speed not visible in the *process visualization*.

Flow efficiency

In Publication P3, an existing metrics, *flow efficiency*, was applied to the context of continuous software engineering. The lean metric measures how much a flow unit is processed in a specific time frame. The amount of value-added activities is divided by the total length of the time frame. In Publication

P3, value-added activity was defined strictly. The only value-added activity was the *development time* of a feature. Once development was completed, the time until the first usage in production environment (sum of *deployment time* and *activation time*) was considered waste. The mean flow efficiency in the case project was 34%. In other words, a feature spent two thirds of the total time in non value-added activities such as waiting for deployment or waiting for a user to actually use the newly released feature.

Measures for eliminating waste: reducing the *deployment time* and *activation time* improves *flow efficiency*. Optimization of *deployment time* and *activation time* guides the software development organization towards a more efficient development process with less latencies and faster feedback from the actual users of the system. The decisions towards higher *flow efficiency* can be made in co-operation of the supplier and the customer.

Development done to first usage

Table 5.1 presents a novel metrics based on activation time. Development done to first usage (*D2FU*) is the sum of deployment time and activation time (Figure 5.9). When development is done, *how long does it take until the feature is actually used?*.

Measures for eliminating waste: the measures for metrics deployment time and activation time. Metrics *D2FU* is conceptual. When the team strives to acquire knowledge about how soon the implemented features are actually used by the users, waste of type *extra features* [134] can be eliminated. For instance, if a certain feature is not used at all, this information may lead to decisions of leaving out other unnecessary features.

Development done to value capture

Development done to value capture (*D2VC*) takes the concept of value creation related metrics further. *When is the invested development effort paid back?* The metrics is based on *development time* and the actual usage events of the feature. The metrics measures how fast a feature acquires actual usage to cover the invested development effort. A context-specific threshold value for development time per times used is presented as a dotted line in Figure 5.10.

Measures for eliminating waste: the measures for metrics deployment time and activation time. When the supplier of software informs the end-users of the new features, waste can be eliminated, since usage of the features reveals possible defects and produces new information on how *value-in-use* is realized. The *D2VC* metrics helps to understand the different value creation

profiles of the features. If the development organization has an expected conception of the usage frequency of a feature, the metrics helps to confirm the conception. By continuously receiving new information on the usage of the features, waste of type *extra features* can be eliminated. For instance, if a certain set of features is used rarely, it may not be meaningful to implemented new features related to these features.

Value capture visualization

The *value capture visualization* shown in Figure 5.10 helps to understand the relationship of metrics *D2VC* to value creation management.

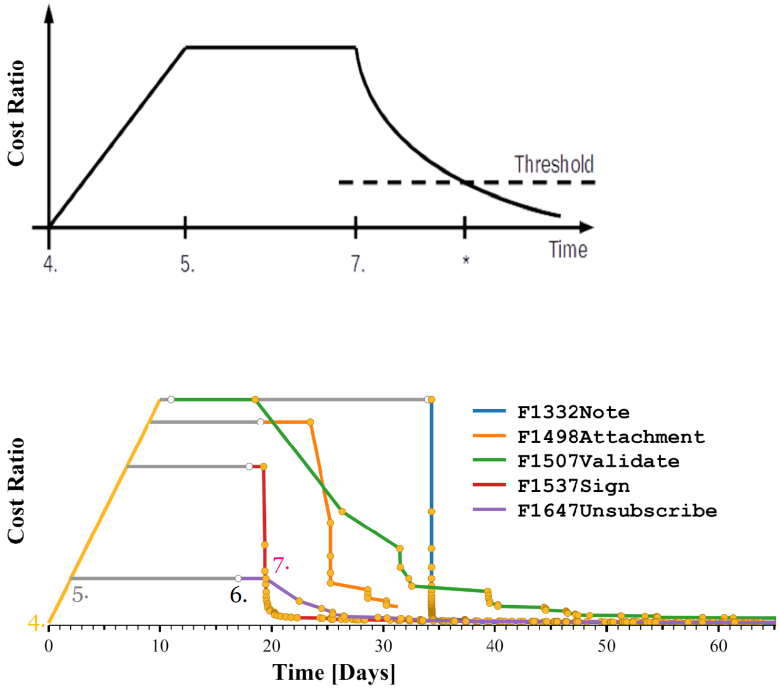

Figure 5.10: The value capture visualization.

For example, in Figure 5.10, the development effort for the sample feature at the bottom left corner was two working days (i.e. *development time* from step 4 to step 5). Then, the feature was waiting for deployment (i.e. *deployment time* from step 5 to step 6). Finally, the first usage of the feature occurred after two days in production in step 7 (*D2FU*). In this phase, the cost ratio was 100% of *development time* per times used. Then, the feature was used for the second, the third and the fourth time. The cost ratio dropped to 50%, 33% and 25% respectively. The threshold in the case

project was set on such a level that approximately 10 usage times for the sample feature were required to reach the threshold. The sample feature was used for the tenth time approximately 30 days after the development of the feature had been done. For all the features measured in Publication P2, the value for *D2VC* varied from 15 to 50 days. This means that for some features it took nearly two months until the feature had reached the threshold value set.

Aho[1] suggests that the area under the curve is related to the amount of waste in the process. The longer the development takes and the longer time it takes for the first usage to occur, the larger the amount of waste is. The *rectangle of unexploited potential* for two separate features is shown in Figure 5.11.

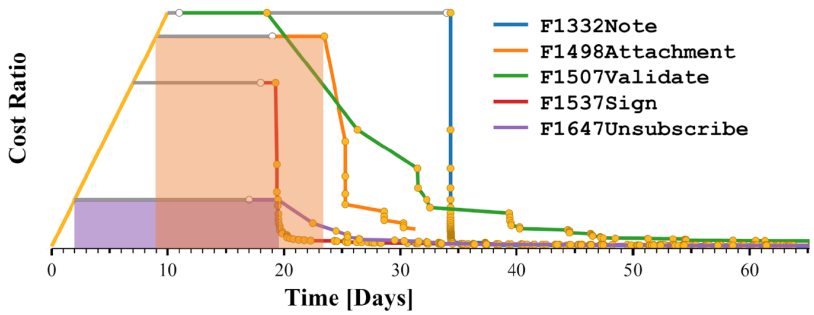

Figure 5.11: *Rectangle of unexploited potential*: the area constructed by invested work multiplied by the number of days the feature is waiting for first usage.

For instance, in Figure 5.11, there are two features with different amount of waste. According to the diagram, the development time of the feature at the bottom left corner was 2 working days. The first usage occurred after approximately 17 days of waiting. This, the amount of waste is 2 * 17 = 34 units. For the other feature in the diagram the amount of waste was 9 * 14 = 126 units, i.e. over three times more.

Measures for eliminating waste related to the *rectangle of unexploited potential*: First, when *deployment time* is kept short, the area of the rectangle of unexploited potential is smaller. Moreover, by reducing the *activation time*, the metrics is improved further. By minimizing *development time* by, for instance, splitting large features to multiple smaller ones, may reduce the amount of waste in the process.

[1]In a private discussion

5.4 Feedback from the practitioners

The chosen research methodology, design science, targets to *utility* [168]. Therefore, the results have been actively communicated both in academic and industrial contexts. The communication has occurred during the development of the artifacts in the publications during the years 2012–2016. Moreover, two more separate feedback sessions presented in the following, have been organized to reflect the results. In the following, we introduce practitioners feedback related to the artifacts developed.

5.4.1 Focus group meeting

Focus group is a group of people assembled in order to discuss a particular subject of the research to reveal their opinions and thoughts [169]. A focus group meeting was carried out in the industrial context in the case company, Solita ltd. In the meeting, the current status of the visualization artifact was presented. There were participants from the following roles: software developers, testing specialists, project managers and other roles from the software company. The output of the meeting produced new ideas to the latest version of visualizations. For instance, a testing specialist presented an idea of *rectangle of quality assurance* which is actually clearly visible in the visualization in Figure 5.12.

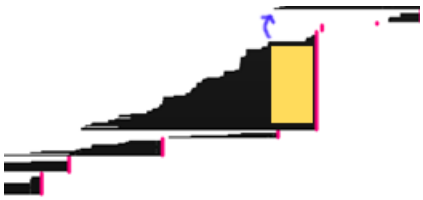

Figure 5.12: The *rectangle of quality assurance* seen by a testing specialist in the focus group meeting.

According to the testing specialist who participated the focus group meeting, this kind of pattern (marked with a yellow square in Figure 5.12) in the information visualization occurs when the customer starts to test the new version to be released. At this stage, the developers switch context to the following release (marked with a blue arrow in Figure 5.12) to implement new features. Therefore, a flat area is constructed into the visualization.

The discussions of the focus group meeting were not analyzed by applying any formal research method, because many of the participants have already been familiarized with the theme of the visualizations. Therefore, the setting

would not have been neutral. However, another session was organized, and the output of the session was interpreted with a qualitative method.

5.4.2 Interview of an agile coach

A separate interview session to reflect the results was organized. The subject of the interview was an agile coach, who had started in the company a week earlier. It can be assumed that this kind of person has a suitable mindset for evaluating the artifacts developed without prior knowledge of the topic. The subject was interviewed and a thematic analysis of the transcript was conducted. The method is described in more detail in research methodology in Chapter 2. In this case, thematic analysis was applied to interpret how the subject understands the visualizations.

The agile coach had not seen the visualizations before, and therefore the information visualization was totally new to the interviewed subject. The interview session started with a presentation of the drawing rules of the visualization. The presentation was neutral and the subject was not led to any certain direction. Therefore, the output of the subject consisted of spontaneous reactions to the visualizations that were presented. The result of the analysis can be interpreted as a sample of how professional people in the field of software engineering can understand and utilize the visualizations.

The thematic analysis revealed certain topics that were constantly repeated in the transcript. The textual output was tagged and common themes were recognized according to the process of thematic analysis presented by Braun et al. [17]. The research question behind the session was *RQ2: How to construct visualizations to demonstrate value creation in continuous software engineering?*. However, the research question was not presented to the subject, because it could have led the discussion to the theme of value creation in continuous software engineering. The subject mentioned three important themes in the discussion which are presented in the following with direct quotes from the transcription of the session.

First, the visualization shows the feedback cycle from the actual usage of the features to the development team. Such information can then be used as a basis for future decisions. A direct quote from the transcription of the session:

Here, a lot of new features are implemented on top of the information gathered in this earlier phase. The feedback is got at the earliest.. exactly here. If I try to summarize... a lot of work is lying in an inventory, and somehow, a lot of capital, too.

The subject mentions high amount of implemented features lying in an inventory with long feedback cycle. This is related to the seven wastes of software development presented by Poppendiecks [134]. They present the problems with batches and queues and state that the idea of lean production is to expose problems as soon as they arise. According to them, batches and queues hide problems. In this case, the concept of inventory and slow feedback cycle is familiar to the agile coach and the subject can connect the concept to the visualization.

Second, the subject mentions concurrent work and work in progress (WIP) [92]. A direct quote from the transcription of the session:

Is it meaningful to implement this many releases at the same time? Instead, they could release one version, get the feedback, tune the process and then the next [version].

Modig et al. [117] present the problem of expensive context switches when there are a lot of concurrent tasks. If more than one release is implemented concurrently, a context switch is a possible source of waste. In the visualization presented in the session, there were even six concurrent releases in the visualization. The agile coach stated that this may be a problem in the software development process.

Finally, the subject mentions visualizations as a good source of retrospective information for the team. A direct quote from the transcription of the session:

They [team] would see themselves that is it really that long? [lead time]

The agile coach states that the visualization could help to set targets for process improvement. The team could then easily see how they have been performing during the past months. According to the coach, the team could see the progress. For instance, they could avoid extra long lead times in the future based on the information the visualization provides.

Moreover, the subject was missing the existence of a broader lead time in the visualization. A direct quote from the transcription of the session:

I can not see what has been the lead time of this task.

The subject was missing the actual development time of a feature. A

new version of the visualization could be constructed based on this feedback. In that version, the lead time would be visible. Then, feedback could be collected and the visualization could be further improved.

5.5 Summary

In this Chapter, novel metrics and visualizations for managing value creation were introduced. The data-driven artifacts construct great opportunities for software process improvement. The key artifacts, namely metrics *deployment time* and Development Done to Value Capture (*D2VC*) together with the *process visualization* and the *value capture visualization* help to improve the process towards continuous value creation. With the novel artifacts, the characteristics of the development process can be demonstrated. Based on the new information, the process can then be improved. Moreover, the team becomes knowledgeable of the effects of the improvement. Software development process is a value creation process. With SPI based on the novel artifacts, value creation is improved.

Based on the feedback from the focus group meeting and the thematic analysis of the interview, two important points of view can be presented. First, professional people in the field of software engineering are able to understand the semantics of the developed artifacts. They are able to evaluate the process based on the visualizations. Second, the developed visualization artifact is found to be useful for process improvement purposes. The artifact could be used for setting a target to the development team and to help to evaluate if the target has been reached.

Chapter 6

Discussion

In this Chapter, we discuss the results by revisiting the research questions and reflecting the results to existing knowledge in the literature. Each research question is introduced and the key results in the publications are then discussed. Moreover, we discuss the validity, limitations and opportunities of future work for the research.

6.1 Metrics for continuous value creation

The first research question addresses metrics for eliminating waste in continuous software engineering:

RQ1: What metrics help to eliminate waste in continuous software engineering?

The research question is addressed in Publications P1, P2 and P3 where novel metrics related to value creation in feature-driven development [128] in the context of continuous software engineering [16] are presented. With the proposed metrics, the development organization is able to continuously receive new information on the value creation mechanisms of the software development process. The presented metrics are based on mining the data automatically generated by the various tools used in software development.

The metrics are based on two points of view. First, they measure value-in-exchange [61] observed from the provider point of view. Value-in-exchange in this context consists of activities related to development and delivery of new features. Second, the customer centric view is important. Value-in-use [61], i.e., the actual usage of the developed features is measured. By covering both the spheres of value-in-exchange and value-in-use, the metrics enable comprehensive measurement and improvement of value creation.

Publication P1 presents three novel metrics: *development time*, *deployment time* and *activation time*. The metrics measure latencies in the various execution environments of the deployment pipeline.

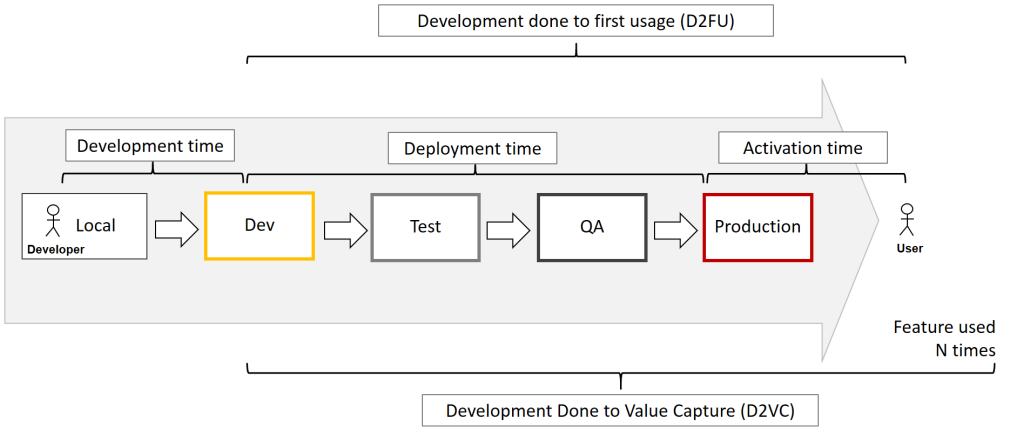

Figure 6.1: Measuring the latencies of the deployment pipeline (modified from Publication P1).

A typical deployment pipeline has multiple execution environments where the software is running. New versions of software are continuously deployed to, for instance, test and production environments. Figure 6.1 depicts a typical pipeline with five environments.

A developer develops a new feature in a Local environment. Each developer has an own Local development environment where a version control system tool is used. In Publication P1, metrics *development time* was presented. In the literature, a similar metrics related to *development time* is presented by Johnson et al. [81] Metrics *DevTime* measures the amount of time a developer spends to different artifacts related to the project. The metrics is measuring the characteristics of the process on a detailed level. The difference between *development time* and DevTime is related to level

of details included in the measurement. *Development time* acts on a higher level.

The work of the members of the development team is then synchronized in the Dev environment. When new commits are pushed to the version control system, the CI system triggers an automated build, deployment and automatic testing. This enables rapid cycle feedback to the developers. In their systematic literature review, Kupiainen et al. [96] present metrics related to fix time of failed build. Recovery is important in order to continue the development work. Sliwerski et al. [152] mined both version control system data and issue management system data. With the information they are able to identify in detail the change sets that caused the defects. In Figure 6.1 software is then released to the Test environment. Testing, for instance, customer acceptance testing, is executed in this phase. A quality assurance (QA) environment can also be part of acceptance testing. Finally, a new version is released to the Production environment. Existing metrics related to throughput [96] can help to evaluate the througput of a deployment pipeline.

Deployment time is the time measured from the moment when the feature was done till the moment when it was deployed to the production environment. In the literature, several related metrics are found [96], for instance, velocity, common tempo time, task done, task's expected end date. Moreover, according to Humble and Farley [73], cycle time is the most important metric related to continuous delivery. They referred to Poppendiecks question: "How long would it take your organization to deploy a change that involves just one single line of code?" [136]. Therefore, naturally, *deployment time* is not a novel metrics itself. However, the combination of metrics and visualizations to reveal the characteristics of the target process, is novel. There is a research gap in this field related to the combination of the research questions of this thesis.

When the feature has been deployed to the production environment, *activation time* is the time it takes until the feature is actually used by a user. In Publication P3, the value for the presented three metrics were measured in a case project during a three month period. The longest latencies were related to *deployment time*. On average, when a feature was done, it took approximately two weeks to deploy it to the production environment. The extra waiting in the various execution environments of the deployment pipeline can be considered waste. By shortening the *deployment time*, waste can be eliminated. When the team strives to optimize *deployment time*, the development organization is driven to a direction where extra latencies postponing

the deployment are eliminated. For instance, latency in acceptance testing may form an obstacle for releasing a new version of software. In this kind of situation, accelerating the acceptance testing schedule may be a means of improvement of the process. According to the observations in Publication P3, *activation time* of features was short and the new features were used for the first time with very short latency. However, in some cases, *activation time* can be a source of waste and effort should be put to, for instance, advertising the new features in the user interface in order to get rapid cycle feedback from the users.

To summarize, with the novel metrics presented, it is possible to automatically gather new information concerning the development process. The information can be utilized in decision making in order to eliminate waste in three ways. First, metric *deployment time* can help to eliminate waste related to extra latencies in the deployment pipeline. Expressed with Poppendiecks' terminology [135] related to seven wastes of software development, reducing deployment time reduces amount of features in an *inventory*. Second, *activation time*, helps to identify features which are not used at all. With Poppendiecks' terminology, *extra features* may be eliminated. Moreover, reducing metric *D2FU*, which is sum of *deployment time* and *activation time*, enables faster feedback from the actual users to the developer. Third, metric *D2VC* helps to identify features which are used infrequently compared to the amount of development effort. The metrics helps to recognize features with high or low usage. This information can then be utilized in future decisions in order to create value and eliminate waste. The novel metrics presented provide new information on the process. Existing metrics related to, for instance, throughput [96] or *flow efficiency*, already provide meaningful information from another point of view. The combination of the metrics and visualization presented in the following, presents a novel approach for value creation management.

6.2 Visualizations for continuous value creation

The second research question addresses the possibilities of information visualization of software engineering data:

RQ2: How to construct visualizations to demonstrate value creation in continuous software engineering?

The research question is addressed in Publications P4, P5 and P6. They construct a series of publications where a novel visualization artifact is developed. To demonstrate continuous software engineering effectively, the visualization artifact depicts the key phenomena of continuous software engineering. Cycle time is seen as an important metric in continuous software engineering [73]. Moreover, the concepts of flow and batch size are central concepts in continuous software engineering [48]. The visualization artifact presents information related to the key concepts effectively. The goal is to enable continuous improvement based on the new information the visualization provides on the development process. The developed visualization artifact demonstrates several aspects of a software development process. Visual representations have multiple benefits which were introduced in detail in Subsection 3.2.2.

The first version of the visualization artifact was developed with an Action Research approach in Publication P4. The publication introduces a visual representation of an agile software development process. The visualization reveals patterns in the data which are not easy to recognize otherwise. For instance, it characterizes the properties of a Scrum sprint [149]. The visualization reveals that the sprint length of the target process is longer than the team claims. Moreover, the flow of the features in the different phases of development can be evaluated. The visualization reveals problems in the flow related to a large number of parallel tasks. The observations the visualization showed to the team were not totally new. However, the visualization provided extra information about the phenomena regarding the development process. For instance, the intensity of communication in the issue management system was new information to the team. Similar timeline based visualization approach has been applied in [30] to show where bugs are concentrated. However, the approach does not show detailed information related to the process. There are also existing solutions that combine version control data and bug system data [47] in order to provide information release history of a project.

Publication P5 presents a mash-up visualization that combines information from multiple data sources. The visualization makes three software engineering phenomena visible. First, the realization of continuous delivery can be observed from the visualization. The visualization shows the trend of continuous delivery based on data from the version control system. If features are delivered continuously, the visualization demonstrates that the

amount of inventory (or waste) stays low. The visualization technique is similar to the information a Kanban board contains [92, 119]. The bottlenecks of the process are revealed in the same way. Second, the *development time* and *deployment time* of features can be seen. The visualization uses issue management data, version control system data and production logs as data sources. This way, the lead time can be tracked from the initial creation of an issue till the actual usage of a feature in the production environment. Finally, the usage frequency of the features can be observed visually. The visualization reveals the variance in the density of usage of features. With this information it is possible to evaluate which new features are used and which are not. The visualization could be used as an ambient visualization in the team workspace in order to spread new information continuously.

In Publication P6, the visualization artifact was further developed to demonstrate a continuous software development process. The drawing rules to construct the visualization are presented in the following. Each task of the issue management system is drawn on the timeline. In the issue management system, a single new feature to be implemented can be split to multiple tasks. Thus, a single line drawn on the timeline may present a single feature or a part of a feature. The vertical order of the features is based on two rules. First, the features are ordered by release date ascending from bottom to top. Second, the features are ordered by the date they were done ascending from bottom to top. Moreover, the trend of the amount of delivered features is presented at the bottom of the diagram with a bar chart.

When the presented drawing rules are applied to tens or hundreds of features, the triangular shapes described in detail in Chapter 5 are shown. The triangles depict key attributes of the process: batch size, feedback speed and cycle time. The batch size of a single release is depicted with the height of the triangle. The feedback cycle is presented on the horizontal axis. The feature that was done first in a release has the longest feedback time. Cycle time can be easily observed in the diagram. Reinertsen [143] presents the cumulative flow diagram which contains information related to queues, arrivals, departures and cumulative quantity. Reinertsen presents example diagrams with different kinds of shapes depicting the target process. In this sense, the solution proposed in Publication P6 is similar Reinertsens who emphasizes the importance of presenting process visually.

According to Reinertsen, reducing batch size reduces cycle time and accelerates feedback [143]. Large batch sizes with slow feedback speed cause problems, for instance, reduced efficiency and lower motivation [117, 143].

A reference process shape in Figure 6.2 was constructed based on the verbal descriptions of the team members of the case project in Publication P6. The team reported they have a major release cycle and a parallel minor release cycle.

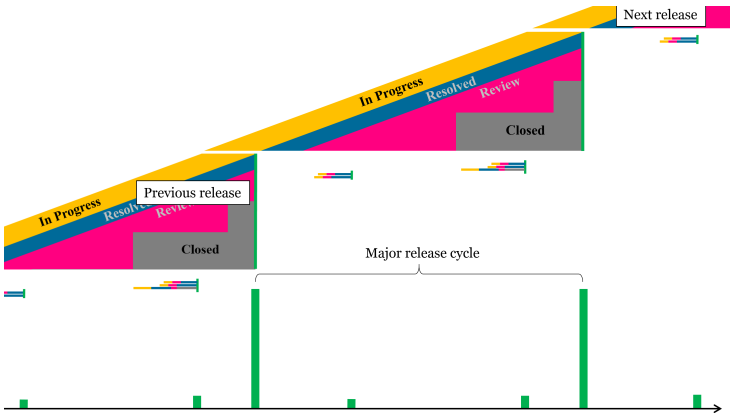

Figure 6.2: A reference process shape with major and minor releases (from Publication P6).

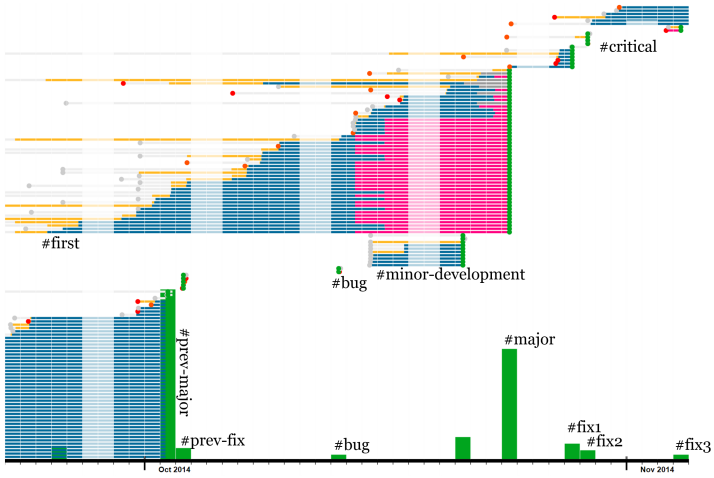

Figure 6.3: A version release with over 50 issues, a parallel minor development release and two fix releases (from Publication P6).

The reference process shape presented consists of one major and two minor releases. The larger major version release consists of tens of tasks. The parallel minor releases are located below the major release. With this information the development organization can find differences between the reference process and the actual process in Figure 6.3.

The two visualizations can be easily compared. Based on visual observations, the actual process mimics the characteristics of the reference process. For instance, the existence of minor development can be ensured. Moreover, the continuity of deliveries can be estimated. With this new information, the development team can ensure if they are following the process they have defined. The team can find possibilities for process improvement.

Existing visualization approaches described in Chapter 4, for instance burndown chart [106] and cumulative flow diagram (CFD) [20, 43, 143] provide partly similar information. However, the process visualization artifact developed in this work contains additional information related to continuous software engineering from two points of view. First, it contains information on multiple concurrent released versions. The separate parallel minor development versions are visible. Second, the visualization summarizes the trend of released versions to the bottom of the diagram. The cycle time can thus be easily observed. Such information is not easily available in existing visualizations.

In the case project, the team reported that the visualization helped to confirm if the quality of the actual software development process is what is intended. According to the team, extra long lead times and number of bug fixes after a version release are effectively revealed by the visualization. The visualization makes it possible to compare the delivered versions. According to the team, the visualization helps to get such an overview of the project which is not available otherwise. The development organization can recognize possible problems in the process with the help of visual representations. The new information can be used in agile retrospective meetings, for instance.

By comparing the visual presentations of the processes, the improvement over time can be perceived. For instance, in the case project, the team did not have a separate parallel minor releases earlier. Kupiainen et al. [96] state that Kanban approach has introduced many useful metrics, for instance, work in progress (WIP). With the visualization, evaluating WIP is a straightforward task. In the visualizations presented in Chapter 5, WIP is revealed. According to the agile coach interviewed in Section 5.4.2, concurrent parallel work is efficiently expressed with the visualization. The team could improve its process based on novel information on WIP.

Moreover, the existence of new issue management system states can be easily perceived. Originally, there was no dedicated review state, but later the team started to use such a state apparently since the process had evolved

during years and a separate step was needed for some reason. t is possible to observe characteristics of the development process based on the information the visualization provides. Reducing batch size reduces cycle time and accelerates feedback [143]. Cycle time of delivered versions can be observed and continuity of delivery can be evaluated. For example, some released versions have long tails which mean delayed feedback for some of the features. By shortening the *deployment time* of features, the tails can be shortened. The visualization helps to improve the process.

The graphical elements in the visualizations present real-world activity by humans. No line would be drawn if there were no people participating in a software project. Visualizations make the invisible visible [143]. From this point of view, the novel visualization artifact is a tool for understanding how human resources are actually developing software. The visualization packs a huge amount of information on the process into a single image which provides new information on the formerly invisible phenomena.

6.3 Managing value creation

Research question RQ3 addresses the usefulness of the developed artifacts in value creation management:

RQ3: How to manage value creation with metrics and visualizations based on automatically generated data?

The use of software development tools generates vast amount of data which provides information on the development process. The metrics and visualizations developed on top of the mined software engineering data help to understand and manage value creation in continuous software engineering. The value creation management in this context is based on three points of view. First, the software development process is a value creation process [140]. By improving the software development process, value creation is improved. In this context, this is enabled by managing the cycle time, batch size, and feedback speed of the development process based on the information the metrics and visualizations provide. Second, value creation is observed through the concept of value-in-use, i.e., the actual usage of the delivered features by the users of the system. Metrics *D2FU* and *D2VC* can be used to observe and improve value creation related to value-in-use.

Third, the decision of which features to implement is not included to value creation management in this context. It is assumed that the features to be implemented have already been selected. Only the development process and value capture through metrics *D2FU* and *D2VC* of the features are taken into consideration. In the following, the results are presented and discussed from these three points of view. A synthesis of the key results related to the novel metrics and visualizations is presented.

In Chapter 3 we found out that "pre-post comparison" is a very common practice in software process improvement [159]. However, a single measure may not be able to show the overall change and benefit of SPI activities. In this sense, a visual approach to support quantitative analysis is promising.

Publication P6 shows the evolution of the development process of the case project. The same project was a target of another case study by Itkonen et al. [77] where the customer and the development team were interviewed. Both stakeholders mentioned several benefits related to value creation due to the process evolution. The collaboration between the developers and the customer was more active and focused on value creation. The customer was more satisfied, because valuable new features were released frequently with shorter latency. The developers described a better, less stressful working environment, improved work morale and job satisfaction. The reduction in stress levels results from the reduced risk of failing deployments because of smaller batches delivered more frequently and elimination of additional work related to manual deployments. In general, long queues in product development have a negative psychological effect [143].

To summarize, value creation can be managed with the novel metrics and visualizations by three means. First, when metric *deployment time* is kept short, feedback is faster, cycle time is shorter and flow efficiency is higher. The team can apply both the metrics and visualization to improve this metric.

Second, the visualization shows if the development process is delivering new features continuously. The cycle time and feedback speed of the delivered batches of new features can be observed visually. Based on this information, process improvements can be planned. For instance, in the presented case project, parallel minor development was a good solution for shortening the cycle time and accelerating feedback.

Third, when the team strives to measure and improve D2VC, the focus is put to value-in-use. Feedback from the users is acquired actively. The

team continuously receives new information on value creation and can make decisions with rapid cycles.

With the metrics and visualizations the team can get such new information on the process which is not easily available otherwise. When the development process is driven towards the continuous delivery of new features in small batches with fast feedback from the users, both the customer and the developers are more satisfied. Value is created continuously.

6.4 Data sources for the data model

The fourth research question addresses the origins of beneficial software engineering data. A key constraint set to the artifacts developed in this work is that no extra work is needed to utilize them.

RQ4: Which data for metrics and visualizations is automatically generated by the tools used in software development?

Continuous software engineering [48] relies heavily on tools. When the development team uses the tools, a vast amount of data concerning the actual software engineering activities is automatically generated. In the context of this thesis, the toolset for feature-driven development [128] consists of three separate systems. Firstly, for each feature developed, there is a task in an issue management system. Using an issue management system is a common practice in software projects. However, not every development project uses an issue management system and thus this information source may not always be available. In this context, we concentrate on development projects that manage their work with an issue management system. Secondly, the change sets to implement the feature are committed to a version control system. This data can be used to track the actual development events. Finally, the production logs are collected and mined with a monitoring platform. This data source can be used to evaluate the actual usage of the feature, i.e., value-in-use presented in Subsection 3.1.1.

With the data sources presented in Figure 6.4, the life cycle of a single feature can be tracked. The events from the beginning of the development till the actual use in the production environment can be tracked. Figure 6.4 presents the event types and data sources for the software engineering

events in the context of this thesis. The data model is used as a basis for the visualizations and metrics presented in this work.

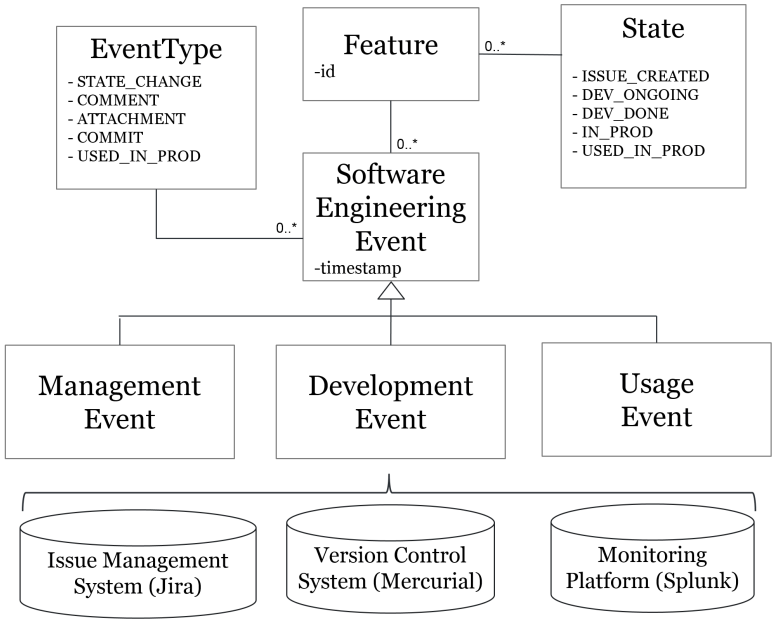

Figure 6.4: Data model of various software engineering events and their sources with sample systems (from Publication P4).

The key concept in the data model is a *feature*. It has a unique identifier which originates from the issue management system. The data model relies on an assumption that the same identifier is used in all tools which are used in the development. The id can then be utilized to track the software engineering activities related to a single feature. To achieve this, the development team may have to take the use of certain practices. For instance, the id of the issue may have to be included in commit messages.

There is a one-to-many relationship from a feature to the software engineering *events*. Each event has a timestamp and an *event type* which may be for instance a state change or a code commit to a feature. The concrete data types of the events are related to management, development and usage of the feature. The data sources for the different types of events at the bottom of Figure 6.4 are: an issue management system, a version control system, and production logs of the monitoring platform. The different event types enable an accurate data collection mechanism concerning the life-cycle of a feature. The events related to development, deployment and usage of the features can be tracked accurately.

Figure 6.5 from Publication P1 presents a state-of-the-art deployment pipeline based on feature branches [36]. The tools used to implement the pipeline produce detailed data related to the life-cycle of features developed.

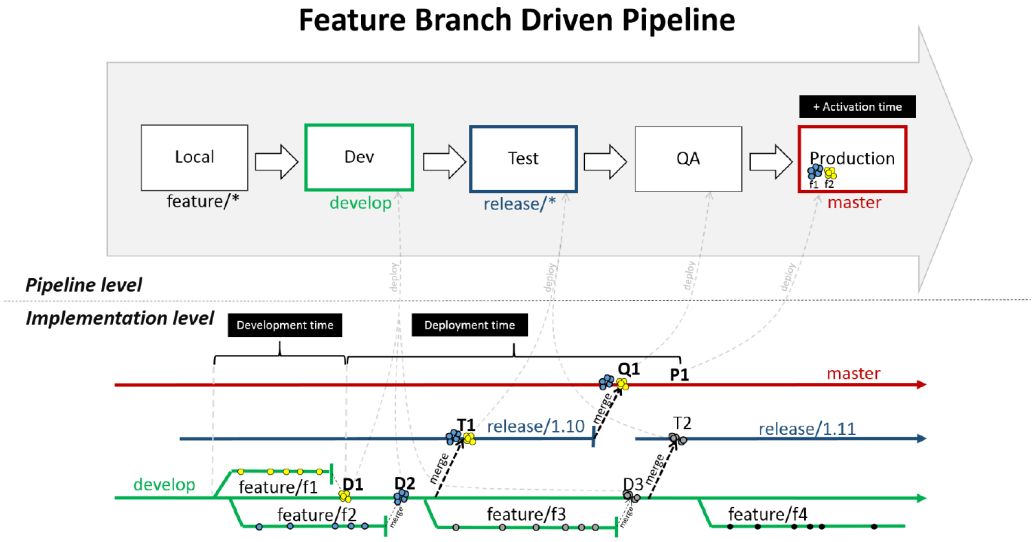

Figure 6.5: A deployment pipeline based on feature branches (from Publication P1).

When the developers use the various development tools related to the deployment pipeline, data for various metrics is produced automatically. The starting time of the development of a new feature is simply the time when the feature branch is created. When the feature is done, the branch is closed and then merged back to the master branch. Moreover, the deployment time of a feature can be tracked accurately from the version control system data. Finally, the timestamps related to the usage of a feature can be tracked from the production logs.

Detailed data concerning the life-cycle of a feature can be stored with the data model presented. The physical implementation of the data model, e.g., a relational database, can then be used as a source for data analysis concerning the software engineering activities. A more detailed description of the physical implementation of the data model and the exact data collection methods are presented in Publications P1, P4 and P5.

6.5 Validity of the research

In this section we address the trustworthy of this research, validity, credibility, reliability and transferability to other domains. The guidelines presented by Hevner [71] are applied to address the validity of this research. The guidelines originally presented in [168] are:

- design as an artifact
- problem relevance
- design evaluation
- research contributions
- research rigor
- design as a search process
- communication of research.

In the following, each of the guidelines is evaluated related to the research conducted in this work.

Design science research must produce a viable artifact. The visualization tool developed in this work in the iterative process described in Section 2.3 is a novel tool for understanding and managing a software development process. The artifact has been developed in an iterative manner in the publications. Moreover, the metrics developed in this work are viable artifacts. The data for the metrics is collected automatically.

The objective in any design science research is to develop technological solutions to relevant business problems. The presented solution in this work is relevant from three points of view. Firstly, continuous improvement needs timely information in order to be efficient. Secondly, the usage of software development practices needs continuous attention on the development of the practices in general. Finally, making novel software engineering concepts visible in industrial context is needful for the co-operation between practitioners and scientists.

The utility, quality and efficacy of a design artifact have to be rigorously demonstrated via well-executed evaluation methods. The iterative development process of the artifacts in co-operation with several industrial case projects has evolved the artifacts with rigorous evaluation methods during the process. The spontaneous needs emerged from the case projects have been fulfilled.

Effective design science research must provide clear and verifiable contributions in the areas of the design artifact. The contributions of the research conducted in this thesis have been concrete. The visualizations and metrics developed have been applied in industrial project contexts and communicated in international scientific conferences.

Design science research relies upon the application of rigorous methods in the construction and evaluation of the artifact. The context where the artifacts have been developed is business centric and industrial. In such an environment accuracy of the developed artifacts have been tested by the practitioners. Errors have been fixed continuously during the development work in an iterative manner based on the statements presented by the practitioners.

The search process of an effective artifact requires utilizing available means to reach desired ends while satisfying laws on the environment of the problem. The design of the artifacts in this work has been a search problem of finding an effective solution. An iterative approach to find solutions that satisfy the business constraints has been used.

Finally, the seventh guideline related to the communication of the research was fulfilled. The technology oriented audiences in the industrial context have validated the work in an iterative manner throughout the whole research process. The customers of the case projects have been the management-oriented audience that has been part of communication related to the artifacts produced in this work.

In general, the validity of quantitative research is defined in [55] by the traditions of positivist roots as whether the research truly measures what it was intended to measure or how truthful the research results are.

Walliman [169] defines data seen as bits of information of several levels of abstraction, namely (from the higher level to the lower level of abstraction) theory, concepts, indicators, variables and values. In addition to this,

the research activities (from higher to lower level of abstraction) are defined as follows: the main question, the sub questions, data types, data measures and measurements. The most concrete level of the former, *values* of *measurements*, define the basis of the new information created in the research. In this thesis, the lowest level of data used in the measurements has been collected from various tools used in the actual development work. In Figure 6.4 we defined the data model for software engineering data and its three data sources. The issue management system data is prone to human errors, since the system is maintained as a side-effect of the actual development work. For instance, a developer or a customer may forget to update the issue state when actual work has already been done. Thus, the correctness of the visualizations presented in this work is not absolutely correct. However, it can be assumed that the typical usage patterns of the tools provide accurate data.

Rosli et al. in their article "Can We Trust Our Results? A Mapping Study on Data Quality" [146] propose a model for describing the lifecycle of research data that goes through in a research. Their systematic mapping study performed defines the terms used for data quality problems as missing data, noisy data, outlier, duplicate data, invalid data, incomplete data, incorrect data, inconsistent data and inaccurate data.

Data Type / Dimensions	**Data source category**	**Example system**	**Data quality problems**
Management Data	Issue management system	Jira	missing data, duplicate data, incomplete data, incorrect data, inaccurate data
Development Data	Version control system	Mercurial	inaccurate data
Usage Data	Monitoring platform	Splunk	inaccurate data

Table 6.1: Data types and their source, sample and data quality problems

Table 6.1 presents data quality problems related to this work presented with vocabulary of the mapping study in [146]. The quality problems for the visualization presented in this research are many. Still, an adequate quality to present the results can be assumed since the subjects use the tools on a day-to-day basis in a way that is rigor. For instance, the issue management system data to create the visualizations has been collected from

a project setting where the team uses the tools together with the customer in an intensive manner, which reduces the possibility of e.g. missing or incomplete data. Moreover, some of the visualizations use version control system data as a source. The data in the version control system reflects the actual events during the development work. However, the actual occurrence of the development work and the data in the version control system may vary. For instance, the developer may write the code for the implementation of a feature and then forget to commit the change sets. This way, the timestamps of the actual development work done and the timestamps of the data may differ.

Researcher bias [53] or a situation in which the researcher's hopes and expectations may affect the results in this kind of industrial research setting is unavoidable thereby creating a threat to internal validity. However, the research conducted in this thesis is quantitative in its nature thus removing many of the problems related to qualitative research. Ethnological research attempting to represent the totality of social, cultural and economic situation [169] faces the problems of the ambiguous meanings of concepts and words and terms related to them. For instance, in this context, interviewing a development team members may result in a situation where the researcher had one meaning to a term while the respondent had another and thus, the epistemological considerations on for example the quality of the collected data are many. Furthermore, any idealistic approach [169] to collect data including human interaction is affected by several biases. In this thesis, materialism (or reductionism) is emphasized. This results in a situation where phenomena are independent of the social factors of the people involved.

From the epistemological point of view [62, 138, 169], an empirical approach in the industrial setting of this research instead of a rationalistic descriptive reasoning is a clear strength of the thesis. The positivist [169] approach of this research of assuming the realism related to software engineering as a technological field of research, leads to results that are convincing and repeatable. The visualization methods described in this research are possible to repeat in numerous software project settings using similar kind of toolset for issue management, development and usage monitoring of the end-users of the system. Moreover, in terms of transferability [58], the results related to the domain of software engineering projects, could be extended to any other domains using digital tools for working, in a broader sense, to any information system.

However, the research conducted, relies heavily on relativism [169] – a

view of the world as a creation of the mind. The process of information visualization is in fact a process of converting existing physical facts from data intensive binary data format to a single visual presentation or a figure that enables the possibilities for speculative knowledge communication performed by the human mind. Thus, the approach applied in this work is naturally both positivism and relativism in a sense of avoiding the key problems in post-modernism, where the meaning of words and discussion in general are ambiguous. The visualization methods developed in this thesis, form an ambiguous language of symbols. For instance, the shapes in the developed visualizations represent truth in a way that is simultaneously both self-evident, ambiguous and speculative. However, the evolution or change of the software engineering in the visualization is self evident but only production of the human mind.

6.6 Limitations

The research was conducted in an industrial context inside a single company. This is a clear limitation of the research. The artifacts developed in this research should be tested in more than one company. However, the case projects chosen for the studies represent an adequate variety of projects as both public and private sector projects were selected, and the size of the projects varied. Moreover, the personnel of the case projects consisted of different people. Therefore, the results are not limited to a homogeneous set of exactly similar projects.

The transferability of the results is satisfactory. The tools the data used in the process relies on are commonly used in software engineering. By using a similar set of tools in a project setting enables the use of the developed artifacts. Moreover, the ideas behind the visualizations developed could possibly be applied to other domains.

6.7 Future work

Possibilities for future work are many. The designed artifacts presented in this work could be applied to a wider range of software projects in diverse

contexts. We could apparently recognize the patterns of fluent processes and develop a tool that helps in improving the process towards a healthy and efficient process. Based on the observations, the artifacts could be further developed.

The visualizations could be developed further based on feedback from the actual users of the visualizations. For instance, Gantt chart [111] could be a suitable layout for showing the software development process data. The amount of details in the visualizations could be either lower or higher in order to emphasize relevant data. With feedback from the actual users of the visualizations, the developed artifacts could be improved further. Fortunately, in Need 4 Speed[1] research program we have an opportunity to collaborate with Finnish universities. The researchers at TUT have been implementing a visualization tool which has been published as an open source project[2]. We could develop the tool further and use it to mine software engineering data from several companies. Need 4 Speed research program is still going on for half a year until May, 2017. This is a tangible opportunity for future work.

The publications of this thesis have also reached international interest. The researcher was contacted by a chief technical officer (CTO) of a large international web market platform. The CTO had read the publications and was interested in the visualizations. Especially the visualizations of the issue management data were considered relevant. Therefore, the developed visualization artifact could be published as an open source project, for instance, in order to reach wider audience.

Moreover, the data sources for visualizations could cover data sets related to usage of the features in a more comprehensive way. We could apply the visualizations as a basis for continuous experimentation where the developers could experiment in a lightweight manner. The information could be presented directly in the development environment in order to experiment continuously on a daily basis. Moreover, the data for the metrics could be directly available for the developers both in the information radiator and the development environment. Furthermore, the metrics presented could be developed further. For instance, metrics D2VC is promising but the threshold value is not easy to use. Therefore, for instance, the threshold value could be replaced with exact number of usages. To present this as a question: how

[1]http://www.n4s.fi/en/

[2]https://bitbucket.org/rimina/n4s-visu

long does it take until the feature has been used, say, 10 times? Is it one day or 21 days? This could characterize the features.

The developed artifacts could be applied to other domains outside software engineering. The visualization artifact developed in this work could provide valuable information to any task-oriented development process in any domain. The tool could provide valuable information for continuous improvement of any process. Moreover, the metrics presented in this work could provide valuable information for improvement.

In the field of analytics there are many promising possibilities. For instance, applying predictive analytics to the data set in order to predict future events could be beneficial. For instance, machine learning algorithms could help in value creation management.

Chapter 7

Conclusions

Visualizations and metrics presented in this thesis form a novel basis for continuous software process improvement. Software engineering tools generate a significant amount of data which can be used for software process improvement purposes.

We used visualizations and metrics to demonstrate the characteristics of software development processes from an industrial context. Visual representations help to understand and evaluate the software development process. The metrics and the visualizations presented in this thesis help to improve value creation from three points of view.

First, the metrics *deployment time* in combination with the visualization artifact developed, helps to manage cycle time, batch size and feedback speed of the process. The visualization provides information on the underlying software development process. According to the qualitative analysis, the visualization can be understood by professionals working in the field of software engineering. The visualization could be beneficial for agile retrospective purposes, for instance.

Second, the novel metrics *D2FU* and *D2VC*, put focus to value-in-use. With these metrics, the team is able to get information on the actual usage of the features. With this information the team may put focus into value creation. By continuously receiving new information on the usage of the features, waste of type *extra features* may be eliminated. By focusing to

improvement of *D2VC* the development organization focusing to continuous value creation.

Finally, the demonstrations of novel software engineering phenomena in an industrial context are contributions themselves. Visualizations representing continuous delivery in industrial software projects produces new information from the practitioners to the researchers. Moreover, the representations of projects shifting towards continuous software engineering are compelling. With the metrics and visualizations developed in this work, this kind of transformation can be explained and understood more clearly.

Bibliography

[1] P. Abrahamsson. Measuring the success of software process improvement: the dimensions. In *Proceedings of the European Software Process Improvement (EuroSPI2000) Conference*, 2000.

[2] D. M. Ahern, A. Clouse, and R. Turner. *CMMI distilled: a practical introduction to integrated process improvement*. Addison-Wesley Professional, 2004.

[3] R. Akerkar. Advanced data analytics for business. *Big data computing*, pages 377–379, 2013.

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

[5] V. R. Basili. Applying the goal/question/metric paradigm in the experience factory. *Software Quality Assurance and Measurement: A Worldwide Perspective*, pages 21–44, 1993.

[6] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[8] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. The agile manifesto. `http://agilemanifesto.org`, 2001. Retrieved: November 2015.

[9] J. T. Behrens and C.-H. Yu. Exploratory data analysis. *Handbook of psychology*, 2003.

[10] S. Berczuk. Back to basics: The role of agile principles in success with an distributed scrum team. In *Agile Conference (AGILE), 2007*, pages 382–388. IEEE, 2007.

[11] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher. *Value-based software engineering*. Springer Science & Business Media, 2006.

[12] E. Bjarnason and B. Regnell. Evidence-based timelines for agile project retrospectives–a method proposal. In *International Conference on Agile Software Development*, pages 177–184. Springer, 2012.

[13] P. Bock and B. Scheibe. *Getting it right: R & D methods for science and engineering*. Academic Press, 2001.

[14] L. Bodo, H. C. de Oliveira, F. A. Breve, and D. M. Eler. Performance indicators analysis in software processes using semi-supervised learning with information visualization. In *Information Technology: New Generations*, pages 555–568. Springer, 2016.

[15] A. Börjesson, A. Baaz, J. Pries-Heje, and M. Timmerås. Measuring process innovations and improvements. In *IFIP International Working Conference on Organizational Dynamics of Technology-Based Innovation*, pages 197–216. Springer, 2007.

[16] J. Bosch. *Continuous Software Engineering*. Springer, 2014.

[17] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[18] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, pages 322–333. ACM, 2014.

[19] R. P. Buse and T. Zimmermann. Analytics for software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 77–80. ACM, 2010.

[20] A. Cabri and M. Griffiths. Earned value and agile reporting. In *AGILE 2006 Conference (AGILE 2006), 23-28 July 2006, Minneapolis, Minnesota, USA*, pages 17–22, 2006.

[21] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[22] K. Charmaz and J. Smith. Grounded theory. *Qualitative psychology: A practical guide to research methods*, pages 81–110, 2003.

[23] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *Computer Graphics and Applications, IEEE*, 18(4):24–29, 1998.

[24] E. Coelho and A. Basu. Effort estimation in agile software development using story points. *International Journal of Applied Information Systems (IJAIS)*, 3(7):7–10, 2012.

[25] M. A. Cohen, J. Eliasberg, and T.-H. Ho. New product development: The performance and time-to-market tradeoff. *Management Science*, 42(2):173–186, 1996.

[26] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato. *Version control with Subversion*. O'Reilly Media, Inc., 2004.

[27] K. A. Cook and J. J. Thomas. Illuminating the path: The research and development agenda for visual analytics. Technical report, Pacific Northwest National Laboratory (PNNL), Richland, WA (US), 2005.

[28] A. Cooper et al. What is analytics? definition and essential characteristics. *CETIS Analytics Series*, 1(5):1–10, 2012.

[29] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284. IEEE, 2011.

[30] M. D'Ambros, M. Lanza, and M. Pinzger. "a bug's life" visualizing a bug database. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 113–120. IEEE, 2007.

[31] T. H. Davenport and J. G. Harris. *Competing on analytics: The new science of winning*. Harvard Business Press, 2007.

[32] R. DeLine. Research opportunities for the big data era of software engineering. In *Big Data Software Engineering (BIGDSE), 2015 IEEE/ACM 1st International Workshop on*, pages 26–29. IEEE, 2015.

[33] P. J. Denning. A new social contract for research. *Communications of the ACM*, 40(2):132–134, 1997.

[34] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[35] A. Dix. *Human-computer interaction*. Springer, 2009.

[36] V. Driessen. A succesful Git brancing model. `http://nvie.com/posts/a-successful-git-branching-model/`. Retrieved: August 2015.

[37] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9):833–859, 2008.

[38] C. Ebert, P. Abrahamsson, and N. Oza. Lean software development. *IEEE Software*, 29(5):22–25, 2012.

[39] C. Ebert, R. Dumke, M. Bundschuh, and A. Schmietendorf. *Best Practices in Software Measurement: How to use metrics to improve project and process performance*. Springer Science & Business Media, 2005.

[40] M. El-Ramly and E. Stroulia. Mining software usage data. In *Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04)*, pages 64–8, 2004.

[41] K. E. Emam, W. Melo, and J.-N. Drouin. *SPICE: The theory and practice of software process improvement and capability determination*. IEEE Computer Society Press, 1997.

[42] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala. Organizing the tasks in complex design projects. In *Computer-Aided Cooperative Product Development, MIT-JSME Workshop, MIT, Cambridge, USA, November 20/21, 1989, Proceedings*, pages 229–252, 1989.

[43] G. Evans. Pattern language of flow release steps. `https://garethevans.geek.nz/2012/01/23/pattern-language-of-flow-release-steps/`, 2016. Accessed: 2016-11-10.

[44] A. Fabijan, H. H. Olsson, and J. Bosch. Early value argumentation and prediction: an iterative approach to quantifying feature value. In *International Conference on Product-Focused Software Process Improvement*, pages 16–23. Springer, 2015.

[45] F. Fagerholm, A. S. Guinea, H. Mäenpää, and J. Münch. Building blocks for continuous experimentation. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 26–35. ACM, 2014.

[46] F. Fagerholm, M. Ikonen, P. Kettunen, J. Münch, V. Roto, and P. Abrahamsson. How do software developers experience team performance in lean and agile environments? In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, number 7. ACM, 2014.

[47] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[48] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9. ACM, 2014.

[49] B. Fitzgerald and K.-J. Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2015.

[50] W. A. Florac and A. D. Carleton. *Measuring the software process: statistical process control for software process improvement*. Addison-Wesley Professional, 1999.

[51] M. Fowler. Continuous delivery. `http://martinfowler.com/bliki/ContinuousDelivery.html`. Retrieved: June 2016.

[52] M. Fowler and M. Foemmel. Continuous integration, 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`, 2012. Retrieved: June 2016.

[53] J. R. Fraenkel, N. E. Wallen, and H. H. Hyun. *How to design and evaluate research in education*, volume 7. McGraw-Hill New York, 1993.

[54] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 99–108. IEEE, 1999.

[55] N. Golafshani. Understanding reliability and validity in qualitative research. *The qualitative report*, 8(4):597–606, 2003.

[56] T. Gollisch and M. Meister. Eye smarter than scientists believed: neural computations in circuits of the retina. *Neuron*, 65(2):150–164, 2010.

[57] R. B. Grady. *Successful software process improvement*. Prentice-Hall, Inc., 1997.

[58] U. H. Graneheim and B. Lundman. Qualitative content analysis in nursing research: concepts, procedures and measures to achieve trustworthiness. *Nurse education today*, 24(2):105–112, 2004.

[59] K. Greaves. Taming the customer support queue. 2011.

[60] S. Gregor. The nature of theory in information systems. *MIS quarterly*, 30(3):611–642, 2006.

[61] C. Grönroos and P. Voima. Critical service logic: making sense of value creation and co-creation. *Journal of the Academy of Marketing Science*, 41(2):133–150, 2013.

[62] E. G. Guba and Y. S. Lincoln. Epistemological and methodological bases of naturalistic inquiry. *ECTJ*, 30(4):233–252, 1982.

[63] E. Gummesson. Exit services marketing-enter service marketing. *Journal of Customer Behaviour*, 6(2):113–141, 2007.

[64] A. Gustafsson, S. Brax, L. Witell, C. Grönroos, and P. Helle. Adopting a service logic in manufacturing: Conceptual foundation and metrics for mutual value creation. *Journal of Service Management*, 21(5):564–590, 2010.

[65] M. Guzdial. Deriving software usage patterns from log files. 1993.

[66] M. Guzdial, P. J. Santos, A. Badre, S. E. Hudson, and M. H. Gray. Analyzing and visualizing log files: A computational science of usability. 1994.

[67] D. Hartmann and R. Dymond. Appropriate agile measurement: Using metrics and diagnostics to deliver business value. In *AGILE 2006 Conference (AGILE 2006), 23-28 July 2006, Minneapolis, Minnesota, USA*, pages 126–134, 2006.

[68] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[69] M. Hassenzahl and N. Tractinsky. User experience-a research agenda. *Behaviour & information technology*, 25(2):91–97, 2006.

[70] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The msr cookbook: Mining a decade of research. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 343–352. IEEE, 2013.

[71] A. Hevner and S. Chatterjee. *Design research in information systems: theory and practice*, volume 22. Springer Science & Business Media, 2010.

[72] C. Hibbs, S. Jewett, and M. Sullivan. *The art of lean software development: a practical and incremental approach.* " O'Reilly Media, Inc.", 2009.

[73] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[74] W. S. Humphrey. *Managing the software process.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[75] J. Hunt. Feature-driven development. *Agile Software Construction*, pages 161–182, 2006.

[76] H. Ishii and B. Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 234–241. ACM, 1997.

[77] J. Itkonen, R. Udd, C. Lassenius, and T. Lehtonen. Perceived benefits of adopting continuous delivery practices. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*, pages 42:1–42:6, 2016.

[78] S. Jaitly, A. K. Mishra, and L. Singh. A systematic review on the impact of metrics in software process improvement. *Compusoft*, 3(3):624, 2014.

[79] P. Järvinen. Action research is similar to design science. *Quality & Quantity*, 41(1):37–54, 2007.

[80] C. Johnson. Top scientific visualization research problems. *Computer graphics and applications, IEEE*, 24(4):13–17, 2004.

[81] P. M. Johnson. Searching under the streetlight for useful software analytics. *IEEE Software*, 30(4):57–63, 2013.

[82] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[83] M. Karlesky and M. Vander Voord. Agile project management. *ESC*, 247(267):p4, 2008.

[84] D. Keim, G. Andrienko, J.-D. Fekete, C. Görg, J. Kohlhammer, and G. Melançon. *Visual analytics: Definition, process, and challenges*. Springer, 2008.

[85] D. Keim et al. Information visualization and visual data mining. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1):1–8, 2002.

[86] D. Keim, F. Mansmann, J. Schneidewind, H. Ziegler, et al. Challenges in visual data analysis. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 9–16. IEEE, 2006.

[87] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of Mozilla Firefox. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 179–188. IEEE, 2012.

[88] T. Kilamo, M. Leppänen, and T. Mikkonen. The social developer: now, then, and tomorrow. In *Proceedings of the 7th International Workshop on Social Software Engineering*, pages 41–48. ACM, 2015.

[89] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, pages 96–107. ACM, 2016.

[90] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[91] S. Kim, T. Zimmermann, M. Kim, A. Hassan, A. Mockus, T. Girba, M. Pinzger, E. J. Whitehead Jr, and A. Zeller. Ta-re: An exchange language for mining software repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 22–25. ACM, 2006.

[92] H. Kniberg. *Lean from the trenches: Managing large-scale projects with Kanban*. Pragmatic Bookshelf, 2011.

[93] B. Kristjánsson and H. van der Schuur. A survey of tools for software operation knowledge acquisition. *Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2009-028*, 2009.

[94] O. Ktata and G. Lévesque. Designing and implementing a measurement program for scrum teams: What do agile developers really need and want? In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 101–107. ACM, 2010.

[95] E. Kupiainen, M. V. Mäntylä, and J. Itkonen. Why are industrial agile teams using metrics and how do they use them? In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 23–29. ACM, 2014.

[96] E. Kupiainen, M. V. Mäntylä, and J. Itkonen. Using metrics in agile and lean software development–a systematic literature review of industrial studies. *Information and Software Technology*, 62:143–163, 2015.

[97] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, and S. Carpendale. Empirical studies in information visualization: Seven scenarios. *Visualization and Computer Graphics, IEEE Transactions on*, 18(9):1520–1536, 2012.

[98] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987.

[99] R. Lengler and M. J. Eppler. Towards a periodic table of visualization methods for management. In *IASTED Proceedings of the Conference on Graphics and Visualization in Engineering (GVE 2007), Clearwater, Florida, USA*, 2007.

[100] M. Leppänen, T. Kilamo, and T. Mikkonen. Towards post-agile development practices through productized development infrastructure. In *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, pages 34–40. IEEE Press, 2015.

[101] J. K. Liker. *The toyota way*. Esensi, 2005.

[102] E. Lindgren and J. Münch. Software development as an experiment system: a qualitative survey on the state of the practice. In *International Conference on Agile Software Development*, pages 117–128. Springer, 2015.

[103] J. D. Little and S. C. Graves. Little's law. In *Building intuition*, pages 81–100. Springer, 2008.

[104] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. "O'Reilly Media, Inc.", 2012.

[105] H. Mäenpää, T. Kilamo, and T. Männistö. In-between open and closed-drawing the fine line in hybrid communities. In *IFIP International Conference on Open Source Systems*, pages 134–146. Springer, 2016.

[106] V. Mahnic and N. Zabkar. Measuring progress of scrum-based software projects. *Elektronika ir Elektrotechnika*, 18(8):73–76, 2012.

[107] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen. On rapid releases and software testing. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 20–29. IEEE, 2013.

[108] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[109] Y. Mashiko and V. R. Basili. Using the gqm paradigm to investigate influential factors for software process improvement. *Journal of Systems and Software*, 36(1):17–32, 1997.

[110] A.-L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väätäjä. Software visualization today: systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, pages 262–271. ACM, 2016.

[111] H. Maylor. Beyond the Gantt chart:: Project management moving on. *European Management Journal*, 19(1):92–100, 2001.

[112] T. Menzies and T. Zimmermann. Software analytics: so what? *IEEE Software*, 30(4):31–37, 2013.

[113] P. Middleton and D. Joyce. Lean software management: Bbc worldwide case study. *IEEE Transactions on Engineering Management*, 59(1):20–32, 2012.

[114] A. Miller. A hundred days of continuous integration. In *Agile Development Conference, AGILE 2008, Toronto, Canda, 4-8 August 2008*, pages 289–293, 2008.

[115] S. Misra and M. Omorodion. Survey on agile metrics and their interrelationship with other traditional development metrics. *ACM SIGSOFT Software Engineering Notes*, 36(6):1–3, 2011.

[116] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272. ACM, 2000.

[117] N. Modig and P. Åhlström. *This is lean: Resolving the efficiency paradox*. Rheologica, 2012.

[118] W. Müller and H. Schumann. Visual data mining. *NORSIGD Info*, 2:2002, 2002.

[119] S. Nakazawa and T. Tanaka. Development and application of kanban tool visualizing the work in progress. In *Advanced Applied Informatics (IIAI-AAI), 2016 5th IIAI International Congress on*, pages 908–913. IEEE, 2016.

[120] M. Natrella, C. Croarkin, and W. Guthrie. Engineering statistics handbook. *Statistical Engineering Division, NIST2003*, 2003.

[121] J. K. Naukkarinen. What engineering scientists know and how they know it. Towards understanding the philosophy of engineering science in Finland. *Tampereen teknillinen yliopisto. Julkaisu-Tampere University of Technology. Publication; 1344*, 2015.

[122] S. Neely and S. Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference,*

AGILE 2013, Nashville, TN, USA, August 5-9, 2013, pages 121–128, 2013.

[123] R. L. Novais, A. Torres, T. S. Mendes, M. Mendonça, and N. Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013.

[124] M. Ohira, R. Yokomori, M. Sakai, K.-i. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: A tool for mining multiple project data. In *International Workshop on Mining Software Repositories (MSR2004)*, pages 42–46, 2004.

[125] T. Ohno. *Toyota production system: beyond large-scale production*. CRC Press, 1988.

[126] A. Ordanini and P. Pasini. Service co-production and value co-creation: The case for a service-oriented architecture (soa). *European Management Journal*, 26(5):289–297, 2008.

[127] A. Paivio. *Imagery and verbal processes*. Psychology Press, 2013.

[128] S. R. Palmer and M. Felsing. *A practical guide to feature-driven development*. Pearson Education, 2001.

[129] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[130] I. Pentinmaki. It measurement: practical advice from the experts. *ACM SIGSOFT Software Engineering Notes*, 28(3):25–25, 2003.

[131] J. Pernstål, R. Feldt, and T. Gorschek. The lean gap: A review of lean approaches to large-scale software systems development. *Journal of Systems and Software*, 86(11):2797–2821, 2013.

[132] M. Petre, E. de Quincey, et al. A gentle overview of software visualisation. *PPIG News Letter*, pages 1–10, 2006.

[133] M. Pikkarainen and A. Mäntyniemi. An approach for using CMMI in agile software development assessments: experiences from three case studies. In *6th International SPICE Conference on Software Process Improvement and Capability Determination*, 2006.

[134] M. Poppendieck. Lean software development. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 165–166. IEEE Computer Society, 2007.

[135] M. Poppendieck and T. Poppendieck. *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

[136] M. Poppendieck and T. Poppendieck. *Implementing lean software development: from concept to cash*. Pearson Education, 2007.

[137] M. Poppendieck and T. Poppendieck. *Leading lean software development: Results are not the point*. Pearson Education, 2009.

[138] K. Popper. *The logic of scientific discovery*. Routledge, 2005.

[139] Z. Racheva, M. Daneva, and L. Buglione. Supporting the dynamic reprioritization of requirements in agile development of software products. In *Software Product Management, 2008. IWSPM'08. Second International Workshop on*, pages 49–58. IEEE, 2008.

[140] Z. Racheva, M. Daneva, and K. Sikkel. Value creation by agile projects: methodology or mystery? In *Product-Focused Software Process Improvement*, pages 141–155. Springer, 2009.

[141] Z. Racheva, M. Daneva, K. Sikkel, and L. Buglione. Business value is not only dollars–results from case study research on agile software projects. In *Product-Focused Software Process Improvement*, pages 131–145. Springer, 2010.

[142] Reese. Jira junkie: Cumulative flow chart: aka a scrummasters dirty little secret! `http://jirajunkie.blogspot.fi/2012/06/cumulative-flow-chart-scrummasters.html`, 2012. Accessed: 2016-11-15.

[143] D. G. Reinertsen. *The principles of product development flow: second generation lean product development*, volume 62. Celeritas Redondo Beach, 2009.

[144] D. F. Rico. *ROI of software process improvement: Metrics for project managers and software engineers*. J. Ross Publishing, 2004.

[145] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo. Continuous deployment of software intensive products and

services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2016.

[146] M. Rosli. Can we trust our results? a mapping study on data quality. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, pages 116–123, Dec 2013.

[147] J. A. Rozum. Concepts on measuring the benefits of software process improvements. Technical report, DTIC Document, 1993.

[148] V. Rubin, C. W. Günther, W. M. Van Der Aalst, E. Kindler, B. F. Van Dongen, and W. Schäfer. Process mining framework for software processes. In *International Conference on Software Process*, pages 169–181. Springer, 2007.

[149] K. Schwaber and M. Beedle. Agile software development with scrum. 2001. *Upper Saddle River, NJ*, 2003.

[150] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[151] T. Skog, S. Ljungblad, and L. E. Holmquist. Between aesthetics and utility: designing ambient information visualizations. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 233–240. IEEE, 2003.

[152] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[153] M. Staron, J. Hansson, R. Feldt, W. Meding, A. Henriksson, S. Nilsson, and C. Hoglund. Measuring and visualizing code stability–a case study at three companies. In *Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on*, pages 191–200. IEEE, 2013.

[154] J. Stasko. *Software visualization: Programming as a multimedia experience*. MIT press, 1998.

[155] G. Svensson and C. Grönroos. Service logic revisited: who creates value? and who co-creates? *European business review*, 20(4):298–314, 2008.

[156] W. F. Tichy. RCS – a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[157] E. R. Tufte. *The visual display of quantitative information*. Graphics Press, 1992.

[158] J. W. Tukey. *Exploratory data analysis*. Reading, Mass., 1977.

[159] M. Unterkalmsteiner, T. Gorschek, A. M. Islam, C. K. Cheng, R. B. Permadi, and R. Feldt. Evaluation and measurement of software process improvement a systematic literature review. *IEEE Transactions on Software Engineering*, 38(2):398–424, 2012.

[160] A. Van Barneveld, K. E. Arnold, and J. P. Campbell. Analytics in higher education: Establishing a common language. *EDUCAUSE learning initiative*, 1:1–11, 2012.

[161] H. Van Der Schuur, S. Jansen, and S. Brinkkemper. Becoming responsive to service usage and performance changes by applying service feedback metrics to software maintenance. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 53–62. IEEE, 2008.

[162] H. van der Schuur, S. Jansen, and S. Brinkkemper. A reference framework for utilization of software operation knowledge. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 245–254. IEEE, 2010.

[163] J. J. van Wijk. Views on visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):421–432, 2006.

[164] M. VanHilst and S. Huang. Mining objective process metrics from repository data. In *SEKE*, pages 514–519. Citeseer, 2009.

[165] M. VanHilst, S. Huang, and H. Lindsay. Process analysis of a waterfall project using repository data. *International Journal of Computers and Applications*, 33(1):49–56, 2011.

[166] S. L. Vargo and R. F. Lusch. Service-dominant logic. *What it is, What it is not, What it Might be*, pages 43–56, 2014.

[167] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, 2009.

[168] R. H. von Alan, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[169] N. Walliman. *Research methods: The basics*. Routledge, 2010.

[170] J. G. Walls, G. R. Widmeyer, and O. A. El Sawy. Building an information system design theory for vigilant eis. *Information systems research*, 3(1):36–59, 1992.

[171] X. Wang, K. Conboy, and O. Cawley. Leagile software development: An experience report analysis of the application of lean approaches in agile software development. *Journal of Systems and Software*, 85(6):1287–1299, 2012.

[172] C. Ware. *Information visualization: perception for design*. Elsevier, 2012.

[173] R. Weber. Still desperately seeking the IT artifact. *MIS quarterly*, 27(2):183–183, 2003.

[174] P. Weißgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 9–9. IEEE, 2007.

[175] E. Whitworth and R. Biddle. Motivation and cohesion in agile teams. In *Agile processes in software engineering and extreme programming*, pages 62–69. Springer, 2007.

[176] J. P. Womack and D. T. Jones. *Lean thinking: banish waste and create wealth in your corporation*. Simon and Schuster, 2010.

[177] W. Wright. Research report: Information animation applications in the capital markets. In *Information Visualization, 1995. Proceedings.*, pages 19–25. IEEE, 1995.

[178] S. Yasutaka, S. Matsumoto, S. Saiki, and M. Nakamura. Visualizing software metrics with service-oriented mining software repository for reviewing personal process. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 549–554. IEEE, 2013.

[179] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie. Software analytics in practice. *IEEE Software*, 30(5):30–37, 2013.

[180] X. Zhang, C. Wang, Z. Li, J. Zhu, W. Shi, and Q. Wang. Exploring the sequential usage patterns of mobile internet services based on markov models. *Electronic Commerce Research and Applications*, 17:1–11, 2016.

Publication I

T. Lehtonen, S. Suonsyrjä, T. Kilamo, T. Mikkonen. Defining Metrics for Continuous Delivery and Deployment Pipeline. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST)*, 2015.

Defining Metrics for Continuous Delivery and Deployment Pipeline

Timo Lehtonen[1], Sampo Suonsyrjä[2], Terhi Kilamo[2], and Tommi Mikkonen[2]

[1]Solita Ltd, Tampere, Finland
timo.lehtonen@solita.fi
[2]Tampere University of Technology, Tampere, Finland
sampo.suonsyrja@tut.fi, terhi.kilamo@tut.fi, tommi.mikkonen@tut.fi

Abstract. Continuous delivery is a software development practice where new features are made available to end users as soon as they have been implemented and tested. In such a setting, a key technical piece of infrastructure is the development pipeline that consists of various tools and databases, where features flow from development to deployment and then further to use. Metrics, unlike those conventionally used in software development, are needed to help define the performance of the development pipeline. In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc. As concrete data, we use data from project "Lupapiste", a web site for managing municipal authorizations and permissions.

Keywords: Agile measurements, continuous deployment, lean software development.

1 Introduction

Software development, as we know it today, is a demanding area of business with its fast-changing customer requirements, pressures of an ever shorter time-to-market, and unpredictability of market [1]. Lean principles, such as "Decide as late as possible", have been seen as an attractive way to answer to these demands by academics [2]. With the shift towards modern continuous deployment pipelines, releasing new software versions early and often has become a concrete option also for an ever growing number of practitioners.

As companies, such as Facebook, Atlassian, IBM, Adobe, Tesla, and Microsoft, are going towards continuous deployment [1], we should also find ways to measure its performance. The importance of measuring the flow in lean software development was identified already in 2010 by [3], but with the emergence of continuous deployment pipelines, the actual implementation of the Lean principles has already changed dramatically [4]. Further on, measuring has been a critical part of Lean manufacturing long before it was applied to software development [5]. However, the digital nature of software development's approach

to Lean (ie. continuous deployment pipelines) is creating an environment, where every step of the process can be traced and thus measured in a way that was not possible before. Therefore, the need for a contemporary analysis of what should be tracked in a continuous deployment pipeline is obvious to us.

In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc (`http://www.solita.fi`). As case studies investigate the contemporary phenomena in their authentic context, where the boundaries between the studied phenomenon and its context are not clearly separable [6], we use concrete data from project "Lupapiste", or "Permission desk" (`https://www.lupapiste.fi`), a web site for managing municipal authorizations and permissions. The precise research questions we address are the following:

RQ1: Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?
RQ2: How should the pipeline or associated process be modified to support the metrics that escape the data that is presently available?
RQ3: What kind of new metrics based on automatically generated data could produce valuable information to the development team?

The study is based on quantitative data and descriptions of the development processes and the pipeline collected from the developer team. Empirical data of the case project was collected from information systems used in the project, including a distributed version control system (Mercurial VCS) and a monitoring system (Splunk).

The rest of this paper is structured as follows. In Section 2, we address the background of this research. In Section 3, we introduce our case project based on which the research has been conducted. In Section 4, we propose metrics for continuous delivery and deployment pipeline. In Section 5, we discuss the results of case study and provide an extended discussion regarding our observations. In Section 6 we draw some final conclusions.

2 Background and Related Work

Agile methods – such as Scrum, Kanban and XP to name a few examples – have become increasingly popular approaches to software development. With Agile, the traditional ways of measuring software development related issues can be vague. The outcome of traditional measures may become dubious to the extent of becoming irrelevant. Consequently, one of the main principles of Agile Software Development is "working software over measuring the progress" [7].

However, not all measuring can be automatically considered unnecessary. Measuring is definitely an effective tool for example for improving Agile Software Development processes [8], which in turn will eventually lead to better software. A principle of Lean is to cut down waste that processes produce as well as parts of the processes that do not provide added value [9]. To this end, one should first

recognize the current state of a process [3]. This can be assisted with metrics and visualizations, for instance. Therefore, one role for the deployment pipeline is to act as a manifestation of the software development process and to allow the utilization of suitable metrics for the entire flow from writing code to customers using the eventual implementation [10].

Overall, the goal of software metrics is to identify and measure the essential parameters that affect software development [11]. Mishra and Omorodion [11] have listed several reasons for using metrics in software development. These include making business decision, determining success, changing the behavior of teammates, increasing satisfaction, and improving decision making process.

2.1 Continuous Delivery and Deployment

Continuous delivery is a software development practise that supports the lean principles of "deliver as fast as possible" and "empower the team". In it the software is kept deployable to the staging and production environments at any time [12, 13]. Continuous delivery is preceded by continuous integration [14, 15] where the development team integrates its work frequently on a daily basis. This leads to a faster feedback cycle and to benefits such as increased productivity and improved communication [15–17]. Similarly, "the final ladder " — continuous deployment — requires continuous delivery. So, continuous deployment [18, 19] takes one step further from delivery. In it software is automatically deployed as it gets done and tested. Taking continuous deployment to the extreme would mean deployment of new features directly to the end users several times a day [20, 21]. Whether software is deployed all the way to production, or to a staging environment is somewhat matter of opinion [18, 22] but a reasonable way to differentiate between delivery and deployment in continuous software development is the release of software to end users. Delivery maintains a continuously deployable software, deployment makes the new software available in the production environment.

Regardless of actual deployment, continuous software development requires a deployment pipeline (Figure 1) [10], which uses an automated set of tools from code to delivery. The role of these tools is to make sure each stakeholder gets a timely access to the things they need. In addition, the pipeline provides a feedback loop to each of the stakeholders from all stages of the delivery process. An automated system is not about software going into production without any operator supervision. The point of the automated pipeline is that as the software progresses through it, different stages can be triggered for example by operations and test teams by the click of a button.

2.2 Agile Metrics

In [8] the authors categorize agile metrics used in industry into metrics relating to iteration planning and tracking, motivation and improvement, identifying process problems, pre-release and post-release quality, and changes in the processes or tools. The metrics for iteration planning offered help with prioritization

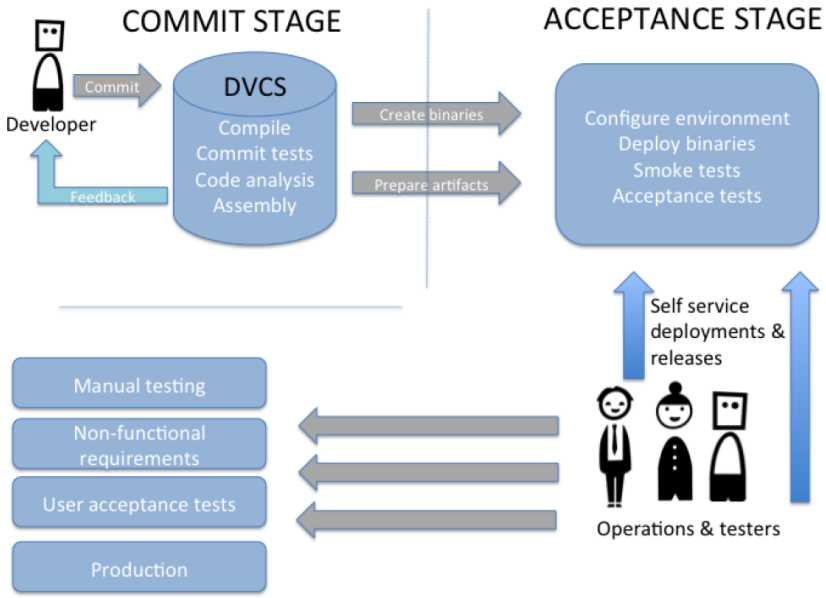

Fig. 1. Anatomy of a Deployment Pipeline according to [10].

of features. These include estimation metrics for measuring the size of features, the revenue a customer is willing to pay for a feature, and velocity of the team in completing a feature development. Iteration tracking include progress metrics such as the number of completed web pages, story completion percentage, and again velocity metrics. In the category of motivation and improvement, approaches such as visualizing the build status and showing the number of defects in monitors were found to lead into faster build and fix times. Using metrics such as lead time and story implementation flow assist in identifying waste and in describing how efficiently a story is completed compared to the estimate. Pre-release quality metrics were found to be used for making sure the product is tested sufficiently and for avoiding integration fails. Post-release quality metrics measure attributes such as customer satisfaction and customer responsiveness. These can be evaluated for example with the number of defects sent by customers, change requests from customers, and customer's willingness to recommend the product to other potential customers. For the final category of metrics for changes in processes or tools, sprint readiness and story flow metrics were found to change company policies to having target values for metrics.

In [11] a more general approach in categorization of agile metrics is used. The authors define the core agile metrics to include product, resource, process, and project metrics. Of these, the product metrics deal with size, architecture,

structure, quality, and complexity metrics. Resource metrics are concerned with personnel, software, hardware, and performance metrics. Process metrics deal with maturity, management, and life cycle metrics, and project metrics with earned business value, cost, time, quality, risk, and so on. Each of these sub-metrics can define a range of additional metrics such as velocity, running tested feature, story points, scope creep, function points, earned business value, return on investment, effort estimates, and downtime. The researchers also point out that teams should invent metrics as they need such, and not use a metric simply because it is commonly used – this might result in data that has no value in the development.

Kunz et al. [23] claim that especially source-code based product metrics increase quality and productivity in agile software development. As examples, the researchers present Number of Name-Parts of a method (NNP), Number of Characters (NC), Number of Comment-Lines (CL), Number of Local Variables (NLV), Number of Created Objects (NCO), and Number of Referring Objects (NRO). All in all, the researchers emphasize the early observation of quality to keep the software stable through the development process.

In their 2009 book [9] the Poppendiecks emphasize the customer-centricity in metrics. They present examples of these including time-to-market for product development, end-to-end response time for customer requests, success of a product in the marketplace, business benefits attributable to a new system, customer time-to-market, and impact of escaped defects.

2.3 Lean Metrics

As lean methods have been developed originally for manufacturing, there are obviously collections of corresponding metrics. For instance, the following has been proposed [3]: Day-by-the-Hour (DbtH) measures the quantity produced over the hours worked. This should correspond to the same rate of customer need. Capacity utilization (CU) is the amount of work in progress (WIP) over the capacity (C) of the process. An ideal rate is 1. On-time delivery (OTD) is presented as the number of late deliveries over the number of deliveries ordered. Moreover, such metrics or signals that help the involved people to see the whole, are mentioned in [24].

Petersen and Wohlin [3] present cost efficiency (CE), value efficiency (VE), and descriptive statistics as measurements for analyzing the flow in software development. A possible way of measuring CE is dividing lines of code (LOC) by person hours (PH). However, they point out how this cost perspective is insufficient as value is assumed to be created always by investment. The increase in LOC is not always value-added as knowledge workers are not machines. On the other hand, VE = (V(output) - V(input)) / time window. V(output) represents the final product, and V(input) the investment to be made. This type of measuring takes value creation explicitly into account, and therefore it can be a more suitable option.

Overall, according to van Hilst and Fernandez [25] there are two different approaches to evaluating efficiency of a process considering Lean ideals. These

views apply models from queuing theory, in which steps of a process are seen as a series of queues. Work advances from queue to queue as it flows through the process, and process performance is then analyzed in terms of starvation(empty queues) and bottlenecks(queues with backlogs). The first approach is to look at a workstation and examine the flow of work building up or passing through. At the same time, the activities on the workstation are studied to see how they either add value or impede the flow. On the contrary, the second approach follows a unit of work as it passes through the whole process. In that case, the velocity of this unit is studied. Considering these two approaches, van Hilst and Fernandez [25] describe two metrics: Time-in-process and work-in-process. Work-in-process is corresponding with the first approach as it describes the amount of work present in an intermediate state at a given point in time. The second approach is measured with time-in-process describing the time needed for a unit of work to pass through the process. For an optimal flow, both of these need to be minimized.

Finally, Modig [24] takes an even deeper look into measuring flow efficiency. This metric focuses on the unit, which is produced by an organization, (*flow unit*) and its flow through different workstations. Flow efficiency describes how much a flow unit is processed in a specific time frame. Higher flow efficiency is often better from the flow units point of view. For instance, if a resource processes the flow unit for one hour, and then the flow unit is placed to a queue of two hours, and then another resource starts to process it for three hours, the flow efficiency is 4 / 6 = 66%. If the length of the queue is shortened to for example half an hour, the flow efficiency is higher (4 / 4.5 = 89%).

3 Case Lupapiste

An industrial single case study was conducted to investigate measuring a state-of-the-art pipeline within a two months time frame of actual development work. The case project, and its deployment pipeline are introduced in the following.

3.1 Description of the Case Project

The application "Lupapiste", freely translated "Permission Desk", is a place for the citizens and companies in Finland to apply for permissions related to the built environment, available at `https://www.lupapiste.fi`. The project was started in 2012, and the supplier of the system is Solita Plc., a mid-sized Finnish ICT company. The end users of the system consist of various stakeholders, with various interests. The Environmental Ministry of Finland owns the project code and acts as a customer in some new functionalities needed to the system.

At the time of research (Fall 2015), the project team consisted of seven developers, a user experience (UX) designer and a project manager that are co-located in a single workspace at the supplier. On the management level there are four more persons in different roles. The team is cross-functional and has also DevOps [26] capabilities. Some team members have an ownership of certain parts of the

system, but the knowledge is actively transferred inside the team by changing the areas continuously and for example by applying agile practices like pair reviewing of code to spread out the knowledge in a continuous manner. The team takes use of a custom Lean Software Development process that includes features from agile Scrum-based processes with lean heritage. The process is ongoing and has no sprints, but milestone deadlines for certain functionalities are set by the product owner team, which consists of project management personnel of the supplier and the formal customer of the project. Furthermore, agile practices, like daily meetings, have been combined with lean practices and tools, like a Kanban board.

3.2 Deployment Pipeline of the Project

The pipeline of the case project has several environments (see. Figure 2) – a personal local development environment (Local), the shared development environment (Dev), a testing environment (Test), a quality assurance environment (QA) and the production environment (Production). Each of these environments serve different needs, and deployments to the different environments are managed through the version control system. Therefore, it automatically provides accurate data and meta data to measure the pipeline, which we have already proposed in an earlier paper [27]. The actual timestamps of deployments are stored in the meta data of the version control system branches.

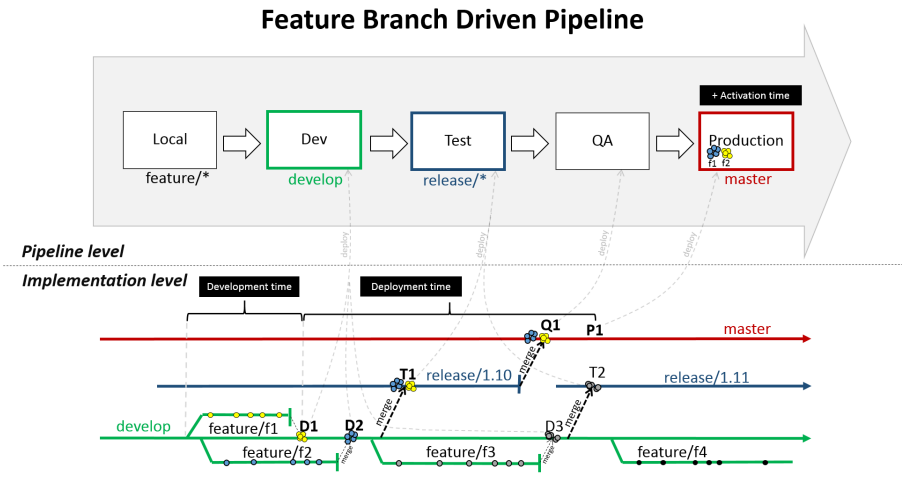

Fig. 2. Deployments to the pipeline environments are triggered by push-events to the distributed VCS. Features f1 and f2 have been merged and pushed to the develop-branch (triggering deployments D1 and D2 to the Dev-environment), then to the Test-environment (deployment T1 to Test-environment) and production environment (P1).

The team uses a VCS-driven solution to manage the deployments to the environments of the pipeline. The team applies the Driessen branching model [28], which utilizes feature branches. Figure 2 presents the connection between the branches and the deployments to the various environments of the pipeline. When the development of a new feature starts, a developer opens a new feature branch and starts developing the feature in the Local environment by committing changes to the new branch. The developer may push the local changes to the version control system from time to time, but no CI jobs are executed in this phase. When the development of the feature is ready, the feature branch is closed and the changes are merged to the develop-branch. When the changes are pushed to the version control system, a stream of CI jobs for deploying a new version to the Dev-environment is triggered automatically (deployments D1, D2 and D3 in Figure 2). The CI jobs build the software, migrate the database, and deploy and test the software.

The deployment to the Test environment is accomplished by merging the develop-branch to a release-branch. Once again, when the changes to a release branch are pushed to the version control system, a stream of CI jobs for building the software, migrating the database, and deploying and testing the software in the Test environment is triggered (deployments T1 and T2). For instance, deployment t1 in Figure 2 was triggered by a push to branch release/1.10, which contained features f1 and f2. Similarly, the production deployment happens by closing the release branch, which is then merged to the master-branch. The new version to be released can then be deployed to the QA (deployment Q1) and production environments (deployment P1) with a single click from the CI server.

In Figure 2, feature f1 flows from the development to production in deployments D1, T2 and P1. Feature f2 flows in deployments D2, T1 and P1. Feature f3 has flown to the test-environment in deployments D3 and T2. In order to deploy feature f3 to the production environment, release branch release/1.11 should be closed and merged to the master branch, which then would be manually released with a single click from the CI system.

Figure 3 presents the correspondence of the branches in the version control system and the CI jobs on the radiator screen in the team workspace. If a CI job fails, the team is immediately knowledgeable of the problems. Moreover, the current status of the functional end-to-end tests running in the Dev-environment is visible to the team.

In case of urgent problems in the production environment, the branching model also allows creation of a hotfix branch. Figure 3 represents a situation where urgent problems occurred after a deployment to the production environment. The automated tests had passed, but the login button was invisible on the front page because of layout problems. In this special case, a hotfix branch was then opened, the layout problems were fixed, the branch was merged to the master branch, and when the changes were pushed and a CI job was triggered manually, the problem was fixed and the users could continue logging in to the system.

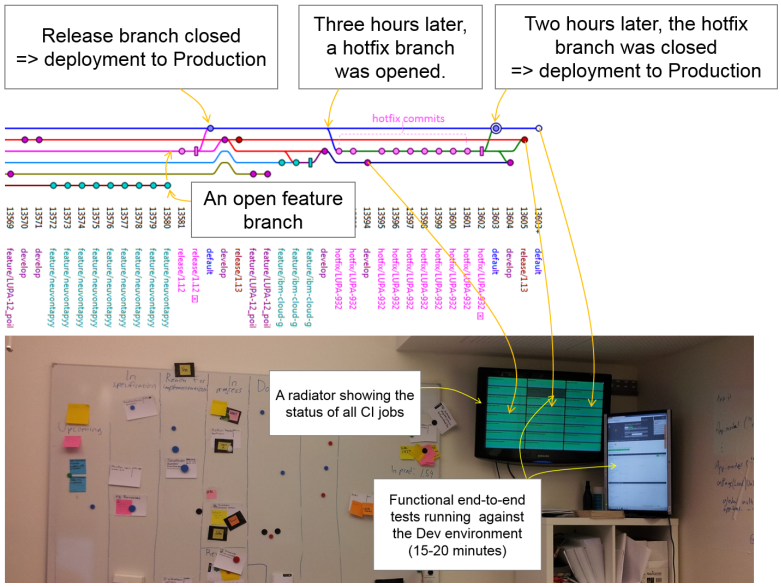

Fig. 3. An actual usage sample of the branching model and its correspondence to the CI jobs.

4 Defining Metrics for Pipeline

In this section, we define several new metrics which describe the properties of the deployment pipeline. The goal of the metrics is to provide valuable information for the team for improving the performance of the pipeline. With them, it is possible to detect bottlenecks, indicate and consequently eliminate, waste, and find process improvements.

We divided the metrics into two categories. First, *Metrics on the Implementation Level* dependent of the toolset and practices used to implement the pipeline. Second, *Metrics on the Pipeline Level* are metrics that are independent of the actual implementation of the pipeline. The metrics in the two categories are discussed in more detail in the following.

4.1 Metrics on the Implementation Level

The availability of data to calculate flow and throughput depends on the implementation of the pipeline and the actual tools and practices used. In essence, development, deployment and activation time must be available for each feature, discussed in more detail in the following.

– *Development time*, or the time it takes for the team to implement a new feature. The development time of a single feature can be measured in our case project, as each new feature is a new branch in the version management

system. The starting time for the new feature is simply the time when the branch is created, and completion time is when the branch is merged with the master branch. See Figure 2 for an example of development time of feature/f1. It is the time from opening the feature branch until D1. In an earlier paper [27] we measured the value of this metric during a three month period. The value was typically one or two days, but for larger features, it was even 12 working days.

- *Deployment time*, or the time it takes to deploy a new feature to production use when its implementation has been completed. There are two dimensions to this metric. One is the execution time of the tools needed for the actual deployment (e.g. seconds or minutes), and the other is the decision process to deploy the features, if additional product management activities, for example acceptance testing, are associated with the deployment (e.g. hours or days). See Figure 2 for an example of deployment time of feature/f1 – the time from D1 to P1. In [27], we measured a mean value of nine working days during a three month period.
- *Activation time*, or the time it takes before the first user activates a new feature after its deployment. Activation time can only be measured for features that are specific enough to be visible in production logs. At times, however, this can be somewhat complicated. For instance when a new layout is introduced, the first time the system is activated can be considered as the first use of the feature. See Figure 2 for an example of activation time of features found in the production log. It is the time from P1 to the first use caught from the production logs. The mean activation time in [27] was three working days while the median was one working day.

Another viewpoint to the time a feature spends in the deployment pipeline, is to count the age of the features that have been done, but are still waiting for production environment deployment. The following metric is based on measuring the current time spent on the pipeline:

- *Oldest done feature (ODF)*, or the time that a single feature has been in development done state, but is still waiting for deployment to the production environment in some of the environments of the deployment pipeline. The metric is dependent on Definition of Done (DoD) [29]. In our case project, this data is available from the meta data of the feature branches: a feature branch closed, but not merged to a closed release branch. At the time of research (Autumn 2015), the value of ODF in the case project is currently six days and the weekly release schedule has kept the value in less than one week constantly.

4.2 Metrics on the Pipeline Level

In the context of continuous delivery and deployment, the throughput of the pipeline used to deliver features to the end user is an important metric. Out of

the existing metrics, flow efficiency, proposed in [24], best captures the spirit of the pipeline. We propose the following new metrics to this category.

- *Features Per Month (FPM)*, or the number of new features that have flown through the pipeline during a month. The metric is based on Day-by-the-Hour (DbtH), which measures quantity produced over hours worked [3]. In the case project, the data for this metric can be collected from the implementation level data (number of feature branches closed and merged to a release branch that has been closed). Apparently, this metric can be measured in many other implementation settings, for example in a project that does not use feature branches. For example, issue management system data or version commit messages following a certain convention, are possible sources for this data. At the time of research, the value of this metric for the case project is 27 FPM during the last three months, which is more than one feature per working day.
- *Releases Per Month (RPM)*, or the number of releases during one month. Long term change of this metric provides information on changes in the release cycle. In the case project, this data is available both in the version control system and the CI server logs. At the time of research, the value of this metric for the case project is 7 RPM, which is one or two releases per week.
- *Fastest Possible Feature Lead Time*, or the actual time the feature spends in the build and test phase on the pipeline. In our case project, there is latency which origins from the use of feature branches and separate build processes for each branch. The code is compiled and packaged multiple times during the different phases of the pipeline. A build promotion approach in e.g. [30], where the code is build only once and the same binary is deployed to all environments, the lead time may be shorter. At the time of research, the value for this metric is two hours (quick integration and unit tests running some minutes in the commit stage and functional end-to-end browser tests running one hour in the pipeline environments). As a shortcut for urgent issues, the team can also use a hotfix branch, which allows making a quick fix in minutes.

5 Results

Working in close cooperation with industry to answer to our research questions has given us important insights over industry tools, processes and needs. Next, we will revisit our original research questions, and give a short discussion regarding our observations.

5.1 Research Questions Revisited

Considering the metrics defined above in the light of data available in version control system has given us high confidence that these metrics can be gathered

in a straightforward fashion, with certain exceptions. However, tools are needed to automate data collection process, and to help visualizing the results [31]. The exact answers to research questions are the following.

> **RQ1:** Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?

Data can be collected regarding development, deployment and activation time from the tool chain that is used by the project. For the two former, data is precise, but requires following certain conventions, such as creating feature branches in the version data base for new features – a feature which is not supported by all version control systems. Regarding feature activation, the situation is less clear, since for numerous features it is not obvious when they are truly activated. When referring to a new function in the system, such as a widget in the screen for instance, the activation produces identifiable traces, whereas a change in layout or in libraries used are harder to pinpoint. Therefore, to summarize, with version control and usage monitoring system data, it is also possible to address the numbers of features in development, deployment, and activation, although for the latter with only some limitations and interpretations.

Regarding practicality, we feel that any team performing continuous delivery and deployment should place focus on metrics listed above. Based on discussions with the team developing Lupapiste, visualizing the data regarding features on the pipeline was found very useful, and exposing developers to it actually led to faster deployment and to less uncompleted work in the pipeline.

> **RQ2:** How should the pipeline or associated process be modified to support the metrics that escape the data that is available?

While actions related to actual development are automatically stored in the version control system, end users' actions are not. Therefore, better support for feature activation is needed. This can not be solved with tools only but require project-specific practices. For instance, additional code could be inserted to record activation of newly written code, or aspect-oriented techniques could be used to trace the execution of new functions as proposed in [32].

In the present setup, there is no link to product management activities. In other words, the pipeline only supports developers, not product management. More work and an improved tool chain is therefore needed, which is also related to the above discussion regarding feature activation. However, it can be questioned if this falls within the scope of the pipeline, or should be considered separately as a part of product management activities.

> **RQ3:** What kind of new metrics based on automatically generated data could produce valuable information to the development team

We presented data collection methods for collecting data for new metrics regarding the deployment pipeline. We proposed multiple new metrics for the deployment pipeline. For example, metric *Oldest Done Feature (ODF)*, which the

team of the case project found especially potentially useful, could be applied for measuring the current state of the deployment pipeline. Exposing such a metric to the team for example on the radiator screen in the team workspace, could improve the release cycle of the project.

We measured the values for the new metrics proposed for the case project. The team was producing more than one feature a day and making releases at least once a week. The *Oldest Done Feature (ODF)* at the time of research was only six days old. According to these metrics, the features are flowing fluently from development to the production environment.

5.2 Observations

To begin with, the deployment to production may have extra latency even in a state-of-the-art deployment pipeline. For instance, in the case project, many of the features suffered from a long latency of even weeks or months between the time the feature was done till the time when it was deployed to the production. The team was shortly interviewed about the obstacles why the features were not deployed to the production environment earlier. The obstacles were often related to features that had been merged to the develop-branch, which then accompanied the develop branch to a state where it was not possible to deploy anymore. For instance in one case, a key functionality was broken and the fix needed data from a design process.

The time after a new feature that has been deployed to the production environment and is waiting for users to use the feature, can be regarded as waste. To eliminate such, the users of the system must be informed regarding newly deployed features, and they also have to have the skills to use them. Because the users in the case project are the municipal authority users nationwide, an announcement sent by email as new features are introduced. Moreover, a wizard, which would tell about the new features, for example after login or in the context of the features, could help the users to find the new functionality.

We discussed about the proposed new metrics with the development team of the case project. *Oldest Done Feature* was found as the most useful metric that could possibly help the team to improve the flow of the pipeline. The team even considered that this kind of metric could be shown on the radiator screen – if the oldest feature is for example two weeks old, the radiator could indicate the problem in the pipeline. However, the actual usage of such a metric is not straightforward. There are times, when the develop branch is not deployable because of, for example, a major refactoring. In this kind of circumstances this kind of metric may disturb the team.

6 Conclusions

A metric should be used for a purpose. A modern deployment pipeline paves a highway for the features to flow from the development work to actual usage in the production environment. The tools on the pipeline produce a lot of data

regarding the development and deployment activities of the new features. We analyzed the tools and practices of an industrial single case study in order to identify which data are automatically created by the several tools of the deployment pipeline. The results show that data for many new useful metrics is automatically generated.

Based on this data, we defined several new metrics for describing the properties of the deployment pipeline. For instance, the metrics proposed can be applied to analyze the performance and the present status of the pipeline. The goal of metrics is to provide valuable information to the team to improve processes and the pipeline. Applying the metrics in a continuous delivery project setting can help to achieve this.

Acknowledgements

This work is a part of the Digile Need for Speed project (`http://www.n4s.fi/en/`), which is partly funded by the Finnish Funding Agency for Innovation Tekes (`http://www.tekes.fi/en/tekes/`). Persons in Figure 1 are designed by Paulo S Ferreira from `thenounproject.com`.

References

1. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.
2. M. Poppendieck and T. Poppendieck, *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.
3. K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software: Practice and experience*, vol. 41, no. 9, pp. 975–996, 2011.
4. M. Fowler, "Agileversuslean," http://martinfowler.com/bliki/AgileVersusLean.html, 2008, retrieved: November 2014.
5. R. Shah and P. T. Ward, "Defining and developing measures of lean production," *Journal of operations management*, vol. 25, no. 4, pp. 785–805, 2007.
6. R. K. Yin, *Case study research: Design and methods*. Sage publications, 2014.
7. K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The agile manifesto," http://agilemanifesto.org, 2001, retrieved: November 2014.
8. E. Kupiainen, M. V. Mäntylä, and J. Itkonen, "Why are industrial agile teams using metrics and how do they use them?" in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 23–29.
9. M. Poppendieck and T. Poppendieck, *Leading lean software development: Results are not the point*. Pearson Education, 2009.
10. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
11. S. Misra and M. Omorodion, "Survey on agile metrics and their inter-relationship with other traditional development metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 6, pp. 1–3, 2011.

12. S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *Agile Conference (AGILE)*, Aug 2013, pp. 121–128.
13. M. Fowler, "Continuous delivery," http://martinfowler.com/bliki/ContinuousDelivery.html, retrieved: November 2014.
14. D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.
15. M. Fowler, "Continuous integration," http://martinfowler.com/bliki/ContinuousDelivery.html, retrieved: November 2014.
16. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
17. A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE '08. Conference*, Aug 2008, pp. 289–293.
18. J. Humble, "Continuous delivery vs continuous deployment," http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, retrieved: November 2014.
19. J. Humble, C. Read, and D. North, "The deployment production line," in *Agile Conference*. IEEE, 2006, pp. 6–pp.
20. D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, p. 1, 2013.
21. J. Humble, "Continuous delivery vs continuous deployment," http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, 2010, retrieved: November 2014.
22. B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.
23. M. Kunz, R. R. Dumke, and N. Zenker, "Software metrics for agile software development," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 673–678.
24. N. Modig and P. Åhlström, *This is lean: Resolving the efficiency paradox*. Rheologica, 2012.
25. M. Van Hilst and E. B. Fernandez, "A pattern system of underlying theories for process improvement," in *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM, 2010, p. 8.
26. P. Debois, "Devops: A software revolution in the making," *Cutter IT Journal*, vol. 24, no. 8, 2011.
27. T. Lehtonen, T. Kilamo, S. Suonsyrjä, and T. Mikkonen, "Lean, rapid, and wasteless: Minimizing lead time from development done to production use," in *Submitted to publication*.
28. V. Driessen, "A succesful git brancing model." http://nvie.com/posts/a-successful-git-branching-model/, retrieved: November 2014.
29. K. Schwaber and M. Beedle, "Agile software development with scrum. 2001," *Upper Saddle River, NJ*, 2003.
30. L. Chen, "Continuous delivery: Huge benefits, but challenges too," *Software, IEEE*, vol. 32, no. 2, pp. 50–54, 2015.
31. A.-L. Mattila, T. Lehtonen, K. Systä, H. Terho, and T. Mikkonen, "Mashing up software management, development, and usage data," in *ICSE Workshop on Rapid and COntinuous Software Engineering*, 2015.
32. S. Suonsyrjä and T. Mikkonen, "Designing an unobtrusive analytics framework for java applications," in *Accepted to IWSM Mensura 2015, to appear*.

Publication II

P. Tyrväinen, M. Saarikallio, T. Aho, T. Lehtonen and R. Paukkeri. Metrics Framework for Cycle-Time Reduction in Software Value Creation. In *The Tenth International Conference on Software Engineering Advances (ICSEA)*, 2015.

Metrics Framework for Cycle-Time Reduction in Software Value Creation

Adapting Lean Startup for Established SaaS Feature Developers

Pasi Tyrväinen, Matti Saarikallio
Agora Center, Department of CS and IS
University of Jyväskylä, Finland
pasi.tyrvainen@jyu.fi, matti.saarikallio@gmail.com

Timo Aho, Timo Lehtonen, Rauno Paukkeri
Solita plc
Tampere, Finland
{timo.aho, timo.lehtonen, rauno.paukkeri}@solita.fi

Abstract— **Agile software development methodologies driving cycle-time reduction have been shown to improve efficiency, enable shorter lead times and place a stronger focus on customer needs. They are also moving the process development focus from cost-reduction towards value creation. Optimizing software development based on lean and agile principles requires tools and metrics to optimize against. We need a new set of metrics that measure the process up to the point of customer use and feedback. With these we can drive cycle time reduction and improve value focus. Recently the lean startup methodology has been promoting a similar approach within the startup context. In this paper, we develop and validate a cycle-time-based metric framework in the context of the software feature development process and provide the basis for fast feedback from customers. We report results on applying three metrics from the framework to improve the cycle-time of the development of features for a SaaS service.**

Keywords-metrics framework; cycle-time; agile; software engineering process; lean startup; feedback; SaaS.

I. INTRODUCTION

The software engineering (SWE) process has traditionally been managed on a cost basis by measuring programmer effort spent per lines of code, function point or requirement. These metrics have also been used to guide software process improvement. In order to align more with business strategy and value production the focus has shifted more towards value creation instead of cost reduction. For example, value-based SWE [1], software value-map [2] and a special issue on return on investment (ROI) in IEEE Software [3] have explored value in software development. As a reaction to move away from a cost-reduction focus, the recent goal of lean thinking has been to optimize for perceived customer value [4]. Thus, we can say that leadership approach for the software development process is moving from a cost focus to a value focus.

Measuring the value of application software and cloud services is difficult to do before it is in use, as you need to consider the value of the software for the potential users, the business value for the firm developing it and the value for other stakeholders [1][5][6]. The current theories of value do not present a simple way of assessing customer value [7]. Although companies put a great amount of effort into increasing customers' perceived value in the product development process, determining how and when value is added is still a challenge even in marketing and management sciences. [7] Further, the software engineering metrics are measuring attributes of the software development process (e.g., cost, effort, quality) while these metrics remain disconnected from the attributes and metrics developed for measuring value (see Table I). Various approaches have been developed to overcome this gap [1][5][6][8][9][10][11][12][13][14][15][16] without any major break-through.

The software engineering community has adopted an iterative approach to software development in form of Scrum [17], XP [18] and other agile [19] methods. These promote fast cycle user interaction and development process to keep the effort focused on customer needs based on fast customer feedback either interactively or through analysis of service use behavior. The startup community has adopted a similar approach and commonly uses the lean startup cycle [20] to evaluate the hypothesis of customer needs using the build-measure-learn cycle, which is repeated to improve customer acceptability of the offering and the business value of the startup. The common theme of these approaches is that instead of trying to estimate or predict the value in advance, try to shorten the cycle time from development to actual customer feedback, which indicates the value of the software in use. That is, from the SWE perspective, the speed of feedback received from users is the best indicator of the value of the newly created software. This indicates that shortening the feedback cycle would drive the SWE process towards faster reaction on customer value and higher value creation.

Although there exists a common understanding about the key role of a fast customer feedback cycle in linking the SWE process to value creation, the measurement methods and metrics available in literature are positioned either as cost-based SWE methods or as value-oriented metrics with little connection to the engineering process providing little guidance for managing and developing the SWE process (see Table I). Thus, the research question of this paper is, what metrics would guide cycle-time-driven software engineering process development in established organizations?

As the answer is context-dependent, a set of metrics will be needed. This paper aims at filling this gap by proposing a metrics framework enabling adoption of such metrics in a variety of contexts where new features are incrementally added to software.

 ISBN: 978-1-61208-438-1

TABLE I. POSITION OF THIS RESEARCH TO BRIDGE COST-ORIENTED SOFTWARE ENGINEERING (SWE) METRICS AND VALUE-ORIENTED BUSINESS METRICS

	Measurement Domains		
	SWE Metrics	***Research Gap Addressed Here***	***Value Metrics***
Scope (measurement target)	SWE Process	Value Creation Cycle	Customer Value of Offering, Value of Startup
Measured Attribute	Cost, Effort, Quality	Cycle Time	Value for Customer, Value for Enterprise
Examples	Function Points per month, Faults per lines of code		Value in Use, ROI, Lean Analytics

Applying the guidelines of the design science method [21], this research has been initiated based on company needs presented in interviews of Software as a Service (SaaS) development firms in a large industry-driven research program [22], to target an issues with business relevance in firms.

In Section II, we construct the metrics framework artifact based on the analysis and synthesis of previous research literature selected from the perspective of the research question. Following the design science research guidelines, we also demonstrate generalizability of the framework artifact to several contexts by choosing from a variety of metrics to target the specific process development needs. We also propose a simple diagrammatic representation for visualizing some of the metrics values in operational use to pinpoint development tracks requiring attention in an organization with multiple parallel feature-development teams.

In Section III, we evaluate the metrics framework by applying it to the case of a firm developing new features for an existing SaaS service and discuss the impact of the findings on revising the target of the next process improvement actions. In Section IV, we summarize the results, draw the conclusions and propose directions for further research.

II. THE CYCLE-TIME METRICS FRAMEWORK

A. Developing the Framework

The flow of new features through a SWE process can be measured at various points in time with an aim to reduce delay between points to reduce cycle time. The scope of the process measured will impact the attention of the software developing organization. In the narrowest scope, the cycle time measured includes the basic software development cycle while the widest cycle takes into account the customer needs and experience and, thus, matches and even expands the lean startup cycle [20].

In the proposed framework (see right side of Figure 1), the feature life-cycle begins with three planning phase events: 1) a need emerges, 2) a software development organization recognizes the need, and 3) the decision is made to develop the feature. In large established organizations, the identification of feature needs has been excluded from the responsibility of the SWE organization to responsibility of the product marketing organization, while the entrepreneurship-oriented startup community has emphasized the value of including the need identification step as an inherent part in the fast business development cycle of the organization developing the software. Sometimes there is an intentional lag between events 2 and 3 as the decision may be to wait for the right time window (cf. real options [23][24]), or features with higher priority are consuming all resources available.

Continuing from the 3 events that form the beginning of the feature life cycle (above) and for the purposes of measuring the value creation cycle, the main development events included in this framework are 4) development starts, 5) development done, and 6) feature deployed. Use of XP, Scrum and other iterative and incremental development (IID) processes has aimed at reducing the time between events 4 and 5 (or fixing that to 2–4-week cycles). The cycle-time from 4 to 5 is here referred to as the Development cycle (see Figure 1). Moving from packaged software to cloud delivery and SaaS development along with moving from an annual or a six-month software release cycle to continuous integration (CI [25]) and continuous delivery (CD [26]) in development operations (devops [27]) has reduced the interval between 4 and 6.

After the event 6, the traditional software engineering process is often thought to be completed, while many entrepreneurship-oriented approaches, such as Lean Startup [20], go further, starting from building a product to measuring the use of it, which produces data used for learning and for producing ideas for the next development cycle (see left side of Figure 1). That is, building the product based on current ideas is only one of the three main events needed for value creation: build–measure–learn [20]. For considering the business and customer perspectives in this metrics framework for the value creation cycle, we need to expand beyond step 6 to include the use, measuring and learning phases: 7) when the feature gets used, 8) when feedback data is collected to support learning, and 9) when a decision is made based on the feedback. Note that events 8 and 9 resemble events 2 and 3 while not all information from customer needs is collected through measuring the use of the current product. It is also commonly assumed that the time from feature deployed (6) to first use (7) is short, while without measured data this can be an incorrect assumption. There have been cases where almost half of software features were never used [28]. Further, if software quality is high, it can take some time to get feedback, and it may require many uses of the feature before customer sends feedback about problems. Additionally, it can take time for a feature to get sufficient number of uses to allow for a reliable analysis of customer behavior (8). Also, the deployment process of the company can delay the decision to act on the feedback (9).

 ISBN: 978-1-61208-438-1

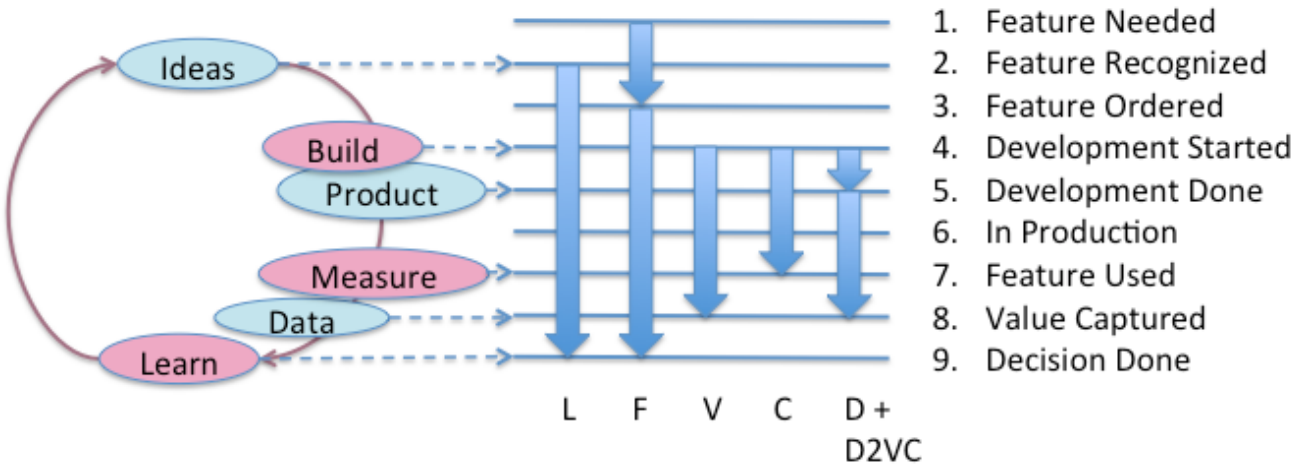

Figure 1. The value-driven metrics framework for driving software engineering cycle-time reduction (on the right), the Lean Startup cycle (on the left) and example cycles, for which cycle time can be used as the metrics driving cycle-time reduction (in the middle).

Figure 1 depicts the proposed framework. On the right side we have the sequence of events identified. On the left side, we have the Lean Startup cycle with horizontal arrows pointing from the phases to related events of the framework. The vertical arrows in the middle represent examples of cycle times that can be used as a target metric for developing SWE process. The cycles in the center are labeled as follows: L = Lean Startup cycle, F = Full cycle including fuzzy front end and full feature development cycle, V = Value cycle from starting the development to value capture, C = Core cycle from development start to first feature use, and finally D+D2VC, where D = Development cycle from start of development to production readiness and D2VC = time from development done to value captured. We emphasize that this list of cycles is not exclusive and new cycle time metrics can be created with this framework on demand for each context.

B. *Changing Process Development Focus through Metrics*

The various cycle-time metrics available in the framework can be used for focusing process development activity to specific process areas based on the need (see Table II). For example, if the basic software development process has been well developed and if some incremental development process, automated testing and continuous integration are applied, it may be useful to shift the attention to continuous deployment. In that case, the metric to be followed can be changed from Development cycle to cycle time between events 4 and 6, from start of development to start of production (see the second line in Table II). Changing the metric will also change the focus of attention and can often result in adjusting the processes, resource allocations or tools used.

TABLE II. EXAMPLE PROCESS DEVELOPMENT TARGETS WHEN USING ALTERNATIVE CYCLE-TIME METRICS

Cycle	Start Event	End Event	Addressed Capabilities	Process Development Focus
D, Development	4: Development Started	5: Development Done	XP, Scrum and other IID processes, automated testing and continuous integration (CI)	Using this cycle-time metrics addresses cycle-time of the basic SW development process
Time to production	4: Development Started	6: In Production	Same as in D, adding continuous deployment (CD) to the measurement scope	Using metrics for this cycle time focuses attention to CD capability
C, Core cycle	4: Development Started	7: Feature Used	Same as previous adding communication (diffusion) to customer base to the scope	Here the focus shifts to integrating customer facing team with development
V, Value cycle	4: Development Started	8: Value Captured	Adding customer analytics and customer feed-back capabilities to the previous scope	Shifts focus to integrating analytics capability to IID+CI+CD capability
Time to Value	4: Development Started	*: Break Even	As Value cycle, but using this metrics assumes that value produced can be evaluated.	As in Value cycle
D2VC	5: Development Done	8: Value Captured	Post-development processes needed to deliver the created value and to get the feedback	Focusing on value cycle capabilities after the basic SW development process.
Fuzzy Front End	1: Feature Needed	3: Feature Ordered	Deep customer understanding (between events 1 and 2) and market understanding (2 to 3)	Measuring capability to find customer needs close to actionable market
...	...	...	...	...

 ISBN: 978-1-61208-438-1

In large organizations, where the product-marketing department is responsible for collecting market requirements and for product launches, the processes crossing product development and product-marketing departments may be problematic. In these cases, choosing the Value cycle, Time to Value or Design done to value captured (D2VC) as a common metric for both of the departments will enforce collaboration between the departments and will likely improve the total value creation capability of the organization, while local metrics within the departments are likely to lead to local optimization leading to non-optimal organizational behavior. It should be noted, that this issue appears mainly in large established organizations rather than in small startup firms, the needs of which the lean startup approach has been developed.

The time to value cycle in Table II ends with the event of reaching the breakeven point, which is marked with an asterisk "*" rather than a number representing a specific ordering in the framework. In some cases a pay-per-use business model provides a basis to determine the break-even point for a feature, while in some cases the break-even point is estimated by qualitative means. A new feature may produce enough value to reach the break-even point when it is published (event 6) or when it is used for the first time (event 7). However, in many contexts this event occurs close to event 8, Value captured, that is, the feature use count is high enough, and sufficient feedback has been received, to ascertain whether the feature was worth the development effort. Based on these examples and the other examples in Table II we can observe, that the choice of applicable metrics is context dependent. Thus there is a need for a framework for metrics, which supports choosing the metric applicable for a specific situation.

C. Depicting Cycle-Time Elements

Depicting the proposed cycle-time metrics makes it easier to decide whether to further develop or even to drop a existing feature and will also help in communicating the cycle-time reduction agenda to software engineers and other parties involved. For this purpose we devised a simple diagrammatic representation presented in Figure 2. In this example, the development starts at point 4 and ends at point 5. The y-coordinate represents the cumulative development time, in line with the cumulative cost for the organization. This linear curve is intentionally simplistic as the focus is on the form of the curve after event 5. In contrast, software engineering oriented representations, such as the Kanban Cumulative Flow Diagram [29], focus on analysis of the development cycle from 4 to 5 and ignore activity after production readiness.

From event 5 on, the horizontal line represents the duration of the waiting time from ready-to-deploy through deployment to first use. The feature is used for the first time in production at event 7. After that the dropping logarithmic curve represents the speed at which feedback has been received. After the second use the curve comes down to half, after the third use to one third of the original, and so on.

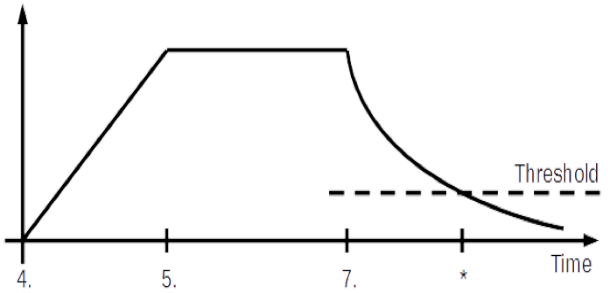

Figure 2. Depiction of the cycle times for feature analysis and process development. The numbers refer to the event number in the framework.

That is, the curve represents development time divided by number of times used. A context-specific target threshold for development time per times used is presented as a dotted line and the time when the curve reaches the threshold is marked with an asterisk "*".

In line with our approach to focus on the cycle times, this graphical representation aims at depicting the cumulative effort invested to the feature during development. There is a risk embedded in this development effort as it has not received feedback from the customers. Thus it is potential waste if customers do not accept the feature. This risk is mitigated along the narrowing gap of the asymptotic curve and the horizontal axis and reaching the threshold indicates that enough customer feedback has been received to ascertain whether it has been worth developing the feature. Event 8, Value captured, is serving this purpose as the event when sufficient user feedback is gained to evaluate the value of the newly developed feature and for adjusting the development plans accordingly, to further develop the feature or to drop it. In addition to guiding value creation, fast feedback from event 8 makes it easier for software developers to fix errors and modify the feature as long as they can still recall the implementation of the feature and have not moved on to new assignments.

Although measuring value is difficult, we would also like to identify the time-to-value cycle, that is the time from starting the development to break even, to the point at which its value to the customer exceeds the development costs. Now we face the challenge that while the cost can be easily measured in terms of time or money, value as a concept is not clearly defined and even if it were it would be hard to measure. We can speculate that the break-even point could be reached already on deployment (for features whose existence provides value even if they are not used, e.g., emergency-situation feature), on first use (when customer finds it), after a certain amount of uses (some use value derived from each), or sometimes a feature can fail to become profitable. Thus, the location of this measurement point cannot, in general, be identified in the sequence of events in the proposed framework, rather it is context dependent.

If we want to measure value, we need to define value. Historically three forms of economic value are the use value, exchange value and price [30]. There are many theoretical divisions of value to support decision making about which

 ISBN: 978-1-61208-438-1

software feature to work on next [2][5][7][8][9][10][11][12] [13][14][15][16][24][30][31], but most theories consider the use value to the customer as essential. For the purposes of metrics development the focus will be on customer use value. It is important to note that due to market mechanism, exchange value is less or equal to use value [30]. This means that we could calculate a monetary estimate for the upper bound of the value captured by the software developer, that can be compared with cost. Still, the issue is problematic.

If we assume that there is use value for a feature, and in some cases the use value can be estimated as equal for each use, we would like to measure directly the cost versus benefits ratio: $\frac{development\ costs}{benefits}$. However, as discussed the benefits are challenging to measure and, at worst, we might need a new metric for each feature. This leads us to suggest that we isolate the hard-to-measure part, benefits, by instead measuring the precisely calculable cost per use $\beta = \frac{development\ costs}{times\ used}$ and only if possible compare it to the estimated value for the user, based on a case-by-base estimation method. Next, we will show, using a case study, that reaching events "*" and 8 produce very similar value for process development and feature decision making and that they can be used interchangeably. Thus, time to receive enough feedback is also a good, practical proxy for value produced.

III. METRICS VALIDATION CASE STUDY

A. Target Organization and Service

We evaluated the metrics framework in a mid-sized Finnish software company, Solita Ltd. The case software development team develops a publicly available SaaS (lupapiste.fi) used by citizen applying for a construction permit related to real estate and other structures. This privately operated intermediary service provides a digital alternative to avoid the time-consuming paper-based process of dealing directly with the public authority. This service is used by employees of the licensing authority in the municipalities (about 100 users), the applicants (citizens and companies, about 100 per month), and architects and other consultants (1-2 per application). The software development process metrics were evaluated with the usage data collected from the process flow of five new features of this SaaS service deployed during the observation period, in mid-2014.

The service has a single page front-end that connects through a RESTful API to its back-end. Each call to the API is recorded on the production log files with a time stamp. We mined and analyzed the log entries together with the development data captured by the version control system. In this case, we chose features that introduced a new service to the API and were thus possible to trace automatically with a simple script that queried the monitoring system automatically. Some manual work was needed to find the features that introduced a new API, but automation of this work is also possible.

B. Results from Applying the Metrics to Sample Features

From the recorded event time stamps we calculated three metrics values for the case features. Development cycle (D) from start (4) to done (5) in working days. Lag to production from done (5) to deployed (6) in calendar days. And finally, D2VC, time from development done (5) to value captured (8). In this context the target company estimated that enough feedback data was collected for learning when the feature was used four times per each day spent on development, which gave the context-specific definition for the value capture event (8). Table III presents the data that is depicted in Figure 3. To enable comparison, all the features are shifted in the time axis to have event 4 (start of development) at day 0. In a daily use, an alternative depiction can show the timeline representing the history of all features to current point of time from which it is easy to identify development peaks and, more specifically, to notice the curves that remain high after the peak which indicates a demand for action. Either a feature has not been deployed and promoted well for the users or there is no user need for the feature.

C. Case Analysis and Discussion

From Table III we can see that for these five features the average of development effort needed to implement and test the features was about eight working days. When the development was done, on an average 12 calendar days was spent on waiting for deployment of the feature to the production environment. We can also observe that the features with lower priority (F1647 Unsubscribe and F1332 Note) have almost double the lag to production compared to the other features.

TABLE III. DESCRIPTION OF THE SAMPLE FEATURES, THEIR PRIORITY, DEVELOPMENT TIME (IN WORKING DAYS), LAG TO DEPLOYMENT (IN CALENDAR DAYS) AND DEVELOPMENT TO VALUE CAPTURED (D2VC; IN CALENDAR DAYS)

Feature id	Priority	Development (days)	Lag to deployment (days)	D2VC (days)	Description of the feature
F1332 Note	2	10	24	24	Authority user can add a textual note that other users cannot see.
F1498 Attachment	4	9	10	N/A	Applicant user can set the target of an uploaded attachment.
F1507 Validate	4	10	1	49	System validates the form prior the user sends the application.
F1537 Sign	4	7	11	15	Authority user can require an applicant to sign a verdict.
F1647 Unsubscribe	3	2	15	28	Authority user can unsubscribe emails.
Average		**8**	**12**	**29**	

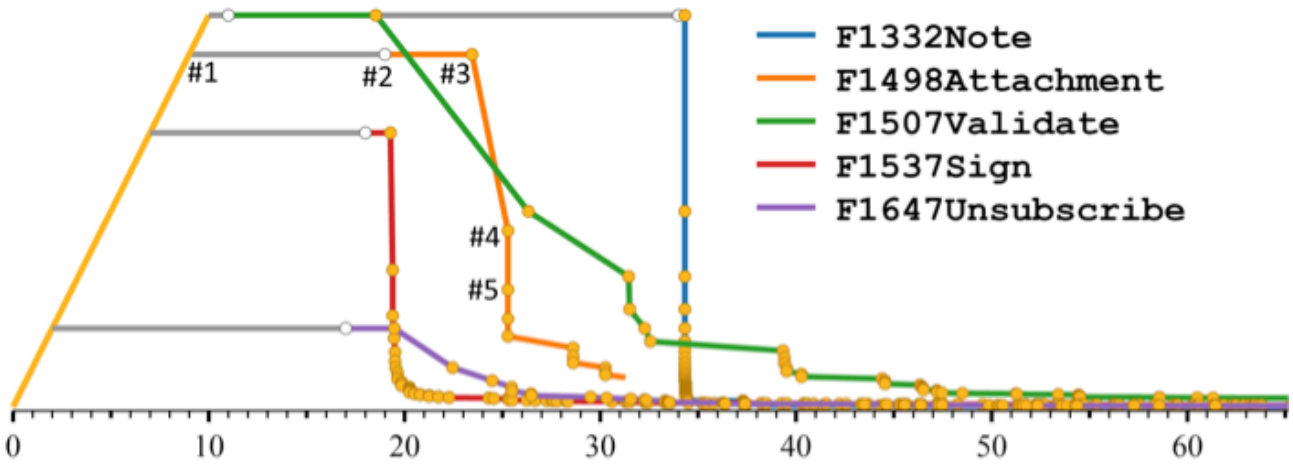

Figure 3. Depiction of the cycle-times of the five features. Development working days share the rising line starting from (event 5) and end in event 6 (start of the gray horizontal line), deployment (7, white dots in the right end of the gray part of the horizontal lines) and usage (yellow dots). To enable comparison, all the features are synced to have event 5 (start of development) at day 0.

The average time from completion of development to value capture is 29 days (this does not include feature F1498 Attachment, which did not reached the number of uses needed for the threshold). From the depiction in Figure 3, we can also see that this feature is no longer used. This feedback triggers the discussion on the reasons for the discontinuation of use of the feature to determine if there is a need to improve it or remove it from the service. When the target is to minimize the cycle times, minimizing the lag from production readiness to deployment (from event 5 to 6) and the means to increase the use of new features are clearly the places where major improvement can be reached much easier than from reducing average development time. By plotting the events in this way, it is easy to identify the places where changes can be made as well as to communicate the need with the development teams.

The results triggered also a discussion on the release practices of the firm. From the service use statistics it is possible to see that the service is heavily used from Monday to Thursday, less on Friday and very little during weekends. Thus it is likely that features released on Mondays will get used sooner than the ones released on Fridays, which provides the additional benefit that the feedback from users (8) would reach the developers when they still recall the software they were working on. Even more profound than the weekly cycle is a similar variation related to the vacation seasons. Deploying new features just prior to vacations will have negative impact on the Value cycle, as described above.

IV. Summary, Conclusions and Further Research

The feedback from practitioners suggests that the current literature lacks metrics that could be used for directing a software development organization from the business perspective to enhance effective value creation and value capture. Although the Lean Startup Methodology proposes to develop the software via the build–measure–learn cycle, we seem to lack the means to measure the value that the delivered software creates. Also the researchers have observed this problem and conclude [7], that the current theories of value do not present a simple way of assessing customer-perceived value. Although companies put a great amount of effort into increasing customers' perceived value in the product development process, determining how and when value is added is still a challenge even in marketing and management sciences [7]. Previous literature on XP, Scrum, lean startup and related approaches has indicated that in the context of SaaS services, delivering new versions of a service to the customer, collecting the usage data and making further decisions based on the data provides the most promising path for the software vendor to understand customer-perceived value. Agile methods have been shown to enable shorter lead times and a stronger focus on customer needs [32].

Shortening the cycle times provides increased flexibility maintaining options to change development direction with speed [20][22] as well as other business benefits for software service firms. This encouraged us to search for metrics that help software firms in the process development towards shorter cycles. On this basis, we formulated the research question as, what metrics would guide the cycle-time-driven software engineering process development in established organizations?

As the proposed solution, we adopted and extended the lean startup [20] value creation cycle and constructed a framework for metrics based on the times between main observable events within the cycle, all the way through to receiving and analyzing user feedback. This focuses attention on fast execution of the value creation within the user feedback cycle. That is, we are not trying to measure value of the results of the cycle, such as the value of the product produced or the value of the startup or progress of the startup in creating the offering, as in lean analytics [12].

 ISBN: 978-1-61208-438-1

By finding the measurable values from within the value creation cycle, the cycle-time metrics framework aims at bridging the gap between cost-oriented SWE metrics and value-oriented business metrics. Cycle-time reduction serves as the intermediary of increased value creation guiding software feature development and software process development. The metrics measure the calendar time between the key events. The first three events are related to feature need identification and the business decision to implement the specific feature (event 3). The core events following this decision are start of the development (4), the feature is ready for deployment (5), the feature is deployed (released, 6), and first use of the feature by a customer (7). These events are followed by feedback related events, the feature feedback data has been collected and analyzed (8) and a decision is made based on the feedback (9). The time intervals between the core events (4-7) are of most interest for the engineering while the other events (1-3 and 8-9) relate to the customer-perceived value analysis of the feature. We also provided examples on how changing the measurement cycle directs the process development to new process areas.

Our empirical focus was at the level of features being added to an existing SaaS offering. In the empirical part, the times between the events in the core cycle were measured for five new features in the development processes of an independent software vendor's SaaS service. The results showed that the core metrics were able to capture and bring up useful characteristics of the business process that triggered both a "drop vs. develop feature" discussion (for feature F1498) and a number of process development discussions. In these five feature development cases the average development time was shorter than the waiting time for the feature to be released. This has negative impact to the efficiency of fixing potential problems emerging during the first uses of the feature by first users, as the developers have already oriented towards another assignment. The detection of the delay of feature releases lead to a further analysis of the vendor's release practices in general and prompted quick improvements to their process.

Although the results from the empirical part showed that the metrics are useful in practice, there are still several avenues of further research that we wish to explore. The empirical part used data from the engineering system and customer feedback data to identify the core events. This seems to be a useful starting point and the firm in our case study would like to extend the collection of data to cover as many of the nine events as possible and as automatically as possible. The time from release readiness to analyzed customer feedback seems to be a particularly useful measurement of deployment performance.

In general, collecting the data can and should be automated using engineering information systems to the extent possible (events 1 and 8 cannot be detected automatically). For the other events, we propose collecting and depicting the data graphically in real-time status displays providing an overview of the development activities for business and engineering management. As we can observe from the empirical case, the results are useful both for focusing process development activities and for making business decisions regarding which features will be developed further, which will be used as they are, and which features will be removed from the service. This way the simplified depiction can provide transparency between the business and the development organization. Thus we encourage further empirical work on the automation of data collection and its depiction based on events identified in the framework.

In startups the result of value creation cycle can be analyzed in the context of the evolution of the enterprise [12]. In context of established feature development processes, this framework adopted the approach of using only cycle times between events as the metrics within the value creation cycle. This is due to limited applicability of suitable previous research results for real-time customer-perceived value analysis beyond A/B testing and similar tools that can be used between events 7 and 8. Although cycle time metrics seems to provide high added value to focus process development in connecting software development with customer value, investigating the value capture events 8 and "*" further is needed. Finding an easy to apply means for estimating the perceived user benefits would enable various new developments supporting the operative business development of a software engineering team.

ACKNOWLEDGMENT

This work was supported by TEKES as part of the Need for Speed (N4S) Program of DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

REFERENCES

[1] B. W. Boehm, "Value-based software engineering: Overview and agenda," in Value-based software engineering, Springer Berlin Heidelberg, 2006, pp. 3-14

[2] M. Khurum, T. Gorschek, M. Wilson, "The software value map - an exhaustive collection of value aspects for the development of software intensive products," Journal of software: evolution and process. Wiley, 2012, 711-741.

[3] H. Erdogmus, J. Favaro, W. Strigel, "Return on investment," IEEE Software 3(21), 2004, pp. 18–22.

[4] K. Conboy, "Agility from first principles: reconstructing the concept of agility in information systems development," Information Systems Research, 20(3), 2009, pp. 329-354.

[5] S. Barney, A. Aurum, C. Wohlin, "A product management challenge: Creating software product value through requirements selection," Journal of Systems Architecture, 54(6), 2008, pp. 576-593.

[6] A. Fabijan, H, Holström Olsson, J. Bosh, "Customer Feedback and Data Collection Techniques in Software R&D: A Literature Review," in Software Business. Springer International Publishing, 2015, pp. 139-153.

[7] J. Gordijn, and J.M. Akkermans, "Value-based requirements engineering: exploring innovative e-commerce ideas," Requirements Engineering 8(2), 2003, pp. 114-134.

[8] M. Rönkkö, C. Frûhwirth, S. Biffl, "Integrating Value and Utility Concepts into a Value Decomposition Model for

 ISBN: 978-1-61208-438-1

Value-Based Software Engineering," PROFES 2009, Springer-Verlag, LNBIP 32, 2009, pp. 362–374.

[9] R.B. Woodruff, and F.S. Gardial, Know your customer: New approaches to customer value and satisfaction. Cambridge, MA, Blackwell, 1996.

[10] C. Grönroos, "Value-driven relational marketing: from products to resources and competencies," Journal of Marketing Management 13(5), 1997, pp. 407–419.

[11] T. Woodall, "Conceptualising 'value for the customer': An attributional, structural, and dispositional analysis," Academy of Marketing Science Review, no. 12, 2003, pp. 1526–1749.

[12] A. Croll, and B. Yoskovitz, Lean Analytics: Use Data to Build a Better Startup Faste,. O'Reilly Media, Inc. 2013.

[13] P. Tyrväinen, and J. Selin, "How to sell SaaS: a model for main factors of marketing and selling software-as-a-service," in: Software Business, Springer, Berlin Heidelberg, 2011, pp. 2-16.

[14] V. Mandić, V. Basili, L. Harjumaa, M. Oivo, J. Markkula, "Utilizing GQM+ Strategies for business value analysis: An approach for evaluating business goals," The 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2010.

[15] M. Saarikallio, and P. Tyrväinen, "Following the Money: Revenue Stream Constituents in Case of Within-firm Variation," in: Software Business. Springer International Publishing, 2014, pp. 88-99.

[16] J. Bosch, "Building products as innovation experiment systems," in: Software Business, Springer, Berlin Heidelberg, 2012, pp. 27-39.

[17] K. Schwaber, and M. Beedle, Agile Software Development with SCRUM, Prentice Hall, 2002.

[18] K. Beck, Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.

[19] A. Cockburn, Agile Software Development, 1st edition, 256 p. Addison-Wesley Professional, December 2001.

[20] E. Ries, The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Publishing Group, 2011.

[21] A.R. Hevner, S.T. March, J. Park, S. Rami, "Design Science in Information Systems Research," MIS Quarterly, Vol. 28, No. 1, 2004, pp. 75-105.

[22] J. Järvinen, T. Huomo, T. Mikkonen, P. Tyrväinen, "From Agile Software Development to Mercury Business," in: Software Business. Towards Continuous Value Delivery, Springer Berlin Heidelberg, LNIB, vol. 182, 2014, pp 58-71.

[23] H. Erdogmus, and J. Favaro, "Keep your options open: Extreme programming and the economics of flexibility," in Giancario Succi, James Donovan Wells and Laurie Williams," Extreme Programming Perspectives", Addison Wesley, 2002.

[24] M. Brydon, "Evaluating strategic options using decision-theoretic planning," Information Technology and Management 7, 2006, pp. 35–49.

[25] M. Fowler, Continuous Integration, 2006. http://martinfowler.com/articles/continuousIntegration.html retrieved: Septmeber, 2015.

[26] J. Humble, and D. Farley, Continuous delivery: reliable software releases through build, test, and deployment automation, Pearson Education, Jul 27, 2010.

[27] P. Debois, "Devops: A software revolution in the making," Cutter IT Journal, vol. 24, no. 8, August, 2011.

[28] J. Johanson, Standish Group Study, presenation at XP2002.

[29] K. Petersen, and C. Wohlin. "Measuring the flow in lean software development." Software: Practice and experience, vol. 41, no. 9, 2011, pp. 975-996.

[30] J.S.Mill, Principles of political economy, 1848, abr. ed., J.L.Laughlin, 1885.

[31] M. Cohn, Agile estimating and planning. Pearson Education Inc. 2006.

[32] M. Poppendieck and M.A. Cusumano, "Lean software development: A tutorial," Software, IEEE, vol. 29, no. 5, 2012, pp. 26–32.

 ISBN: 978-1-61208-438-1

Publication III

T. Lehtonen, S. Suonsyrjä, T. Kilamo, T. Mikkonen. Continuous, Lean, and Wasteless: Minimizing Lead Time from Development Done to Production Use. In *Euromicro Conference series on Software Engineering and Advanced Applications (SEAA)*, 2016.

Continuous, Lean, and Wasteless: Minimizing Lead Time from Development Done to Production Use

Timo Lehtonen[1], Terhi Kilamo[2], Sampo Suonsyrjä[2], and Tommi Mikkonen[2]
[1]Solita Plc. FI-33000 Tampere, Finland
[2]Tampere University of Technology, FI-33720 Tampere, Finland
timo.lehtonen@solita.fi, terhi.kilamo@tut.fi, sampo.suonsyrja@tut.fi, tommi.mikkonen@tut.fi

***Abstract*—Modern software organizations invest substantial effort in building and automating their tool chain. The goal is to maximize both the speed of development, and how rapidly new software is deployed. This paper presents results from a descriptive and exploratory single case study from an ongoing project of Finnish company Solita. Based on data from version control and production logs, we investigate the feature flow in the project to study the effect of lean processes and the continuous deployment tool chain to waste produced by the deployment pipeline. We find that flow efficiency can be improved simply by minimizing the idle time the feature waits in the production process after its implementation has been finalized. This reduction of waste benefits both end users and developers – the users get access to new features, and the developers receive timely feedback.**

I. Introduction

Software companies are paying increasing attention to their prompt ability to deliver value to end users. The goal of Lean Software Development (LSD) [1] – or applying the principles of Lean manufacturing to software – is to postpone decisions to the latest possible moment while delivering value as soon as there is value to deliver. Already completed features are constantly improved based on data regarding the actual use of software. In the spirit of LSD, actions that do not serve these goals and delays in delivering the features to end users can be regarded as waste. Extensive infrastructure is needed to maximize the speed of development and deployment. Such pipeline, including a version control system, build and test servers, and automated production installations, enables delivering features to end users more rapidly than ever before [2].

In this paper we study what kind of LSD processes and tools direct the development towards rapid continuous deployment, and how giving timely information throughout development can support team workflows. As concrete data, we use the project ”Lupapiste”, freely translated ”Permission desk”, a web site for managing municipal authorizations and permissions created by Solita, a mid-sized Finnish software company specializing in the design and implementation of web systems, analytics, and business intelligence. We investigate the feature flow of the project in regard to the lean concept of waste. To recognize waste produced in the deployment pipeline, we have applied the Value Stream Mapping (VSM) [3] method to the project's continuous delivery pipeline.

II. Background

Lean Software Development (LSD) refers to applying the principles of lean manufacturing in the development of software systems [1]. In LSD, independent teams deliver software where new features are constantly made available to end users, and features that already exist are constantly improved based on data regarding the actual use of software. Actions that do not serve these goals, as well as delays deploying the features, can be regarded as waste, which is to be eliminated.

One way to advocate LSD is Continuous Delivery, or the practice of keeping the software in such a condition that it can be deployed to staging and production environments at any time [4]. This enables faster feedback, increased productivity, and improved communication. Continuous Deployment (CD) [5] takes one step further from delivery by automatically deploying software as it gets done and tested. When taking CDto the extreme, new features are deployed to the end users several times a day [6].

Regardless of actual deployment, continuous software development requires a tool chain – a deployment pipeline, which uses an automated set of tools from code to delivery. The role of these tools is to make sure each stakeholder gets a timely access to the things they need. In addition, the pipeline provides a feedback loop to each of the stakeholders from all stages of the delivery process. In this paper we use the concept of a deployment pipeline based on the ideas of Humble and Farley [4] as the reference model. The pipeline they propose is widely used by practitioners especially in the field of web application development both in the industry and in open source projects. The approach of [4] is to make use of a distributed version control system (DVCS), build once, and then deploy the same binary to any execution environment.

III. Waste and the Deployment Pipeline

In manufacturing context, value stream is a collection of actions that are required to bring a product through the two main process flows, starting with a raw material and ending with the customer [3]. *Production* is the flow from raw material to the customer, and *design* is the flow from concept to launch. In the context of the deployment pipeline, the production flow is essential. It is the process where the specifications are converted into the elements of the software that produce business value to the stakeholders, as flow items go through the deployment pipeline. The design flow is out of the scope

for this paper, as it occurs outside the deployment pipeline, mainly in requirements management and design phases.

In a lean process, resources add value to a *flow unit* [7]. A flow unit is typically material, information or people. *Lead time* is the time it takes for the flow unit to flow through the process from beginning to end. In software development context, a flow unit is a new feature implemented into the software, which may be, for example, new functionality, bug fix, or technical refactoring. The resources that add value to the feature are the team of developers, customer-side stakeholders, and the users of the software. The servers of the deployment pipeline also add value to the feature as they can verify the code automatically in the different environments. This shortens the lead time as no human work is needed for verifying that the system is still working after the updates.

The efficiency of a lean process is divided to two different views: *flow efficiency* and *resource efficiency* [7]. *Flow efficiency* describes how much a *flow unit* is processed in a specific time frame – here, the time it takes for a feature from completion of development to its first activation in production use. *Resource efficiency* is about maximizing the usage of the resources, which in general means maximizing the time that resources, in this case people or machines, spend executing their work. As this issue is hardly a key question in software development, we focus on flow efficiency of new features from development to production use.

In LSD, any activity in the value stream that does not produce value to the flow unit can be regarded as waste [7] Thus, waste within software development ranges from unnecessary code getting written to any delays that occur during the development process [1]. In this paper, the focus is on waste produced by delays in the deployment pipeline.

All activity that happens in a value stream [3] can be divided into two categories: value added (VA) activity and non-value added (NVA) activity. We define value added activity related to a new feature in the context of this paper very strictly. From our viewpoint, the only value added activity is the development phase. Once development is completed, the time until the first use in production environment is considered waste. In this context, we have identified three sources of waste – extra steps in quality assurance, waiting for use after deployment, defects in production – each contributing to extended lead time. This is wasteful as with short lead times, the feedback cycle is also short and effective, whereas with longer lead times, there are more flow units flowing simultaneously, which forces the developers to context switching and restarting unfinished tasks.

Extra Steps – Testing and quality assurance: Depending on the type of application being developed, extensive testing can be interpreted as extra processing steps on an already finished feature, i.e. waste. If new features are implemented, the application is verified to work with extensive regression testing. The risk that the application does not work at all, or causes harm that should then later be cleaned up, is usually rather low. Instead, when the team deploys the software into the production environment, hidden bugs and defects rise to prominence. Moreover, the team gets timely feedback from the actual users of the system. The developers may not understand the user needs thoroughly and the deployment makes the conflicting interpretations visible.

Countermeasures: Whenever new features are done, deploy them to the production the next working day. Plan the deployment schedule with the customer to, for example, once or twice a week. The definition of done can even be pushed to mean delivered [4]. In many cases, developer testing locally is enough, and the users in the production environment can do the rest of the testing, often without considering (or even knowing) this. If there are bugs, the users of the system will report them. Furthermore, the monitoring platform may catch some of the defects and report them automatically.

Waiting, including the time that a feature waits for first use after deployment to production environment: By definition, inventory is waste, and in this case, every execution environment is an inventory where a feature may be pending. If a new feature has been deployed into an execution environment and its testing process is slow, waste may be produced. Moreover, if a new feature has been deployed into production environment and is waiting for use, the time spent waiting is in this context regarded as waste. The reasons why the users do not use the new feature can vary. For example, it is possible that the users do not know about the addition of the new feature, or they may not need the new feature yet.

Countermeasures: Plan the customer acceptance testing in advance, so that it happens with a short feedback cycle after the development is done. Communicate effectively with the testers and inform them about the new features. When the testers know what to test and how to test it the possible acceptance testing performed by the testers should proceed fluently. Demand the customer to test rapidly in the quality assurance environment and inform the users about the new features effectively. Help the users find the new features by keeping them up to date efficiently. One way to inform the users in some settings is to contact them personally. Others include sending email announcements or including an introductory wizard to the user interface of the system to guide users through the newly added features.

Defects in production environment not caught by developer tests: If regression and acceptance testing of a new feature have not been extensive enough, defects that arise in the production environment produce waste. Part of the waste is the context switch – if the developer has already started the implementation of another feature, the context switch to fixing a critical bug in production environment, and then switching back to the earlier task is waste. Another form of waste is the extra communication needed for example for browsing the issue management system, reading the bug reports and asking for more information from bug reporters.

Countermeasures: One way to avoid context switching is to agree on a duty officer where one team member at a time is responsible for handling the urgent bug requests. This way, developers' context switching – obviously waste – is avoided.

The fundamental problem behind all the three wastes of deployment pipeline — Extra Processing Steps, Waiting, and

Defects — is the long lead time. The first two of the three wastes lengthen the lead time of a new feature being implemented directly. If the feature is waiting in an inventory, or if it is processed further in vain, the lead time is prolonged. When the feature is processed in the testing phase and when the feature is waiting for deployments of waiting for users, the lead time is prolonged. The third waste, defects in production, also prolongs the lead time, because a broken new feature was incomplete, and the lead time of the new feature can be interpreted to end only when the feature was completely implemented. Moreover, the management work needed for handling the defects is waste, which prolongs the lead time.

The larger the amount of flow units flowing simultaneously, the more choices there are for context switching and for restarting unfinished tasks. Context switch for a programmer is an expensive operation [8] that should be avoided in order to be more effective in the development process. Restarting a previously unfinished task produces waste in terms of doing the same things again.

Finally, the latency between the deployment and the first use in production makes the feedback cycle even longer. With latency we mean the extra time a flow unit spends in a non-value added activity. For example, if a new feature is waiting for the deployment to the production environment for 10 extra days, the total lead time is prolonged in vain from the flow units point of view. There are many reasons for deployment latency. For example, the current version may have been refactored in a way that breaks the regression tests.

IV. Case Lupapiste

A descriptive and exploratory single case study industrial case study [9] was conducted to investigate waste in a state-of-the-art pipeline within a two month's time frame of actual development work. Focus was on what kind of a lean development process and toolset direct the development processes towards CD and how describing timely workflow information during development can help to detect bottlenecks. The application "Lupapiste" is available at https://www.lupapiste.fi/. The permission types applied contain e.g. building permissions, environmental permissions and digging permissions. There is also a lightweight free form information request for the citizen to ask geolocated questions from the municipal authority. The municipal authority can start a dialog with the citizen through the chat functionality of the application, and then, if needed, the citizen can apply for a formal building permit with detailed information on the location and properties of the building including architectural drawings and plans as attachments. Also some other parties, like an architectural designer, can be invited to the application easily by email. When the applicant thinks that everything is ready to be submitted, the information is then transferred to a municipal decision making system. Some time later, when the authorities have handled the application usually in a formal meeting, the system sends email to the citizen, who is then redirected back to the application to see the details of the decision. The usage of the system nation-wide is emerging and some tens of municipalities have taken the system into use. The amount of municipal authority users is in the hundreds and the amount of citizens using Lupapiste is hundreds per month.

The project was started in 2012. The supplier of the system is Solita Plc., a mid-sized Finnish software company. The end user of the system consists of various stakeholders, with various interests. The Environmental Ministry of Finland owns the project code and acts as a customer in some new functionalities needed to the system. The business model is unusual as the marketing of the system has been delegated to the supplier, providing an opportunity for the supplier to collect transactional costs of the permissions applied by the citizens and companies. This motivates the supplier on marketing the application to the municipal authorities in Finland. Indeed, in comparison to the old fashioned paper based system, Lupapiste has drastically shortened the lead times of building applications. For example, as reported at the web site https://www.lupapiste.fi/, in the city of Järvenpää the lead time of the permission handling process was shortened from three weeks to three days.

At the time of research (Fall 2014), the project team consisted of four developers, a user experience (UX) designer and a project manager co-located in a single workspace at the supplier. On the management level there are four more persons in different roles. Some team members have an ownership of certain parts of the system, but the knowledge is actively transferred inside the team by rotating these continuously and for example by applying agile practices like pair reviewing of code to spread out the knowledge in a continuous manner. The team makes use of a custom Agile process that includes features from Scrum, but with Lean heritage where also Kanban tools, e.g. WIP-limit or kanban cards [10] are applied. The case project's pipeline environment consists of a personal local development environment, the common development environment, a testing environment, a quality assurance environment, and the production environment. The deployments to the different environments are managed through the version control system, which then provides the accurate and automatic metadata to measure the process.

The study is based on quantitative data and descriptions of the development processes and the pipeline collected from the developer team. Additional informal discussions were carried out to gain knowledge on team practices for deployment. Empirical data of the case project was collected from the project's DVCS – Mercurial VCS (http://mercurial.selenic.com/) – and monitoring system Splunk (http://www.splunk.com). The data gathered was then exported and analyzed using statistical tool R (http://www.r-project.org).

To depict the project's development and deployment pipeline direct, informal discussions with the developers were held by one of the researchers. The researcher works in the case company and has been working in the case project earlier, so the current team could be contacted directly by the researcher to collect the required information. Software measurement as a process of representing software entities, like processes, in quantitative numbers [11], was further applied

in the study. We also applied descriptive statistics to get an understanding of the data collected.

Researcher bias in this types of study settings is unavoidable. The researcher responsible for the description of the case projects deployment pipeline has been working in the case project. Another internal threat is selection bias: the case project brings with it a plethora of aspects, such as development and customer culture, that affect the study. A clear threat to external validity of the research is that it investigates a single case. Thus, it poses a threat to the generalization of the results.

V. Results and Discussion

Identifying Waste: We applied Value Stream Mapping (VSM) to recognize the waste in the lean process used together with the deployment pipeline. The VSM current state map [3] is depicted in Figure 1, with an extension depicting a branching model [12] on a conceptual level. The figure is a combination of a process description combined to the inventories of the deployment pipeline. The terms used in the diagram are derived from the branching model and from Kanban [13] terminology and the terms that software engineering practitioners use. The Kanban terminology comes from the team's development practices. The team has for example used a Kanban board to visualize the state of the process [13] in order to recognize the bottlenecks on the deployment pipeline.

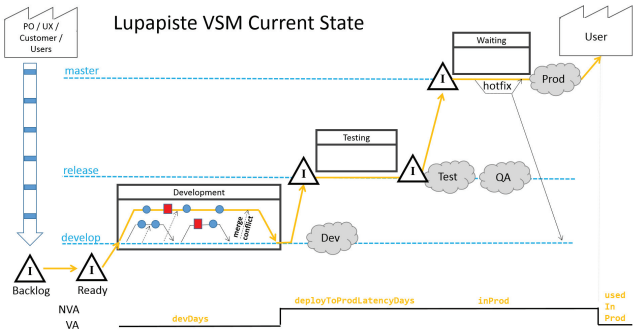

Figure 1. A value stream mapping of the case project.

The development of a new feature is started when developer starts working on the feature from the ready buffer. Physically the ready buffer is a column on a Kanban board indicating the items there are ready to be implemented. The product owner fills the ready buffer, selecting the most important features at the moment from the product's backlog. The backlog in turn is filled with user needs and requirements from the customer, or with requirements up with which a UX designer or the product owner have come. The ready buffer always contains a small number of the most important features needed to the system being developed which the developer can start working on. In the case, the concrete day-to-day software development process of the team is based on the use of the version control system and the Hudson continuous integration system (http://hudson-ci.org/).

At the beginning of the development of a new feature, its ticket on the Kanban board is moved to the "development" column, marking the start of the programming phase. The developer opens a new local feature branch in the distributed version control system. The state of the development branch has to be green according to the information radiator [14], implying all tests in the development branch must pass. If broken code has been committed to the development branch, a lot of waste is generated, and, in addition, other team members are not able to start the development of new features or synchronizing with the development branch. Each developer works on a single item at a time by writing the source code with comprehensive tests. Value is added in a continuous manner with local commits [15].

The development process includes programming activities that introduce value. If the lead time of development phase is long, the risk of merge conflicts rises, because the longer the development lead time of a feature is, the more merge operations are needed, as the develop-branch changes often on an hourly or daily basis. The developer has to stay up-to-date with the changes of the development branch at latest when the feature is ready.

The deployment to the test environment in the pipeline is made by merging the development branch to the release branch. When changes are pushed to the release branch, a build to the test environment is triggered automatically. After this, there may be some manual testing in this phase performed by for example the product owner or a user experience designer, or the different user roles at the customer. For some features, even some formal acceptance testing for the new features is needed before deploying them to the QA or production environments. In the context of this research, this latency in testing is considered to be waste, because when the feature is ready, a strict definition of done [16] would enforce it to be done in the sense of "ready for production".

The deployment to the production environment happens by merging the release branch to the default branch and then building and deploying the project with a single click in the CI. When a manual step is required, it is easier for the team to keep control of the publishing schedule. Another manual step is to move the tickets on the Kanban board between the columns. In the case project, the schedule for production deployments is agreed on with the customer in weekly product owner team meetings, where involves both the supplier and the customer.

To summarize, the three wastes of the case project are:

- *Extra Processing Steps*: Testing procedure is too long.
- *Waiting*: Features are waiting for use in production environment, as users do not know about them.
- *Defects*: There are defects in the production environment not caught by developer tests.

Feature Flow: During the two month research period, the team completed a total number of 38 new features (Table I). The number of features that contained new API interfaces, and were thus fully trackable through the logs in the monitoring system, was 5. The mean development days per feature was 5 working days. Thus, on average, if a developer starts devel-

opment on a new feature today, the same developer can start the implementation of another feature on the same weekday next week.

Table I
METRICS DURING THE RESEARCH PERIOD.

	Development (days)	Deployment latency (days)	Business value latency (days)	Flow efficiency	Development done to production use (days)
f1	10	0	1	91	1
f2	4	0	1	80	1
f3	3	0	1	75	1
f4	1	2	1	25	3
f5	7	3	1	64	4
f6	7	7	1	47	8
f7	4	2	1	57	3
f8	1	4	1	17	5
f9	11	4	1	69	5
f10	1	5	1	14	6
f11	12	5	1	67	6
f12	1	5	1	14	6
f13	2	7	1	20	8
f14	2	8	1	18	9
f15	1	8	1	10	9
f16	9	9	1	47	10
f17	1	10	1	8	11
f18	7	10	1	39	11
f19	2	11	2	13	13
f20	7	12	1	35	13
f21	2	8	1	18	9
f22	9	8	2	47	10
f23	10	8	1	53	9
f24	2	10	1	15	11
f25	10	11	1	45	12
f26	1	14	1	6	15
f27	5	17	1	22	18
f28	10	17	1	36	18
f29	1	18	1	5	19
f30	2	18	1	10	19
f31	10	18	0	36	18
f32	6	19	1	23	20
f33	9	20	1	30	21
f34	1	20	1	5	21
f35	2	24	1	7	25
f36	2	1	1	50	2
f37	1	1	1	33	2
f38	2	1	1	50	2

The mean latency in deployment to production was 9 working days. The value is high, as any extra waiting prolongs the lead time in vain, which leads to the problems related to a long lead time described in Section III. The mean latency for production use for the five features that it was possible to collect the data for was 3 working days. Median was 1 working day and mode was 0 working days. We assume that latency of one working day is the best assumption of typical production use to occur, i.e., when the new feature is deployed, it will be used the next day. Assuming a production use latency value of 1 working day for the rest 33 features, the mean of flow efficiency was 34% and the mean time from completed implementation to production use was 10 working days. In other words, when the development was done, it took two weeks to get actual feedback from the production environment from the real users using the new features. Most of the latency occurs in the deployment phase (9 out of 10 days).

Summary: A modern development toolset includes a deployment pipeline built on top of DVCS. Such a toolset enables rapid and continuous deveplopment, while the obstacles of the rapid development are visualized and eliminated, thus continuously optimizing the workflow. The demonstration with VSM and the feature flow efficiency shows that the flow efficiency can be improved simply by deploying more often. With frequent deployment, waste is eliminated. Lowering the latency of deployment leads to faster use and shorter feedback cycle.

VI. CONCLUSIONS

We studied the flow efficiency of a software project, with focus on the lead time from development done to production use. The goal was to recognize waste in a state-of-the-art deployment pipeline combined to a lean software development process. The results show that when the lead time is longer, the amount of waste increases in terms of context switching, starting unfinished tasks anew, and longer feedback cycles. In contrast, together with rapid deployment of new software, the ability to manage the lean flow efficiency results in shorter feedback cycles, fewer unfinished tasks and context switches, and, eventually, eliminating waste.

ACKNOWLEDGEMENTS

This work is supported by Tekes (http://www.tekes.fi/) and Digile's Need for Speed program (http://www.n4s.fi/). Thanks to Antti Virtanen for the idea of considering lead time to cover the time until the first production use.

REFERENCES

[1] M. Poppendieck and T. Poppendieck, *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

[2] K. B. Stone, "Four decades of lean: a systematic literature review," *International Journal of Lean Six Sigma*, vol. 3, no. 2, pp. 112–132, 2012.

[3] M. Rother and J. Shook, *Learning to see: value stream mapping to add value and eliminate muda*. Lean Enterprise Institute, 2003.

[4] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[5] J. Humble, C. Read, and D. North, "The deployment production line," in *Agile Conference*. IEEE, 2006, pp. 6–pp.

[6] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, p. 1, 2013.

[7] N. Modig and P. Åhlström, *This is lean: Resolving the efficiency paradox*. Rheologica, 2012.

[8] R. C. Martin, *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011.

[9] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2014.

[10] J. Boeg, *Priming Kanban: A 10 step guide to optimizing flow in your software delivery system*. Trifork, 2011.

[11] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[12] V. Driessen, "A succesful git brancing model." http://nvie.com/posts/a-successful-git-branching-model/, retrieved: November 2014.

[13] H. Kniberg, *Lean from the trenches: Managing large-scale projects with Kanban*. Pragmatic Bookshelf, 2011.

[14] M. Pikkarainen and A. Mäntyniemi, "An approach for using cmmi in agile software development assessments: experiences from three case studies," in *SPICE 2006 conference, Luxemburg*, 2006, pp. 4–5.

[15] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" Corvallis, OR: Oregon State University, Dept. of Computer Science, Tech. Rep., 2014.

[16] K. Schwaber and M. Beedle, "Agile software development with scrum. 2001," *Upper Saddle River, NJ*, 2003.

Publication IV

T. Lehtonen, V. Eloranta, M. Leppänen, and E. Lahtinen. Visualizations as a Basis for Agile Software Process Improvement. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, 2013.

Visualizations as a Basis for Agile Software Process Improvement

Timo Lehtonen
Solita plc.
Tampere, Finland
timo.lehtonen@solita.fi

Veli-Pekka Eloranta, Marko Leppänen, Essi Isohanni
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
{firstname.lastname}@tut.fi

***Abstract*—Software projects have usually a lot of software engineering data available in different kinds of repositories. This data can be mined and used for software process improvement purposes. In general, agile methodologies emphasize reflection, making problems visible, and learning from the past. As the human mind is powerful in interpreting visual representations, visualizations could help in recognizing problems and areas of improvement in an agile software development process. In this paper an action research approach was taken to carry out software process improvement in an industry project. The research resulted in a visualization of the issue management system's data. The visualizations were a tool to identify problems in the development process and to make them visible for all stakeholders. The results show that this kind of visual approach can be used successfully to point out problems in the process. The visualizations form a basis for communication on possible software process improvement.**

Keywords - visualization; agile software development; SPI

I. INTRODUCTION

One of the cornerstones of Scrum and agile is continuous improvement. It is based on regular reflection and improving the efficiency of the development team by learning and improving the development process. In Scrum this reflection is carried out in sprint retrospective [20] [21] [18]. In addition, other systematic methods have been proposed [16].

Sometimes the development team might get complacent or just blind to their opportunities for process improvement. This "happy bubble" [17] might be hard to pop and the process improvement may stagnate. To tackle this, we will present in this article a way to visualize the software process to make the problems visible and to encourage the team to do software process improvement (SPI). The visualizations are a tool that can be used by the team to identify the problems, and consequently to monitor if the changes in the development process are alleviating the problems.

The study is based on data from an industrial ERP development project. Action Research [1] approach was taken to find out what kind of visualizations would surface the problems the best and would be usable for the development team and to the customer too. In the end, visualization was used to identify problems in sprint length, release cycle, task size, lead time and communication between the development team and the customer. Statistical analysis on the visualized data was carried out to see if the visualization really illustrated the problems which were recurring frequently. Finally, the same visualization approach was applied in another project to see if the result can be generalized to other similar projects.

This paper is organized as follows. Section 2 describes our research methodology, i.e. Action Research. Section 3 characterizes the development method that the team used before this study was initiated. Section 4 describes the case project and the subsequent iterations of the action research are explained in sections 5-7. Section 8 describes the outcome of the last iteration and thus presents the results of the study. In section 9 the visualization approach is validated by applying it to another software project and showing the results of a statistical analysis. Related work is discussed in section 10 and section 11 gives some concluding remarks.

II. APPROACH

The chosen methodology for the research was Action Research (AR) [1] as it gave us opportunity to work within the complex environment of software development. The objects of our research, i.e. the problems of the software project and the visualizations, cannot be separated from their context in the software development project. The actions of improving the software process do not have a meaning independent of their associations in the project. AR is a situational research method that suits for research work in such a context [5]. It is also collaborative suiting for the co-operation of researchers and practitioners and adapts to the process, like in this case the yet unknown software project [5]. Action Research is iterative in nature and lasts until a satisfactory result has been achieved. In the beginning of the first iteration we had only a very general idea of the problem: the customer has reported some quality problems in the project. So, AR suited our study perfectly, as in the beginning we were not certain how we could solve the problem and what are the real reasons behind the problem. Altogether, we carried out four iterations during time period from November 2012 to May 2013.

Action Research consists of the following three steps [1]:

1. Diagnosis of the problem

2. Action intervention
3. Reflective learning

The first step in AR is to diagnose the problem. In our case, the purpose was to find out the way the development team is working and what kind of problems they have in their development process which caused the quality problems. Additionally, we needed to map out what kind of process data is available in our use and how we can use it to catalyze the team's SPI process. Diagnosis of the problem is described in more detail in Section V.

Second, we took the initial steps towards solving the problems we found out in the step 1. After the first attempt to solve the problems, reflective learning was carried out together with the development team, customer and software development process experts from the university and from the company. Finally, we repeated steps 2 and 3 three more times until we were satisfied with the end result.

Once our visualization approach seemed to make problems visible in the development project, we wanted to validate the approach in another project to make sure that the results are applicable to other projects too. Additionally, we carried out a statistical analysis to validate that the visualization highlighted issues that were the real problems in the development.

The initiation of the Action Research project was partially made by the researchers and partially by the practitioners. The researchers were looking for suitable projects to test out their ideas for measuring the effects of SPI and the effect of applying different software development practices. At the same time, the target project members were looking for various ways to improve quality and their development practices.

Company participants had the authority in the AR project. So, researchers only suggested possible solutions and the final call if the action to be taken was on the organization side. The development team has to deliver features to the customer and we did not want to disrupt this process. This affected the research in almost all iterations as the development team had to decline from trying out practices suggested by researchers.

The structure of the AR project was between formal and informal. Researchers were bound by a non-disclosure-agreement, but otherwise the role of the researchers was informal and there was no official contract on the research project.

III. SCRUM IN A NUTSHELL

Scrum is a well-known and widely adopted software development framework. Here it is presented as described in [19]. Usually practical implementations have taken this approach as a starting point and evolved from there by continuous improvement. Scrum aims at agility by providing ways to develop software in an incremental and iterative way. In addition, the development cycle is time-boxed.

Scrum framework is rather simple as it has only three roles for the people involved: Scrum Master, product owner and the development team. For organizing work for the team there are two backlogs: the product backlog and sprint backlog. The product owner (PO) is responsible for updating the product backlog and it should include all work that the team should do as product backlog items. These items are ordered so that the items which produce the most return on investment should be first on the backlog. The PO orders the backlog by consulting all stakeholders of the product. All items have their work estimates and estimates of the business value attached. These items can be a variety of things: product features, use cases, user stories, requirements, bugs, documents, technical issues, architectural refactoring and so on.

The Scrum Master's (SM) role is to keep an eye on the Scrum process. SM removes impediments from the work and makes it possible for the team to achieve the desired end result. The team is a small self-organized group of people who commit to implementing the items in the product backlog. They take the items and split them into tasks and divide the work autonomously. They keep track of the finished tasks and hold daily meetings with the SM. In this stand-up meeting, the team members tell what tasks they have accomplished, what tasks they're going to do next and what problems they have encountered.

The work is done in sprints, which are fixed length time spans. The original source says it would be 30 days, but usually companies have implemented shorter spans. Every sprint starts with a planning meeting where all Scrum roles gather to check the product backlog and let the team choose which items they can commit to. After that, these items are split into tasks whose work amounts are estimated. A task usually lasts from a half an hour to or 16 hours at maximum. A task has a definition-of-done, which will clearly state when a certain task is done and will not require work spent on it anymore. The team commits to a work amount that can be accomplished in one sprint. This amount is called the team velocity. After the team has begun the development, the sprint backlog cannot be changed. If some change is mandatory and the team cannot commit to get everything done, then sprint must be aborted. After the sprint is completed, there is a demonstration of the new features of the software and a retrospect meeting, where the sprint is reflected upon and the team and Scrum master decide on what they should improve, what to keep and to discard from the process.

A Scrum project is controlled by means of frequent inspection of the project and has a goal of making progress and problems visible [20]. A visual representation is universal and represents complicated concepts with a common language [3]. People participating in a Scrum project often want to understand the project with their own vocabulary [20]. Therefore, visualizations may work as a common language in a Scrum project.

IV. CASE PROJECT: LEHTIPISTE ERP ELMO

The case project used in this paper is Lehtipiste ERP Elmo. The customer is Lehtipiste, a marketing and distribution organization of single copies of newspapers and magazines in Finland. The vendor of the system is Solita plc. [22]. The ERP system manages the logistics of the delivery chain of magazines. Elmo project involves seven people on

the customer side and seven people on the vendor side. The development of the system started in 2005 and the system has been up and running in production since 2010. The project is in the maintenance phase which consists of bug fixing and developing of new features.

The current software development process used is a derivate of Scrum framework: an iterative process, which produces increments to the software after each sprint. The product owner role is held by a person on the customer side. There was no named Scrum master role, but project manager could be seen as a Scrum master. However, the role of the project manager as a part of the process improvement was unclear to the researchers.

The development is carried out in sprints. According to the project manager and the development team the length of a sprint is four weeks. The customer lists all features and issues she wishes to be implemented or resolved in the next increment. In other words, the customer plans the sprints beforehand and, typically, there are a couple of sprints planned beforehand as there is already more work than can be implemented in one sprint. This deviation from Scrum makes it hard for the team to commit to the work as they cannot select the items they take to the sprint by themselves.

As a notable deviation from the usual Scrum approach, each sprint consists of two phases: a development phase and a system testing phase. The version control branching model of the project supports this approach: in the development phase, all code changes are committed to the trunk. When the testing phase begins, a new branch is created in SVN. When the branch is created, a decision on which features to include in the production release is made. The bugs found in the testing environment are corrected to the branch and these fixes are also merged to the trunk.

In each sprint there is a reserved time for the bugs that are found in the testing environment from the previous sprint's increment. Typically, half way through the sprint, the team gets feedback from the testing environment and they fix the bugs found. This causes branching in the source control as the development will continue in the usual fashion, while the testing environment produces reports about bugs that should be fixed.

During the sprint, the developers write code and test it on their own development environment on their workstations. This local testing environment is a light-weight implementation of the production environment with a limited amount of logistics data. In the end of the sprint, the development version is deployed to the testing environment, which can be used to more extensive testing. Testing environment has the same inbound operative integrations to other systems as the production environment. However, the real end users are not using the testing server, so their input is not part of the test framework. Many new features can only be thoroughly tested in the production environment, because they may need, for example, real world sales and shipment data to work properly.

After the testing is over and the bugs are fixed, the system is deployed to the production environment. The testing phase takes four weeks and is scheduled so that the system is deployed in the middle of the current sprint. The customer will give feedback and bug reports after the deployment whenever necessary.

The issues and bug reports are collected with Jira issue management system [8]. It is used extensively in the project for both bug tracking and initiating development tasks of new features. The project has a policy to avoid emails and use Jira for communication instead.

Jira has a specified workflow for all development tasks in this project setting. When a developer starts the development, she changes the state of the task to *In Progress*. When the development is done, the state is changed to *Fixed*. When a test build is done, the fixed tasks go to state *Delivered to Test*. The customer carries out the testing, and when it is completed, the state is changed first to *Ready for Production* and then on release day to *Delivered to Production*. Then after some time, the task ticket is *Closed*.

Team members must have a shared understanding of what it means for work to be complete [20] [21]. Definition of "Done" in this project is task state *Ready for Production* in Jira. It means that the task is ready to be installed to the production environment.

In the beginning of year 2013, the project had some problems with the process and the customer reported quality problems in the new versions delivered. The project manager was contacted by the researchers and the action research was started. The goal was to investigate the project and to identify the root problems the software development process had. The responsibility on improving the process was left to the development team as that the team must be willing to make the changes or otherwise the SPI process wouldn't be successful.

V. ITERATION 1

A. Diagnosis of the Problem

In the beginning of the action research cycle it seemed that there is a rather large and complex ERP system at hand. There were hundreds of open tasks in the issue management system and many of them seemed to be actively worked on simultaneously. It was hard to form a clear big picture, so it seemed that metrics and visualizations could help to have a better understanding of the status quo.

The researches met the project manager in Jan 2013. The project manager explained the software development process and how Jira was used in the project. One essential observation was that all communication with the customer was carried out using Jira, so the researchers could have a good track of communication with the customer in their disposal. One of the researchers had been working in the project as a developer some years ago, so the project was partly familiar to the research team.

The project manager reported that there are problems with some of the Jira issues. For example, few implementation tasks have been open for months causing problems in an iterative time-boxed approach. The tasks were such that they were never "done" and thus there was no commitment to deliver all selected tasks at the sprint end as some of the tasks would fail anyways. Also, there were tasks

that have been closed for a while and they are then reopened. Reopening could happen for various reasons.

B. Action Intervention

The researches had an idea to use metrics for analyzing the project data. Metrics could help to see what kind of things can be said, for example, about the lead time of a user story. In the first step the researchers planned mining the data from the version control system and issue management system to reveal problems in the development process.

The mined data could be represented visually so that it helps to form a big picture of the project for SPI purposes. It was not known, which kind of visualization would be to most beneficial for SPI. The researchers also planned to use metrics to describe the project. The first metrics the researchers had in mind included bug tracking and lead times for the issues.

First ideas for visualization consisted of visualizing the data in version control system (SVN). There are lots of tools like SVNStat [23], which make it possible to visualize the events in the version control system. So, the SVNStat tool was used to generate the graphs. However, the graphs were not useful as the problems of the project seemed not to be source code dependent. So, the research direction was changed towards the issues in Jira, because Jira was used extensively and because the tasks contain a lot of useful metadata. The metrics on lead times seemed to indicate that the team had longer sprint cycle than the aforementioned 4 weeks. This had to be investigated further.

The people in Elmo project had a conception of their software development process as described in the previous chapter. However, it is hard to see, if the process is under control and performing well or not, which is crucial start point for process improvement [6]. Also, different team members had differing conceptions of the status quo. So, visualizations of the Jira content could help in forming the common conception of the project as the main problem with hundreds of active issues in Jira having tens of comments for each issue is that the data is in textual format. So, it is challenging to get a clear overview what is going on.

The Jira content was transformed to a visual form as follows. First, a Jira filter was applied to find issues from the past two years. The issue event data consists of state changes, comments and assignments. For example, an issue may have changed to state *In Progress* and it may have 10 comments and three assignments to different people. All these changes to an issue have timestamps.

The first draft of the visualization of process data was created by gathering the data from Jira with a simple Jira (JQL) query and exporting it to HTML format. Then D3 JS-library [2] was applied to generate a graphical representation. The visualization at this stage was not very useful, but the iterative process towards a useful visualization had begun.

C. Reflective Learning

The visualization made it possible to point out some problems in the project, such as long bug lead times. However, the data was just a flat export of bugs from Jira. The data did not contain any time based event data (issue state changes, assignments) and it did not take into account Jira comments. Thus a richer data source was needed, which would have more detailed data of the Jira events. Therefore, the visualization should illustrate more Jira's metadata information. Thus, we decided to drop the idea of visualizing any of the version control data and focus on data mined from Jira.

Time is one of the hardest variables to map in any complex system, but it also reveals a lot of important information [13]. The data in this iteration was flat and most of the time dimensional data was missing. The goal in the second iteration would be to get richer data from Jira and to represent it on a timeline.

VI. Iteration 2

A. Action intervention

An issue in Jira encounters many events during its lifetime. For example, its state can be changed, it can be assigned to another person or it can be commented on. To have the time dimension visualized better, the researchers needed an access to issue event data. However, the Jira export files do not contain this data. Thus, another way had to be used to acquire it. A direct connection to Jira database was not possible in this case because of company level security policies, so a custom Jira Web Crawler was implemented. The crawler was implemented with Robot Framework [15] and Python scripts. These technologies were chosen because of their familiarity to the researchers.

The visualization at this stage (Fig. 1) contains a timeline going from left to right, commenting events and state changes.

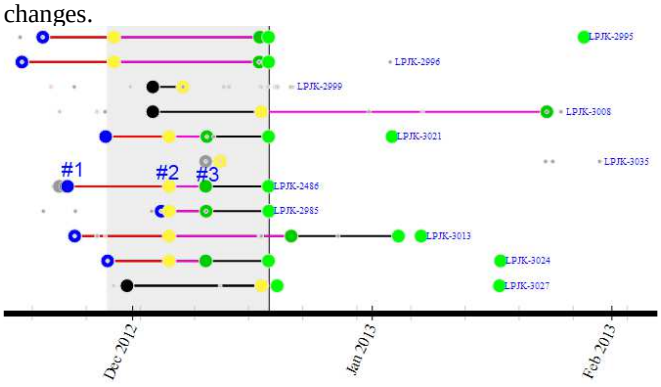

Figure 1. The issue life cycle is rendered according to the time based data.

The visualization in Fig. 1 contains a lot of data per issue. The two circles at point #1 in the figure show the time when the issue was started working on and then completed. Point #2 shows the time when the issue was installed to the test environment and point #3 shows the event when the task was ready for production. The task was then later installed to the production among other tasks. Comments are drawn with small gray bullets.

B. Reflective Learning

The diagrams provided some new information of the process. For example, now it was possible to see the length

of the testing time of an issue. The visual layout of the diagrams was still rather poor and the diagrams were cluttered by several colors. The visualizations now contained more information, but they were still rather hard to interpret. The goal of the next iteration would be to make the visualization easier to read and interpret. It would help to identify the problem points in the process.

The project manager also reported that large amount of tasks simultaneously in testing phase is becoming bigger and bigger problem. This was visible in our visualization, but we needed to make that more prominent, so the development team could easily see the problem from the visualization.

VII. ITERATION 3

A. *Action intervention*

As mentioned, the goal of this iteration was to improve the readability of the visualization. In addition, we wanted to emphasize the amount of ongoing work (development and testing). Kanban based software processes [10] take use of Work in Progress (WIP) limits. The idea of WIP is to limit the amount of simultaneous ongoing work. In Scrum, this principle is called single-piece continuous flow. This approach keeps individual tickets flowing on the Kanban board [10]. For example, the product owner may limit the number of tickets available in the development buffer [10] to five. This makes the priority of the chosen tickets higher and makes them flow more fluently.

At this stage, there was an effort to add a WIP limit to the diagrams. A trend curve summing up the total amount of ongoing tasks was added to the bottom of the diagram. The diagram is illustrated in Fig. 2.

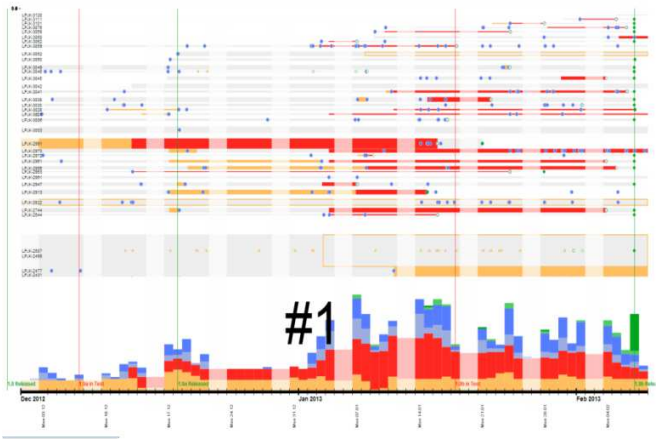

Figure 2. A diagram with a trend curve included.

The tasks are presented as rows in the visualization and the height of the row illustrates the task size. In Fig. 2 each of the sprint's tasks were visualized in the diagram. Comments are marked with blue circles. Red color marks the testing time of the issue. Green vertical lines mark the start and the end of the sprint. Red vertical lines mark the date when new software version is transferred from the testing environment to the production environment. Weekends are shown as white vertical lines.

The daily summary can be seen at the bottom of the diagram. All *in progress* tasks are summed with yellow color, and in testing tasks with red. For example, at point #1 in Fig. 2, the sum of the red blocks on the bottom starts to rise. This happens because there are more red lines on the top part of the diagram telling that there is a lot of testing going on. Also the number of comments and assignments per day were summed to the bottom to show a trend.

B. *Reflective Learning*

The visualizations at this stage started to be mature enough to get feedback from the customer. The researchers met the customer in the beginning of March 2013. The diagram in Fig. 2 was shown to the customer's representatives and it was used as a basis for communication. The researchers explained some details about the ongoing tasks of the project based on the diagrams. For example, some tasks seemed to have a lot of comments during their testing time.

In total, three persons on the customer side were interviewed by the researches: the IT manager, customer's project manager and a person responsible for infrastructure of the system. The researchers discussed with them asking general questions about how the process is going and if the current situation be seen from the visualization.

Every one interviewed agreed that the work during sprints is fragmented, as the team should focus on producing new features. During the sprint the team members still make bug fixes to the features from the last sprint. One important observation was that the visualization of the comments on the time span seems to show that some tasks have unclear requirements and some tasks get reiterated in testing. This might be due to the fact that the specifications are documented too early compared to their development.

The customer representatives could point out from the graph that the testing phase takes clearly too long for features. Visualizations could help the customer to motivate more active testing. This helps in shortening the lead times.

Visualization also illustrates the lead times of the tasks. Lead times were in many cases longer than the sprint length and thus, tasks were not completed within a sprint. Some features even have so long development time that they aren't needed anymore when they finally get deployed. Visualization could help the customer to see all features currently under development. As task IDs are shown in the visualization, the customer could identify if the feature in a sprint is such that it is not needed anymore and could be terminated. The main result here, however, was that the visualization revealed that the lead times are too long and a way to shorten them should be sought out.

The interviews also gave some information on the reasons why testing phase for some issues is so long: it is not possible to test some features before they have been running in the production environment for several weeks. The customer comments were used as the basis for improving the visualizations in the next iteration. In addition, UX specialists at Solita were heard to improve the layout and colouring of the diagram.

VIII. ITERATION 4

A. *Action intervention*

In iteration 4, the visualizations were simplified a bit. For example the color theme was changed based on the suggestions by the UX specialists.

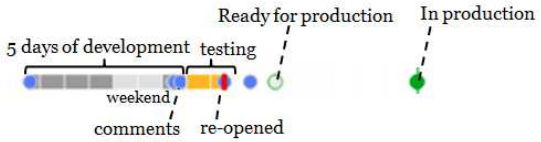

Figure 3. Lifeline of a single implementation task in a sprint.

The final form for representing a single issue is shown in Fig. 3. The task is first in ongoing development state (gray color) for five working days (there is a weekend shown in white in between). Then the testing phase (yellow color, or lighter gray if the image is in gray scale) of two days starts next week. During the testing phase, the task is re-opened (red line) and the task is returned to state "open". Finally, the task is deemed ready for production and on delivery day it is deployed to the production environment. Blue dots in the figure are comments. In theory, there should be gray color after the red line, because the task was now again in implementation. People tend to forget to change the state of the task, which happened in this case, too. In this case, there was a comment and then the task was ready for production. This implies that there is no need for such a state in Jira set up.

The real value of visualizing the sprint is that the diagrams can be used as a basis for communication. The project manager and the product owner could now get a better understanding of what is happening during the sprints. Generating the visualizations does not need any extra reporting work from the team – all information is pulled automatically from Jira.

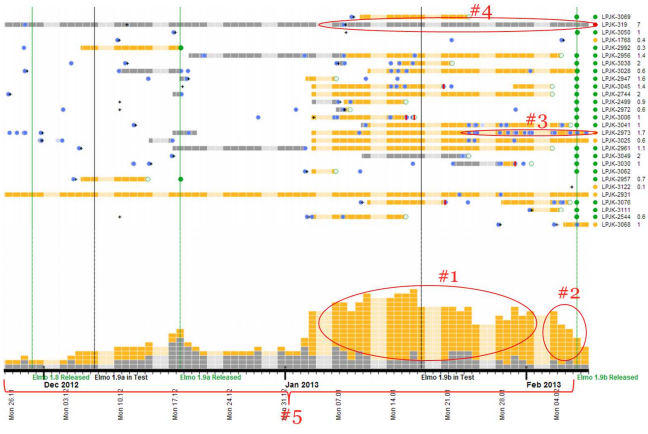

Figure 4. Tasks of one sprint. For a higher quality image, see http://www.cs.tut.fi/~tile/agile-visualization/

These observations could be pointed out in Fig. 4:
#1: Lot of tasks are simultaneously in testing phase, but testing does not seem to proceed during next four or five weeks. #2: Testing seems to proceed very quickly before the deadline in the end of the sprint. #3: A task has a lot of commenting going on in the testing phase. This might indicate a problem in testing. #4: Implementation of a task takes the whole sprint. Perhaps it should be split into smaller parts. #5: Sprint length should be four weeks, but it seems to be over 10 calendar weeks (there is one week Christmas vacation in the timeline of the diagram).

For example, observation #3 was not new to the project, but now, when the fact was visible in the diagram, it was easier to discuss about it and possibly find a solution to fix the problem. In this case, some people in the project decided that they could use a chat tool for communication instead of Jira commenting to make communication more interactive and quicker.

In Fig. 4 it can be seen that there is a period of four weeks when testing does not seem to proceed very much. Lean agile practices suggest one piece continuous flow. Kanban based process often use WIP limits [10] for ensuring the flow. Observation #1 is against these principles, because there are over 10 tasks in testing phase simultaneously. This can be clearly seen from the diagram. In general, observations from the diagram could be used as a basis for SPI. Carrying out the actual SPI was out of this study's scope.

The visualization presented in Fig. 4 contains a lot of information. There are tens of tasks and hundreds of events shown in a compact form. Card et al. [3] has many examples on figures that contain a lot of data in an understandable visual form. Like those diagrams, these visualizations offer a good basis for discussion between the project members.

IX. VALIDATION

A. *Statistical analysis*

Does the proposed visualization reveal the problems of the project? To validate that the phenomena perceived from the visualizations are relevant, a statistical analyst who was not involved in the action research team investigated numerical data related to the project. The data contained information about all tasks worked on in the project during one year (6/2012 - 5/2013). Altogether, there were 10 sprints and 461 tasks of which 137 had failed and 226 succeed. Failing in this context means that the lead time of a task exceeded the limit of 30 working days. The limit for failing was chosen to match the actual average sprint length of the 10 sprints examined. Lead time was defined as the time from state *In Progress* to *Ready for Production*.

On the left of Table I, we present descriptive values of seven variables (lead time, time spent, etc.) that are related to the proposed visualizations. These variables were identified as important factors in customer interviews and thus were selected for statistical analysis. The amount of tasks varies between the rows of the table because all the variables were not available for all tasks. For instance, some of the tasks were still unfinished and therefore we do not have the lead time and the time estimate was not conducted for all the tasks.

The purpose of the statistical analysis was to chart which characteristics of a task predict failing and compare if they

Table 1 - Statistical analysis of data that were used in the visualization.

	All Tasks			Succeeded			Failed		
	N	Mean	Std. Deviation	N	Mean	Std. Deviation	N	Mean	Std. Deviation
Lead time	363	35.43	41.13						
Time spent	381	1.04	1.27	211	0.85	0.88	130	1.38	1.67
Original estimate	376	1.16	1.13	174	0.98	0.85	113	1.36	1.57
Amount of comments	461	8.25	8.25	226	7.02	6.08	137	11.96	11.08
Commenting persons	461	3.30	1.90	226	3.10	1.71	137	4.08	1.94
Amount of assigns	461	5.71	4.67	226	5.10	3.74	137	7.91	5.72
Assigned persons	461	3.28	1.64	226	3.19	1.46	137	4.03	1.68

are visible in the visualization. If these characteristics are shown in the visualization it can help in paying attention to the right tasks already during the sprint before the task has failed and cannot be delivered.

Independent samples T-test was used to analyze the difference between the tasks that failed and tasks that were accomplished. On the right of Table I, we present means and standard deviations for the succeeded and failed tasks separately. For all the variables, the differences in the means between the failed and succeeded tasks are statistically significant ($p<0.01$). The differences between the means in lead times are not presented because failure of a task is defined by its lead time.

According to the statistical analysis, the tasks that failed in the first sprint were bigger (both in the actual size and in the estimate), there were more comments and assigns in these tasks, and there were more people involved in commenting and working than in the tasks that had been accomplished. In [14] Rising et al. suggest that task size should be less than five days to promote success. Our results are in line with this conclusion. The number of comments in a Jira issue indicates that the issue management system is used in communication instead of just documenting the tasks. Cockburn [4] claims that the communication efficiency is poor in using textual message exchange instead of face-to-face communication. Thus, this is a pain point in project and should be dealt with. Furthermore, this is an indication that the specifications are not enabling [21] for the team. However, there exist additional metrics of successful project management, but in this context they were not considered as important ones.

Analyzing the proposed visualizations from this point of view, we can say that it is relevant that the visualization shows the comments marking them with blue. This will help paying attention to the tasks getting multiple comments. When a task starts getting comments, the size of the task can be checked from the visualization. If the task is big the multiple comments are particularly critical. A shortcoming of the visualization is that the amount of different people involved in a task does not show. However, adding too much information in the visualization might make it disorganized. This is an aspect that should be discussed and considered in the further development of the visualization.

B. *Another project context*

To validate the generalizability of the visualization it was tested in the other project. The goal was to ensure that the crawler can be used in a different project context and visualization could provide value for the SPI in another project, too. The other project was a large public sector project at Solita plc. The project was also in the maintenance phase with some development of new features. The visualization for a period of three months is seen in Fig. 5.

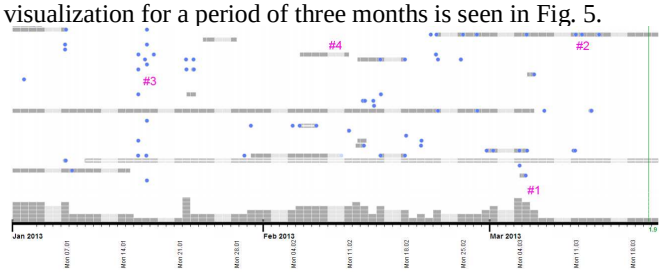

Figure 5. Visualization applied to another project.

The project uses three week sprints, but every sprint's results are not released to the customer. A separate long term release plan contains the release dates. In a release, the results of multiple sprints are delivered to the customer. During the inspection period the team had a release date at the end of March 2013.

The tasks in this project seem to flow fluidly and there are only few tasks *in progress* simultaneously (#1). Testing phase is not visible in the Jira task states at all. Instead, testing is organized separately before the production. Most of the tasks are short and implemented in a couple of days at longest (e.g. #4). Some tasks (#2) are taking a longer time to be implemented. This might be something this project might want to discuss in detail if it is a problem. Commenting seems to be on a low level and the larger groups of comments (#3) are comments made in a project planning meeting. This project seemed to have fewer problems in the development process when interviewed, and the same can be observed from the visualization too. Visualization approach was useful to some extent in this project, too, even though they had a different way to work with Jira than Elmo project had.

X. Related Work

Software process management is about successfully managing the work process associated with developing and maintaining software systems [6]. It consists of four key responsibilities – define, measure, control and improve the process. The visualizations developed in this work were mainly targeted to measuring the process. For example, the sprint length in Elmo project was reported to be four weeks, but the visualizations show that there is some activity in the issues during a ten week period. Also lead time as defined in [9] is easy to measure with visual perception – it can be pointed out that tasks in Fig. 4 have a very long lead time. Statistical fact supports this perception as the mathematical mean is 35 working days, more than the 4 weeks the project aimed at.

Graphical inventions of all sorts serve two distinct purposes [3]. Their first purpose is communication (a picture is worth of ten thousand words) and the second is to discover the idea by yourself. In both cases, visual representations produce cognitive amplification by, for example, extending a person's working memory. We might imagine a stockbroker,

watching computer displays of financial data, rushing to act on events. Whatever the activity is, mental work and perceptual interactions of the world are likely to be interwoven.

The most important ways in which visualizations can amplify cognition are: increase processing and memory resources, reduce searches, enable pattern detection, and perceptual inference operations. Visualization may pack a huge amount of data into a small space and allow patterns in the data to reveal themselves. They make such inferences easy which are not easy otherwise. [3]. Also Larkin and Simon in their classical study [11] illustrated three basic ways why diagrams helped. By visualizing the data, large amounts of search can be avoided. Using location to group elements reduces the load on working memory. Also, visual representations automatically support a large number of perceptual inferences that are extremely easy for humans [11].

Visual displays provide a high bandwidth channel from the computer to the human [11]. We acquire more information through vision than through all of the other senses combined. Improving cognitive systems often means tightening the loop between a person, computer-based tools, and other persons. [3]

Diagrams are effective in the same way as the written words on this page are effective. We must learn the symbols and conventions of the language, and the better we learn them, the clearer that language will be [3]. The same applies to the visualizations created in this study – the customer company people quickly learned the language of the visualizations.

XI. Conclusions

When a project uses an issue management system, it is possible to visualize the tasks that are involved in the project. It makes possible to analyze the diagrams and point out violations of certain agile practices. Visual perceptions and inference based on them are easy to humans. Visual approach often makes it possible to point out such problems that would be hard to find otherwise.

Visualizations represented in this paper offer a good basis for communication. It is easier to form a common conception of the project state when a visual diagram is used as a basis.

The visualizations created in this research gave the developers a new view on the data. There are still many ways to improve the visualizations further. In addition, a lot of useful data for SPI remain still unanalyzed in Elmo project. For example, it could be useful to combine version control system data with the issue management system data. Furthermore, there are other repositories that could still be combined with the already analyzed data to make more useful visualizations. Future research could investigate if these visualizations are useful in other software development methodologies than Scrum. Additionally, automatic detection of bad practices basing on the visualized data could be developed to help the project managers.

References

[1] D. Avison, F. Lau, M. D. Myers and P. A. Nielsen, ”Action Research,” Communications of the ACM, 42, issue 1, pp. 94-97, January 1999.

[2] M. Bostock, 2012. [Online]. Available: http://d3js.org.

[3] S. Card, J. D. Mackinlay and B. Shneiderman, Readings in information visualization: using vision to think, Morgan Kaufmann, 1999.

[4] A. Cockburn, ”Characterizing people as non-linear, first-order components in software development,” In International Conference on Software Engineering 2000, 1999.

[5] R. D. Evered and G. I. Susman, ”An Assessment of the Scientific Merits of Action Research,” Administrative science quarterly, pp. 582-603, 1978.

[6] W. A. Florac, R. E. Park and A. D. Carleton, ”Practical Software Measurement: Measuring for Process Management and Improvement,” Carnegie Mellon University, Pitsburgh, 1997.

[7] A. E. Hassan and X. Tao, ”Software intelligence: the future of mining software engineering data.”, ACM, 2010.

[8] ”JIRA Issue Management System”, Atlassian Corp, [Online]. Available: http://www.atlassian.com/software/jira

[9] A. Johnsen and J. Solberg, ”Quantifying the Effect of Using Kanban versus Scrum: A Case Study.”, IEEE Software, Vol. 29, issue 5, pp. 47-53, 2012.

[10] H. Kniberg, Lean from the Trenches: Managing Large-Scale Projects with Kanban, The Pragmatic Bookshelf, 2011.

[11] J. Larkin and S. Herbert, ”Why a diagram is (sometimes) worth ten thousand words.,” Cognitive science 11.1, pp. 65-100, 1987.

[12] T. Lehtonen, ”Agile Visualization,” Tampere University of Technology, 2013. [Online]. Available: http://www.cs.tut.fi/~tile/agile-visualization/

[13] M. Lima, Visual Complexity: Mapping Patterns of Information, Princeton, 2011.

[14] L. Rising, N.S. Janoff, ”The Scrum Software Development Process for Small Teams,” IEEE Software, Vol. 17, nro 4, pp. 26-32, 2000.

[15] ”Robot Framework - A generic test automation framework,” [Online]. Available: http://code.google.com/p/robotframework/

[16] O. Salo and P. Abrahamsson, ”An iterative improvement process for agile software development,” Software Process: Improvement and Practice, osa/vuosik. 12, nro 1, pp. 81-100, 2007.

[17] Scrum Pattern Community, ”Pop the Happy Buble,” 2013. [Online]. Available at: https://sites.google.com/a/scrumplop.org/published-patterns/very-old-patterns/pop-the-happy-bubble

[18] K. Schwaber, ”Scrum Development Process,”Proceedings of the 10th Annual ACM Conference on Object-Oriented Programming Systems. Languages and Applications (OOPSLA), 1995.

[19] K. Schwaber and M. Beedle, Agile Software Development with SCRUM, Prentice Hall, 2002.

[20] K. Schwaber, Agile Project Management With Scrum, O'Reilly Media, Inc., 2004.

[21] K. Schwaber and J. Sutherland, ”The Scrum guide – the definitive guide to Scrum: The rules of the,” 2011. [Online]. Available at: https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf.

[22] ”Solita Oy,” [Online]. Available: http://www.solita.fi

[23] SVNStat, 2013. [Online]. Available: http://svnstat.sourceforge.net/

Publication V

A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen, and K. Systä. Mashing Up Software Issue Management, Development, and Usage Data. In *Proceedings of the 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE)*, 2015.

Mashing Up Software Issue Management, Development, and Usage Data

Anna-Liisa Mattila*, Timo Lehtonen†, Henri Terho*, Tommi Mikkonen* and Kari Systä*
* Tampere University of Technology, Korkeakoulunkatu 10, FI-33720 Tampere, Finland
Email: {anna-liisa.mattila, henri.terho, tommi.mikkonen, kari.systa}@tut.fi
† Solita Plc., Åkerlundinkatu 11, FI-33100 Tampere, Finland
Email: timo.lehtonen@solita.fi

***Abstract*—Modern software development approaches rely extensively on tools. Motivated by practices such as continuous integration, deployment and delivery, these tools are used in a fashion where data are automatically accumulated in different databases as a side-effect of everyday development activities. In this paper we introduce an approach for software engineering data visualization as a mashup that combines data from issue management, software development and production use.**

The visualization can show to all stake holders how well continuous delivery is realized in the project. The visualization clearly shows the time spent to specify and develop the features as well the length of the delivery cycle. Further more the visualization shows how much work is unfinished and waiting for delivery. This can help the development team to decrease the amount of unfinished work and by that help them to keep up in continuous delivery mind set. In addition to development data usage of the features is also visualized.

***Index Terms*—Information Visualization, Software Analytics, Continuous Delivery**

I. Introduction

Modern software development approaches rely extensively on tools. Due to practices such as continuous integration, deployment and delivery, these tools are used in a fashion where data is automatically accumulated in different databases as a side-effect of everyday development activities. Browsing and visualizing the content of these repositories then gives an idea what has taken place in a project.

Motivated by the web mashups that have become increasingly popular we propose combining software engineering data from various sources and origins into an integrated experience. Techniques to accomplish this are similar to those commonly used in mashups – cleansing, unification, and visualization.

In this paper, we demonstrate how to mash software engineering data up from various data sources into an easy-to-read and easy-to-interpret visualization. For the visualization we have collected data from three different sources. These sources include management data from the issue management system, development data from the version control system, and actual end-user usage data from the monitoring platform.

The results are based on data collected from a single project at a mid-sized Finnish software company Solita (http://www.solita.fi), a service provider that wished to obtain and efficiently react to customer behavior and feedback. The data are from project lupapiste.fi, a service which can be used by citizens and companies to apply for permissions related to the built environment in Finland.

The rest of this paper is structured as follows. In Section II, we address background and motivation of the study. In Section III, we introduce our data model and data collection method, and in Section IV we demonstrate our visualization approach. Finally, in Section V, we draw some final conclusions.

II. Background and Motivation

Software engineering data can be almost anything from version control logs to software usage data. In many software projects software engineering data is collected automatically by tools used e.g. for project management, software development, and software usage analysis [1].

The data sources for management, development and usage are however separated, and even if the tools provide visualizations and analysis from the data, only the perspective of one data source is usually covered. Combining data from the different sources and analyzing the combined data set can give better overall picture from the project. However, such combination of data is non-trivial to create. The tools store data using different data models and formats which makes combining data difficult. Cleansing and homogenization of the data is thus needed in order to combine the data for analysis.

Various methods, including statistical analysis and visualizations, are available to analyze the software engineering data. In general, information visualization is a powerful tool not just for presenting results of statistical analysis but also for exploratory purposes. Good visualization can present large amounts of data in a relatively small space and pinpoint insights of what to analyze further [2].

In our study data from issue management system Jira[1], version control system Mercurial[2] and software monitoring platform Splunk[3] is combined and visualized to study development of a software feature from specification to use. The motivation of the study is to explore how continuous deployment is realized in the project lupapiste.fi.

We have chosen to use a timeline based visualization method as we have used a similar visualization successfully in our previous work to explore realization of sprints in terms of

[1]Issue management system – https://www.atlassian.com/software/jira
[2]Distributed version control system – http://mercurial.selenic.com/
[3]Software monitoring platform – http://www.splunk.com/

features completed [3]. The timeline is also a natural choice when the attribute we are interested in is time.

III. Data Collection

As already mentioned, the data used in the study was collected from three sources of empirical project data, reflecting different, but inter-related dimensions of a typical software engineering project. The data collected were converted into a homogenized data model and combined. In the following, we cover the aspects of the homogenized data model and the actual data collection process.

A. Data Model

When data is collected from various sources, a data model that unifies the data entries from different sources and gives the data a meaning in its use context is needed. In the following we describe the homogenized data model for software engineering data we used in the study.

Because we explore development and usage of features in a software, a software feature is a central concept in our data model. Software feature have states that indicate different phases in the life-cycle of the feature. In addition to state changes there are other events like adding of a comment or an attachment to the Jira ticket or usage of the feature by a customer. All events are time-stamped and called *software engineering event* in this paper. The data model and the data sources used are depicted in Figure 1.

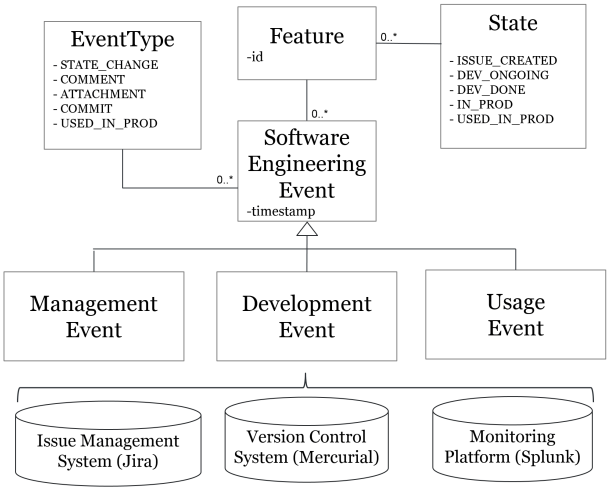

Fig. 1. The homogenized data model. The model presents the concepts of software process and their relations. The model is used to mash the software engineering data up.

A feature consists of a unique identifier that is also used as its name (e.g. `LUPA-1537`) and a number of time stamped software engineering events. The unique feature identifier originates from the issue management system, and it is used as the common denominator throughout its development to track different engineering activities. The state of the feature is initially `ISSUE_CREATED`, followed by `DEV_ONGOING`, `DEV_DONE`, `IN_PROD`, and finally ending at `USED_IN_PROD`.

Each **software engineering event** is either *management event*, *development event* or *usage event*. Details of the different types of events are the following:

- Management events originate from the issue management system, where an issue is created per feature. When an issue is created, the lifetime of a feature begins with state `ISSUE_CREATED`. The issue can then be commented (event type `COMMENT`), or a specification may be uploaded (event type `ATTACHMENT`).
- Development events originate from the version control system. When the development of a feature begins, the state of the feature changes to `DEV_ONGOING`. The development event of type `COMMIT_TO_FEATURE` is produced from version control system commits. When the development is done, the state of the feature changes to `DEV_DONE`. Furthermore, when the feature has been delivered to the production environment – that is, made available to end users – the state is changed to `IN_PROD`.
- Usage events originate from the production logs, which are accessible through the monitoring platform. When a feature is used in the production for the first time, its state changes to `USED_IN_PROD`.

B. Data Collection Process

The data collection process consisted of three steps, where data was collected from the version control system, the issue management system, and the monitoring platform. In this section the three steps of the data collection process is explained in detail.

Combining and collecting the data relies on certain practices the development team have. The team uses HG flow plugin[4] that implements the Driessen branching model [4]. In the model each feature is developed in its own feature branch and merged to a release branch, which is then closed when deployed to the production environment. Moreover, the team has a practice to have one Jira issue per feature and the team uses Jira issue identifier as the prefix for the feature branch name. Without an issue identifier included in the branch name, the data can not be easily combined with the issue management system data. The usage data was merged manually because there were no easy way to deduce automatically to which feature a use event is related to. The R tool[5] was used to convert the data sets into the homogenized data format and combine it to one CSV file.

First in the data collection procedure, data from the distributed version control system (VCS) was collected using custom scripts. The VCS contains the actual time stamps for feature development starting from the beginning of the development (state change to `DEV_ONGOING`), continuing with the moment when development was done (`DEV_DONE`). Release branches are used for deploying the software to the

[4]HG flow, A Mercurial extension – https://bitbucket.org/yujiewu/hgflow/
[5]The R Project for Statistical Computing – http://www.r-project.org/

production environment, thus closing times of the release branches provide a time stamp for state change to IN_PROD. The identifiers of the features developed are also deduced from the version control data and used in collecting the data from Jira.

As the second step, data from the issue management system Jira was collected. Issue creation time and commenting activity was captured. Also time stamps for uploading files were collected, because file uploads in this case are usually specifications created by user experience designers. The data was collected using a custom web crawler that visited the issue pages and captured the time stamps and attributes of the issue. The crawler visits only the issue pages that match the issue identifiers found from version control system. The crawler is described in detail in [3].

Finally, as the third step, data was collected from the Splunk monitoring platform using customized scripts. Log data from feature usage in production was a list of events of action type USED_IN_PROD, where every call to the back-end API of the application was collected. The API call data was collected only for five features that introduced a new API interface. There was no easy way to automatically combine the API call path to the issue identifier, and thus the data was merged manually.

IV. Data Visualization

The visualization presented in Figure 2 shows a mashup from issue management, version control, and feature use monitoring system data. The visualization includes some tens of features developed in the project lupapiste.fi during a 2,5 month period.

The goal of the visualization is to show how long it takes to specify, develop and deliver a feature as well as the time it takes to customers to find the feature from the product and start using it. The visualization shows also the time spent waiting – waiting for the development to be started after specification is finished, waiting for the delivery after the development is completed and waiting the customers to find the feature. Realization of continuous delivery is made visible also showing the amount of features waiting for delivery. The visualization is developed using D3JS[6] visualization framework.

In the visualization (Figure 2) each feature has its own lane that represents the lifespan of the feature. The states, described in previous, are color coded to the feature life span – light blue color for state ISSUE_CREATED (light gray in gray scale), yellow color for state DEV_ONGOING (darker gray in gray scale), dark gray color for DEV_DONE and pink vertical lines for state USED_IN_PROD. The small black dots in the visualization mark commits, the larger red dots mark file uploads and the dots with blue outline and white filling mark comments. At the bottom of the visualization amounts of features under work and features waiting deployment are shown as a stacked bar chart. The dark gray bars show the amount of features waiting delivery. Yellow bars (lighter gray in gray scale) show amount of features under development. The x-axis is time, which increases form left to right and weekends are marked with lighter color in order to make it easier to count number of weeks. The software events introduced in the data model are annotated to the Figure 2 as follows: management events (*ME*), development events (*DE*), and usage events (*UE*).

From the visualization we can observe the lead times of features as well as the general progress of feature development, specification updates and commit and delivery frequencies. As an example in the middle of the visualization in, spot #ME-1, there is a feature that has a lead time from ISSUE_CREATED to IN_PROD of four weeks. The dot in the beginning of the feature #ME-1 represents a comment in Jira. The red dot on top of text #ME-1 represents a file uploaded to Jira, which in this case is a specification. When the development of the feature starts, a new specification is uploaded. The feature has one day of no commits and then yet another specification uploaded. There is also a one week period of development without commits. Then, in the beginning of the next week, a final commit was pushed to the version control system and the feature was deployed to the production environment two days later in a delivery that is annotated with #DE-DELIVERY-3 in the visualization.

When a lot of features are delivered simultaneously, the height of the gray bars at the bottom of the visualization (#DE-trend) fall down. On the left in spot #DE-problem-1, the number of features waiting for delivery, was large. We can see a drop in the height of the gray bars in November showing that the process has been delivering continuously at that time. The drop in the delivery cycle time can be due that the development team was informed about this research and its goals at October.

The delivery in top left corner of the visualization, annotated with #DE-delivery-1, contained a feature that it was possible to collect usage data for. The usage data (the short vertical pink lines) are shown on the feature lane, and it can be observed that the usage ended suddenly at point marked with #UE-1. The reason was that the API service name was changed as a part of code refactoring, and the data were no longer gathered for the feature.

There were two features in delivery #DE-delivery-2, which usage data were also collected for. During week #UE-2 the two features were used approximately equally, but before and after that week, the usage density seems to differ. This kind of rough observations about the usage of the features can be made on basis of the visualization.

The visualization makes four software engineering phenomena visible. First, the realization of continuous delivery can be observed from the bar chart. Second, the lead times of features can be seen from the feature lanes. Third, the timing of specification work can also be observed. Finally, as the visualization contains usage data of the features, the density of usage can be evaluated. The bar chart of the visualization can also help developers to arrange their work as it shows how much work is started but not finished.

[6]Data Driven Documents – http://d3js.org/

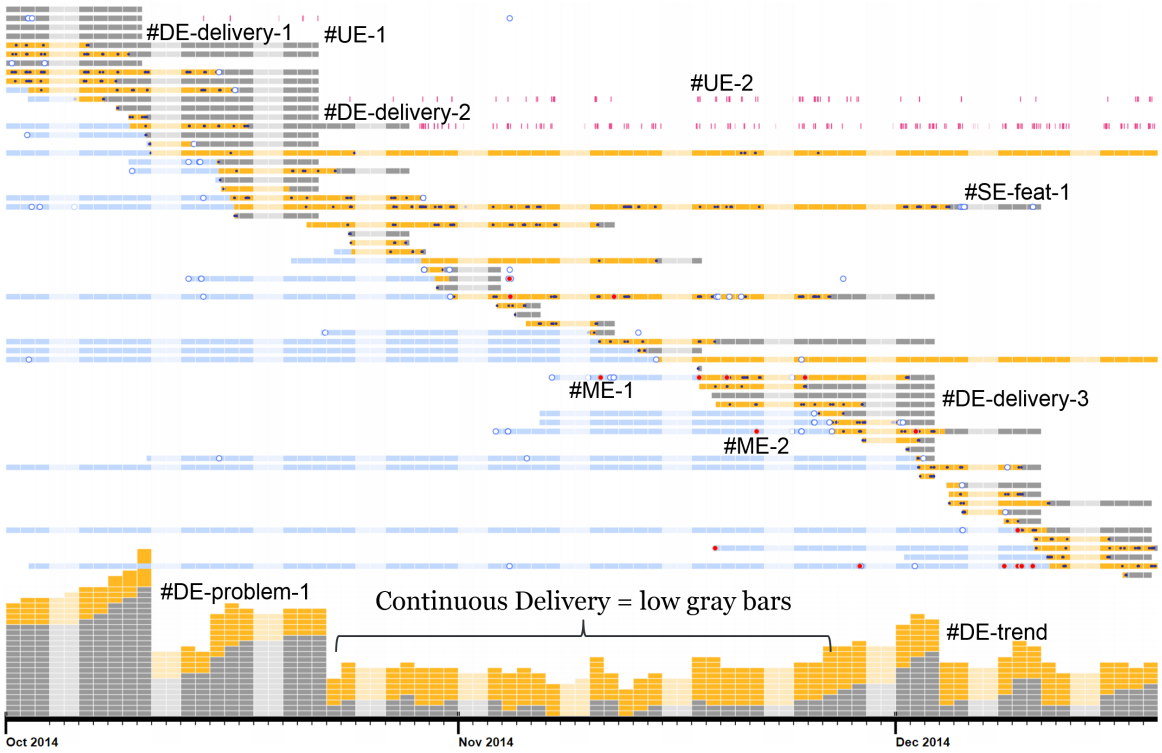

Fig. 2. A mashup visualization of the issue management, development, and usage data. On the upper part of the visualization software feature lifespans, one feature per line, are presented. The states related to a feature are color coded: light blue for ISSUE_CREATED, yellow for DEV_ONGOING and dark gray for DEV_DONE. Other events are presented as different shapes – dots mark commits (black color), comments (red color) and file uploads (light blue outline with white filling color) and vertical lines (pink color) mark usage. At the bottom of the visualization is stacked bar chart that presents amounts of unfinished features (yellow) and amount of finished features waiting for delivery (dark gray). The x-axis is time, which increases form left to right and weekends are marked with lighter color.

V. Conclusion

For this study, we implemented a software engineering data visualization that mashes data up from three different sources of an industrial software project. These sources cover issue management, development and usage data of an Web-based service. We collected, filtered, transformed, and merged data from Jira (features to be developed), Mercurial (feature development and deployment data) and Splunk (production usage logs). The merged data was shown in a single visualization. The visualization and data collection procedures are still work in progress. In future we aim to develop the homogenized data model further to cover more aspects of software process as well as improve the visualization format.

The visualization presented can show to all stake holders how well continuous delivery is realized in the project. The visualization clearly shows the time spent to specify and develop the features as well the length of the delivery cycle. Further more the visualization shows how much work is unfinished and waiting for delivery. This can help the development team to decrease the amount of unfinished work and by that help them to keep up in continuous delivery mind set. In addition to development data usage of the features is also visualized. This information is valuable for analyzing how much time it takes users to find a feature. Moreover some conclusions on feature usefulness can be drawn from the usage data. However, further analysis of the use data is left as future work.

Acknowledgment

This work is a part of the Digile Need for Speed project (http://www.n4s.fi/en/), which is partly funded by the Finnish Funding Agency for Innovation Tekes (http://www.tekes.fi/en/tekes/). In addition, this research has been supported by Foundation of Nokia Corporation.

References

[1] T. Menzies and T. Zimmermann, “Software Analytics: So What?” *Software, IEEE*, vol. 30, no. 4, pp. 31–37, 2013.

[2] E. R. Tufte, *The Visual Display of Quantitative Information*. Graphics press Cheshire, CT, 1983.

[3] T. Lehtonen, V.-P. Eloranta, M. Leppänen, and E. Isohanni, “Visualizations as a Basis for Agile Software Process Improvement,” in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, vol. 1. IEEE, 2013, pp. 495–502.

[4] V. Driessen, “A Successful Git Branching Model,” 2010. [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/

Publication VI

T. Lehtonen, T. Aho, T. Mikkonen, and K. Kuusinen. Visualizations for Software Development Process Management. In *the 26th International Conference on Information Modelling and Knowledge Bases (EJC)*, 2016

Visualizations for Software Development Process Management

Timo LEHTONEN [a], Timo AHO [a], Kati KUUSINEN [b] and Tommi MIKKONEN [b]
[a] *Solita PLC, Åkerlundinkatu 11, FI-33000 Tampere, Finland*
[b] *Department of Pervasive Computing, Tampere University of Technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland*

Abstract. Software development projects have increasingly been adopting new practices, such as continuous delivery and deployment to enable rapid delivery of new features to end users. Tools that are commonly utilized with these practices generate a vast amount of data concerning various development events. Analysis of the data provides a lightweight data driven view on the software process. We present an efficient way of visualizing software process data to provide a good overall view on the features and potential problems of the process. We use the visualization in a case project that has become more agile by applying continuous integration and delivery together with development and infrastructure automation. We compare data visualizations with information gathered from the development team and describe how the evolution can be understood through our visualizations. The case project is a good example of how a change from a traditional long cycle development to a rapid cycle DevOps culture can actually be made in a few years. However, the results show that the team has to focus on the process improvement continuously in order to maintain continuous delivery all the time. As the main contribution, we present a lightweight way to software process visualization. Moreover, we discuss how such a heuristic can be used to track the characteristics of the target process.

Keywords. software visualization, continuous delivery, DevOps,

1. Introduction

Implementing a modern development tool chain calls for several technical enablers, such as continuous integration [1] and smooth deployment [2]. In general, aim at the reduction of the time it takes from completed implementation to deployment has resulted in DevOps [3] where developers and operators work as a team to deliver value to end users with an intensive feedback loop.

In such a setting, where tools are constantly playing a major role in the fashion the development advances, developers' actions are reflected as recurring patterns like version history commit, deployments and issue management events. These patterns form traces to information systems. This software engineering data can be processed and mined in the databases.

In this paper, we investigate such traces in the light of the evolution of the development process. The paper is built on mined data from the issue management system of an industrial project executed by Solita PLC, a Finnish software development and

consulting company that specializes in web software and business intelligence. We have analyzed thousands of issue management system tasks in detail with a visual approach to describe the actual changes in the development process. The project in question is a public sector web-based data intensive software project that uses a set of typical software development tools that have automatically generated data for analysis during the years.

In this study, we apply information visualization to demonstrate the evolution of the software development process during a five year long period. During the time frame both development tools and practices have evolved. In the beginning, the process could be described with manual long-lasting implementation periods and the Scrum culture approach. After five years, the approach has transformed to rapid cycles and automated mechanisms for infrastructure with monitoring and quality assurance as an integral part of daily development work. As a tool for analysis, we use visualizations of the data stored in the issue management system. Furthermore, we collected the opinions regarding the visualizations from the developers and the project manager of the case project. Our exact research question can be formulated as:

RQ: How to demonstrate a software development process by using automatically generated data?

The rest of this paper is structured as follows. In Section 2 we present the relevant related work. We continue in Section 3 by going through the case study regarding the transition between the different development models. Section 4 analyzes the result and in Section 5 we discuss the results in more detail and finally, Section 6 draws some concluding remarks.

2. Background

During the recent years, numerous software companies have invested considerable effort in building and automating their development tool chain often referred to as ”Climbing the Stairway to Heaven” [4]. The evolution of organizations for adopting continuous integration, continuous delivery and even continuous deployment [5] is often a step by step procedure [4]. Continuous integration is a requirement for continuous delivery, which in turn is a requirement for continuous deployment [6]. These strategies can then be applied in transformation towards DevOps [9] and reliable, predictable release engineering [10].

This extensive infrastructure, needed to maximize development and deployment speed as well as feedback collection mechanisms, commonly includes a version control system, a build server, a test server, automated production installations, and number of other tools to support development and management. These components form a *deployment pipeline* [5], which uses an automated set of tools from code to delivery. Feature-driven development [7] is one approach for designing and delivering valuable changes to a software. The development team often manages the features to be implemented in an issue management system, for instance Jira[1]. The issue management data can then be mined, for instance for software process improvement (SPI) [8] purposes.

Various methods are available for analyzing the software engineering data produced by the tools. For instance, machine learning algorithms could predict forthcoming software engineering events. In general, information visualization is a powerful method not

[1] `https://www.atlassian.com/software/jira`

just for presenting results of statistical analysis but also for exploratory purposes. In particular, good visualization can present large amounts of data in a relatively small space and pinpoint insights of what to analyze further [11]. Visualizations amplify the capabilities of the human brain [12] as they increase processing resources, reduce searches, enable pattern detection, and perceptual inference operations. Moreover, visualizations can expand the working memory used for problem solving [13].

In the literature, there are two major disciplines of visualization [15]. *Scientific visualization* refers to processing of physical data while information visualization refers to processing of abstract data. However, the distinction between scientific visualization and information visualization is not clear [15]. Moreover, software visualization is a term for applying information visualization to the domain of software engineering [14]. Diehl et al. [15] present the goal of software visualization as improving the productivity of the software development process. They define software visualization as the visualization of artifacts related to software and its development process. This covers a wide variety of artifacts from program code and documentation to bug reporting and visualizing the structure and behavior of the software. Software evolves over time through program code changes to extend the functionality of the system or simply to remove bugs [15]. In a narrower meaning, software visualization is often used interchangeably with *program visualization* which means the visualization of the software as an executable program [16]. In this sense, software visualization is related to visualization of computer programs. Moreover, according to Petre et. al [17], software visualization uses visual representations to make software visible.

There are multiple examples of applying information visualization to data concerning software development. Chuah et. al [18] use glyphs for viewing software project management data. They applied a visual approach to highlight interesting patterns and anomalies in the data set. Gall et. al [19] apply information visualization to study the release history of a software system. They conclude that information visualization technologies can be effectively applied to the analysis of software evolution and to uncover valuable information. Ohira et. al [20] collected data for software process improvement from configuration management systems, mailing list managers and issue tracking systems and presented the data visually. They mention that real-time visualizations motivated developers to fix bugs, since they were aware that there were still unresolved issues. As a problem, they report that visualizations can be too complicated to understand. In [21], the authors mine version control system data and examine how developers work together. With the visualizations they are able to find interesting phases during the evolution.

To our knowledge, there is a research gap in applying information visualization to software engineering data. We have already studied the relationship of issue management and other software engineering data in an earlier paper [22]. We developed a mash-up of information from multiple sources including issue management system, version control system data and monitoring platform. We applied visualizations to analyze the detailed development process based on the development data.

3. Case Project

This research is based on mining software process data from an industrial project executed by Solita PLC[2], a Finnish software development and consulting company. The time frame we cover is five years during which both development tools and practices have evolved. The evolution has started with long-lasting implementation periods and manual deployment and moved to rapid cycles and automated mechanisms for infrastructure, monitoring, and quality assurance. The data we study is produced by a public sector web-based data intensive software project that uses an issue management system to manage the process. The teams uses the issue management system very intensively in their daily work. The tool is used in in daily meetings and in communication with the customer.

Over the years, there have been several changes in the infrastructure, practices, and operations related to the system as described in Table 1.

Table 1. Major changes in the case project

2011	Dev / Ops	CI-server (Hudson) was taken to use
2012	Dev	Scrambled production database dump (nightly dump automatically available)
2013	Dev	Build tool evolution (from Ant to Maven)
	Dev	Automatic database migration
2014	Dev	Environment independent build (Jan)
	Ops	Scripted production deployment (Jan)
	Ops	Server configuration automation (Ansible + Vagrant) (Nov)
2015	Dev	Automatic database cloning
	Ops	Application sets to be installed declared in a text file
	Ops	Automatic deployment to customer acceptance testing environment triggered by commits
	Ops	Interfaces for monitoring, smoke-testing, and radiator

The changes in Table 1 are divided into categories ”Dev” and ”Ops”. If the changes is related to development, the category is ”Dev”. For example, scrambled database dump from the production in 2012 was a change that boosted development. Moreover, automatic deployment to customer acceptance testing environment in year 2015 was a change related to operations part of DevOps.

The team described that their development process consists of two parts. First, they use a major project-based development cycle, where development is divided into projects of length of 1-2 months each yielding a release. Second, the team uses a continuous minor development cycle that consists of smaller releases. The goal of minor development is to deliver smaller development items continuously to the production environment in short cycles. The goal is not to fix bugs, but if there are any, they are fixed and deployed with a short cycle.

[2] `http://www.solita.fi`

4. Results

We applied an interactive visualization tool [24,25] to a case project where a transformation from older software development methods towards a novel short cycle DevOps-culture has happened during years. We developed the visualization tool further to contain metadata based rules that enable the creation of reference process shapes which can be compared with visual shapes generated from actual software project data. Next, we use the tool to generate various views of the target process. We start by introducing the tool output with an example.

4.1. Sample Issues View

Figure 1 presents issue states and three sample issues extracted from the issue management system data. The initial state for an issue is *Open*.

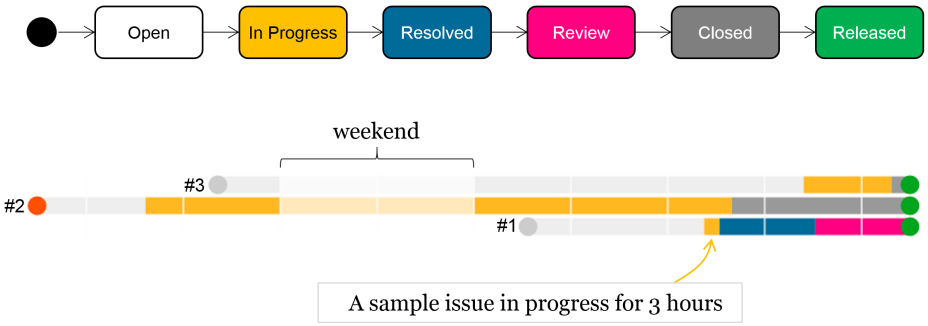

Figure 1. Issue management system states and three sample issues to demonstrate the visualization rules.

In Figure 1, the open state is presented with a dot in the beginning of the issue time line on the left. The color of the dot is gray for issues with default priority and red color indicates higher priority. For instance, issue #2 in Figure 1 has a higher priority.

Issue #2 was created to the issue management system first. The development of the critical issue was started approximately one day after its creation. The developer changes the state to *In Progress*, which is presented with yellow color. Then, apparently nothing happened during the weekend, and finally, issue was done or *Resolved* after six days of development. Then, in a few seconds, the issue was put to *Review* state and then immediately *Closed*. Thus, no blue or pink color is visible for issue #2 . Issue #1 was put to *Resolved* state (blue color) after three hours of development work. Then the issue went to state *Review* (pink color), which means acceptance testing performed by the customer.

The issues are ordered from bottom to top according to the time stamp of *Resolved* state. Thus, issue #1 is on the bottom, because it was resolved first. Then, issue #2 is in the middle because and issue #3last, because it was the latest task that had *In Progress* activity i.e. was resolved last. Next, we construct a reference process shape according to these drawing and ordering rules to Figure 2 and then apply the visualization technique to some tens of released issues presented in Figure 3.

4.2. *Version Release View*

Figure 2 presents the larger major release cycle and minor release cycles below it. The reference process in this case has one major release and two minor releases during a few weeks period.

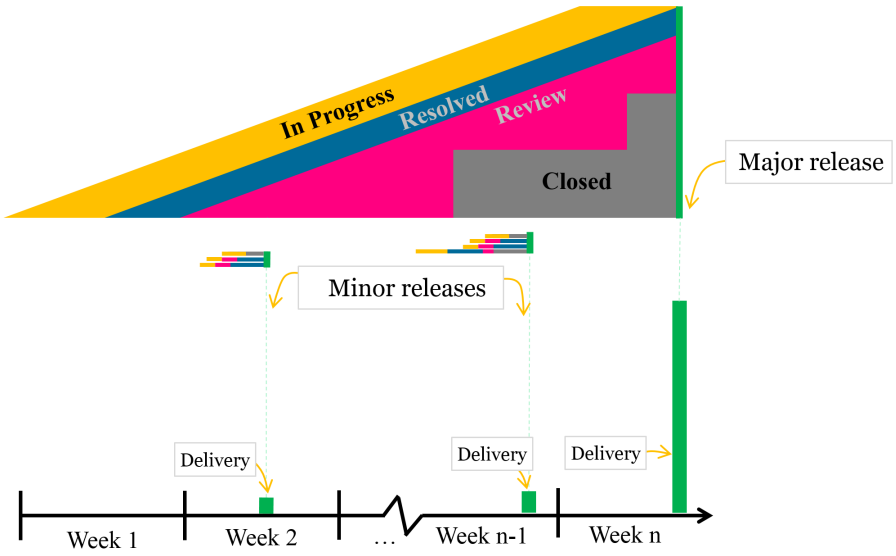

Figure 2. A reference process visualization that reflects issue management system states and the verbal description presented by the team.

Because of the ordering rules described in section 4.1, the reference process forms a triangular shape. The release date of the major release forms a sharp edge to the right. The released issues are drawn to the bottom of the diagram as a green bar which indicates the amount of issues delivered. In the reference process diagram there are three versions delivered – one major version and two minor versions. States *In Progress* and *Resolved* are in the reference visualization drawn with equal length but in practice, their length varies from seconds to weeks. Because the customer reviews many tasks at once, the *Review* states form a shape of a stairway. The number of tasks in one release varies significantly. The team stated in discussions that a total number some tens or hundred tasks would be suitable for their needs.

The visualization in Figure 3 is an actual version released in 2014. The visualization can be compared to the triangular reference process shape in Figure 2.

We can now point out some spots in the visualization that follow the reference visualization characteristics. The version contained over 50 issues that were released in the end of October in one major release annotated in the figure with label `#major`. In spot `#first`, the implementation of the first task belonging to the released version was finished and the state then changed to *Resolved* (blue color). Approximately three weeks later (the weekends are highlighted with translucent white color) the task then went to review state (pink color) in the middle of October. The issue was released in the major version (`#major`) among tens of other features. After the weekend, fix versions `#fix1`,

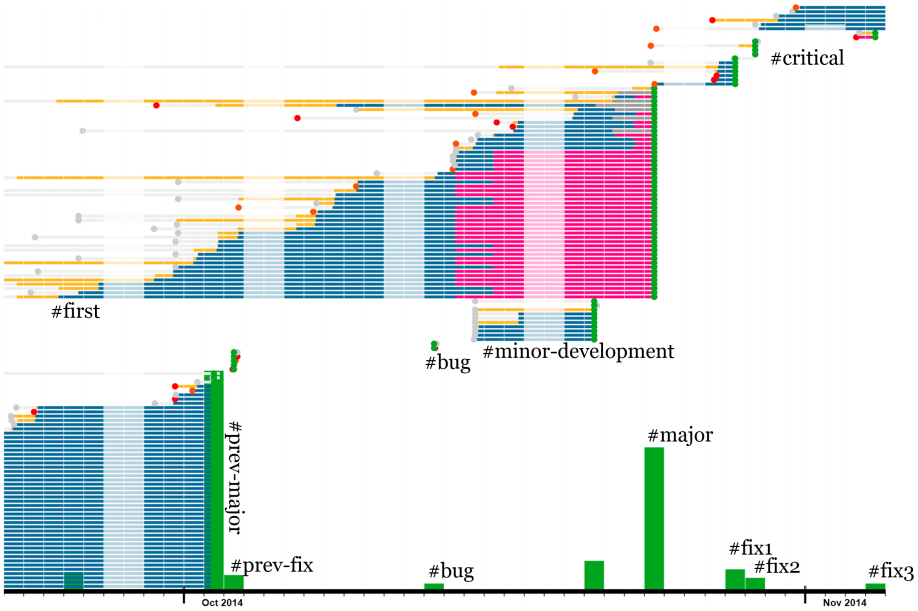

Figure 3. Sample major version released that was released in the end of October 2014. Two fix versions and a smaller minor development release are shown.

`#fix2` and `#fix3` were released. Some of the tasks in `#fix1` was a critical bug pointed out with a red spot (`#critical`). In the mean time, there was also a parallel minor version released in spot `#minor` on the bottom of the visualization. The minor version contained approximately ten tasks. Moreover, in spot `#bug` there was a critical bug fix deployed with a lead time of some hours.

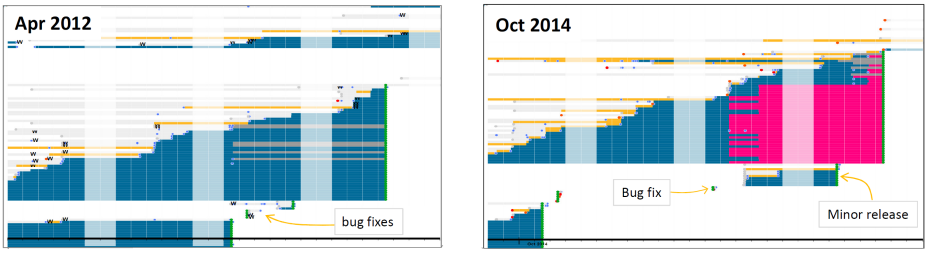

Figure 4. The evolution of the process – in the past there was no minor parallel development, but nowadays the team uses rapid cycle minor development.

Figure 4 presents an example of a change in the process. The sample release visualization from Oct 2014 on the right shows that there is a minor release in parallel to the major release. There was also a critical bug fix in the middle of October in 2014. When we compare this visualization to the other visualization from Apr 2012 on the left, we can point out three differences. Firstly, the earlier release visualization is missing a minor release. However, there are bug fixes visible. Secondly, in the earlier visualization, work estimates were made per issue, which is shown with character `'W'` in the visualization.

Finally, there is no *Review* state (pink color) in the earlier release. The team took *Review* state to use in September 2012. According to these simple visual observations, it is known that the process has evolved during years. The team has found out a way to fulfill the urgent needs of the customers with a short feedback cycle. Moreover, the focus of the team has changed from work estimating to value adding activities.

4.3. Version Release Series View

Figure 5 presents the reference process shapes of a series of versions. The development work of the previous version continues immediately as development work in the next version. Continuous delivery is visible at the bottom of the diagram.

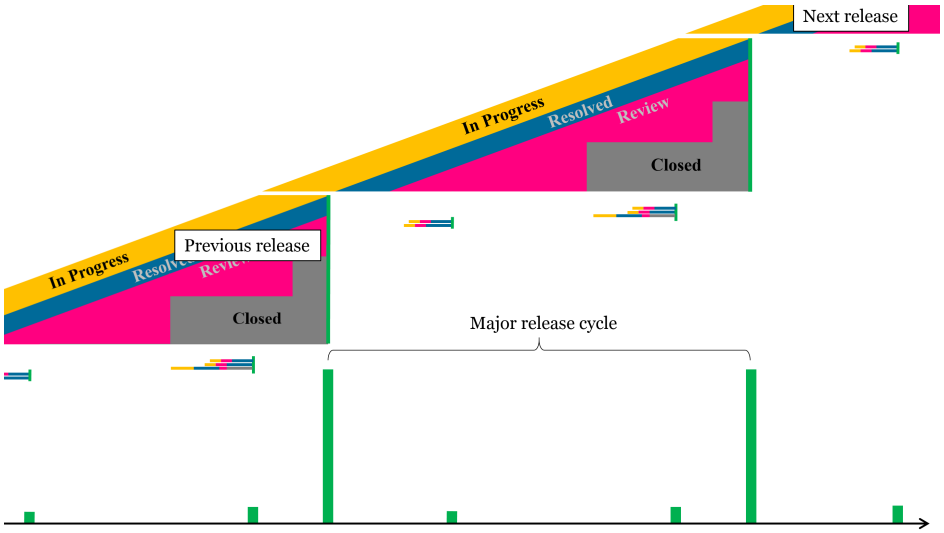

Figure 5. A series of reference process visualizations with certain major release cycle and continuous delivery of parallel minor versions.

Figure 6 presents a wider perspective to the version presented in Figure 3. It is possible to observe that there are multiple simultaneous development tasks going on in the mean time in the upper part of the diagram. Furthermore, the number of delivered features per day are presented in the bottom of the diagram. In spot `#empty` in February 2015, there is a break in the continuous delivery of features apparently because of the giant version release on the top of the visualization annotated with label `#major`.

4.4. Evolution View

To get a total view of the process evolution, we combined an infographic presenting over 5 000 issues from years 2011 to 2015 into Figure 7. There is a reference process in the bottom left corner of the infographic which gives a hint of what the layout of the actual process shapes should be. In this case, the reference process consists of 150 issues released per version with major release cycle of two months, which accompanies the verbal description in section 3. The combined bar chart presenting the throughput on the top of the infographic presents the number of issues released per year half. For instance, dur-

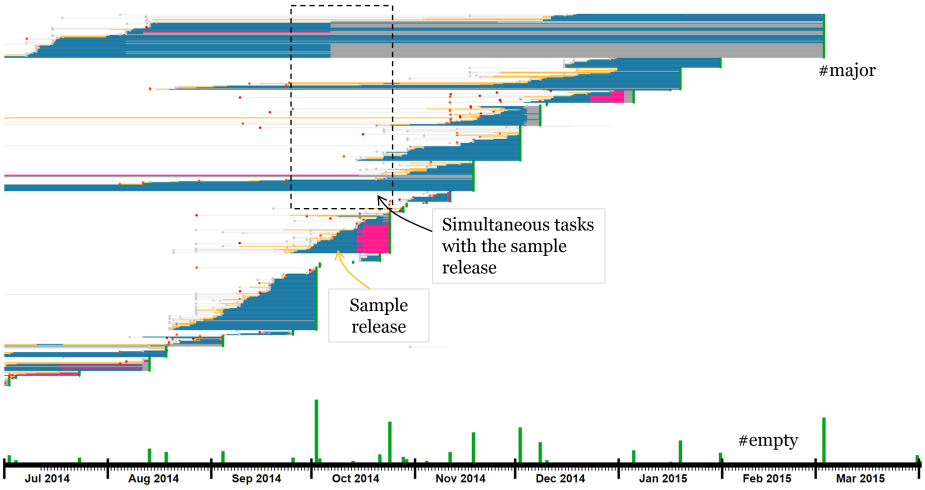

Figure 6. A wider perspective to the version released in figure 3

ing the first year half of 2012, a total number of approximately 500 issues was released. Some of the issues released in the beginning of the year 2012 were developed during the end of year 2011.

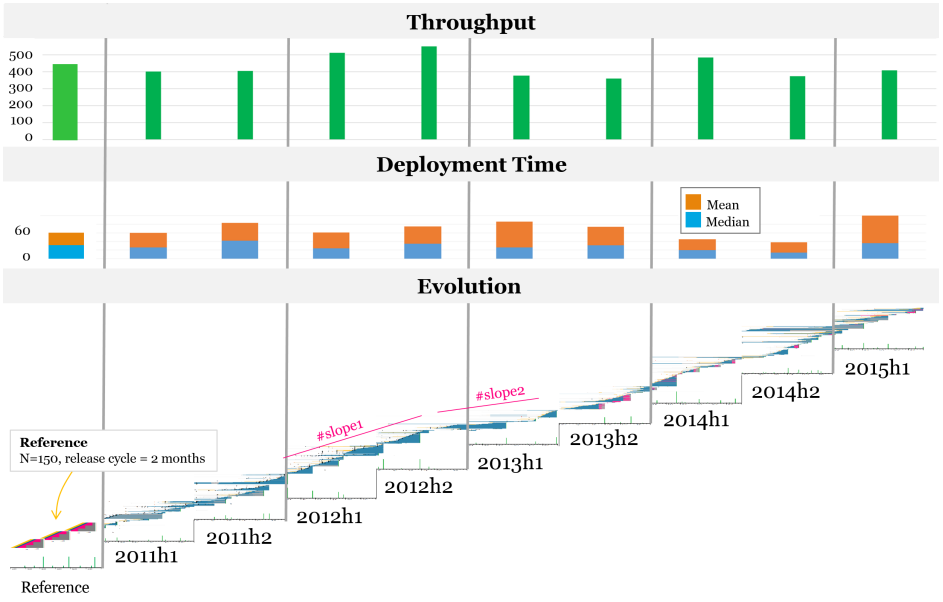

Figure 7. An infographic of the evolution of the software development process during a five year period.

Annotation `#slope1` presents the slope factor reflecting the high throughput during year 2012, which was higher than the slope factor `#slope2` in the beginning of year 2013. The throughput of the team was higher in 2012 than it was in 2013. Naturally, the throughput is affected by, for instance, the size of the tasks – if features are split to smaller

tasks than before, the throughput increases. Moreover, *deployment time* or the lead time from *Resolved* to *Released* is shown in the middle of the infographic. *Deployment time* answers to the following question: when a feature was done, how long did it take until the feature was deployed to the production environment? When we observe the throughput bar graph and the deployment time bar graph visually, the first half of year 2014 seems to have low mean and median deployment time with a high throughput. If we then compare the first half of year 2014 visually to for example the first year half of 2015, the difference is clear. Year first year half of 2015 contains issues with very long tails, of even half year long. This leads to longer deployment times.

5. Discussion

In this section we reflect the results to our research question: *How to demonstrate a software development process by using automatically generated data?*. We start by discussing the results and then reflect them to the opinions collected from the project manager and the developers.

Tools play a major role in novel software development work. The data set of traces the tool usage produces creates a great possibility to evaluate the evolution of the development process. In this paper we combined tens of thousands of events related to issue management system tasks to a compact visual format. From the visualizations, we are able to recognize changes in the development process. For instance, the change towards a development process with separate major and minor cycles can be recognized. Moreover, the continuous delivery phenomenon is made visible and can thus be evaluated. We pointed out problems in continuous delivery with the new visual representation which enables pattern recognition and quick inferences of the process. By combining simple statistics concerning throughput and lead times into a single infographic, we are able to evaluate the evolution of the process.

The visualizations produced by the tool presented in this paper forms a basis for software process evaluation. However, data quality problems related to software engineering data collected are many. Keim et al. [14] list the following error sources as threats to visual data analysis: noise, outliers, low precision, missing values, coverage errors and clones. Problems in raw data quality can jeopardize the conducted visualizations. Moreover, Rosli et al. in their mapping study [26] recognize several typical flaws for data quality in software engineering research. In the context of this research, a typical source for inaccurate data is the everyday usage of the issue management system. For instance, a developer can forget to update the issue management system task state when the actual development work starts or ends. This leads to inaccurate time stamps which affect the visualization. However, we assume that the data is adequately accurate for visual analysis. The data describes actual real-world events that were performed by persons participating to the development of the software. In this sense, the data consists of facts and thus the visualizations describe real-world events that actually happened.

The visualizations reveal interesting facts about the software development process. In Figure 3 it is noteworthy that there are three fix releases after the major release. The reason for them is unknown, but one obvious explanation is that targeting to zero bugs is expensive. The team deploys features actively to the production environment with a short feedback cycle and lets the end-users partly report of the bugs. Naturally, critical

and serious bugs have to be avoided. For instance, bugs in a global marketing system related to billing functions may have expensive consequences and thus have to be avoided. However, bugs related to non-critical sections in a standard public sector software are not life critical and thus partly acceptable.

We collected the opinions regarding the visualizations from the development team in an informal manner. The project manager mentioned that the visualizations make it possible to get an overview of the project with a glance. Especially the comparison of different versions or projects is made possible. According to the project manager, such a comparison would not be possible otherwise. As an improvement, the project manager mentioned that textual labels to the versions would make the visualization easier to read.

The developers of the case project mentioned several points that the visualizations present effectively. Firstly, the visualizations reveal low quality versions by presenting the number of fix versions needed after the release. Secondly, tasks with exceptionally long lead times are revealed. According to the developers, exceptionally long lead times are always a signal of a problem in the development process. Thirdly, the visualization reveals the amount of continuous bug fixing needed. The bug fixes are visible beneath the versions released. Finally, the visualizations indicate if smaller versions are released continuously or not. According to a developer, this reveals if value is produced to the customer continuously or not.

Moreover, the visualizations were also criticized by the developers. They mentioned that errors in the visualizations are many. For instance, some work may not be entered to the issue management system and is thus not visible. Furthermore, one of the developers mentioned unknown correlations as a problem. The interpretations made based on the visualizations may not reflect what actually happened in the real world. Use of other analysis methods to explain the events is needed. As a future work, one of the developers was interested of the impact of using visualizations as a method for reflection in the organization.

As future work, the visualizations techniques could be developed further. The tools developed should be applied to more than one project context in order to evaluate generalizability of the results. Moreover, an analytic pipeline which could demonstrate the status of the project in a continuous manner, could produce valuable information to the project stakeholders. The data collection methods described in this paper can be automated. A tool chain covering all steps from the initial data collection to the visualization of the data can be implemented.

6. Conclusions

In this paper we presented a novel visualization approach for illustrating software engineering projects based on issue tracking tool data. It is important to note that this kind of data is usually generated automatically as a side effect of the project when the tools are used. Data is usually readily available and does not need any extra activities to be used as a basis for visualizations.

Visualization is a light-weight way to get a good view on the overall development process. In addition, it can be used to understand the process more deeply by showing what kind of sprint lengths and common deployment times actually exist, for instance. On the other hand, also anomalies like uncommonly long delivery times for some features are easily noticeable.

We used the visualization to analyze a case software project. In this, we demonstrate how evolution towards a more agile process can both be validated and the effects recognized. The general lead time and feedback cycle has significantly reduced and amount of waste in process diminished. The developed interactive visualization tool can be applied in different scenarios and with different levels of abstraction.

As future work, we are interested in using this visualization tool for multiple projects in different kinds of environments. This way we could visually recognize differences in the patterns of software processes and ask if they actually exist. It would be very intriguing to find some kind of general fingerprint for a healthy process and see how actual process visualizations differ from it.

Acknowledgments

The work was supported by Tekes DIGILE Need for Speed project. We would also like to thank Solita, the case project, and Finnish Broadcasting Company for support and the possibility to perform this research.

References

[1] M. Fowler, "Continuous integration," Available at http://www.martinfowler.com/articles/continuousIntegration.html, 2006, accessed 27.11.2015.

[2] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[3] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[4] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, pp. 392–399.

[5] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[6] M. Fowler, "Continuous integration," http://martinfowler.com/bliki/ContinuousDelivery.html, retrieved: November 2014.

[7] S. R. Palmer and M. Felsing. *A practical guide to feature-driven development*. Pearson Education, 2001.

[8] W. A. Florac and A. D. Carleton. *Measuring the software process: statistical process control for software process improvement*. Addison-Wesley Professional, 1999.

[9] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[10] A. Dyck, R. Penners, and H. Lichter, "Towards definitions for release engineering and devops," in *Proceedings of the Third International Workshop on Release Engineering*. IEEE Press, 2015, pp. 3–3.

[11] E. R. Tufte and P. Graves-Morris, *The visual display of quantitative information*. Graphics press Cheshire, CT, 1983, vol. 2, no. 9.

[12] S. K. Card, J. D. Mackinlay, and B. Shneiderman, *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[13] D. A. Norman. *Things that make us smart: Defending human attributes in the age of the machine*. Basic Books, 1993.

[14] D. Keim, F. Mansmann, J. Schneidewind, H. Ziegler *et al.*, "Challenges in visual data analysis," in *Information Visualization, 2006. IV 2006. Tenth International Conference on*. IEEE, 2006, pp. 9–16.

[15] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[16] J. Stasko. *Software visualization: Programming as a multimedia experience*. MIT press, 1998.

[17] M. Petre, E. de Quincey, et al. A gentle overview of software visualisation. *PPIG News Letter*, pages 1–10, 2006.

[18] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *Computer Graphics and Applications, IEEE*, 18(4):24–29, 1998.

[19] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 99–108. IEEE, 1999.

[20] M. Ohira, R. Yokomori, M. Sakai, K.-i. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: A tool for mining multiple project data. In *International Workshop on Mining Software Repositories (MSR2004)*, pages 42–46, 2004.

[21] P. Weißgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 9–9. IEEE, 2007.

[22] A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen, and K. Systä, "Mashing up software issue management, development, and usage data," in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*. IEEE Press, 2015, pp. 26–29.

[23] J. Pearl, "Heuristics: intelligent search strategies for computer problem solving," 1984.

[24] T. Lehtonen, V.-P. Eloranta, M. Leppanen, and E. Isohanni, "Visualizations as a basis for agile software process improvement," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, vol. 1. IEEE, 2013, pp. 495–502.

[25] A.-L. Mattila, T. Lehtonen, K. Systä, H. Terho, and T. Mikkonen, "Mashing up software management, development, and usage data," in *RCoSE'15*, 2015.

[26] M. Rosli, "Can we trust our results? a mapping study on data quality," in *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, Dec 2013, pp. 116–123.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3899-5
ISSN 1459-2045