Department of Computer Science
Series of Publications A
Report A-2015-7

Software Developer Experience: Case Studies in Lean-Agile and Open Source Environments

Fabian Fagerholm

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Hall 5, University Main Building, on the 4^{th} of December, 2015, at noon.

University of Helsinki
Finland

Supervisor
Tomi Männistö, University of Helsinki, Finland

Pre-examiners
Tracy Hall, Brunel University London, United Kingdom
Kari Smolander, Lappeenranta University of Technology, Finland

Opponent
Robert Feldt, Blekinge Institute of Technology and Chalmers University of Technology, Sweden

Custos
Tomi Männistö, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911, telefax: +358 9 876 4314

ISSN 1238-8645
ISBN 978-951-51-1746-5 (paperback)
ISBN 978-951-51-1747-2 (PDF)
Computing Reviews (1998) Classification: K.6.1, D.2.9, K.6.3, K.7.4
Helsinki 2015
Unigrafia

Software Developer Experience: Case Studies in Lean-Agile and Open Source Environments

Fabian Fagerholm

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
fabian.fagerholm@helsinki.fi
http://www.cs.helsinki.fi/people/fabian.fagerholm

PhD Thesis, Series of Publications A, Report A-2015-7
Helsinki, December 2015, 118 + 68 pages
ISSN 1238-8645
ISBN 978-951-51-1746-5 (paperback)
ISBN 978-951-51-1747-2 (PDF)

Abstract

Human factors have been identified as having the largest impact on performance and quality in software development. While production methods and tools, such as development processes, methodologies, integrated development environments, and version control systems, play an important role in modern software development, the largest sources of variance and opportunities for improvement can be found in individual and group factors. The success of software development projects is highly dependent on cognitive, conative, affective, and social factors among individuals and groups. When success is considered to include not only fulfilment of schedules and profitability, but also employee well-being and public impact, particular attention must be paid to software developers and their experience of the software development activity.

This thesis uses a mixed-methods research design, with case studies conducted in contemporary software development environments, to develop a theory of software developer experience. The theory explains what software developers experience as part of the development activity, how an experience arises, how the experience leads to changes in software artefacts and the development environment through behaviour, and how the social nature of software development mediates both the experience and outcomes. The theory can be used both to improve software development work environments and to design further scientific studies on developer experience.

In addition, the case studies provide novel insights into how software developers experience software development in contemporary environments. In Lean-Agile software development, developers are found to be engaged in a continual cycle of Performance Alignment Work, where they become aware of, interpret, and adapt to performance concerns on all levels of an organisation. High-performing teams can successfully carry out this cycle and also influence performance expectations in other parts of the organisation and beyond.

The case studies show that values arise as a particular concern for developers. The combination of Lean and Agile software development allows for a great deal of flexibility and self-organisation among developers. As a result, developers themselves must interpret the value system inherent in these methodologies in order to inform everyday decision-making. Discrepancies in the understanding of the value system may lead to different interpretations of what actions are desirable in a particular situation. Improved understanding of values may improve decision-making and understanding of Lean-Agile software development methodologies among software developers. Organisations may wish to clarify the value system for their particular organisational culture and promote values-based leadership for their software development projects.

The distributed nature and use of virtual teams in Open Source environments present particular challenges when new members are to join a project. This thesis examines mentoring as a particular form of onboarding support for new developers. Mentoring is found to be a promising approach which helps developers adopt the practices and tacit conventions of an Open Source project community, and to become contributing members more rapidly. Mentoring could also have utility in similar settings that use virtual teams.

Computing Reviews (1998) Categories and Subject Descriptors:

K.6.1 [Management of Computing and Information Systems]: Project and People Management—life cycle, training;

D.2.9 [Software Engineering]: Management–life cycle, productivity, programming teams, software process models;

K.6.3 [Management of Computing and Information Systems]: Software Management—software development, software process

K.7.4 [The Computing Profession]: Professional Ethics

General Terms:
Human Factors, Management, Performance, Theory

Additional Key Words and Phrases:
Ph.D. thesis, developer experience, agile software development, lean software development, open source software development, values, onboarding, virtual teams, cognition, conation, motivation, affects, social factors

Acknowledgements

A doctoral thesis is a solitary effort, but there is a great deal of support and input from others. The work is personal so far as the author has succeeded in selecting, combining, and refining contributions of different forms from others and fusing them with his or her own thoughts and labour into a single whole. Perhaps only the author is in a position to see what nuance of the work can be traced back to its original source; to this article, to that remark, to those insights gained together during mutual discussion. I am grateful for each bit and wish to thank all those who have been part of this process in one way or another.

First, I would like to thank my supervisor Tomi Männistö for his support and guidance. In the latter phases of my PhD studies, his advice has been particularly important. His open attitude has made it easy to discuss anything I might have had on my mind, and his interest towards research methods and the eternal questions of research rigour and validity helped me make sense of my own work as it approached completion.

I thank my preexaminers, Tracy Hall and Kari Smolander, for their insightful feedback and encouraging remarks. Through their helpful advice, I have been able to improve many aspects of this thesis. I also wish to thank my opponent, Robert Feldt, whose work on psychological and social aspects in software engineering has strengthened my intuition of the importance of such concerns, and shown that it is possible to publish high-quality research in that area.

I have been fortunate to receive a great deal of support in different forms throughout my PhD studies. Docent Marko Salmenkivi initially supervised the thesis, and our long, interesting, and often humorous discussions on various topics were welcome distractions from the mundane. Professor Pekka Abrahamsson secured funding that supported my project during its first years, and he was not afraid of initiating and opening doors for risky, ambitious endeavours. Prof. Dr. Jürgen Münch generously supported my research through funding from his FiDiPro research project, and we have enjoyed a period of fruitful cooperation in writing articles. He has taught

me a lot about executing research studies and disseminating scientific results that I could otherwise only have picked up through years of trial and error.

During my studies, I have come to appreciate the importance of a supportive and well-working research group. The group I have been part of has gone through several transformations since I joined in 2009. I wish to thank three persons from the early days in particular, for being good co-workers and for their part in the work that comprises this thesis. Petri Kettunen is a loyal colleague and deep thinker, and I have been able to count on his steady flow of out-of-the-box ideas. Virpi Roto supported me greatly during our close research cooperation. Her encouragement helped me dare go into the field, meet research participants in their environment, and to trust my skills to collect and analyse empirical data. I remember our Affinity Wall data analysis sessions with a smile. With Marko Ikonen, many hours were spent on pondering matters regarding both work and life, and I greatly appreciate the value of his support.

There are also some persons to thank from a later group constellation. Patrik Johnson helped in tremendous ways with organising and improving the Software Factory project course at the Department of Computer Science. Kati Kyllönen has been similarly supportive and I have been able to count on her in many situations. Alejandro Sánchez Guinea helped turn some of these projects into research studies, resulting in a series of publications, two of which are part of this thesis. Max Pagels supported me in many ways, and helped me in particular with one of the publications in this thesis. I thank him not only for that, but also for the many nice lunches we enjoyed together and for generously giving me countless rides home after work.

In our current group, I wish to extend my thanks to Simo Mäkinen, Sezin Yaman, Myriam Munezero, Leah Riungu-Kalliosaari, Hanna Mäenpää, Juha Tiihonen and Antti-Pekka Tuovinen, who have been patient with me during the final phases of completing this thesis. I have also found it valuable to be able to work alongside them in ongoing research projects. I have learned a lot and our discussions, ranging from the academic to the social and personal, has made me feel at home.

I would also like to thank Jay Borenstein at Stanford University and Facebook, without whom the study on onboarding in Open Source projects would not have been possible. Jay did not hesitate to let me utilise the Open Academy program he has developed, for which I am very grateful.

I would also like to acknowledge the support of the Department of Computer Science at the University of Helsinki, which has provided an excellent working environment and high-class administrative and IT services. This support has extended beyond my thesis work to all other activities

and duties I have had at the department. Esko Ukkonen and Jukka Paakki have both supported my efforts as heads of department at different times, and I thank them for their leadership and availability. Jukka's advice and support for this thesis have also been crucial. In addition, I wish to extend my warm thanks to all those in the administration and IT services who keep the department running every day.

Thanks are also due to the students whom I've had the privilege of meeting in the Software Factory project course and other courses I've taught at the Department of Computer Science. I count more than two hundred names so far, and it is not possible to list them here. Some of these students have also been participants in the studies contained in this thesis, some directly and some in pilot runs. I strongly suspect that I have been learning more than they have.

This thesis would not exist without the companies that have devoted their resources to support this work, and I am grateful for their contribution. Also, I extend my thanks to the numerous employees in these companies who have generously shared their time and experiences.

This work has been supported by the Department of Computer Science, University of Helsinki, and Tekes – the Finnish Funding Agency for Technology and Innovation, as part of the Scalable High Performing Software Design Teams research project, and the Cloud Software and N4S Programs of DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business), as well as the Cloud Software Factory FiDiPro project. This work was supported in 2013 by a Nokia Foundation scholarship.

Finally, and most importantly, I would like to extend my loving gratitude to my family. My dear wife Jonna, thank you for all your support, patience, and love. Thank you for taking this journey with me. My two strong and beautiful daughters Aurora and Enid, thank you for the insights about the joy, awe, and wonder of being in the moment to discover the world. And to the one who has not yet arrived: I eagerly look forward to yet more moments . . .

Helsinki, November 2015
Fabian Fagerholm

List of Original Publications

This thesis is based on the following peer-reviewed publications, which are referred to as Articles I–V in the text. The contributions of the present author are described below. The publications are reproduced with permission from the copyright holders at the end of the thesis.

Article I Fagerholm, F., Münch, J. (2012). Developer Experience: Concept and Definition. In International Conference on Software and System Process (ICSSP 2012), pp. 73–77. IEEE.

Article II Fagerholm, F., Ikonen, M., Kettunen, P., Münch, J., Roto, V., Abrahamsson, P. (2015). Performance Alignment Work: How Software Developers Experience the Continuous Adaptation of Team Performance in Lean and Agile Environments. Information and Software Technology, vol. 64, pp. 132–147. Elsevier.

Article III Fagerholm, F., Pagels, M. (2014). Examining the Structure of Lean and Agile Values Among Software Developers. In Proceedings of the 15th International Conference on Agile Software Development (XP 2014): Agile Processes in Software Engineering and Extreme Programming. Lecture Notes in Business Information Processing, vol. 179, pp. 218–233. Springer International Publishing.

Article IV Fagerholm, F., Sanchez Guinea, A., Münch, J., Borenstein, J. (2014). The Role of Mentoring and Project Characteristics for Onboarding in Open Source Software Projects. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2014), Article 55. ACM, New York, USA.

Article V Fagerholm, F., Sanchez Guinea, A., Borenstein, J., Münch, J. (2014). Onboarding in Open Source Software Projects. IEEE Software, vol. 31, no. 6, pp. 54–61. IEEE.

For Article I, the author developed the initial idea, contributed to reviewing the literature, and wrote the text for the article, considering comments and suggestions from the second author. For Articles II and III, the author carried out the main part of all phases of the studies, from ideation, initial review of literature, study design, data collection and analysis, and writing. For Articles IV and V, the author initiated and coordinated the study, contributed major parts of the study design and analysis, and wrote major parts of the text. None of the articles have been used in previous dissertations.

Contents

Chapter 1

Introduction

Software development is an inherently human, intellectual activity. Software is written by humans and writing it requires abstract thought and the ability to apply strategies for limiting and handling complexity. It is also a highly social activity, as software development involves collaborating with others to clarify what the software being written should do – its requirements – and to coordinate the writing of several pieces of software that should be combined into a larger system.

Software engineering research has shown that human factors are the most important determinants for outcomes such as individual productivity, team performance, software quality, and effectiveness of development methods. Although there are many studies on various human factors, they are not easily combined into a framework for guiding research and practice. A particular question is how development outcomes and the subjective experience of developers themselves can be explained in terms of human factors. A framework to structure this particular aspect of human factors in software engineering is missing.

This thesis aims to advance the empirical knowledge of how software developers experience the activity of developing software. It provides a theoretical contribution, grounded in empirical studies, that increases the understanding of software developer experience in relation to team performance, values, and team formation, especially in Lean, Agile, and Open Source development environments. Apart from the intrinsic value of increased knowledge, the findings in this thesis can be of use in both theory and practice. Increased understanding of software developer experience can be a foundation for new studies. The findings can also be used to improve development processes, methods, tools, and environments in studies and practice. Such improvements can increase the possibilities of success for software development projects. Conversely, a lack of such understanding

causes risks and problems when attempting to take processes, methods, and tools into use, as practitioners may not see personal relevance in such new solutions. Improved understanding can help make studies more relevant and practical deployments more likely to succeed.

1.1 Motivation

Traditionally, the field of software engineering has attempted to provide solutions for many problems related to software development by applying an engineering paradigm [Basili, 1996; Rombach, 2011]. For example, software architecture shows how the software itself can be structured in different ways to accomplish specific results. A certain architecture can result in performance improvements, while another architecture can result in greater modularity or make it easier to evolve the software for varying needs.

Software engineering has developed the notion of software process to codify the steps that should be taken to proceed from an initial need or idea, through the different stages of software development, to a released version of the software, and often onward to the deployment, maintenance, and sometimes eventual decommission of the software [Madachy, 2008]. These steps can be performed partially, iteratively, incrementally, in different sequences, using different levels of quality requirements, and with different kinds of technical methods and tools. An important reason to codify these steps is to coordinate the activities of several people working on the software. A timely delivery and sufficient level of quality are two other important goals to which a high-quality software process can contribute.

Software processes themselves can be considered a kind of software that can be analysed and improved for different purposes [Osterweil, 1987]. Software processes should be considered to be development or design processes rather than manufacturing processes [Humphrey, 1989; Basili, 1996; Conradi and Fuggetta, 2002; Freeman and Hart, 2004]. While manufacturing is constrained by the physical limitations of materials, design involves exploration of a potentially infinite space of ideas. The value of a certain set of choices from that space depends on many factors, including what is relevant for a specific customer at a specific point in time. The decision-making process in software design requires the involvement and collaboration of many stakeholders, bringing the full complexity of human social interaction into software development work.

Despite wide agreement on the importance of considering human aspects in software development, research on this topic remains fragmented [Warfield, 2010; Amrit et al., 2014]. Simultaneously, the importance of understanding

human aspects in software development is increasingly important, as rising numbers of people are engaged in software development work due to digitalisation. Exact numbers are difficult to obtain, but some estimates may be provided to show the order of magnitude. Scaffidi et al. [2005] estimated that while official census figures predicted less than three million professional programmers in 2012 in the USA, 90 million people would use computers in their work, and 13 million would perform some kind of programming by that year. A report published in late 2013 estimated that there are more than 11 million professional software developers in the world based on workforce, education, and macroeconomic statistics from 90 countries [International Data Corporation, 2013]. As software pervades society, more organisations become involved in the development of software-intensive systems in one way or another. Many kinds and sizes of organisations form software communities and ecosystems in which they can act as both producers and consumers of software. Some observers argue that in the near future, "every company will be a software company" [Meijer and Kapoor, 2014].

Understanding human factors in software development is key to improving both software development performance and well-being [Känsälä and Tuomivaara, 2013; Laanti, 2013]. While human factors concerns in software development have always been important, they are becoming even more central in today's development environments. Lean-Agile software development emphasises people, collaboration, customer involvement, and flexible adaptation to circumstantial changes, as central to the development activity [Dingsøyr et al., 2012]. Globally distributed development, often desired for potential cost and performance benefits, is highly dependent on human factors [Herbsleb and Moitra, 2001], and even more so when combined with Lean-Agile approaches. Open Source software development elevates collaboration and code reuse to a global scale, but comes with its own set of complexities and challenges.

When considering individuals and teams, it appears promising to turn to existing research on human factors in software engineering as a starting point. Current research indicates that behavioural outcomes are moderated not only by skill (e.g., Bergersen et al. [2011]; Bergersen and Gustafsson [2011]), but by motivational and affective constructs on the individual level (e.g., Shaw [2004]; Beecham et al. [2008]; Franca et al. [2012b]), and by communication and coordination aspects on the team level (e.g., [Herbsleb and Mockus, 2003b; Cataldo et al., 2008; Cataldo and Herbsleb, 2013]). In other words, a plausible approach to improving software development is to improve the cognitive, conative (motivational), and affective traits of software development projects, processes, and methods in a way that provides benefit

to the individuals and teams using them. However, besides considering behaviour, it is also important to consider subjective experience if one wants to improve both outcomes and working conditions (c.f. Csikszentmihalyi [1975]). Despite a growing body of literature on human factors in software development, it is not clear how the multitude of factors can be brought together into a framework that captures the experience of professional software developers and that is understandable to them.

1.2 Research Questions and Scope

Previous research does not provide a framework for understanding how developers experience software development. The subjective experience of software developers has an impact on externally observable outcomes, but it is difficult to explain and understand the impact and outcomes without an understanding of the subjective experiences. Indeed, software engineering research lacks a viable model of people that would take into account both overt behaviour and its correlates in the subjective mind of individuals. Such a model should relate individuals to their social context, since much of human experience and behaviour arises in interaction with the social environment. Understanding software developer experience is of interest not only for improving outcomes, but also for improving working conditions and creating sustainable work environments for software developers themselves.

In this thesis, we consider a software developer to be a person who is employed primarily to construct software or directly supervise the construction of software. Examples of such jobs include programmers, software architects, and software testers, but job roles such as software project manager also fit into the description in cases where the work is connected to software artefacts. For example, a project manager may have to understand and reason about substantial portions of the software and its architecture in order to manage a development project. That person may thus influence even technical decisions during the construction of software, and is included in our definition of a developer.

We ground our inquiry in two popular forms of modern software development: Lean-Agile software development and Open Source software development. These are selected partly because they represent two well-established boundaries of innovation in the field, and partly to control the scope of this thesis. Agile software development has become widely used in the software industry since its introduction in 2001 [Boehm and Turner, 2004] and is often combined in practice with Lean software development [Poppendieck and Poppendieck, 2003]. Lean-Agile software development thus

represents a contemporary form of software development to which a major portion of software development professionals are exposed. It also puts special emphasis on people as the main factor for success, which makes it attractive for studies on human factors. Lean-Agile software development is the primary context in which we develop a theory of software developer experience in this thesis.

Open Source software development is also a well-known phenomenon and Open Source software is in wide use in contemporary organisations [Crowston et al., 2008]. Open Source development tools and the Open Source Linux operating system can be considered de facto standards in many application areas. Open Source software development represents a perspective which is orthogonal to Lean-Agile: its primary characteristics are open access to source code – both in terms of licensing and online availability – fast, fluid, and self-organised development, and nearly exclusive use of online means of communication and collaboration. It can be considered a particular form of globally distributed software development where projects transcend organisational boundaries and participants can range from paid, professional developers to volunteer hobbyists. Open Source software development thus comes with a unique set of challenges but also promises unique benefits. In particular, its nature exposes human aspects of software development that may not be as clearly visible in other settings. Open Source is a secondary context for this thesis, in which we examine how a specific area of developer experience can be improved.

The main research problem addressed in this thesis is:

How do software developers experience the activity of software development?

In order to address this problem, we divide it into four research questions. First, we consider the developer experience construct in general, seeking a conceptual framework to inform the inquiry. Such a framework should provide a theoretical lens or perspective through which the main problem can be examined, with key concepts and their relationships stated. The first research question is:

RQ 1: How can software developer experience be conceptualised?

Second, we approach developer experience in Lean-Agile environments by considering it against the concept of performance. Software development organisations strive for high performance in order to meet demands for economic viability. However, software developers working in teams must

often balance potentially conflicting demands, such as meeting a deadline and ensuring future software maintainability or a low number of defects. Lean-Agile approaches to software development put heavy emphasis on self-organising teams [Moe et al., 2008]. The responsibility of performing well rests on the team as a whole as well as on team members individually. However, current research does not sufficiently explain how developers manage this responsibility nor how they experience it. Better understanding of this area could help explain achieved levels of performance, provide guidelines and training to improve self-organisation in teams, and to give better definitions of the performance concept in Lean-Agile settings. The second research question is:

RQ 2: How do software developers experience team performance in Lean-Agile environments?

Third, we approach developer experience in relation to values. Human values are abstract motivations that play an important role in human behaviour and in how humans experience their world. Values are core ingredients of human culture. Values propagate in and between groups, and consistent patterns of values, or value systems, can emerge, forming national, organisational, professional, and other kinds of cultures. An internally consistent value system which the members of a group share and agree with to a large degree is beneficial for well-being and for aligning activities towards common goals. As can be observed in foundational literature, Lean-Agile approaches have at their core attempts to codify specific value systems. For instance, Lean software development emphasises respecting individuals and empowering teams [Poppendieck and Poppendieck, 2003]; Agile software development explicitly values "individuals and interactions over processes and tools" [The Agile Alliance, 2001]. Theoretically, this emphasis can be seen as a value statement in favour of self-direction and universalism, and, more generally, of openness to change. Studies on Lean and Agile software development often refer to values as underpinning the development methodology but do not provide an explanation for why this matters. If values are to be treated as an influence factor in this context, they should be better understood. It is not clear how developers interpret the Lean-Agile value system. The third research question seeks to uncover how developers actually understand the value system rather than how the value system is depicted in literature:

RQ 3: How do software developers experience the value system of Lean and Agile approaches?

Fourth, we examine how a specific aspect of a specific software development environment can be influenced in terms of developer experience. We focus on team formation and performance in virtual team environments, especially in Open Source Software projects. Distributed and global software development is increasingly common in many organisations as they seek to lower the cost of labour and increase productivity. In addition, organisations may need to combine their resources with other organisations to tackle problems of a size or complexity that are beyond the capabilities of a single organisation. It is therefore important to understand how to support teams in such "virtual" environments, especially in situations where team composition is changing. Transferring project-specific knowledge and culture as well as fostering creation of social connections between new and more senior project members can be seen as contributing to a positive developer experience. Here, we use Open Source Software projects as the concrete context. The fourth research question is:

RQ 4: How can new developers be supported in virtual team (Open Source) environments?

The four research questions do not exhaustively address all aspects of the main research problem. Their function is to demarcate the research so that it is possible to conduct within the scope of a PhD thesis. Research questions 2–4 are tied to a case study context and a particular theme or construct within that context. The thesis addresses the main research problem with respect to each context and theme or construct. Table 1.1 shows the relationship between the case study contexts, articles, themes, and research questions in this work.

1.3 Research Contributions

This thesis provides a novel contribution by investigating software developer experience, particularly in Lean-Agile and Open Source environments. Advancing the understanding of developer experience enables improvements in projects' end results and developers' work conditions, allows researchers to structure their inquiry on developers' realities, reveals the complex nature of developers individually and in groups, and gives a structure to manage this complexity. It should be noted that the contributions of this thesis do not consider software development outcomes directly. Instead, they address the way software developers construct their reality, which has an indirect effect on outcomes. In this section, we briefly describe the contributions of the five original, peer-reviewed articles of which this thesis is comprised.

Table 1.1: Relationship between case study contexts, articles, themes, and research questions.

		Lean-Agile		**Open Source**
	RQ 1 Concept	RQ 2 Team Performance	RQ 3 Values	RQ 4 Onboarding
Article I	×	×	×	×
Article II	×	×	(×)	(×)
Article III			×	
Article IV				×
Article V				×

× Article addresses research question.
(×) Article provides background and motivation for research question.

Article I: Developer Experience: Concept and Definition

We propose to consider the experience of software developers in their work, and define the concept *developer experience* [Article I]. We transfer the user experience concept into the domain of software developers. Developer experience focuses on the individual software developer in the context of a software team but also situates the individual and the team into the larger organisational context. We do not limit the developer experience concept to only the technical tools used for software development, but include the complete environment including the software development process and work methods. Article I provides a theoretical lens for the thesis and thus addresses all our research questions.

Article II: Performance Alignment Work: How Software Developers Experience the Continuous Adaptation of Team Performance in Lean and Agile Environments

The term "performance" often refers to the ability to reach desirable outcomes. From a managerial perspective, software developers are expected to meet requirements of schedule and scope – to finish a certain set of work in a certain time. However, there are many more aspects to performance in software development. We examined the developer perspective of performance in a professional setting [Article II]. We conducted a multiple-case study

with five companies, studying how software developers experience team performance in Lean-Agile environments. We developed the *Performance Alignment Work* theory, which explains how software developers negotiate the meaning of performance in different situations, interpret their current performance, and adapt it to changing circumstances. In the context of this theory, high-performing teams are those which are particularly good at Performance Alignment Work and engage with their environment to influence performance expectations. Our findings illustrate differences and similarities in performance experiences among professional software developers in different types of companies. Values and the team formation process were highlighted as factors with an important relationship to developer experience and performance.

Article II addresses RQ 2 by presenting an in-depth study of developer experience in a professional context where Lean-Agile software development is used. It directly addresses the research question and contributes the Performance Alignment Work theory, explaining the constant process of negotiation in which developers are engaged. It also provides increased detail for answering RQ 1. The article provides additional motivation and background for research questions 3 and 4, since we found that the value system underpinning Lean-Agile methods, and the special consideration of team formation, are among the important factors contributing to how developers experience performance in Lean-Agile environments.

Article III: Examining The Structure of Lean and Agile Values Among Software Developers

The findings from Article II indicate that values are an important factor in understanding how developers experience performance. Article III addresses the values aspect of developer experience in Lean-Agile environments in more detail. We present results from a quantitative study examining Lean-Agile values. Apart from contributing a questionnaire instrument for assessing values, the study reveals a number of findings regarding how today's software developers perceive the values behind Lean-Agile software development. Values are strongly emphasised in these methodologies. We seek here to uncover how developers understand the value system associated with the methodologies. Making the value system explicit can have benefits in terms of increased understanding of the methods, better adaptability to national and cultural values, and improved developer experience. Article III addresses RQ 3 by presenting an empirically grounded model of Lean-Agile values with 11 dimensions, and by relating the model to earlier research on universal human values and personality.

Article IV: The Role of Mentoring and Project Characteristics for Onboarding in Open Source Software Projects

The findings from Article II indicate that the team formation and reformation process is important for understanding how developers experience performance. In today's organisations, the desire for dynamic resource allocation means that changes in team composition is a common occurrence. Team formation is also important in contexts where team and project membership is fluid, such as in Open Source software development. Many organisations already operate in a manner where software development crosses organisational boundaries and developers may enter and exit projects frequently. Onboarding, or organisational socialisation, is a process that helps newcomers become integrated members of their organisation by learning the knowledge, skills, and behaviour they need to succeed and be productive in their work. In Article IV, we present a study examining onboarding in an Open Source context, where virtual teams are used. We studied mentoring by experienced project members as a support mechanism for onboarding. We compared the performance of newly introduced developers receiving onboarding support to that of unsupported developers. Through Article IV, we address RQ 4 by examining mentoring as a support mechanism for onboarding new members into virtual teams in Open Source projects. We found that mentoring can be beneficial for onboarding, that onboarding is a learning process which does not proceed linearly, and that differences in project characteristics mediate the degree of success of onboarding.

Article V: Onboarding in Open Source Software Projects

Article V continues the analysis presented in the previous article. We examined mentoring as a potential onboarding mechanism in more detail, including the returns on resources spent on mentoring. Article V addresses RQ 4 by providing additional findings and guidelines regarding mentoring as an onboarding support mechanism. The findings indicate that mentoring can have a positive impact on the early stages of onboarding, but also that mentoring is associated with costs in terms of reduced development performance among developers carrying out the mentoring function. Since the mentor is also a developer whose experience with the onboarding process is of importance, we suggest some ways in which to motivate and compensate mentors for their efforts.

1.4 Thesis Structure

This doctoral thesis consists of five original research articles and the present summarising report. The remainder of this summarising report is organised as follows. Chapter 2 reviews and discusses literature in order to develop a theoretical background regarding the nature of software development. The chapter examines the relationship between software products, processes, organisation, and human factors, and provides some fundamental theory on human factors. Chapter 3 describes the components of a research design in general, and explains the research design of this thesis in particular. Chapter 4 draws together the research conducted in Articles I–V and provides the research results answering the research questions. The chapter also summarises the articles as a whole from the perspective of the main research problem. Chapter 5 discusses the implications of the answers for research and practice, their validity, the ethics of the research, and potential future work. Finally, Chapter 6 concludes with a summary.

Chapter 2

Theoretical Background

In this chapter, we review and discuss literature on software development in order to illuminate its nature from different perspectives and to provide the necessary theoretical background for the remainder of the thesis. In order to understand how software developers experience software development, we must first understand what software development is. We must understand, in general terms, the activity system in which software development is conducted: the object of software development, the rules and assumptions which govern it, and the instruments, mediating artefacts, and methods of labour division that are used to achieve desired outcomes. We must simultaneously understand how human factors come into play in software development.

Three broad groups of questions characterise the development of software: those concerning the product, the process, and the people involved. Questions in the first two groups ask which attributes the software should possess, and how the activity of building the software should be performed in order to ensure that those attributes are present to the desired degree. The first group of questions has been addressed from a variety of perspectives, ranging from methods to structure the software – its architecture – to methods of eliciting and managing requirements, and further to designing the patterns of software use, a topic addressed by usability and interaction design. The second group of questions has mainly been addressed within the field of software engineering research in terms of *software process*, with different process models and related methods being key contributions. The third group of questions concerns the people involved in carrying out software development. Who are they? What characteristics do they possess? What is the interplay between these characteristics and the software development activity? What is the relationship between people in a group developing software, and between people, the product, and the process?

2.1 Software Products

Software programs are instructions for digital computers to perform computations and interact with input and output devices. A software product refers to a product whose primary component is software [Kittlaus and Clough, 2009]. In some cases, software is an underlying product part that cannot be bought separately. For instance, embedded software can be considered software that is integrated in a non-software product and thus not sold as a stand-alone product. Software products can be delivered as part of a physical product, be standalone, delivered on a physical medium or via the Internet either to be installed on the user's computer or as a remotely accessible service. Frequently, software products and services occur together, such as in the case of smartphones or tablet computers, which are embedded software products that connect to software services over a network, making them platforms for delivering new features to users. We use the term software product to include software services unless otherwise noted.

2.1.1 Programming

Software programs and products can be seen from the context of development, which is the viewpoint of this thesis. In the context of development, software programs are written as source code by developers and built into object code and associated data, to be executed and processed in the context of use. On a fundamental level, programming involves identifying algorithms, data structures, and other kinds of programming language and support library constructs that can be used to write each part of the program. Foundational research in computer science deals with these aspects (c.f. e.g., Knuth [1997a, 1998, 1997b, 2011], Dahl et al. [1972], and Nygaard and Dahl [1978]). Programming platforms consist of one or a few languages, support libraries which contain reusable program code for common tasks, and tools to compile the source code into executable object code or, in the case of interpreted languages, an interpreter for running the code. Modern programming platforms are sophisticated frameworks with ready-made code for a variety of purposes, and come with integrated development environments that provide powerful functions for writing and editing source code in a solitary or group setting.

2.1.2 Software Architecture

The amount of program code required for a given product can be very large, and the relationships within the code and between the code and data can

be very complex. Large size and high complexity can make it difficult to manage programming: making code changes or testing the code can become intractable. Software architecture addresses these difficulties by introducing a higher level of abstraction that partitions software into pieces and governs the relationship between the parts. Software architecture has other roles besides structuring the software, such as allowing for software reuse and providing means to divide programming tasks between several programmers.

The importance of software architecture has been emphasised by many researchers (e.g. Parnas [1972]). Brooks [1975] considers developing software architecture a creative design problem, and asserts that organisations must take the problem of software design seriously. He notes that good designs can arise from good practices, but "great designs come from great designers", emphasising the people aspect of software development. The need for an intermediate design layer – the architecture – has been noted by other authors (e.g., [Brooks, 1975; Mills, 1985; Royce and Royce, 1991; Perry and Wolf, 1992]; for an overview, see e.g. Kruchten et al. [2006]).

2.1.3 Software Product Design

The role of a software developer encompasses more than programming. In order to be able to write source code, developers must know several things about the context of use. For example, they must have details regarding the technical execution and processing environment, the users, the tasks to be carried out by the users, and the situations in which those tasks are to be carried out. This information can be expressed as documentation or models of different degrees of formality, but it is also present in the mental models that individual developers carry in their minds. Through a design process, which includes activities such as ideation, planning, prototyping, experimentation, analysis, and refinement, developers use their understanding of software requirements to develop a program or set of programs that constitute the product.

A pertinent question is how to organise the software design and development activity. Software development can be structured by processes and organisation-scale frameworks to standardise, manage, and improve projects, products, knowledge, and the workforce. Within those structures, the nature of software development can be seen broadly as a design activity, encompassing "all the activities involved in conceptualising, framing, implementing, commissioning, and ultimately modifying complex systems" [Freeman and Hart, 2004]. Design may be viewed as a phased activity, with each phase including generation and selection of design alternatives, their representations, procedures for solving design problems, human approaches to design, and

design structures [Simon, 1969]. The final outcome of any design process is uncertain, and thus its eventual value is uncertain [Baldwin and Clark, 2006]. Methods to manage risk are hence important. Having modularity-in-design is of high economic interest, since it means that the design process can be split into modules and carried out in parallel. For this reason, designing and identifying modularity in different forms and at different levels is an important part of software development work. This includes the product level [Pohl et al., 2005], but also the levels of technical architecture and detailed program structure.

There are several possible approaches to software design on all levels. Before examining these in further detail in the next section, we briefly establish a more general frame of reference regarding design. Moran [1996] identifies four broad conceptions of design in the literature (see Figure 2.1). Kalfoglou et al. [2000] characterise the four conceptions as follows. *Decomposition and synthesis* involves recombining existing components and then abstracting from the resulting combinations into new meta-knowledge. *Design as search* treats the problem as a traversal of a design space, with the aim of finding paths to goals. *Design as negotiation and deliberation* treats design problems as "wicked problems" which have no objectively definable criteria for solution correctness. In this view, design is an activity that supports a community in conducting an effective debate over a range of possible solutions. Finally, *situated design* refers to a reflective activity where the designer obtains new knowledge through interaction with the design object – a kind of experimentation which continually results in new versions of a design through observing errors. Kalfoglou et al. [2000] observe that combinations are also possible. For example, the Experience Factory concept (c.f. Basili et al. [1992]; Basili [1993]; Basili et al. [2002]) can be characterised as both design as search and situated design.

Design is also an organisational concern. Clark and Fujimoto [1990] argue that integrity is crucial for the success of products. Integrity means good performance and value, satisfaction for customers in every respect, including intangible aspects such as look and feel. Companies that consistently develop products with high integrity have two major traits: internal and external integration. Internally, they are coherent and there is a large degree of alignment on all levels of the organisation, ranging from strategy and structure on the managerial level to the skills, attitudes, and behaviour of individual employees. Externally, companies are integrated with their customers so that the customer becomes a part of the development organisation. Clark and Fujimoto [1990] further argue that integrity begins with a product concept that captures the customer's viewpoint and continues with the concern of

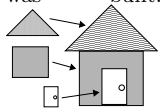
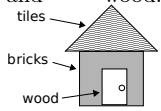
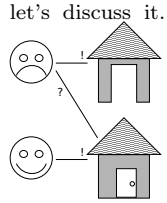
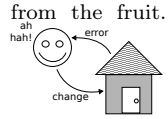
Design paradigm	**Decomposition and Synthesis**	**Search**	**Negotiation and Deliberation**	**Situated**
Driver	Reuse libraries	Heuristics	Conflicts	Errors
Early work	Alexander [1964]; Alexander et al. [1977]	Simon [1969]	Rittel and Webber [1973]; Finkelstein et al. [1994]	Schön [1983]
Example	Build a house using parts found when the last house was built. 	Build a house via a search of the space of what is known of bricks, tiles, and wood. 	Let each stakeholder design their preferred house, then let's discuss it. 	Build the house, find out that the living room gets too hot in the summer, plant fruit trees for shade, make and sell the jam from the fruit. 

Figure 2.1: Four broad descriptions of design paradigms. (Adapted from Moran [1996] and Kalfoglou et al. [2000].)

satisfying what the customer wants and how completely the product concept has been embodied in the product's details. For software developers, the degree of internal and external integration in the organisation and in the approach they use to develop software can have a significant impact on the nature of their work.

2.2 Organisation of Software Development

Organising software development concerns structures and approaches to manage all aspects of the software development activity with the purpose of ensuring that desired outcomes are achieved in a timely fashion and within budget constraints. How software development is organised and what is emphasised in the organisation can have far-reaching effects for the outcome. For example, an exploratory survey of software factories led Cusumano to propose that factory implementations in the U.S. and Japan fall into a spectrum of varying control, and that the Japanese factories placed significantly more emphasis on the concept of reusability [Cusumano, 1989]. Cusumano reported that the Japanese organisations which had introduced more structured ways of work displayed significant improvements in productivity, quality, and process control over several years. However, he notes that the Japanese organisations used the factory facilities mostly for products that were similar to ones they had made in the past. New development was sourced from less-structured subsidiaries or suppliers, or performed as special projects outside the factories. Cusumano concludes that, at the time of the study, the weakness of the Japanese approach was their focus on process improvement rather than product innovation or development of product packages. This example illustrated that differences in how software development is organised can have a major impact on both the technical outcome as well as the utility of the end result.

2.2.1 Software Projects

Projects are temporary endeavours with specific aims, and they are unique in the sense of not being routine operations [Project Management Institute, 2013]. Project management organises software projects. The objective is to produce the right software product, with the right level of quality, at the right cost, and within the right schedule. Scope, quality, cost, and schedule are among the fundamental attributes that must be balanced in software project management. Software development projects often use a project management framework, life-cycle model, or process to structure activities.

2.2.2 Software Processes

The use of a software engineering process is a key ingredient in professional software development. Software engineering processes, referred to here simply as software processes, are concerned with "work activities accomplished by software engineers to develop, maintain, and operate software, such as requirements, design, construction, testing, configuration management, and other software engineering processes" [Bourque and Fairley, 2014]. A software process has been defined as a goal-directed activity in the context of engineering-style software development [Münch et al., 2012]. Software processes are the foremost means in software engineering for structuring software development over time. Software processes address the question of how to build a software artefact. Research on software processes strongly indicates that there is a dual relationship between software products and processes [Osterweil, 1987], and that there is value in choosing the right process and adjusting and improving it for one's specific purposes and environment.

Software process encompasses three broad, overlapping areas: process modelling, process life-cycle models, and process improvement [Madachy, 2008]. Process modelling represents processes in the abstract: as mathematical models that focus on a particular aspect of a process as currently implemented (as-is), or as planned (to-be). These models can be used to evaluate the process, simulate its behaviour under particular conditions, identify improvement areas, or provide training to practitioners. Process life-cycle models determine the steps to be taken in software development or evolution, the order of the stages, and the transition criteria between the stages. Finally, process improvement refers to activities which aim to improve some aspect of software development – such as reducing the number of product defects, increasing development speed, or reducing costs – by improving a related part of the development process.

Software process life-cycle models are structures which determine the approach taken to the tasks or activities that take place during the activity of developing software [Madachy, 2008]. The model determines which steps are taken during software development or evolution, the order of the stages, and the criteria for transitioning between stages. Different models specify these to varying exactness, and there is a multitude of models to choose between. Models differ in their underlying assumptions about user requirements, technology, system complexity, system quality criteria – such as reliability, safety, and reusability – schedule, cost limitations, stakeholder involvement, developer skills, and the organisational context in which the development project is performed. Some of these assumptions are explicit in the model,

while others may be implicit or unknown. Thus selecting a model which is appropriate for a given situation is a difficult but important task, since it can determine whether a resulting system satisfies user needs, or cause changes in project cost and schedule [Alexander and Davis, 1991; Cockburn, 2000]. In practice, models must be adapted and tuned to account for the environment in which they are applied and the type of product that is to be developed.

Software process improvement consists of activities which aim to produce an understanding of existing processes and change them in order to increase product quality, lower costs, or reduce development time [Sommerville, 2011, chap. 26]. Software process improvement owes its legacy to process improvement in manufacturing. Through an approach known as statistical process control (see e.g., Shewhart [1931]; Deming [1975]; Oakland [2008]), a physical manufacturing process can be measured in terms of how many product defects it produces, and the process is analysed and modified in order to reduce the defect ratio and increase the capability of detecting defects. Humphrey [1988] was among the first to argue that the same type of technique can be applied to software engineering.

However, software processes exhibit important differences compared to manufacturing processes, and Humphrey's [1988] argument is not universally accepted without reservation [Sommerville, 2011]. Software processes are design processes involving knowledge and information, and rather than operating through physical laws, they rely on human cognition, communication, and collaboration. For software, there is always an inherent reliance on individual creativity when developing solutions [Basili, 1996]. Taking process models into use is as important as, but frequently more difficult than, constructing them [Münch et al., 2012]. Thus, the question of how to get people to follow the process as intended is often neglected. More fundamentally, we may ask whether people should adapt to the process or whether the process should be tailored to its users – the software developers. The way we think about the software design activity may result in different answers to this question.

Different types of software development process models approach software design in different ways. Following the design terminology introduced in Section 2.1.3, more linear, sequential models are closer to decomposition and synthesis, while iterative and incremental models are closer to design as search. Lean-Agile approaches, detailed below, can be characterised as design as search due to their iterative and incremental nature, but they also emphasise the aspect of negotiation and deliberation through their involvement of the customer. For example, the Scrum method [Sutherland

and Schwaber, 1995; Schwaber and Beedle, 2001] involves the customer through the Product Owner role, whose responsibility it is to participate in story creation and in managing the product backlog.

The various life-cycle models can be categorised according to type. Some models stand out as "classical" examples. The simplest model is a single-pass, sequential process that consists of requirements analysis, design, coding, testing, and operation stages. Variants of this model are commonly known as waterfall models. Perhaps the most distinguishing trait of this type of model is that it aims to develop a complete set of requirements during the first phase [Bourque and Fairley, 2014]. The requirements are then rigorously controlled and any changes to requirements are based on documented change requests that must be approved by a control board. There is some confusion as to where this type of model originates. Some have attributed it to Royce [1970] although Royce's model proposed several improvements on a purely sequential process, including the recommendation to perform the process twice, with the first run being a kind of in-depth prototyping stage. A more accurate description may be that the model originated in a few early works which described experiences and lessons learnt in the practice of software development, including that of Benington [1983] (the referenced article is adapted from a presentation given by Benington in 1956), Rosove [1967], and Lehman [1969].

One modification to the waterfall model is the incremental approach, where the sequential process is repeated in multiple passes to meet predefined requirements in successive steps. Each increment adds functionality to the previously released increment, reducing the time required to provide initial versions of the system to the customer. The complete set of requirements is defined first, after which it is partitioned into increments and each increment is then implemented [Bourque and Fairley, 2014]. Some flexibility may be permitted in revising the requirements as the software evolves.

Iterative models produce software in small, usable pieces that build upon each other. Iterative models were first documented by Basili and Turner [1975]. Their approach proceeds in a top-down, stepwise, heuristic fashion. A skeletal sub-problem is implemented first, and acts as an initial guess for the final implementation. Subsequent steps iteratively enhance the current implementation, implementing each requirement until the final implementation is completed. A key trait of this approach is that the developer can take advantage of what was learnt during the previous iterations.

2.2.3 Lean-Agile Software Development

Lean software development and Agile software development are relatively recently described approaches to software development that have become very popular. Both concepts can be found in manufacturing production methodology. In manufacturing, Lean strives to eliminate waste in the manufacturing process, with Agile being the next step enabling quick response to customer needs and market changes; the two may also be viewed as complementary [Naylor et al., 1999].

Although the terms are relatively new in software development, the underlying ideas have existed for a long time. Larman and Basili [2003] document instances of iterative and incremental processes from as early as 1957. They demonstrate that the practices of these approaches have been used in several large, software-intensive technology development projects in both governmental and private organisations. They also link the evolution of iterative and incremental processes to that of the waterfall approach. They note that at the beginning of the 1980s, iterative and incremental processes were in a minority position compared to the sequential approach. During the 1980s, sequential processes were subject to increasing criticism, leading in that same decade to evolutionary and risk-driven approaches, and, in the 1990s, to more variants of iterative and incremental processes. Eventually, in the early 2000s, a small group of people interested in promoting lightweight, iterative and incremental processes summarised what they felt was the essence of software development process under the umbrella term "Agile". They published the Agile Manifesto [The Agile Alliance, 2001] to codify their position as a set of core values. Agile software development was a response to perceived challenges arising from a turbulent business environment [Cockburn and Highsmith, 2001; Highsmith, 2001]. Several Agile software development methods have been proposed, such as Extreme Programming [Beck and Andres, 2004], Scrum [Sutherland and Schwaber, 1995; Schwaber and Beedle, 2001], and Dynamic Systems Development Method [Stapleton and Constable, 1997]; for an overview, see e.g. Abrahamsson et al. [2003].

Since the introduction of the term, Agile methods have become very popular [Boehm and Turner, 2004]. Agile methods are designed to facilitate evolution of the software requirements as the implementation process unfolds in a project [Bourque and Fairley, 2014]. Agile methods emphasise rapid delivery of customer value, customer involvement in the software development process, and reliance on human relationships rather than documentation [Cockburn and Highsmith, 2001]. Agile software development does not refer to any single process model, but is rather an approach that aims to empower software development teams to take charge of their own

activities and focus only on activities which are essential for delivering customer value. Agile software development can be seen as a reformulation of iterative and incremental software development [Larman and Basili, 2003], but with an added emphasis on people. The focus on humans in Agile software development can be observed through the fact that in contrast to life-cycle models with formal, technical definitions, Agile is founded on a values-based manifesto accompanied by a set of high-level principles (see The Agile Alliance [2001]; Highsmith [2001]). Descriptions of Agile software development methods usually refer to, and repeat, the central tenets of the manifesto, and use its value statements and the accompanying principles to justify methodological decisions.

Lean software development was introduced in 2003 in a book titled "Lean Software Development: An Agile Toolkit" [Poppendieck and Poppendieck, 2003]. This treatment positions Lean within Agile software development. Lean software development also emphasises creation of value for customers and emphasises people as a key success factor for software development projects [Poppendieck and Poppendieck, 2003]. Interestingly, Lean links back to the early foundations of software processes by highlighting the removal of wasteful activities – those that do not contribute to customer value – as a central activity. The term "Lean" emerged from a number of studies of the automotive industry, particularly findings from Toyota in Japan (as part of MIT's International Motor Vehicle Program (IMVP); some important findings are reported in e.g. Krafcik [1988]; for a summarising example of related studies, see e.g. Shimokawa [2010]). The Toyota Production System (TPS) was described as a key reason behind the success of Toyota in the car manufacturing industry [Ōno, 1988; Womack et al., 2007; Liker, 2004; Morgan and Liker, 2006], although the validity of some of the IMVP data, and the correctness of the interpretations and conclusions reached based on the IMVP studies have been questioned [Coffey, 2006; Dybå and Sharp, 2012]. Nevertheless, the terms Lean and Lean thinking have entered the field of software engineering, resulting in a body of work that complements Agile.

As with Agile, Lean thinking has been linked to a philosophy with deeply rooted values. Many researchers claim that Lean cannot be implemented effectively without simultaneously understanding and implementing the philosophy and value system on which it is based (e.g., [Ōno, 1988; Holweg and Pil, 2001; Liker, 2004; Hines et al., 2004; Saruta, 2006; Womack et al., 2007; Liker and Hoseus, 2008]). Whether this applies in the context of software development is open to debate, but it appears reasonable to assume

that successful implementation of both Lean and Agile depends on more than straightforward use of daily practices.

There is confusion as to whether Lean software development should be considered an Agile method or a different approach with a close relationship to Agile [Lane et al., 2012]. There are important common aspects to Lean and Agile: a focus on customer value, mapping the stream of value adding activities and workflows, and, importantly, an emphasis on individuals as the key ingredient for success. In contrast to Agile software development, the Lean movement has not produced comprehensive software development lifecycle or development process models. However, the Kanban task scheduling method, which is part of the Lean toolkit, has been used both on its own and in combination with Scrum [Ladas, 2008; Ahmad et al., 2013; Oza et al., 2013; Corona and Pani, 2013]. Lean thinking is supported by many practices and techniques recommended by Agile methods [Poppendieck and Poppendieck, 2003; Hibbs et al., 2009]. Extreme Programming has been considered to embrace Lean thinking, but has also been said to emulate craft work rather than Lean production [Middleton and Sutton, 2005; Lane et al., 2012]. Lane et al. [2012] consider Lean software development as a broader concept than Agile, encompassing an overall business perspective, asserting that it should inform the creation and application of software development methods. When organisations implement Lean and Agile, the two are merged into a single methodology that has traits from both parts. It therefore makes sense to consider the two approaches together as Lean-Agile software development as we do in this thesis.

Neither Lean or Agile on their own, nor the Lean-Agile combination, are unproblematic. Saruta [2006] notes that Lean thinking relies on a deeply ingrained corporate culture that may be detrimental to employee well-being and cohesion of work groups. A critical analysis of the discourse of key methodological contributors to Agile software development reveals values that are expressed in practice rather than present in foundational documents [Lawrence and Rodriguez, 2012]. Enlightenment – valuing knowledge and insight – was found, but also power – valuing possibly coercive influence to affect policies. Values of wealth and skill were less expressed, while rectitude, respect, affection, and well-being were only weakly expressed. Lawrence and Rodriguez [2012] interpret this as a legitimisation strategy to improve diffusion and industry adoption of Agile. Conboy [2009] notes that Agile software development was initially driven primarily by practitioners and consultants. A lack of rigorous conceptual studies resulted in a fragmented understanding of Agile, unclear definitions, and even direct contradictions. Dybå and Dingsøyr [2008] have called for more empirical

research on the core ideas in Agile software development. Dingsøyr et al. [2012] note the need for theory-based research to separate true innovations in Agile practices from reformulations of old ones. As Lean-Agile software development is adopted, or considered for adoption, by software development organisations, it is of great importance to examine how it impacts the daily life of software developers. This motivates the first two research questions posed in Chapter 1.

2.2.4 Global, Distributed, and Open Source Software Development

Global Software Development (GSD) refers to a mode of development where development tasks are distributed to sites across the world. GSD has been motivated by the need to access scarce, skilled resources in a cost-effective way regardless of location; the business advantages of market proximity; the quick and dynamic formation of virtual corporations and teams to exploit market opportunities; the pressure to decrease time-to-market by doing round-the-clock development using time zone differences; and the need for flexibility to take advantage of merger and acquisition opportunities [Herbsleb and Moitra, 2001]. As more companies take GSD into use, an increasing number of developers and software development teams participate in global software development [Mockus and Herbsleb, 2001; Holmstrom et al., 2006; Šmite et al., 2010].

A trait of GSD that is of primary relevance here is the concept of virtual teams. Virtual teams are small, often temporary groups of geographically, organisationally, temporally, and culturally dispersed knowledge workers who use information and communication technologies to accomplish one or more organisational tasks [Powell et al., 2004; Ale Ebrahim et al., 2009]. Virtual teams are often assembled on demand and can be short-lived. Due to their dispersion and reliance on technology for communication, virtual teams face particular challenges in addition to those encountered by co-located teams. Many of these challenges revolve around issues of communication and coordination [Herbsleb and Mockus, 2003a; Espinosa et al., 2007].

Open Source Software (OSS) is attractive to companies as a source of low-cost innovation and productivity [Grand et al., 2004; Bonaccorsi et al., 2007; von Krogh and Spaeth, 2007], similarly to GSD. OSS development is difficult to define, as it includes a multitude of aspects, including licensing, mode of development, and community aspects. OSS can be thought of as a movement that promotes unencumbered access to software source code, partly because of ethical reasons (e.g. [Mingers and Walsham, 2010]) and partly because of the belief that it leads to better development outcomes

(e.g. [Raymond, 2001]). OSS projects can be very loosely structured and a large portion of contributors can be volunteers, but there may also be a significant portion of paid, employed contributors [von Hippel and von Krogh, 2003]. OSS has always had a strong relationship with commercial companies, and many OSS projects are nowadays driven primarily by paid employees in companies. For example, "well over 80% of all [Linux kernel] development is demonstrably done by developers who are being paid for their work" [Corbet et al., 2015]. However, OSS projects vary in this regard.

OSS projects are attractive cases for research due to the availability of data. Both in-depth analyses of exceptional projects (e.g., Mockus et al. [2002]; Capiluppi and Izquierdo-Cortázar [2013]; von Krogh et al. [2005]; Alali et al. [2008]) and census-like analyses of large numbers of projects (e.g. Capiluppi et al. [2003]) have been performed. Despite a growing body of research, and the potential benefits of OSS, guidance on how to manage OSS projects is still scarce in many respects [Crowston et al., 2008], perhaps due to the diversity of projects. Some studies have pointed out the importance of mentoring for sharing and reuse of knowledge [von Krogh et al., 2005; Sowe et al., 2006; Crowston et al., 2008; Sowe et al., 2008].

2.3 Managing People in Software Development

Organising software development is an important part of ensuring a high-quality and timely delivery of the individual components of a software product and the product as a whole. Managing the software development workforce is another relevant challenge. DeMarco and Lister [2013] note that "companies that sensibly manage their investment in people will prosper in the long run". Managing the workforce is important for knowledge and experience management. Seaman et al. [2003] note that the goals of experience management include such abstract notions as improving development processes and making development work more productive and satisfying. They also list a number of more practical and concrete reasons for pursuing it, such as loss of important information due to employee turnover, slow integration of new employees, bottlenecks due to a limited number of experts with important specialised knowledge, and unnecessary reinvention due to lack of knowledge of what has already been done in other parts of the organisation. The ultimate goal of people management is both to maintain a high level of performance, but also to maintain employee well-being and thus sustaining performance in the long run.

2.3.1 Software Development Teams, Performance, and Success

High-performing teams have been defined as those that outperform "all reasonable expectations as well as all other similarly situated teams" [Katzenbach and Smith, 1993]. Teams can offer greater adaptability, productivity, and creativity than any single individual [Salas et al., 2005; Gladstein, 1984; Hackman, 1987]. However, simply grouping skilled individuals together is not enough to gain the benefits of teams [Hackman, 1998].

Team performance research aims to improve the work outcome of teams. Models of team performance can help to describe team performance in specific cases or explain what leads to high or low performance in general. Software development performance can be thought of as a prerequisite for success. On the technical level, performance is related to quality, which can be operationalised as correctness, maintainability, or computational performance. On the level of software projects, success means timely delivery of the right product within budget constraints. The notion of "right product" refers to quality from the perspective of use. In this sense, the goal of software development is to construct software products that fulfil requirements for usability. Usability, or *quality in use*, can be defined in terms of effectiveness, efficiency, and satisfaction for the user [Earthy et al., 2001]. Product attributes, such as understandability and learnability, contribute to usability, as do technical attributes internal to the product, such as computational performance and correctness.

On the level of teams, the relationship between team performance and project success is unclear and can be understood in different ways [Freeman and Beale, 1992; Agarwal and Rathod, 2006]. The number of productivity factors ranges in the hundreds or thousands [Endres and Rombach, 2003], and it is not known which ones ultimately have the highest impact on outcomes. Also, software engineering lacks a sound measure for success [Ralph et al., 2013], making it difficult to even conceptualise the relationship. Success includes not only meeting schedules and making profits, but also employee well-being and public impact [Ralph and Kelly, 2014]. Software product managers have been observed to be dissatisfied with available means of evaluating performance [Cedergren and Larsson, 2014]. Although several team performance models exist in other disciplines, they may not be directly applicable to software development teams [Dingsøyr and Dybå, 2012].

Rather than being a choice among a set of readily defined performance factors, some research suggests that performance (and, by extension, success) should be seen as a dynamic process of negotiation and deliberation, echoing the corresponding design paradigm, as mentioned earlier. Perfor-

mance can then be understood in many different ways depending on the viewpoint [Freeman and Beale, 1992; Adolph et al., 2012; Ralph and Kelly, 2014], and viewpoints may conflict [Ikonen et al., 2011; Kettunen, 2013; Ralph and Kelly, 2014].

2.3.2 People CMM

Since software development is an inherently human activity, the most important asset for software development organisations are employees. People CMM is an example of a framework which can be used to manage human resources in a software development organisation. People CMM belongs to the Capability Maturity Model Integration (CMMI) family of models. They are collections of best practices that allow a software organisation to assess the status of their processes in terms of maturity levels and identify potential improvement areas [CMMI Product Team, 2010]. CMMI is based on the well-known Capability Maturity Model (CMM), which in turn is based on an earlier framework by Humphrey [1988; 1989].

While other CMMI models focus on improvement of work processes, they do not provide specific guidance for the development and improvement of management processes for the workforce itself. People CMM aims to increase the workforce capability of an organisation [Curtis et al., 2001]. It is motivated by the need for a workforce "with the knowledge and skills to make rapid adjustments and the willingness to acquire new competencies" [Curtis et al., 2001]. People CMM is positioned as a framework for implementing advanced practices for human capital management, such as workforce training, formal mentoring programs, and improvement of information sharing. It also supports team-based work design, helps to clearly communicate the organisation's mission, and can be used in situations where the company needs to downsize and restructure its workforce.

People CMM views workforce practices as regular organisational processes that can be continuously improved in the same way as other work processes [Curtis et al., 2001]. Like the CMM and CMMI models, People CMM provides a roadmap which an organisation can use to improve itself by transforming its work practices. People CMM consists of five maturity levels, which follow those established by the CMM framework: initial, managed, defined, predictable, and optimising. At each level, a new system of practices is added to the ones on the previous level. Each successive step raises the level of sophistication of the workforce development practices in the organisation. Table 2.1 summarises the maturity levels.

People CMM illustrates how software development organisations can differ with respect to how they manage their workforce. Workforce management

Table 2.1: People CMM Maturity Levels [Curtis et al., 2001].

Level	Summary	Process Areas
1 – Initial	Inconsistent management	—
2 – Managed	People management	Compensation, Training and Development, Performance Management, Work Environment Communication and Coordination, Staffing
3 – Defined	Competency management	Participatory Culture, Workgroup Development, Competency-Based Practices, Career Development, Competency Development, Workforce Planning, Competency Analysis
4 – Predictable	Capability management	Mentoring, Organisational Capability Management, Quantitative Performance Management, Competency-Based Assets, Empowered Workgroups, Competency Integration
5 – Optimising	Change management	Continuous Workforce Innovation, Organisational Performance Alignment, Continuous Capability Improvement

and software processes combine into complex systems which have an impact on developers' working life both in the short and the long term. As can be seen, higher levels of maturity in workforce management are associated both with increased sophistication in development and operations processes, but also with softer issues such as culture, competency development, mentoring, empowering work groups, and continual improvement on all levels of the organisation, including the individual level. The software development process is thus interlinked with workforce management and with the overall process of continual development and renewal of the organisation.

2.3.3 Onboarding

Onboarding, or organisational socialisation, refers to processes that help newcomers become integrated members of their new organisations [Bauer and Erdogan, 2011]. As part of onboarding, new members learn the knowledge, skills, and behaviour they need to succeed and be productive in their work. Human guidance has been found to be the most important factor in helping newcomers understand source code and software development project issues [Dagenais et al., 2010]. Mentoring is one such form of human guidance, and it has been described as a fundamental knowledge transfer mechanism in companies [Swap et al., 2001]. It is also present on level four of the People CMM framework (see Table 2.1).

While onboarding is well understood in regular company settings, onboarding in virtual team environments, particularly in OSS communities, is less well known. However, it is of prime relevance in such environments. For example, Ducheneaut [2005] found that joining an OSS project requires newcomers to go through a complex socialisation process, in which they need to present themselves as skilled and well versed in software development in order to be accepted as developers. The process varies between OSS projects, but some common characteristics are the relative lack of documented procedures compared to corporate onboarding programs, the lack of or volunteer nature of persons guiding newcomers (e.g. mentors), and primary reliance on newcomers themselves to take the initiative and prove their ability to contribute.

2.4 Human Factors in Software Development

Numerous studies suggest that managing human resources is of key importance to software organisations, and some even show that other factors have comparatively little impact on cost or outcomes. Human factors are among the most important to consider in software development [Curtis et al., 1988].

Avison et al. [1999] note that "failure to include human factors may explain some of the dissatisfaction with conventional information systems development methodologies; they do not address real organizations". In this section, we discuss the role of human factors in software development. We consider research on cognitive, conative, and affective aspects, and present some fundamental theory on values. Finally, we discuss software development as an experience.

2.4.1 The Role of Human Factors in Software Development

Previously in this chapter, we have shown a selection of important technical concerns and managerial means to organise software development – addressing the product and process aspects. However, people remain the most important factor. Based on more than sixty software development project studies, Boehm [1981] concluded that the largest source of opportunity for improving productivity can be found in personnel attributes and human relations activities. In a study of several professional software development teams, the use of production methods, such as software methodologies and automated development tools, could not explain the variance in software product quality or team performance [Sawyer and Guinan, 1998]. The same study found that social processes, such as the level of informal coordination and communication, intragroup conflict-solving ability, and the degree of supportiveness among team members, can account for a quarter of the variation in product quality. Many studies have demonstrated that individual abilities and social factors in teams are significant cost drivers for software engineering projects, often exceeding all other factors [Boehm, 1981; Cockburn and Highsmith, 2001; DeMarco and Lister, 2013; Sawyer and Guinan, 1998].

Jacobson and Seidewitz [2014] suggests that software engineering is a mixture of practices largely adapted from other disciplines: "project management, design and blueprinting, process control, and so forth". They question whether the engineering paradigm makes sense at all. Sawyer and Guinan [1998] find it paradoxical that software development teams are brought together to create variability, while production methods are used to reduce it, and that team-level social processes may predict team performance better than production methods. They suggest that software methodologies should be designed as socially-centred methodologies rather than production-centred ones. Lean and Agile methods may be considered attempts at taking such a perspective.

Improvements in social and human factors should yield large returns in practical software development, but the research on such concerns remains

fragmented [Warfield, 2010; Amrit et al., 2014] despite documented insights into such factors from at least the late 1960s (see e.g. Rouanet and Gateau [1967]). Throughout the history of software engineering, researchers and practitioners alike have agreed that the software developers involved in developing software are a key factor in determining project outcomes and success (e.g., Weinberg [1971]; Brooks [1975]; Boehm [1981]; Sawyer and Guinan [1998]; Cockburn and Highsmith [2001]; Feldt et al. [2008]; DeMarco and Lister [2013]). However, the focus in software engineering has been mainly technical. Glass [2002] found that most software engineering research has tended to be "diverse in topic, narrow in research approach and method, inwardly-focused from the viewpoint of reference discipline, and technically focused (as opposed to behaviourally focused) in level of analysis", and others have made similar findings (e.g., Shaw [2003]; Sjøberg et al. [2005]; Zannier et al. [2006]). Compared to the amount of technically-oriented studies, there are relatively few studies that focus on the particular conditions of professional software developers in their work [Wallgren and Hanse, 2007; Feldt et al., 2008; Lenberg et al., 2014]. In a recent systematic literature review, Lenberg et al. [2015] find that only a few human factors concepts have been covered by existing research, and multiple levels of analysis are seldom used in a single study.

One of the reasons that research on human aspects of software development is still in its infancy may be that there is no generally accepted framework to guide such research. Instead, researchers borrow from other disciplines, such as psychology, social psychology, sociology, and economics, to increase their methodological toolkit and enable access to research questions that are otherwise unreachable [Warfield, 2010]. However, Curtis [1984] notes that one source of limitations for psychology of programming studies stems from the difficulty of integrating "theory and data from the mixture of paradigms borrowed from psychology", and suggests that research should focus on the cognitive science paradigm. An example of a study following this suggestion is that of Hansen et al. [2012] which relates computational models of the human mind to measurements of cognitive complexity in systems.

Other researchers have also observed the need for multidisciplinary research into software development. Herbsleb [2005] notes that computer science is necessary but not sufficient for investigating many of the issues in software engineering, and points to examples of cognitive and organisational theories to complement it. Lenberg et al. [2014] propose that research on human aspects in software development should use models and methods from organisational psychology and behavioural economics to study "behavioral

and social aspects of software engineering activities performed by individuals, groups or organisations". The unit of analysis in such studies can be on the individual, group, or organisational level. They further propose that the field should proceed by identifying relevant psychological constructs on each level. The model is similar in many respects to the one proposed by Curtis and Waltz [1990], with the exception that the latter is more fine-grained and takes into account not only the individual, group, and organisation, but also places these in an environmental context, such as a market or business milieu. Lenberg et al. [2015] argue that the aspects group and organisation should be broadly understood, to include different types of teams and other task-focused groups, as well as loose organisations such as communities.

Amrit et al. [2014] note that many diverse concepts and theories are included in the area of human factors, but that the most studied human factors in software engineering research include coordination [Kraut and Streeter, 1995; Amrit and van Hillegersberg, 2008] and collaboration [Herbsleb and Mockus, 2003a; Mockus et al., 2002; Amrit and van Hillegersberg, 2010] in the development process, trust [Sabherwal, 1999], expert recommendation [Cubranic et al., 2005], program comprehension [Brooks, 1983], knowledge management [Rus and Lindvall, 2002; Espinosa et al., 2007], and culture [Carmel and Agarwal, 2001]. Lenberg et al. [2015] find that in terms of SWEBOK [Bourque and Fairley, 2014] knowledge areas, most human factors publications concern software engineering professional practice, software engineering management, software engineering economics, software engineering models and methods, and software engineers themselves without connection to a specific knowledge area. Amrit et al. [2014] further note that while software engineering research has developed theories of its own which include or address human factors, it can also be beneficial to borrow theories from other disciplines. They argue that existing theories from the social and behavioural sciences can inform the design of new empirical studies. Such theories may contribute to the development of frameworks that can be useful for organising existing concepts in the software engineering literature, or to develop and evaluate knowledge claims, e.g. hypotheses. Also, borrowing theories may increase the explanatory power of the theory because its constructs and propositions are better explained [Hannay et al., 2007; Sjøberg et al., 2008].

In this thesis, we attempt to organise human factors in software engineering into a framework that allows us to reason about how developers experience the development activity. The overall frame of reference for human factors in this thesis may be considered to be social psychology with some additional material from related disciplines. The classical conceptu-

alisation of the human mind as cognition (thinking, memory), conation (motivation, volition), and affect (emotion, feelings, mood) is borrowed from the philosophy of psychology, and is present in some form in all modern individual psychology. Situating the individual human in a social context, and the concept of human values, are applications of social psychology. The concept of onboarding is borrowed from organisational psychology, which may be considered closely related to social psychology. Human experience can be considered the main focus of phenomenological psychology. By combining these aspects, we obtain a theoretical lens for examining the main research problem and the first research question. We view the product, process, and people aspects of software development through this lens.

2.4.2 Cognitive Aspects

Cognition refers to mental processes involving knowledge, including attention, remembering, and reasoning, and also to the content of the processes, such as concepts and memories. The cognitive aspects of software engineering are related to attention, memory, problem-solving, and decision-making. Research exists on, e.g. skill [Bergersen et al., 2011; Bergersen and Gustafsson, 2011; Sudhakar et al., 2011], knowledge [Liang et al., 2007], competence [Hall et al., 2007], and logical reasoning [Calikli and Bener, 2010]. Early work in this area focused on programming language design and the process of learning programming languages. Researchers examined how humans form mental models of software code, and how these models relate to language, semantics, and other information structures in the human mind [Hoc et al., 1990]. However, even in early work, larger units of observation have been considered, such as teams and organisations [Weinberg, 1971].

One important consideration is the cognitive aspects of software itself. Wang and Patel [2009] argue that software has several inherent cognitive constraints that may be considered fundamental. Software is *intangible*, because it is an abstract rather than physical artefact. Similarly, artefacts describing the process of software development are also intangible. Software is *complex* because of its intricate internal and external connections, which makes it difficult express or cognise as a whole. This includes the complex relationships between the software and the internal and external data on which it operates, the behaviour of the software when interacting with inputs and outputs, and the complexity of its interactions with the environment, such as other (software) systems and users. The *indeterminacy constraint* means that a large portion of software behaviours and event sequences are only determinable at run-time. Comprehending the space of possible run-time sequences can be extremely difficult due to the large

Table 2.2: Key knowledge concepts in the cognitive sciences. Adapted from Robillard [1999].

Key Knowledge Concept	Viewpoint
Procedural/Declarative	Knowledge nature of content
Schema	Knowledge internal structure
Proposition	Formal knowledge representation
Chunking	Representing unit of knowledge
Planning	Managing knowledge structures

number of possibilities. Software is also highly *diverse*, with a multitude of types, styles, architectures, behaviours, platforms, application domains, and other variations, which multiply the requirements for knowledge for software developers. Software is *polymorphic* in the sense that its design and development are open-ended problems, where solutions and paths to solutions are diverse, and demonstrating their optimality is hard. Thus weighing the possible options is a difficult cognitive task. Wang and Patel [2009] further argue that it is inherently difficult to express the architecture and behaviour of a software system. Due to the *inexplicit embodiment* of software, new notations, such as formal notations, diagrams, and programming languages, are needed in order to make it possible and easier for humans to handle it. Finally, the authors argue that while some quality aspects of software may be quantified, *unquantifiable quality measures* still exist, such as design quality, usability, and reliability. Wang and Patel [2009] argue that the productivity of software developers is constrained by their cognitive apparatus, i.e., the brain. Before software can be written, a representation must be generated in the brain. Therefore, a cognitive metaphor for software development may be closer to the actual nature of software development than a metaphor based on manufacturing or production.

On the technical level, software development research and practice has developed many concepts to manage the cognitive complexity of the development activity, such as information hiding, modularity, and objects [Robillard, 1999]. These can be described in terms of cognitive processes and concepts as shown in Table 2.2. *Procedural knowledge* is "know-how" which is acquired in practice from interacting with the environment. *Declarative knowledge* is based on facts regarding properties of objects, persons, and events, and their relationships. Whereas procedural knowledge is often hard to communicate, declarative knowledge tends to be easy to communicate. Declarative

knowledge can be episodic – related to sequences of events – or semantic – related to a specific topic, such as the meaning of words and concepts.

The concept of knowledge *schema* arises from the assumption that knowledge is stored in memory according to a predetermined structure. Such schemas are collections of topical and episodic knowledge. The notion of schema is important from a cognitive standpoint because it does not only structure knowledge in memory, but also structures the acquisition of knowledge so that a certain interpretation occurs while knowledge is being acquired. Schemas are individual, but people may develop similar schemas due to, e.g., a shared cultural, professional, or educational background. Nevertheless, schemas explain why individuals may think and reason differently even when given the same information as a starting point.

A *proposition* is the smallest discrete unit of knowledge. Propositions represent well-specified information, preserve the meaning but not the form of a statement, and naturally support reasoning and inferences. In software engineering, propositional knowledge is mainly applicable to well-defined problems. As noted, software design problems are usually ill-structured, or polymorphic [Wang and Patel, 2009], or at least a mixture of ill- and well-defined problems [Robillard, 1999].

The human cognitive process can only handle a finite amount of information at a time. The concept of *chunks* illustrates the limitation. Chunks are general and independent of the information content being processed, and are a measure of the unrelated knowledge that can be processed naturally. With accumulated expertise, humans learn to handle increasingly large units of information, which can eventually be processed as a single chunk, thus making more efficient use of the available cognitive capacity. Robillard [1999] mentions encapsulation, information hiding, modularisation, and abstraction, as examples of software development approaches that deal with the chunking phenomenon.

Finally, *planning* is a higher-order cognitive function that involves the integration of a large amount of information from many interrelated knowledge domains. Plans are needed in order to overcome the limitations of working memory. Plans organise knowledge according to various criteria and guide the tasks to be done by the mind. Plans have a heuristic nature, guiding mental activity towards the most promising options at any given time. They optimise the use of memory by keeping only critical information directly available and abstracting the rest. Plans enable a higher level of control than what can be derived from the detailed activity being processed at present. Expert plans have been shown to be hierarchical with multiple levels and

explicit relationships between levels, based on basic schema recognition, and internally well connected [Hoc, 1988].

Since so much of the software development activity is intangible and exists only as a representation in the individual software developer's mind, a pertinent problem is how software developers communicate and share this otherwise hidden information. Several studies have explored cognition in team environments. Robillard et al. [1998] use coding of video sequences to show that technical review meetings are composed of three types of cognitive activities: review, synchronisation, and elaboration of alternative solutions. They suggest that synchronisation enables meeting participants to ensure a common representation of design solutions and evaluation criteria. Half of the meeting time was spent on this activity, highlighting the difficulty of sharing mental models. More generally, Misra and Akman [2014] explore cognitive aspects of various types of meetings in the software development process, ranging from meetings related to requirements and design to meetings related to implementation, review, and testing. They propose a cognitive evaluation model which can be used to support replacement of some meetings with other tools and techniques.

Part of the cognitive difficulties in software development relates to the diversity of knowledge that must be processed, and the complex interactions with the environment that occur during development. Hazzan and Dubinsky [2003] argue that software developers are required to "think on various levels of abstraction", moving between levels to consider both a global view of the system being developed as well as a local, detailed view of its parts. For example, a global view is needed to understand customer requirements, while programming a specific program part requires a local view. Moving across this spectrum of viewpoints is challenging: it is not always obvious to developers when and how to shift the perspective.

Although cognitive factors have been widely studied, many open questions remain. Apart from fundamental questions regarding, e.g., reading and understanding source code, the interaction between cognitive factors and the organisation of software development work has not been thoroughly explored in the software engineering literature. For instance, it is not well known how changes in the software development process interact with cognitive processes on the individual level.

2.4.3 Conative Aspects

Conation refers to the mental process that activates or directs behaviour and action [Hilgard, 1980]. Motivation is the conative construct that has received by far the most attention in software engineering research. Motivation

has been examined using case studies [Baddoo et al., 2006], by examining differences between developers and project managers [Sharp et al., 2007], and the research has been summarised by systematic literature reviews [Beecham et al., 2008]. More recent work proceeds towards explanatory theories of motivation [Franca et al., 2012b,a], and specific examinations of influence factors on motivation among software engineers [Sach and Petre, 2012]. Other aspects of conation have been studied in connection with software engineering education [Lingard and Berry, 2002] and under the heading of commitment, although this construct is ambiguous and includes both cognitive and affective aspects in addition to conative ones [Abrahamsson, 2001]. Despite the growing body of research on motivation in software development, research results have so far been inconclusive regarding how and by what software engineers are motivated, and what the benefits of motivating them are [Beecham et al., 2008; Franca et al., 2011].

Motivation is an important factor in software development methodology adoption. Riemenschneider et al. [2002] studied the adoption of a custom-made software development methodology by more than one hundred software developers in a large organisation. They examined how five models of individual acceptance of information technology tools could explain the variance in developers' intentions to use the methodology. They found that the prospects for successful deployment may be severely undermined if the methodology is not perceived as useful by developers. They found that methodology adoption intentions are driven by an organisational mandate, by the compatibility between the methodology and how developers perform their work, and by coworker and supervisor opinions towards using the methodology. However, they emphasise that organisational mandate alone is not sufficient for successful adoption, since although the method may have long-term benefits for the organisation, it may have short-term negative consequences for individuals. Also, compliance with subjective social norms can prevent adoption of a methodology even though it may be useful and compatible with existing work practices. This illustrates the importance of conative factors in software development work, and shows that while these factors have an outcome on the individual level, they are actually mediated on the group level.

Work environments using an Agile software development approach have been found in many studies to be particularly motivating for software developers (e.g., Syed-Abdullah et al. [2005]; Mannaro et al. [2004]; Melnik and Maurer [2006]). However, other studies have found that this may depend on high technical, social, and communication skills (e.g., Coram and Bohner [2005]; Highsmith [2001]). Work environments using one approach

may be difficult and demotivating for developers who are used to a different approach. For example, Beecham et al. [2007] found that eXtreme Programming (XP) supported developers from a traditional, heavyweight development environment in several ways, such as helping to track progress and encouraging communication through visual display of story cards, regular meetings, and the planning game. However, it was found to be at odds with developers' motivational needs in other ways. The difficulty of identifying individual contributions among the highly collaborative activities can be demotivating for developers wanting to demonstrate their performance in the hope of promotion or career progression. Pair work can demotivate the more passive person as the more dominant person may dictate the path for development [Law and Charron, 2005]. This in turn may lead to a sense of unfairness, as experience, education, skills, and seniority are not matched by benefits gained from the organisation, such as salary, recognition, and opportunity for achievement. Finally, XP may be perceived as repetitive, with developers demotivated by lack of variety and eventual dependence on pair work over individual work. While these observations were made on particular Agile methods, their relevance concerns more general human factors. As Agile methodology is increasingly popular in the software industry, and is used in many educational environments to teach students, demotivation due to non-Agile aspects of development environments could become more common.

Although motivation has received much interest in the software engineering research community, other conative factors have not been extensively studied. Volition, intent, and impulse are examples of conative factors that are not extensively researched in the this field. Furthermore, conative factors have important connections to social settings; motivation and desire often occur as a response to the social environment and are linked to affective outcomes in the individual. Conative factors are thus an important part both of what goals are set in software projects and how software developers experience their work.

2.4.4 Affective Aspects

Affect is often used in psychology as the most general term for the emotion domain. It can refer to the entire cascade of cognitive, neurological, physiological, motivational, and behavioural components, as well as the feeling component. Affect and cognition are sometimes considered to be distinct and competing processes [Isen, 2004]. However, rational thinking does not preclude feelings, and while it is possible to give too much weight to feelings in decision-making, it is also possible to do the opposite. Affect,

cognition, and motivation occur together and influence each other [Isen, 2004]. Many studies have shown that especially positive affect may improve the decision-making process by enabling people to take more factors into account and to integrate them while making decisions (e.g., Estrada et al. [1997]; Isen et al. [1991]; Isen [2004]). Mild positive affect has been found to promote intrinsic motivation and to lead to more enjoyment of an activity without negative impact on work behaviour [Isen and Reeve, 2005], and to pro-social behaviour and flexibility in social perception [Isen, 2004].

Developers have been shown to experience several emotions in their work, and these emotions have been shown to change over time [Shaw, 2004]. Programming tasks such as debugging can be influenced by moods [Khan et al., 2011]. Enthusiasm [Wrobel, 2013], and emotional valence and dominance [Graziotin et al., 2013, 2014a,b,c], can have a positive effect on performance. Frustration is a risk factor with possible negative influence for performance [Wrobel, 2013]. Affect is sometimes discussed or implied in research on cultural aspects of software development. Hertzum et al. [2007] suggest that developers may experience frustrating systems differently than users, and that culture plays a role in how people perceive the concept of usability.

Software development work includes many sources of both positive and negative affect. As with the conative factors, there is a relative shortage of software engineering research addressing affective factors directly. Part of the difficulty lies in explaining what consequences affects may have in software development work, why they are unavoidable, and in suggesting what to do about them. Among the first steps is therefore to clarify their role for development.

2.4.5 Values

Values are deeply rooted, abstract motivations that guide, justify, or explain attitudes, norms, opinions, and actions [Schwartz, 1992]. They prime attitudes and guide the selection of behaviours and events [Rokeach, 1973; Feather, 1996; Schwartz and Bilsky, 1990, 1987]. They can be seen as arising from universal requirements for human existence, individual biological needs, the preconditions for coordinated social action, and the survival and welfare needs of groups [Schwartz and Bilsky, 1987, 1990]. Since values can guide the selection of behaviours and evaluation of events [Schwartz and Bilsky, 1987], they impact organisational outcomes [Weeks and Kahle, 1990], business ethics [Feather, 1995; Mumford et al., 2003], and managerial behaviour [Smith et al., 2002].

Values are an example of a social psychological construct where cognition, conation, affect, and sociocultural processes are integrated. Values comprise concepts or beliefs that serve as standards for desirable end-states or behaviours. They are deeply linked to affect and the self-concept [Rokeach, 1973]: when a value is activated, a corresponding feeling also occurs. Values are thus an important component in motivation [Rokeach, 1979], and influence performance and overall work experience. Values do not focus on specific objects or situations [Rokeach, 1973, 1979], but rather explain behavioural patterns over longer periods of time rather than behaviour in specific situations [Bond et al., 1992]. However, values have practical consequences in many everyday situations. They are the most abstract type of social cognition used to guide general responses to classes of stimuli [Kahle, 1996].

In Section 2.2.3, we discussed how Lean-Agile software development rests on a foundation of values. Such underlying assumptions may become visible in practical software development in different ways. Values are among the soft factors influencing software development teams [Sudhakar et al., 2011]. Incompatible personal values can lead to conflict and lower performance [Liang et al., 2007]. In this thesis, we contribute to the body of knowledge regarding values in Lean-Agile software development by addressing a specific research question on the matter.

2.4.6 Experiencing Software Development

There are many fundamental works and theories that attempt to conceptualise and explain the nature of "experience". Many of them originate in philosophy or psychology. User experience (UX) is a sub-field of human-computer interaction that considers the experience of users as they interact with a system or product [Hassenzahl, 2004; Roto et al., 2011]. However, experiencing software development occurs in the context of development rather than the context of use, and so UX cannot be applied as such to all aspects of the context of development. This thesis aims to provide a theoretical framework of developer experience that addresses this problem.

Floyd [1992] considers software development as reality construction: a social cognitive process in which individuals interact to perform a particular kind of design activity where a problem area is grasped and an "appropriate solution worked out and fitted into human activities of meaning [Floyd, 1992, p. 87]. From the perspective of cognition, this highlights the social-cognitive aspect of software development and the nature of *constructing* the reality in which the tasks of software development should be performed.

Forlizzi and Ford [2000] distinguish between three types of experience. First, *experience* is a "constant stream that happens during moments of consciousness". It is internal to the person and becomes manifest in the form of self-talk. Second, *an experience* requires consciousness combined with the external world as a source of influences. A certain set of influences combine into an entity describing an episode of life with a beginning and an end – an experience. Third, *experiences as stories* refer to externalisation of experiences as narrated stories. Stories are narrative vehicles that condense experiences and help people remember them. Stories communicate experiences and allow us to share them with others. Forlizzi and Battarbee [2004] continue the conceptualisation of social experience and add co-experience to the list of experience types. Co-experiences are episodes which are experienced together with other people and where the social context is an important part of the experience.

McCarthy and Wright [2004] base their UX framework on a constructivist stance: "people actively construct or make sense of experience – reflexively and recursively – in a way that seems to fold back into the experience itself". This framework is descriptive. In contrast, Hassenzahl's [2004] outcome-centric model of UX aims to guide actual design. In this model, *product character* is a combination of features, particular and unique to a certain product, that summarise the product's attributes, reduce cognitive complexity for the user, and provides affordances that allow the user to form possible strategies of how to further interact with the product. These product characteristics have both pragmatic and hedonic attributes. Designing a product character involves deliberately combining product features and communicating the intended character to users before and during product use. For the user, experience of product character becomes apparent through interaction.

An important insight from Hassenzahl's work is that the intended product character is not the same as the experienced product character. There is no guarantee that users will perceive and appreciate the product in the way the designers intended. This has potential implications for the design of software development methods and environments: the experienced character of a method or environment is probably not the same as the intended character. Developers will experience them differently from what designers have intended. Therefore, it is important to ask how developers experience their work, and what value they see in a method or environment. This may answer questions about the degree to which people should be made to adapt to a process or vice versa. Kroeger et al. [2014] suggest a model of quality for software engineering processes. The model suggests

that practitioners perceive four quality attributes: suitability, usability, manageability, and evolvability. The judgements are influenced by key properties related to the semantic content, structure, representation, and enactment of the process. The model indicates that the attributes correspond to particular organisational perspectives. The differing views may explain role-based conflicts in the judgement of process quality.

Furthermore, methods and environments should consider affordances that invite developers to perform in the desired manner – e.g. to calibrate their emphasis on quality and productivity according to current project needs. This is echoed by research: for example, Conradi and Fuggetta [2002] observes that "software work, like other design work, is not like mechanised or disciplined manufacture" and that "it has a strong creative component involving human and social interaction that cannot be totally preplanned in a standardised and detailed process model." Discipline and creativity must be balanced in software development [Glass, 2006]. Dogan et al. [2011] note that in order to adequately address human factors in system of systems development, processes should allow individuals at lower levels of an organisation to act while also sustaining control and coordination. Means should exist to give individuals responsibility and authority beyond the standard operating procedures and against social and organisational norms and conventions. These are among the prerequisites for both innovation and satisfaction in the workplace.

While UX focuses on the product in order to influence how it is experienced, a particular question for designing experiences for software developers is how to conceptualise the value of experience itself. Csikszentmihalyi [1990] defines the concept of *flow*, which is an optimal experience. Flow occurs when a person's perceived skill is in balance with the perceived challenge of a task. Such an experience is inherently gratifying and self-motivating. Characteristics of flow experiences include intense concentration, merging of action and awareness, absence of reflective self-consciousness, and loss of sense of time. Flow can occur in daily tasks just as well as in creative work. From this perspective, a good software development method or environment includes affordances that help balance perceived skills and challenges, increasing the likelihood of optimal experiences.

Chapter 3

Research Design

Creswell [2009] considers a research design as an overarching framework that guides the research in all aspects of a study, from the philosophical ideas behind the inquiry to the detailed data collection and analysis procedures. The purpose of a design is not only to guide the researcher, but also to enable the audience to understand and evaluate the research and its results. The design involves the intersection of philosophy, strategies of inquiry, and specific methods. This chapter details the research design of this thesis following Creswell's framework and terminology.

3.1 Elements of Research Design

Detailing a research design is important for several reasons. A research design is a plan or proposal to conduct research. An explicit design allows the researcher to situate the research in the proper scientific context. This in turn helps readers to understand the underlying assumptions and limitations of the research, its goals, and its results [Creswell, 2009]. Creswell details three framework elements that constitute a research design:

- Its *philosophical worldview*, which states the assumptions about what constitutes knowledge claims.
- Its *strategies of inquiry*, which are the general procedures of research.
- Its *research methods*, which are the detailed procedures of data collection, data analysis, and writing.

In the following subsections, we briefly describe the three elements.

3.1.1 Philosophical Worldview

The philosophical worldview of a research design consists of claims about what knowledge is (ontology), how we know it (epistemology), what values are related to it (axiology), how it is expressed in written form (rhetoric), and what processes are used to study it (methodology) [Creswell, 2009]. These knowledge claims include researchers' assumptions about how and what they will learn during their inquiry: the paradigm within which the researcher operates. Creswell discusses four schools of knowledge: postpositivism, constructivism, advocacy/participatory, and pragmatism.

Postpositivism refers to a deterministic viewpoint in which causes determine effects with some probability. Its ontology is that of critical realism: objective reality exists, but humans can never reach certain knowledge regarding it. Postpositivist epistemology states that researchers should strive for objectivity, but can only reach results that are true with some given probability. Researchers should strive to refine claims or abandon them in the face of contradictory evidence. Postpositivist methods tend to be quantitative and deal with measurement and numbers.

Constructivism refers to a viewpoint where the researcher relies as much as possible on the participants' views of the phenomenon under study and attempts to uncover the meaning constructed by participants. The researcher also takes part in constructing the meaning. Constructivist ontology is that of relativism: all truth is constructed by humans and situated within history and a social context. Multiple meanings may exist. Constructivist epistemology states that researcher and participants are linked, constructing meaning together. Constructivist methods are generally qualitative.

Advocacy/participatory research contains an action agenda which aims to change the lives of the participants within their context of work or life, and the life of the researcher. A variety of ontological viewpoints can be included. The distinction between researcher and researched is blurred; the researcher participates in the lives of the latter. Advocacy/participatory research may use any methods that are appropriate for accomplishing the action agenda.

Pragmatism is concerned with solving problems through action. Pragmatist ontology is varied, but the emphasis is less on what "truth" is and more on "what works at the time". Knowledge claims arise through actions, situations, and consequences rather than antecedent conditions. Pragmatist epistemology accepts many different viewpoints and works to reconcile them, but does not aim for consensus but rather for a solution of the problem at hand. Pragmatist research may use any methods which are appropriate for solving things in practice.

3.1.2 Strategies of Inquiry

Strategies of inquiry refer to general procedures of research [Creswell, 2009]. These are often expressed on a continuum ranging from quantitative to qualitative, but mixed methods are also possible. Examples of quantitative research strategies include controlled experiments, quasi-controlled experiments, correlational studies, factorial designs, and structural equation models. In qualitative research, some well-known strategies are Grounded Theory, ethnography, phenomenological research, narrative research, and case study research. Mixed methods approaches include strategies for combining both quantitative and qualitative data and embedding one in the other.

Some overarching research procedures provide reasoned solutions that may be considered as templates for strategies of inquiry. They may include specific instructions for all or most steps of the research process. Examples can be found in case study research [Eisenhardt, 1989; Yin, 2009; Runeson et al., 2012], action research [Susman and Evered, 1978; Avison et al., 1999; Stringer, 2014], and design science research [Hevner et al., 2004; van Aken, 2004]. These overarching procedures may include qualitative, quantitative, and mixed methods approaches but may also include a particular logic that guides and structures the inquiry in other ways, such as in the development of research questions, identification of the unit of analysis, or generalisation of results.

3.1.3 Research Methods

Research methods are the detailed procedures of data collection, data analysis, and writing [Creswell, 2009]. Data may be collected through different kinds of measurements, ranging from direct measurement of observable quantities to indirect and possibly subjective questionnaire-based approaches. Data may also be gathered in the field by direct observation without predefined questions, or through interviews with different degrees of structure and open-endedness. Data analysis methods depend on the type of data gathered and the type of inferences that the researcher is seeking.

3.2 Description of Research Design

This thesis utilises a mixed-methods design consisting of individual case studies, some of which are qualitative and others which are quantitative. The design is an exploratory sequential design, with the qualitative parts occurring towards the beginning of the design, and quantitative parts, investigating more specific aspects, occurring towards the end of the design [Creswell,

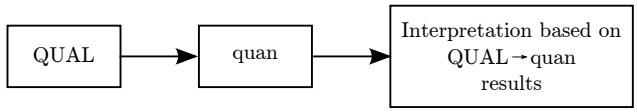


Figure 3.1: Idealised exploratory sequential design. Adapted from Creswell [2009].

2009]. The emphasis is on the qualitative part, and the quantitative part provides additional detail. The final outcome of the research is an analysis of the entire set of findings, and an interpretation of the quantitative parts in light of the qualitative findings. Figure 3.1 shows an idealised exploratory sequential design, while Figure 3.2 shows the design as carried out in this thesis. In this section, we detail the aims and objectives, philosophical worldview, strategies of inquiry, research methods, and research setting of the study.

3.2.1 Aims and Objectives

This thesis aims to advance the empirical knowledge of how software developers experience the activity of developing software. The thesis approaches the problem area with the assumption that software development is an intellectual, social activity in which experience is constructed within and between individuals and is influenced by a multitude of factors that stem from the practice of software development, such as processes, methods, tools, and issues related to organisations and markets. The thesis is exploratory in the sense that it aims to increase the understanding of the phenomenon of interest, but does not begin from a precise problem statement or hypothesis. Its scope is limited to theory generation. Theory testing in the sense of performing statistical tests or controlled experiments, or statistical generalisation to a population at large, are not the primary aims of the thesis, although some of the articles include an element of causal research design. The research questions were introduced in Chapter 1 along with more detailed motivations.

3.2.2 Philosophical Worldview

Traditionally, empirical software engineering has been inspired by the post-positivism of the physical sciences (c.f. Humphrey [1989]; Basili [1996]; Rombach [2011]). The aim is to conduct experiments which codify the relationship between variables into universal or at least contextually valid

laws. While this thesis does not seek to develop such laws, it does consider the object of study to be the external world, not the researcher's personal experience. The researcher must thus turn outward and base the research on observation of different kinds rather than rely on a purely analytical approach – in this sense, the thesis is situated in the field of empirical software engineering research. However, a large part of the phenomena under study in this thesis are not available for immediate observation in the physical world. Many of the antecedents or reasons for observed outcomes are present in the minds of individual actors, and many are socially constructed by the participating actors. Each participant has a "local reality" – a personal, subjective understanding of the meaning of events and actions.

This thesis places emphasis on understanding the individual and socially shared meanings that govern interpretation of events and choice of actions. Thus the main contributions of this thesis can be considered as constructivist knowledge claims, and the focus of the thesis is theory generation rather than verification, as would be the case in a purely postpositivist study. That said, the thesis also aims to take a pragmatic approach to theory: the theory is problem-centred and heavily grounded in real-world practice. Here, the criteria for successful theory is its ability to abstract from individual cases into more general knowledge which is still relevant for practice.

3.2.3 Strategies of Inquiry

The overall strategy for this thesis is a sequential exploratory mixed-methods design, in which a qualitative part is followed by a subordinate, detailing quantitative part [Creswell, 2009]. The mixing point occurs last and is reported in this thesis as the final analysis of the findings in the article set. Articles I and II use a purely qualitative approach while Articles III–V use a quantitative approach. Article II utilises case study methodology with multiple cases. Case studies aim to investigate phenomena in their context [Runeson and Höst, 2009; Runeson et al., 2012]. They are suitable for exploratory and explanatory research questions [Yin, 2009]. Case studies can be used to inductively build theory in many different ways [Eisenhardt, 1989; Eisenhardt and Graebner, 2007]. The remaining articles focus on gaining more detail on selected findings from Article II. Article III uses a quantitative survey approach, while Articles IV and V can be considered quantitative multiple-case studies that observe a practical intervention.

3.2.4 Research Methods

As stated above, both qualitative and quantitative research methods are used in this thesis. In Article I, an initial framework is developed to conceptualise developer experience on the individual level. The article uses existing literature as well as the authors' experience to transfer the notion of user experience to developers and software processes. The method can be characterised as qualitative. Article II uses an exploratory, embedded multiple-case design with a qualitative Grounded Theory method in which the questions of interest and data collection are guided by the initial framework developed in Article I. A rich set of data collected through interviews with professional software developers is analysed. Based on the findings from Article II, Article III uses a survey design with descriptive and inductive statistical techniques. The article uses a quantitative data set with survey responses collected from professional software developers. Articles IV and V are based on a multiple-case design with quantitative analysis of a specific intervention. As part of an educational program, onboarding support, with mentoring as a major component, is used to help student developers enter existing Open Source projects. Their progression is followed quantitatively by collecting a data set of activity metrics which is then compared against a data set representing developers who did not receive systematic onboarding support.

3.2.5 Research Setting

The research reported in this thesis has been conducted in two different settings. The first can be broadly characterised as commercial companies whose products or services are software or software-intensive. In Article II, the setting consists of five Finnish companies utilising Lean and Agile software development methods. In Article III, the setting is similar but includes research participants from a different set of companies and also includes participants from companies which are not located in Finland. The articles describe the samples in detail.

The second setting consists of Open Source Software projects. These projects are highly distributed, have participants from multiple geographical locations, and utilise virtual teams. In this thesis, they are used to represent a modern form of distributed and global software development, with a very lightweight structure – they generally do not use defined or documented processes but rather rely almost exclusively on self-organisation and project-specific cultural norms for coordinating work, as well as tool support which automate certain parts of development and may thus be considered a type

of process automation. Articles IV and V include more detailed descriptions of the projects. It should be noted that in this environment, participants range from novices, such as students, to very advanced developers. The diversity of participants may be considered to be larger than in Articles II and III. The relationships between research settings and articles are shown in Figure 3.2.

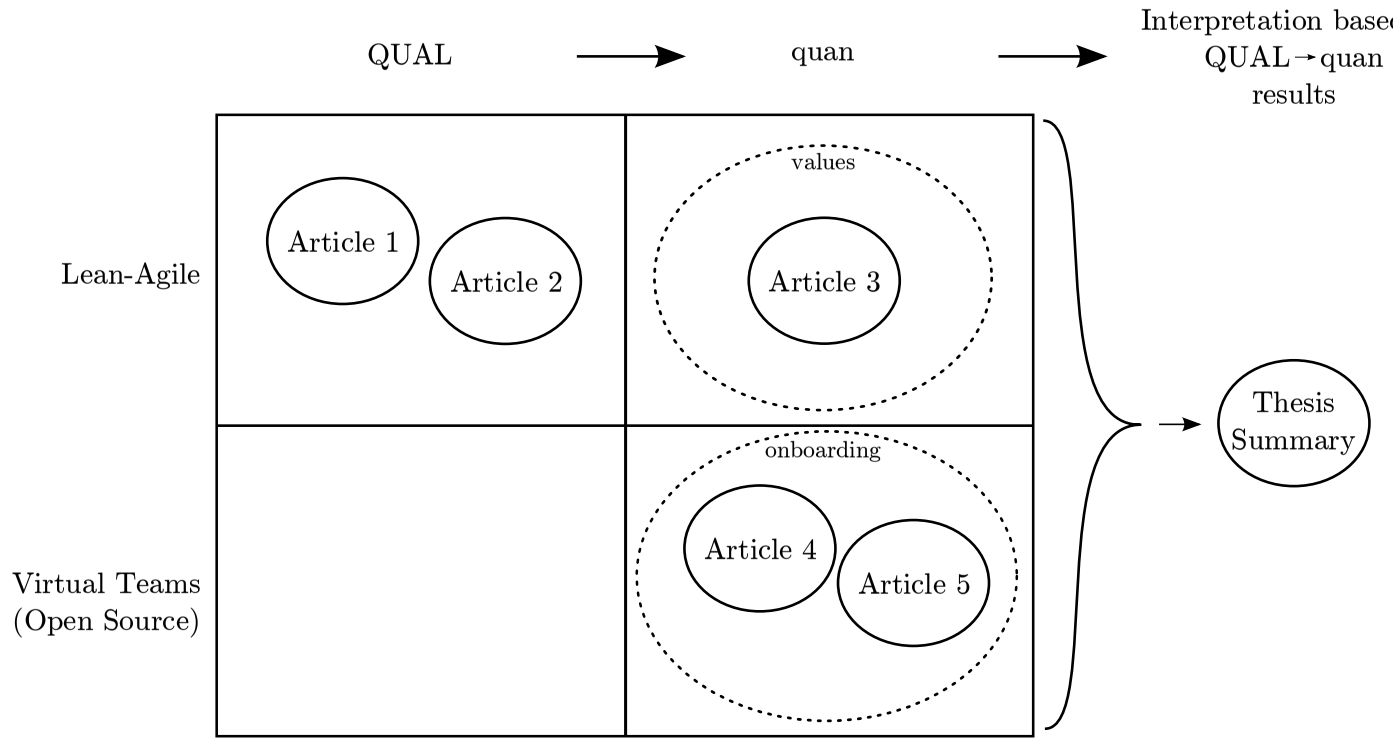


Figure 3.2: Thesis research design: relationships between research settings (left), articles, and phases of exploratory sequential design (top) as used in this thesis. Themes selected for quantitative examination shown as dotted ovals.

Chapter 4

Software Developer Experience

As software development is an inherently human activity, it can be conceptualised in terms of how a human – the software developer – experiences it. An analogy can be made between software developers experiencing the activity of developing software and users experiencing the activity of using software [Article I]. However, there are important differences between the context of software use and the context of software development. The analogy between user and developer experience must be understood as a metaphorical vehicle for intuition rather than a direct mapping. In order to understand developer experience, the construct must be examined in its proper context. In this chapter, we present the results of Articles I–V, answer the research questions posed in this thesis, and summarise the answers as a theoretical framework that addresses the main research problem. The sections of this chapter provide elaborate answers to each question. Table 4.1 provides condensed, summarised answers and acts as an index to the longer answers in the remainder of this chapter.

4.1 Concept and Definition

We address RQ 1 through Articles I and II. Article I reports on an analysis and definition of the developer experience concept, while Article II reports on a study to deepen and elaborate the concept based on interviews with professional developers. The aim of our conceptualisation is to abstract from the large number of human factors variables that play a role in software development, and to create an intuitive model that can be used to draw attention to and reason about how developers experience software development in different situations [Article I]. The conceptual framework is used to focus the inquiry on the developer perspective. The experience in developers'

Table 4.1: Answers to the research questions and contribution to research problem (RP).

	Answer	Details in Section
RQ1	Developer experience can be conceptualised as a combination of cognitive, affective, and conative mental processes that interact with the social and technical environment. See Figure 4.1.	4.1
RQ2	Developers experience team performance in Lean-Agile environments as a continuous process of negotiation on all levels of an organisation. See Figure 4.2.	4.2
RQ3	Developers experience the Lean-Agile value system as consisting of 11 value dimensions. See Table 4.3.	4.3
RQ4	Developers in Open Source projects can be supported through an onboarding process with mentoring by experienced members of the project's developer community.	4.4
RP	Developer Experience Theoretical Framework. See Table 4.4 and Figure 4.4.	4.5

minds is based on perceptions of an inner dialogue and interactions between themselves, other people, and artefacts. The processing of that stream of perceptions forms an experience. The findings of Articles I and II are summarised in Figure 4.1.

In Article I, we argue that the creation of software can be considered an experience in itself. Methods that govern the creation of software has an impact not only on the end goal, but also on the experience of individual developers engaged in the development activity. Developer experience is situated in the mind of individual developers. Following a classical division of the mind, its three faculties are cognition (attention, memory, producing and understanding language, problem-solving, decision-making), affect (emotion, feelings, mood), and conation (impulse, desire, volition, motivation, striving). Thus developer experience forms through an interaction between cognitive, affective, and conative factors.

We also emphasise that the social environment is an important aspect of developer experience [Article I]. The human mind is capable of processing social information, as evidenced both by research on externally observable social behaviour and research on responses to social stimuli in the brain (e.g. Gallese et al. [2004]). We further elaborate this aspect in Article II: software developer experience rests on basic social psychological mechanisms,

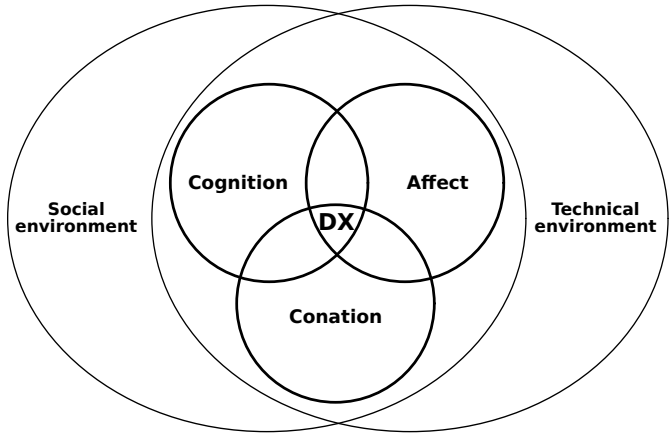


Figure 4.1: The Developer Experience Concept. (Adapted from Articles I and II.)

such as beliefs, norms, values, and group dynamic mechanisms such as group formation and identity. Developers have local theories and beliefs regarding how their work influences team and organisational performance. They reason about performance on all levels: in terms of individuals, teams, organisations, the marketplace, and also in terms of customers, such as satisfying customer wishes and needs. Processing social information is part of software development work.

Evidently, software development involves an environment full of technical artefacts. Programming languages, code written in those languages, tools to write code, plans and diagrams, and processes and methods to organise the software development activity over time, are all examples of such technical artefacts. Developer experience consists of interactions with all kinds of artefacts and activities that a developer may encounter as part of their involvement in software development [Article I]. Artefacts and activities are represented in the mind, and experience arises when the mind interacts with the internal representations as well as with the external environment. Each artefact can be experienced as such, but since software development in the large is a group endeavour, many of the environmental interactions are interpersonal, or social, in nature. Some social interactions may be mediated through technology, such as in the case of an online meeting, while some interactions may use the technological artefacts as shared objects that are the topic of the interaction, such as when developers point to a UML diagram when discussing how to implement a particular part of an architecture.

4.2 Experiencing Team Performance in Lean-Agile Software Development Environments

We address RQ 2 through Articles I and II. We provide the initial frame to focus the inquiry on developer experience in Article I. We use the conceptual framework in Article II, but also modify it based on empirical observation, as discussed in the previous section. In the article, we analyse developer experience in a specific kind of software development environment – Lean-Agile software development. The starting point of the inquiry is the notion of performance in software development teams. As software development organisations seek to improve their overall performance, they implement methods and processes which they believe will influence performance. In this thesis, it is of particular interest to understand how developers experience the continual striving for performance in Lean and Agile environments. The main result of the article is a rich, grounded theory that describes how developers experience the pursuit of high performance in software development teams.

4.2.1 Performance Alignment Work

Despite long-ranging attempts at defining and measuring performance, the concept remains elusive in the companies we studied [Article II]. It may be easy to define performance on a very high level, but understanding what it means in concrete terms is difficult. For developers, the elusiveness of the performance concept may be a kind of disturbance for developer experience. Developers have limited possibilities to influence overall performance, creating a conflicted situation that they must solve.

The study reported in Article II show that developers are involved in defining the meaning of performance through a continuous negotiation process with stakeholders. They discover, shape, and define performance for each situation. This process, which we call Performance Alignment Work, consists of becoming aware of performance concerns (Performance Awareness), defining whether current results match the desirable level of performance (Interpreting Performance), and adapting performance to the current understanding of desirable performance (Performance Adaptation) (see Figure 4.2).

Performance Awareness includes the level of self- or other-orientation in subjects' perception of performance – in other words, to what extent an individual is aware of performance on the individual, team, unit, organisation, and market levels [Article II]. Interpreting Performance describes the desirable level of performance: meeting or exceeding predefined objectives, or transcending them by becoming an active participant in their definition

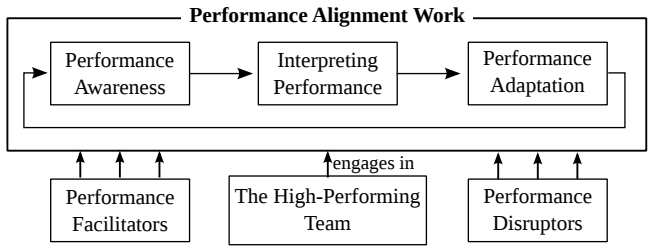


Figure 4.2: Performance Alignment Work is a continuous process in which people work to negotiate the meaning of performance in different situations and adapt their performance to changing conditions. Several factors can facilitate or disrupt this work. High-performing teams engage actively in Performance Alignment Work and are particularly good at it, resulting in superior performance. [Article II]

and assessment. Performance Adaptation refers to the attitude towards adapting to change in general, and to the means of adaptation which software developers have at their disposal. Several factors can facilitate and disrupt Performance Alignment Work (Performance Facilitators and Disruptors). Performance Disruptors can be categorised into continuous and single events, while Performance Facilitators can be divided into concrete, organisational factors, and soft, environmental factors.

High-performing teams reach high levels of performance because they engage actively in Performance Alignment Work and are particularly good at it [Article II]. High-performing teams are considered high-performing because they influence the criteria by which they are judged. Developers perceive high-performing teams in terms of group processes that link skilled and motivated developers to a powerful team identity. High-performing teams consider their possibilities to perform best when they are allowed free access to all stakeholders. This description of a high-performing team differs from the notion of a team which meets or exceeds predefined goals. A team which performs highly in terms of the Performance Alignment Work theory is proactive not only with respect to a predefined goal, but with the definition of the goal itself, and with contributing to alignment of several parts of the organisation to support actions that create value. This is also often what is actually desired: software developers who can think for themselves, can contribute their expertise when project and organisational goals are set, and can proactively alter the goals when circumstances change or new information becomes available.

4.2.2 The Individual and the Organisation

The particular traits of different organisations may have an impact on developer experience. We compared the findings between organisations through a cross-case analysis, which illustrates how similarities and differences in organisations can cause differences in developer experience [Article II]. We found differences and similarities in experiences within companies of different types, sizes, fields of industry, and degrees of globalisation [Article II]. We found that all developers in all companies displayed a common set of experiences regarding performance.

Performance was a more ambiguous concept in larger organisations [Article II]. With larger organisational size, it appears to be more difficult for developers to understand how their work fits into the systemic level, and to understand what expectations come with their job role. In larger organisations, concerns regarding planning of work – how goals should be set and pursued – were more common. In larger organisations, performance facilitators were more complex, and the larger social structures changed the importance of many factors.

Our findings indicate that although developers' experience of performance varies with the size and type of organisation, a common core experience exists within all organisations [Article II]. The largest differences in how developers experienced performance alignment in differently sized companies were related to how they became aware of performance-related concerns. The common core experience consists of a number of items, shown in Table 4.2.

4.3 Experiencing the Lean-Agile Value System

We address RQ 3 through Articles I, II, and III. We provide the conceptual frame for examining values among software developers in Article I, and present qualitative findings regarding the role of values in everyday software development work in Article II. In Article III, we report on a study that analyses the structure of Lean-Agile values among software developers using a survey approach. The article describes how values underpin the Lean-Agile approach and presents a Lean-Agile value structure with 11 dimensions.

Lean-Agile methodology is usually adopted with the specific purpose of improving some or several aspects of performance. However, the style of continuous development and adaptation to changing circumstances mean that performance is not a simple concept in such environments [Article II]. We found that values are an important aspect of how developers experience Lean-Agile software development, but also that values may present a challenge for developers. In situations where Lean-Agile methodology leaves

Table 4.2: Categories of core experiences related to team performance in a Lean-Agile environment. Each part is sorted by prevalence in the interview material. (Adapted from Article II.)

Degree of support	Core experience category
Strong support regardless of organisational size	Need for communication Improving the process Team setup Organisational learning Organisational support Ways to see success Reward Team identity Adapting to change Learning from failures
Support in medium and larger companies	Re-organisation Tools Decision power Facilitating communication Planning of work Seeing the big picture Collaboration and cooperation Personal development Understanding job roles Goal setting Social skills Control of my own work Intrinsic motivation to perform Prioritisation Testing

things unspecified in order to gain flexibility and encourage context-specific solutions, the values underlying the methodology should give developers the means to derive a consistent and effective solution [Article III]. However, difficulties may arise in case individuals interpret the underlying values differently. When methods leave room for flexibility, they come to rely increasingly on developers themselves to find solutions to problems as they arise in everyday work. This means that the methods ultimately rely on psychological and social psychological mechanisms – human relationships – for solving ambiguities arising in specific situations due to method flexibility.

The development philosophy in Lean-Agile can be understood in different ways, leading to different interpretations of performance and different notions of what concrete actions (behaviour) should be taken [Article II]. Developers interpret Lean-Agile as meaning decision power should be within teams. The notion of self-organisation means that decision power should be close to those performing the work. They perceive the spirit of continuous performance improvement to be central to the Lean-Agile approach. They believe that deviation from documented processes, procedures, and methods is often warranted in certain situations and they believe that such deviation is and should be encouraged based on Lean-Agile values. However, developers may confuse external change – e.g. a change in the marketplace – that must be responded to, with the goals and ideals of the approach – e.g. being open to change. Instead of detecting and dealing with actual change, they may begin to generate change themselves even where it is not needed, because the methodology primes them to focus on and expect it, and they lack concepts and vocabulary to process and discuss the underlying values.

The ambiguity of the Lean-Agile approach can be seen in how developers describe processes. Developers can see processes as reusable procedures which reliably produce a certain output [Article II]. They can see processes as decision-making tools which avoid human error and bias. But they can also see processes as something that needs to be tailored for very specific, context-dependent use: teams should own (select, create, and maintain) their own processes. However, since teams often need to interact, some way of making the processes work together is needed. Our findings indicate that developers perceive an underlying value system that Lean-Agile processes are built on. Such a value system gives criteria for prioritisation and decision-making, and thus forms a common foundation on which to build compatible processes and for treating processes as ephemeral, changing objects.

4.3.1 Lean-Agile Value Dimensions

We found a structure of Lean-Agile values among software developers consisting of 11 dimensions (see Table 4.3) [Article III]. Some of these dimensions are consistent with what could be assumed by examining some basic literature on Lean-Agile methodology, while some are less obvious. *Valuing Reliance on People* is perhaps the most obvious value to emerge, as it is so central to Agile software development and mentioned as the first value statement in the Agile Manifesto. However, some of the other dimensions are less obvious. *Valuing a Narrow Work Focus* emerged as a value characterised by the opinion that software developers should be highly specialised on technical work and not deal with work management or process issues, or with stakeholders such as users. This can be seen as contrary to the ideals of self-organisation, process ownership, continuous self-improvement, and inclusion of the customer, which are often emphasised in the Lean-Agile literature. At the same time, it is reasonable to assume that a high degree of special knowledge and a strict focus on technical work will lead to desirable results, such as timely delivery of high-quality code.

Valuing Planning and Preparation is characterised by a preference for preparation and planning before work starts. While a long-term perspective on quality and preference for slow and thorough decision-making are present in Lean thinking, this value can be seen as conflicting with the Agile notion of reactivity, responding to changes, and basing work on fast feedback cycles with an in-progress software artefact. *Valuing the Freedom to Organise* concerns self-organisation and responsiveness. This value emphasises the view that individuals and teams should be allowed to organise themselves freely and choose the manner in which they carry out their work.

The remaining values found concern cooperation, a sense of purpose, and a broad involvement of stakeholders [Article III]. Our findings demonstrate that Lean-Agile software development methodologies are linked to a professional and organisational culture. Values are a central aspect of how developers experience software development methods and methodologies, particularly Lean and Agile approaches which are explicitly based on values. Developers experience Lean-Agile values as consisting of a mixture of human aspects on individual and group levels, concerns regarding process adherence and flexibility, and notions of what is essential to meaningful work. Lean-Agile values cover software development work as a whole, not only specific sub-areas.

Table 4.3: Lean-Agile value dimensions found in Article III.

Lean-Agile Value Dimension	Descriptive Summary
Adherance to the Process	Processes should be strictly followed.
Broad Stakeholder Involvement	Software developers should involve the customer to co-create software.
Collaboration	Software developers should work together in close collaboration.
Discipline	Software developers should have discipline but discipline does not mean lack of responsiveness to change.
Flexibility in Task Execution and Leadership	Software developers should be freely working on many tasks, switching around at will; experts should have the authority to decide.
Freedom to Organise	Software developers should have the freedom to learn and organise their work themselves, from tools to architectures to team organisation to choice of tasks.
Narrow Work Focus	Software developers should focus on their technical work and not deal with stakeholders or management of work.
Planning and Preparation	Software developers should plan and prepare before work starts.
Predictability and Justification	Software developers should base action on evidence and observation rather than prescribed rules or unjustified orders.
Reliance on People	Software developers should be responsive to people, and know contractual obligations; planning a software project is impossible (instrumental value, aiming to increase performance, not necessarily to improve well-being).
Sense of Purpose	Software developers should know the purpose of their own work and its role for an end goal.

4.3.2 Relationship to Universal Human Values and Personality

A pertinent question is whether the value dimensions found in Article III are a reflection of a larger value structure, such as national values. When comparing the value structure against Schwartz's model of universal human values [Schwartz and Bilsky, 1987], we found that Lean-Agile values were more specific and detailed, and there were more pronounced differences in opinions regarding them among our study participants. These differences concern a continuum ranging from high preferences for bureaucratic order to people-orientation. Similarities between Lean-Agile values and the Schwartz model lay on a continuum ranging from self-focus to other-focus. These findings indicate that values are an important component of experiencing Lean-Agile approaches and that the experience may differ between individuals in terms of how the values are interpreted. We found that Lean-Agile values are different from universal human values, suggesting that the Lean-Agile value system is not a simple reflection of national culture although the latter can be expected to be an important influence on work behaviour Schwartz [1999].

We found that software developers can be placed on a value continuum ranging from an open, inclusive, and self-enhancing view to more authoritative, plan-based, and conforming values, and that software development approaches can be placed on a value continuum ranging from "bureaucratic" to "people-oriented" [Article III]. Lean-Agile values can be placed on a continuum ranging from a focus on the self, through a collective view, to carefreeness and flexibility to the degree of giving up control.

Differences in Lean-Agile values are more pronounced compared to universal human values when it comes to the bureaucratic–people-oriented continuum [Article III]. This seems natural because Agile, and to some extent Lean, have been coupled with a powerful "reactionary" message against the waterfall style of software development. The rhetoric of Agile proponents includes the claim that waterfall-type development is something that is fundamentally incompatible with a people-orientation (c.f. Lawrence and Rodriguez [2012]). For this reason, it appears reasonable that this particular dimension is highly emphasised by developers as they may echo the sentiment expressed by visible proponents.

We found other interesting relationships between universal human values and Lean-Agile values [Article III]. A collective focus in universal values was congruent with collaborative decision-making and benefiting the group. A focus on hedonism in universal values was congruent with relinquishing personal ambition and following a direction chosen by others. Self-focus

in universal values was congruent with valuing individual decision-making, control, and ambition. These similarities raise the question of whether broader universal human values may to some extent be an underlying factor that influences what an individual developer pays attention to in the Lean-Agile value system.

When considering Lean-Agile values in relation to personality, the former can be put on a continuum ranging from adherence to processes and roles, and submission to leadership (discipline/obedience), to a more collaborative and social approach to work (collaboration/equality) [Article III]. A preference for social values in Lean-Agile had a weak connection to personality dimensions Extroversion and Agreeableness. A preference for systematic, creative, and organisational values in Lean-Agile had a weak link to Openness to experience, Emotional stability, and Conscientiousness. A preference for processes, roles, and leadership values in Lean-Agile did not seem connected to personality. The results concerning personality, however, were only weakly indicative.

4.4 Supporting Developers in Virtual Team Environments

We address RQ 4 through Articles I, IV, and V, while we provide background for the question through Article II. Article I provides the conceptual frame to focus the inquiry: we suggested that team formation and maintenance are experienced as important concerns by developers, and are perceived as critical for the creation of high-performing teams. We also empirically observed the importance of team formation for developer experience in a professional environment [Article II]. In Articles IV and V, we concentrate on analysing team formation in terms of onboarding in an Open Source software development environment using virtual teams. The study empirically examines mentoring as an onboarding support mechanism in four Open Source projects. It demonstrates that mentoring can have a significant impact on the success of the earliest stages of onboarding, and that Open Source project characteristics moderate the level of success.

4.4.1 Onboarding in Virtual Team Environments

Creating high-performing teams and changing their membership composition requires facilitation; it is not merely a selection of the best individuals but also a process of adjustment for the team [Article II]. Onboarding refers to a deliberate process for facilitating inclusion of new members into an

organisation. Several of the core experience categories listed in Table 4.2 are linked to the onboarding theme. Most importantly, "Team setup" is among the three most prevalent experience categories reported. Based on the study reported in Article II, we may draw a number of conclusions regarding the role of team setup for performance and developer experience. Team setup is related to personal development through skill and motivation. High-performing teams are described as self-directed and resourceful. Team creation is important for high-performing teams, and developers think teams should be created by team members themselves: members should be involved in selecting and accepting new team members, as it is important for teams to maintain their distinct identity. Members of high-performing teams notice individuals' intrinsic motivation for high performance and their social skills, and want to include them in their teams. Developers think high-performing teams should be allowed to recruit new members themselves; they can tell who is motivated to work for high performance. Through peer selections, a powerful team spirit instilling pride in the team members is thus created and maintained.

In Open Source projects, the natural way of forming and changing teams is precisely through selection by existing members based on merit. It is therefore of particular interest how to introduce new members into existing teams in such an environment. Mentoring is one way to transfer the team spirit, identity, and other soft factors of the team culture to a new member [Article II]. In Articles IV and V, we hypothesise that successful mentoring results in an increase of observable actions among new developers, and that expert support from core project members can help new developers by influencing their motivation, increasing cohesion among community members, and reducing the gap between required skills and knowledge and actual developer ability.

4.4.2 Mentoring Supports Onboarding

We found that mentor support can accelerate the early stages of onboarding, but that project maturity is also a significant factor that amplifies the effect of a systematic onboarding process [Article IV]. Deliberate onboarding practices can have an impact on integration of new developers in OSS projects. Developers receiving onboarding support in OSS projects were more efficient and effective than unsupported developers. Supported developers made more commits over time, were engaged in more collaborative activities and communication, and generally surpassed the performance of unsupported developers. This indicates that mentoring may positively impact the effectiveness of an onboarding process in OSS projects in general. The effect of

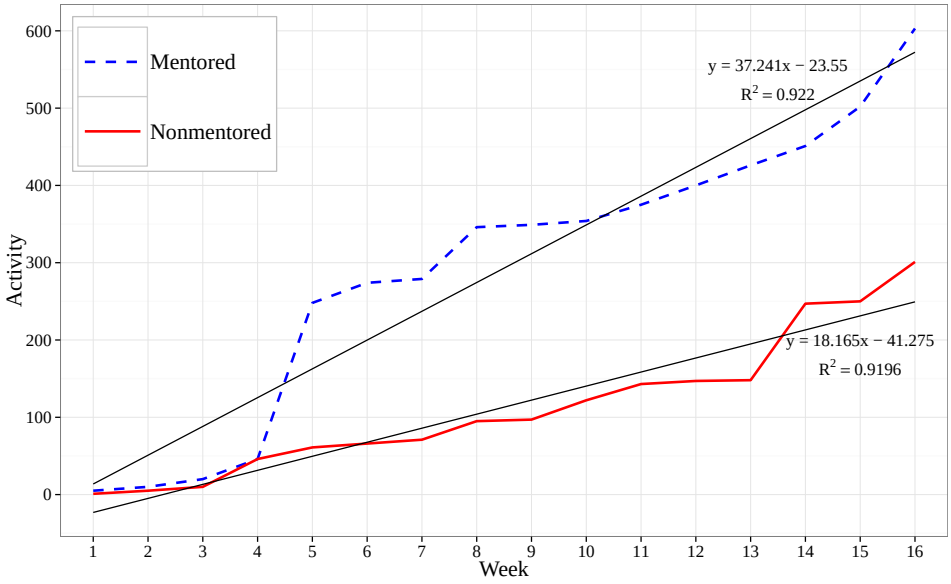


Figure 4.3: Comparison of observed activity (sum of number of commits, pull requests, and discussion interactions) over time between mentored and non-mentored developers. The black regression lines with formulae show the overall trend. R^2 shows the models fit the data well, explaining more than 90% of the variance. (Adapted from Article V.)

mentoring is reflected as an increase in activity, meaning a greater degree of participation (see Figure 4.3).

We found differences in onboarding results that could be attributed to project and community characteristics [Article IV]. Mature projects are better equipped to receive new developers. Both efficiency and effectiveness of supported developers were influenced by project size, appeal, and lifetime. Project characteristics should therefore be taken into account in support activities.

Finally, we found that being engaged in the mentoring activity reduces mentors' productivity in terms of contributions [Article V]. However, mentors did continue to contribute to their projects throughout the observed period, and returned to previous productivity levels after completing the onboarding program. The cost of mentoring in terms of lost productivity is thus temporary but should be taken into account in decision-making.

Our study indicates that mentoring is an effective support mechanism for onboarding in OSS projects [Article IV, V]. Mentoring was an important

element in the obtained results, although other factors may have played a role as well [Article IV]. Taking limitations into account, mentoring may therefore be suitable in other kinds of virtual team environments which display similar characteristics as Open Source environments. Among these are traits also found in the Lean-Agile approach: self-organisation, a high degree of autonomy, placement of decision-making close to those carrying out work, and high reliance on individual skills rather than high reliance on prescriptive process.

4.5 Summary

The articles comprising this thesis each address part of the main research problem expressed in Chapter 1. In this section, we contemplate the results of the individual articles as a whole. We summarise the results as a theoretical model of developer experience that may be used in future studies and as a sense-making tool in practice. The model addresses the main research problem stated in Chapter 1.

To obtain the model, we analysed the contents of the findings in Articles I–V. We examined the quantitative results obtained in Articles III–V against the qualitative results obtained in Articles I and II. We strived to remove, as far as possible, the case-specific aspects of the findings in order to arrive at a theoretical model with a high level of abstraction, but still with origins in the case evidence. The model is not meant to represent the individual case studies in every detail, but rather to be a more general theoretical model of developer experience. Following our constructivist stance, the model intends to capture the meaning constructed by the participants and the present author.

Figure 4.1 shows developer experience as arising in the mind of an individual as a combination of cognitive, conative, and affective factors. While those factors can be examined in isolation and combination, developer experience can also be seen as such, without reducing it to individual factors. This view provides a response to what the experience of developing software is like for the developer, bringing the developer experience concept from individual factors into a theory without examining each and every factor in detail. As a whole, developer experience may be decomposed into seven aspects, shown in Table 4.4. These aspects interact to form developer experience as an individual interacts with other developers (and other stakeholders) and with external objects in an organisational, physical, or virtual environment (see Figure 4.4).

Table 4.4: Aspects of Developer Experience, illustrated with example based on Article II.

Aspect	Description	Example (using Article II)
Experience object	What is experienced	Team performance in Lean-Agile environments
Experience formation	How the experience is formed	Performance Alignment Work theory
Experience influencers	Factors that influence the experience	Performance Facilitators, Performance Disruptors
Experience content	The parts or elements of the experience	Performance Awareness, Interpreting Performance, Performance Adaptation, and subcategories
Experience progression	How the experience changes over time	Performance Alignment Work is a constant cycle; new details emerge continuously
Behaviour outcome	How the experience leads to or moderates behaviour	Performance Alignment Work influences goal-setting and the focus of development work
Object outcome	How the experience or resulting behaviour leads to or moderates changes in one or more artefact or phenomenon	Performance Alignment Work influences performance expectations and alters what the team delivers

The theoretical model shown in Table 4.4 and Figure 4.4 may be used as a frame to structure inquiry into developer experience. The seven aspects comprising the model may be used to describe an experiential view into the software development activity from a developer's perspective. An *experience object* is that which is experienced: an artefact or phenomenon that may be concrete or abstract. This includes objects such as technical artefacts, methods, and process models, that may be internal or external to the context in which the developers act. The experience object is perceived and represented in the mind of the individual developer. Defining the experience object means defining what is experienced in a certain situation or over a period of time. *Experience formation* refers to how an experience regarding the experience object is formed. Defining experience formation means describing how an experience is formed when the individual encounters the experience object.

Experience influencers moderate experience formation. Experience influencers are present in the external environment at large, in the small-group context in which developers interact, and at the individual level. Individual and situational differences come into play to act as parameters in experience formation. Experience formation results in *experience content*, which is a subjective evaluation of the experience. This is, for example, where judgements regarding the positive or negative valence of the experience are to be found. When experience formation occurs repeatedly over time, the experience can change according to a description of *experience progression*. While "experience" can refer to an instantaneous stream or to a single episodic occurrence in a certain situation, it can also refer to an overall cumulative experience as the result of prolonged or repeated exposure over a period of time. Depending on the aims of a study or practical intervention, a shorter or longer experience progression may be considered. Developer experience has a *behaviour outcome* which refers to its role in creating or moderating overt behaviour. Behaviour may itself feed back into developer experience as the individual interacts with the environment. Finally, developer experience has an *object outcome*, as the experience may lead to changes in an external artefact or phenomenon through behaviour. Subsequent experience formations are then based on a changed environment and a changed external object. It should be noted that the object outcome is not limited to the original experience object; many objects are likely to be involved in real situations.

The theoretical framework presented here may be used to examine developer experience both in broad, holistic settings as well as in narrow, specific settings. We use three examples to demonstrate its use. First, we

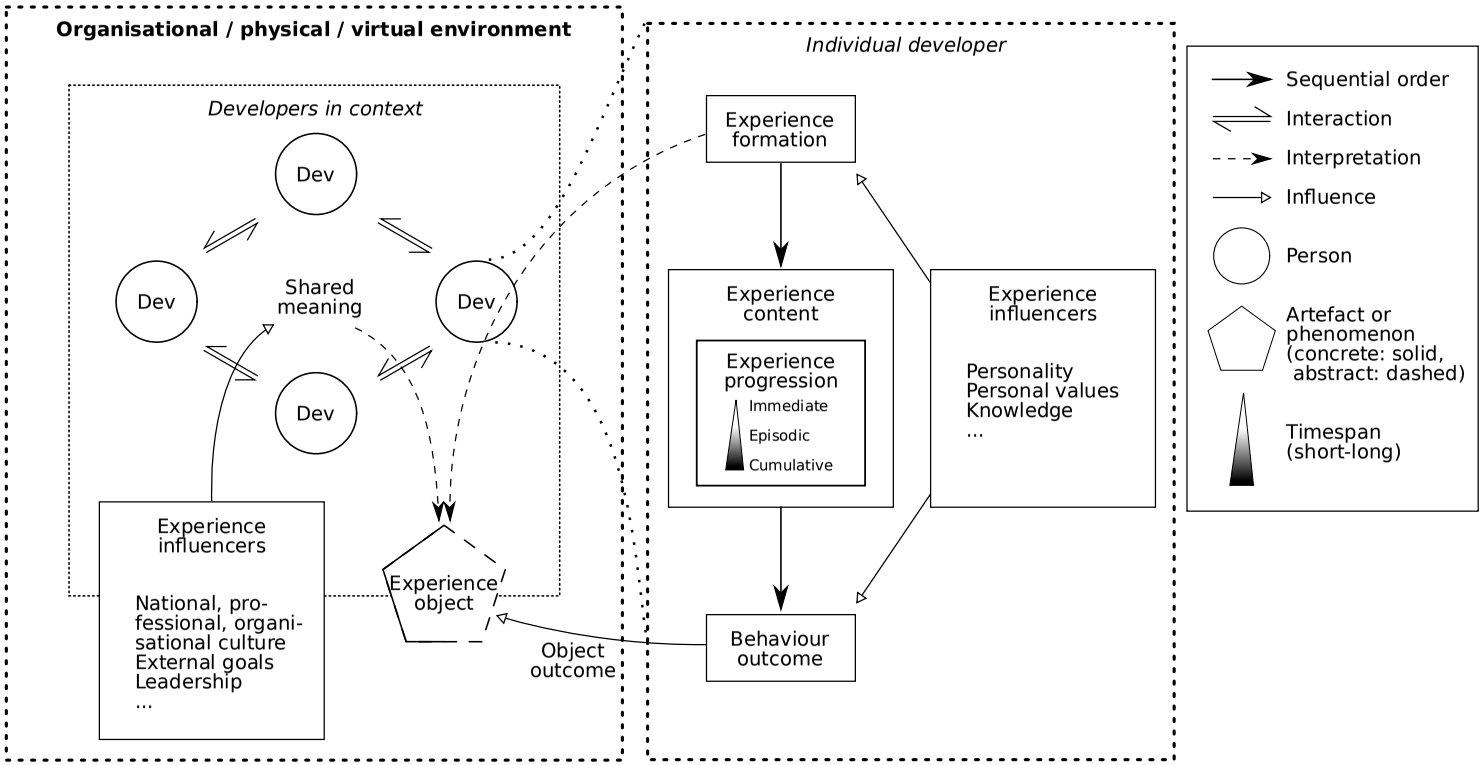


Figure 4.4: Developer Experience Theoretical Framework. Also see Table 4.4.

reinterpret the findings presented Article II (see Table 4.4) and connect them with the findings in Articles III–V through the theory. In Article II, the experience object – what is experienced – is team performance in Lean-Agile environments. This is an example of a broad, holistic setting. The theoretical framework suggests categories of questions that can be used to understand developers' experiences with regard to the experience object: how is the experience formed, what are its contents, what are the influencers and outcomes? In the present study, the Performance Alignment Work (PAW) theory constitutes experience formation, as it describes how the experience is formed: through a continuous process of negotiation. The PAW theory also provides the experience influencers, as it details facilitator and disruptor factors. Experience content is also provided as the detailed categories of the PAW theory. Article III provides more detail on values both as experience influencers on the individual level and as an experience object in its own right.

The PAW theory also contains the experience progression aspect, as it describes a continuous cycle, although it does not fully answer how the experience may change over extended periods of time. The behaviour outcome consists of changes in what goals are set and how, and what the team does in terms of development work in order to meet the goals. Finally, the object outcome is a change in performance expectations and a potential alteration of what the team delivers. This new state in turn is the basis for subsequent experience formation.

The second example maps Articles IV and V to an external view. In those articles, experience formation and influencers are manipulated by introducing mentoring as an onboarding support mechanism. Mentoring can be seen as an experience influencer in the environment outside the individual developer. The behaviour outcome is visible as increased developer activity in the virtual team environment and the object outcome is visible as a change in the source code and other technical artefacts stored in the projects' online collaboration systems. We may hypothesise that the experience content is also altered, but the study did not address that aspect. This example shows how individual components of the framework may be used to define intervention and measurement points. Not all components need to be addressed in a single study, and the theory can be used to reason about the omitted aspects.

Finally, we may use the framework to sketch a narrow, specific study as a hypothetical example. Assuming that we are interested in developers' experiences with integrated development environments (IDEs), we choose one or more IDEs as experience objects. We may then follow the experience

formation as developers attempt to use the tool, unpack the experience content over time, and observe behaviour outcomes at different stages. The framework suggests that we should also examine whether there is an object outcome towards the IDE itself – developers may configure the IDE according to their preferences and level of expertise, influencing their experience – and whether other developers in the examined context are somehow influencing the individual experience. This demonstrates how the framework can be used to differentiate between a usability study, which would likely focus mostly on cognitive aspects of the IDE, and a developer experience study, which examines not only task outcomes but also picks apart the formation and contents of the subjective experience that occurs with using the tool. Furthermore, the framework helps to differentiate between a user experience and a developer experience study. The latter occurs in the context of development, and researchers should pay attention to the particular traits of that context. By carefully defining, e.g., the experience object and experience influencers, researchers can make their study more relevant for the context of development and for developers.

Chapter 5

Discussion

The four research questions posed in Section 1.2 are answered in Chapter 4. Together, they partly address the main research problem of this thesis. In this chapter, we discuss the implications, validity, and ethical concerns of the thesis contribution, and suggest some possible directions for further research.

5.1 Theoretical Implications

To the best of our knowledge, this thesis is the first to investigate software developer experience in general, and in Lean-Agile and Open Source environments in particular. This thesis can be considered an opening for further research. It provides a theory of developer experience that describes a model of the software development activity that places the developer on centre stage. It also shows results from empirically applying such a model in real software development environments.

RQ 1 asks how software developer experience can be conceptualised. Our choice has been to base the answer on a human perspective grounded in behavioural and social science: software developer experience arises from a combination of cognitive, affective, and conative mental processes that interact with the social and technical environment. An alternative approach would be to centre the answer around technology. However, based on the abundance of evidence that human factors play a major role in software development (e.g., Weinberg [1971]; Brooks [1975]; Boehm [1981]; Sawyer and Guinan [1998]; Cockburn and Highsmith [2001]; Feldt et al. [2008]; DeMarco and Lister [2013]; see Chapter 2), such a conceptualisation appears artificial and it would divert attention from the largest impact factors on productivity and quality in software development.

Our answer to this research question suggests that research on software development benefits from being firmly grounded in behavioural and social science. It is possible to view software development as a technology-centred activity where repeatable processes direct individual work. Human factors would then be variables moderating or influencing how process instructions are carried out, ultimately affecting software development outcomes. The knowledge produced by carrying out research with these assumptions can be thought of as empirical rules that are generalisable with contextual limitations (c.f. [Basili, 1996; Rombach, 2011]). This thesis proposes a way to see software development as a fundamentally human and social activity where the instructions for how work is to be carried out are also subject to constant development and interpretation by those performing the work and, to some extent, by special process developers. In this view, developers do not carry out a predefined process which is separate from themselves; the process is, to a large extent, embedded in their personal and social structures. Human factors are inherent in all software artefacts, including process descriptions and work instructions, and are not merely moderating factors but also direct causes and effects. Software artefacts are processed in a social psychological system rather than being idealised entities outside it. The knowledge produced by carrying out research with these assumptions can lead to an increased understanding of what the experience of being a developer means. The conceptual framework developed in this thesis can be used as a starting point for further investigation into this topic.

This thesis supports calls in earlier research for integrating a social and behavioural science perspective into software engineering research (e.g., Wallgren and Hanse [2007]; Feldt et al. [2008]; Warfield [2010]; Amrit et al. [2014]; Lenberg et al. [2014, 2015]; Graziotin et al. [2015]), suggesting that behavioural and social science should not merely be integrated but could be the basis of a line of research on software development. This thesis supports that notion by demonstrating that explicitly taking a psychological or social psychological starting point is a feasible approach.

RQ 2 asks how software developers experience team performance in Lean-Agile environments. Our answer states that team performance in Lean-Agile environments is experienced as a continuous process of Performance Alignment Work (for a full description, see Chapter 4 and Article II). There is more ambiguity and more complex performance facilitator factors in larger organisations. However, a common core experience exists in all organisations. The largest differences between differently sized organisations relate to performance awareness. Experiences of team performance are also strongly related to values.

One implication of this is that static notions of performance may lead to incorrect conclusions in software development team performance research when the aims include the developer perspective. Performance should be considered against the realities of software product and service development in Lean-Agile contexts. A static notion of performance, which could be motivated in scenarios where software developers are focused exclusively on one major development activity – e.g. requirements, design, implementation, or testing – is not in line with how software developers in Lean-Agile environments perceive their work. Lean-Agile software development is connected to the negotiation-and-deliberation and situated design paradigms [Moran, 1996; Kalfoglou et al., 2000] where exploration, discovery, learning, and corresponding rapid changes in direction are of key importance. The uncertainties and ambiguities experienced by developers are part of the motivation for Lean-Agile software development: in a fast-moving market, targets frequently move and goals change. However, such changes may also be a consequence of the Lean-Agile approach itself, as it promotes embracing uncertainty – a notion that means it is acceptable to keep options open and negotiate rather than decide and stick to decisions. For developers, this can mean that negotiation is constant and only relatively few things will ever be finally decided. Therefore, performance targets which are based on assumptions regarding details of the outcome are likely to become invalid during the development process. This is a challenge to research on performance management and measurement in software development because it means that the assumption that a baseline for comparison can be found may be invalid.

RQ 3 focuses in greater depth on the finding that values are of particular importance, and asks how software developers experience the value system of Lean and Agile approaches. Our results indicate that the value system is experienced as having a distinct structure based on the values expressed in foundational literature. In our sample, this was visible as a value structure with 11 dimensions.

Agile values is a frequent topic for opinion and advocacy articles, and is frequently mentioned in research on Agile software development, but we have not found empirical studies dealing specifically with Agile values. In this work, we consider Lean and Agile together, because in reality, it is not possible for developers to fully separate the two in their work. Values have been extensively investigated both in work environments in general and in Lean manufacturing in particular (e.g., Liker [2004]; Liker and Hoseus [2008]), but our study is, to the best of our knowledge, the first to quantitatively investigate values related to the combination of Lean and Agile software

development as experienced by developers. Although developers come into contact with the Lean-Agile combination in practice, it is far from being consistently defined or understood. Lane et al. [2012] summarise Lean values based on existing literature. However, their notion of Lean software development is that it is "not one of the Agile methods" but rather a broader concept that considers software development from a business perspective. They find a set of 12 values that characterise Lean software development. Two overarching values address an internal and an external focus: reduction of waste and delivery of customer-defined value. Our 11-dimension Lean-Agile value system is similar in many respects although it is based on empirical data from practitioners. Our findings can serve as a basis for further empirical work to validate the Lean-Agile value structure. However, the precise dimensions themselves may be less important than the finding that software developers perceive the value system as ambiguous, leading to differences in interpretations of situations and desirable behaviour. Our findings suggest that future research should not assume that developers understand the approach in a uniform way. Also, developers may not be able to formulate similarities and differences between individuals or even report on relevant values-related issues in Lean-Agile studies. Thus conclusions based on reported observations of overt behaviour or even reports regarding the use of work practices should be drawn with caution, as the true underlying causes for observations may be hidden. Research designs should take these fundamental biases into account.

We found that values are related to, but distinct from universal human values and personality. This too should be seen as a caution to research on values and personality in software development. Too direct conclusions may be drawn regarding how they influence software development. For example, it is tempting to use personality differences as causal explanations for performance differences. Our results suggest that a less direct explanatory pathway should be investigated: better or worse results could arise because of a better or worse fit between individual traits, environmental traits, and the assumptions underlying the software development methodology in use. As another example, it is tempting to try to measure "agility" or "leanness" based on checklists of methodological procedures. We would suggest rather to attempt to capture the notion of a Lean-Agile mindset: a way of thinking that says software development work should be governed by a flexible, forward-looking but responsive, and reflective, self-improving, socially inclusive, and dialogic style of working. This contrasts the software developer who knows the details of Lean-Agile procedures by heart and upholds them as normative standards, with the software developer who is

seeking to perfect individual and team work based on an underlying belief that improvement is always possible. Thus a connection between Lean-Agile values and the body of knowledge on motivation in software engineering could be established.

RQ 4 asks how new developers can be supported in virtual team (Open Source) environments. Providing deliberate onboarding support was shown to have a beneficial effect. In particular, mentoring by experienced project members was shown to be a promising onboarding support mechanism. To our knowledge, our study is the first to treat the process of onboarding in OSS directly.

Team formation and introduction of new members to an existing team are linked to classical problems of how to add resources to a software project in order to increase performance – or at least not worsen it. Brooks [1975] observed that the introduction of newcomers into projects are impeded by the ramp-up time it takes for them to learn enough about the technical content to be productive, and by the increased communication overhead due to increased need for coordination. In Open Source environments, where globally distributed work is the norm, bringing new developers into projects is an even greater challenge than in traditional, co-located, single-company settings. Open Source project communities may be considered as special kinds of virtual teams, where a majority of software developers work without direct human contact, and often are not part of the same organisation. Thus establishing relationships to key team members using online communication is important both for productivity reasons and for enabling a positive project experience.

In Open Source projects, developers are often seen as self-motivated and driven by personal interest (e.g. Shibuya and Tamai [2009]). As they may not be part of the same organisation, processes and management procedures may not extend to them all. Thus mechanisms for onboarding which are appropriate in more traditional company environments may not be directly applicable in an Open Source context. While the mentoring support examined in this thesis showed positive results, the effect might be temporary. Some evidence suggests that deliberate onboarding programs may not result in long-term contributors although participants who succeed in them do find them valuable [Labuschagne and Holmes, 2015]. This suggests that at least for some Open Source projects, mentoring has limited utility. Developer motivation could remain the largest factor that determines long-term contribution in such environments. However, as we showed, project characteristics are also an important factor that influences the benefits of mentoring as an onboarding mechanism. With only few pieces of evidence,

final conclusions cannot be drawn regarding the effectiveness of onboarding in OSS projects in general. More research is needed before drawing conclusions regarding this area in future studies.

Returning briefly to our main research problem – how software developers experience the activity of software development – our results indicate that their experience concerns a very broad range of phenomena. Software developer experience goes beyond programming-related concerns. Factors on the individual, group, organisation, market, and society levels have an impact on how software developers experience software development. What is experienced varies from situation to situation, but the formation of an experience can be described as an interaction between the individual and the environment, where the individual processes internal representations of objects and events occurring in the environment into an experience. Together with situational and individual traits, the experience influences behaviour intentions and ultimately plays a role in overt behaviour. This may be considered a model of software developer experience that is grounded in social psychological research results and theories as well as in empirical research on software development. The model could be utilised to design studies on software development where the aim is to understand the developer perspective while taking the broad context into account.

5.2 Practical Implications

Although the main contribution of this thesis is theory development, our findings also have practical implications. It should be noted that these are limited to Lean-Agile and Open Source contexts, although future studies could refine this limitation.

One implication is that paying attention to and improving developer experience in software development organisations is important. Developer experience can, as our results show, be traced back to organisational functioning and the functioning of development methodologies which are likely to have implications for overall performance, e.g. in terms of productivity and quality. Good developer experience could also support the well-being of developers, providing opportunities for meaningful work and personal growth. Beyond this, we discuss some more detailed implications below.

An important issue emerging from our research is the role of ambiguity in Lean-Agile software development methodologies. We hypothesise that some of the ambiguities experienced by developers stem from an ongoing transition in software development. Software development is increasingly a particular kind of product and service development, and the larger activity of devel-

oping those products and services is what software developers are actually contributing to. Methodologies such as Lean-Agile implicitly consider both product development and technical levels, but the link between the product or service development process and the technical software development process is left undefined. Developer experience suffers when the product development and software development processes and their relationships are not clear. Developers may have to perform product development tasks in the software development process, sometimes at an incorrect stage of development. In the companies studied in this thesis, we could detect confusion among developers regarding this issue. As organisations try to implement Lean-Agile approaches, they adopt an incomplete model, resulting in lack of clarity regarding how software product and service development should be carried out.

In addition, the Lean-Agile approaches may be seen as approaches for the software development organisation only, leaving other parts of the organisation to operate in a different mode without an understanding of how to interface with the development organisation and vice versa. As a result, developers are faced with questions of "customer value" or other questions which they are ill-equipped to handle. As a result, their developer experience with regard to the Lean-Agile approach suffers, as the level of needless uncertainty increases – uncertainty that results not from real questions about the market or technical challenges, but from an improperly designed work environment. Thus an improvement in this area could lead to a better developer experience. Also, developer experience is an indicator that can reveal whether the organisation functions properly in this respect. Many new approaches attempt to establish the connection – e.g. Lean Startup [Ries, 2011] and different approaches to scaling Agile software development (c.f. [Laanti, 2014]) – but software developers' skills and knowledge, and software development education, have not yet caught up. Software development organisations should work to clarify these ambiguities and provide adequate training and skill development in order to avoid inefficiencies and risks.

These concerns imply that there is a need for education that extends the technical skills and knowledge of software developers so that they can work better in a product and service development organisation. An understanding of how products and services are developed, and the ability to contribute technical knowledge to such development is of critical importance. With greater understanding of how software development fits into the overall organisational activities, we assume that developers' attitudes towards ambiguity and uncertainty could improve. Developers could also be capable of articulating better how they contribute to the overall process, which could

have a positive effect on motivation. On the whole, improvements in this regard should be visible in practice as better developer experience. The framework developed in this thesis could be used as a basic tool to identify developer experience concerns in software development work activities and to support improvement of work and work-related artefacts with the aim of improving developer experience.

There are some implications related to values in software development. First, both theory and our findings indicate that practitioners can benefit from making implicit values more explicit in their work. It could be beneficial to base software development methodology on values. Values increase the adaptive fitness of individuals, providing them with flexible patterns of behavioural response options [Michod, 1993]. Thus, in contrast to very detailed action specifications, values-based approaches allow dynamic reactions in new and unforeseen situations, which are commonplace in today's rapidly evolving software organisations. Also, compatibility between work and cultural values increases the odds of a work process to be accepted by practitioners, improving the developer experience. Integrating values into the software development process, or even basing the software development process on values, is thus warranted. Lean-Agile software development has put emphasis on a set of core values. This may explain why these approaches have gained such widespread adoption in the software industry: they appeal to emotions because when triggered, values are associated with an affective response. It would be beneficial for organisations to consider whether they are prepared to manage software developers who work with values-based methodologies and if so, how to do so in an effective, efficient, and ethical manner.

Developer experience also comes into play in Open Source development. Many organisations are developing their software products and services into ecosystems and platforms where multiple organisations and individuals may participate, developing, using, sharing, selling, and buying software in different forms. In such environments, software developers may have a direct influence on technology choices which may in turn strengthen or weaken a particular platform or ecosystem. Lack of support from developers may spell the end of a company's platform, restricting their capabilities to deliver products and services to customers. Conversely, platforms favoured by developers will become the basis of new products and services, drawing more customers to it, and enabling more opportunities for monetisation. Providing developers with a superior experience may be a key competitive advantage. Our findings imply that organisations wishing to derive benefits from OSS projects should carefully consider developer experience in such

contexts. Managing the experience of developers during their integration into OSS projects is important. Onboarding in the form of mentoring is one option, but costs in terms of lost productivity for developers acting as mentors, project characteristics, and the particular traits of OSS development in general should be taken into account.

5.3 Threats to Validity

The question of result validity is present in all scientific studies [Creswell, 2009]. Although the basic notion of validity can be considered the same, different types of studies proceed from different assumptions and have different aims, and therefore, the operationalisation of validity and the emphasis on different aspects of validity can differ between studies [Glaser and Strauss, 1967; Creswell, 2009; Merriam, 2009; Runeson and Höst, 2009; Wohlin et al., 2012]. This thesis takes a constructivist worldview, which assumes that participants are engaged in a social process of creating shared meanings which govern interpretation of events and choice of actions in their work. The knowledge claims of the thesis follow a constructivist stance: the results aim to capture the meaning created by the participants and the researcher. The aim is also for pragmatism, in that the thesis aims to generate theory which is grounded and can be used in real-world practice. The contributions of this thesis are grounded in the local environments and contexts in which they were developed. The accuracy of the results is limited by several factors, and they may be generalised or re-used in other contexts only with certain limitations.

Questions of validity also concern the chosen strategies of inquiry and research methods. This thesis combines qualitative and quantitative methods and may be considered a mixed-methods study [Creswell, 2009]. It is therefore necessary to consider the validity criteria of mixed-methods studies in addition to criteria for the qualitative and quantitative parts. However, the main part of the thesis is conducted using qualitative methods, and the quantitative part can be considered subordinate to the qualitative part. In the past, several researchers have considered it possible to utilise quantitative data and analysis as part of qualitative research. For example, Grounded Theory, as described by Glaser and Strauss [1967], considers any kind of data source to be usable. Therefore, the validity criteria for this thesis are based primarily on the qualitative tradition.

5.3.1 Validity Criteria

Software engineering research has made use of well established validity standards and corresponding terminology for quantitative studies (e.g. Runeson and Höst [2009]; Runeson et al. [2012]; Wohlin et al. [2012]). While the qualitative research tradition has developed terminology of its own to discuss validity issues, case study researchers and experimenters in software engineering often use terminology from the quantitative tradition to broadly classify validity criteria (e.g. Yin [2009]; Runeson and Höst [2009]; Runeson et al. [2012]; Wohlin et al. [2012]). Here, we follow the latter approach and discuss the qualitative terms under the four aspects of validity listed by Runeson and Höst [2009] and Wohlin et al. [2012], based on Yin [2009]:

- Construct validity – the extent to which the operational measures studied actually represent what the researcher intends and what is investigated according to the research questions.
- Internal validity – whether the study controls for potential confounding factors when examining causal relationships.
- External validity – the extent to which it is possible to generalise findings and the extent to which findings are of interest to people outside the investigated case.
- Reliability – the extent to which data and analysis is dependent on specific researchers.

5.3.2 Construct Validity

Construct validity concerns the extent to which the operational measures studied actually represent what the researcher intends and what is investigated according to the research questions [Runeson and Höst, 2009; Wohlin et al., 2012]. A construct is an abstraction, created by the researcher, the purpose of which is to represent and conceptualise the latent, non-observable variable which is the cause of some observable effect. The researcher must identify correct operational measures for the concepts being studied [Yin, 2009]. High construct validity helps ensure that research participants and researchers understand what is being studied in the same way.

Construct validity is not unproblematic in qualitative research, since not all qualitative designs can have a priori constructs in the same way as, e.g. quantitative survey studies. However, it is possible to interpret construct validity in qualitative studies as how well they succeed in reconstructing the realities of participants [Guba and Lincoln, 1989]. The investigation here

focuses on the construct *developer experience*, and part of the aims is to develop the construct itself.

A number of mitigation strategies were applied to address construct validity. First, the construct was conceptualised and given at least a preliminary definition based on previous research. Article I documents this conceptualisation and definition. The subsequent articles contribute and add to the conceptualisation. In Article II, which uses case study methodology, construct validity was strengthened by using multiple sources of evidence, the establishment of a chain of evidence, and informant checking of study results, as suggested by e.g. Yin [2009] and Merriam [2009]. Multiple sources of evidence were present since we interviewed more than one informant from each organisation, providing different perspectives on the same organisation. We maintained the chain of evidence during analysis by carefully tracking how each piece of raw interview material was turned into higher-order categories. Thus each result can be traced back through the entire analysis process and can be connected to the exact raw interview fragments from which it originated. Finally, we asked representatives from each organisation to check and comment on our initial analysis results before drawing final conclusions.

In Article III, which included a quantitative survey focusing on the values aspect of developer experience among Lean and Agile software developers, construct validity was strengthened by conducting multiple rounds of survey piloting with respondent feedback in order to select good survey items and improve the survey item wordings. Also, the survey embeds two well-established psychometric instruments with well-developed constructs. In Articles IV and V, the general construct *onboarding* is well established in the literature on organisational psychology. The construct validity of the measurements was strengthened by using existing theory from studies on Open Source Software projects and by making a detailed operationalisation of the measurements using the Goal-Question-Metric approach.

5.3.3 Internal Validity

Internal validity concerns whether the study controls for potential confounding factors when examining causal relationships [Runeson and Höst, 2009; Wohlin et al., 2012]. Internal validity is of concern mainly in Articles IV and V, in which onboarding is examined in a causal research design. Since the study in these articles is most properly classified as a natural experiment – one where researchers cannot control the assignment into treatment or control groups and the treatment occurs "naturally" rather than being introduced by the researchers – we were not able to eliminate nor control

for all identified potential confounding factors. Some important potential confounding factors are i) the use of student subjects, whose skills and motivation may be different from professional developers; ii) the possibility of the treatment group consisting of developers of above-average knowledge and skill levels due to screening and selection into the educational program; iii) unknown treatments or events which could cause the observed effects besides the onboarding support given by mentors; and iv) regression, where participants with extreme scores are selected for an experiment by chance, but later regress towards mean scores. In addition, the nature of the study means that there may be a number of unknown confounding factors which limit the internal validity of the study.

For Articles I–III, and the thesis overall, internal validity is, in the sense described above, not of primary concern since testing causal relationships is not the primary aim of the research, and it is of an exploratory nature (c.f. Yin [2009]). However, in qualitative research, internal validity may be understood as credibility: how well research findings match reality [Merriam, 2009]. Several mitigating strategies are available to address credibility, e.g. triangulation, respondent validation, and peer review. For Article II, we used data and researcher triangulation, case study replication logic, and respondent validation to ground the results in the real-life context of the participants. Although Article III uses a quantitative approach, its design is correlational rather than causal, and thus internal validity concerns only the operational quality of the study, which has been evaluated through peer review. Multiple rounds of piloting and respondent feedback were also used.

5.3.4 External Validity

External validity can be defined as the extent to which it is possible to generalise findings and the extent to which findings are of interest to people outside the investigated case [Runeson and Höst, 2009; Wohlin et al., 2012]. However, the notion of external validity as used in quantitative survey research, where findings regarding a sample are intended to generalise to a larger population, is incorrect when applied to qualitative and case study research. In qualitative research, the term transferability can be used, referring to the possibilities for a third party to transfer findings to another context [Merriam, 2009]. Here, "the burden of proof lies less with the original investigator than with the person seeking to make an application elsewhere" [Lincoln and Guba, 1985]. The mechanism of generalisation itself is also different. Case study research relies on *analytic* generalisation rather than statistical generalisation [Yin, 2009]. The generalisation is not from a sample to a population (with a known and calculable margin of error)

but from a set of case-grounded results to a broader theory. The theory may be validated through replication, in which it is used to examine other relevant cases and determine whether it holds in those cases as well [Yin, 2009; Eisenhardt and Graebner, 2007; Eisenhardt, 1989].

The external validity of the contributions developed in this thesis have been strengthened in a number of ways. The primary strategy is the use of replication logic through multiple-case studies in Articles II, IV, and V. Subjects involved in the individual studies expressed interest towards the studies and their results, and member checking procedures confirmed the relevance for the participants. Articles III–V replicate parts of the findings of Article II in a different setting. For these reasons, we consider it justified to claim that the results from Articles II and III are applicable in a professional environment utilising Lean-Agile software development, and the results from Articles IV and V are applicable in an Open Source environment which may have a mixture of professional and non-professional developers. We have aimed to analytically generalise the overall theoretical contribution developed in this thesis to apply broadly. With some reservations and proper contextual tailoring, we consider the overall results regarding developer experience to apply in a Lean-Agile environment with or without virtual teams. To some extent, it may be possible to generalise further, but this thesis does not include the necessary evidence to assess the validity of the results in other contexts. We consider the detailed reservations and need for contextual tailoring below.

None of our individual studies examine software developers in a "traditional" development setting. This means that no evidence can be provided regarding developer experience in settings other than those examined in this thesis. This choice was partly a practical scoping decision to limit the study for a PhD thesis, but also because we sought to examine human factors at the boundaries of the state of the art, where outcomes are likely to depend to a large degree on the factors we are interested in. When attempting to generalise beyond the chosen settings, care should be taken to adjust and adapt the findings. We argue that despite the lack of empirical evidence, theory suggests that many of the findings are likely to apply outside the settings we investigated. It may be the case that generalising findings on human factors is less dependent on the "technical" aspects of the context, such as development methodology, and more dependent on aspects that are common to the specific kind of knowledge work that software development constitutes. This should not be taken as claiming that generalisation can then be done freely, but rather that parties wishing to transfer the findings may be able to do so if they find underlying commonalities that are not

dependent on the Lean-Agile or OSS context variables. The contributions in this thesis should be tested or investigated in other kinds of settings, and there is no reason to avoid applying the contributions in other kinds of settings, as long as the potential limitations are taken into account.

Some specific issues influencing generalisation are the cultural context, existing organisational procedures, and the level of knowledge of Lean-Agile software development methodology and knowledge of human factors in software development. The cultural context of this study is primarily Finnish, with most company cases being located in Finland and having primarily Finnish participants. Although not confirmed at present, some factors particular to this cultural setting may limit the transferability of the results. Examples of such factors include educational background and leadership structures that are based on social hierarchy norms. Generalisation to other cultural contexts would require further case replication. However, many of the findings in this study can still be applied if cultural context is taken into account and suitable adjustments made. Compatibility with existing organisational procedures may be an issue for generalisation. Adherence to standards and norms internal or external to an organisation may prevent the application of some of the results. The results are aimed mainly at software development organisations or organisational units which are not constrained by, e.g., regulatory requirements. Examples include software-intensive product and service development in small companies, such as startups, and research and development units with a focus on software-intensive product and service development in medium-sized and larger companies. Considering these limitations, the findings may be transferred to other contexts.

Factors on the individual level may also limit the generalisability of the results. The level of knowledge of Lean and Agile software development, and of human factors in software development, may limit the possibilities to utilise the results. The assumptions of Lean and Agile software development, which include high autonomy, iterative and incremental development rather than linear development, and tolerance for an empirical approach in which trial and error is permitted, may not be suitable in some contexts. Furthermore, knowledge of human factors in software development is needed before the findings in this thesis can be used. Concepts and underlying theories from psychology and social psychology, such as cognition, conation, affects, and values, their relationships, and their impact on behaviour, must be understood before the results can be applied in practice. The contributions of this thesis may not be usable without such knowledge.

Finally, since the study in Articles IV and V is performed with student participants, there is a justified concern that these results may not be generalisable to a professional context. Previous studies examining whether students can represent professionals as research subjects show that the question does not have a simple answer. Some studies report that student subjects and professionals perform similarly (e.g., Höst et al. [2000]; Svahnberg et al. [2008]), others conclude that students are not representative of professionals (e.g. Remus [1989]), while still others are inconclusive (e.g. Runeson [2003]). As observed by Berander [2004], the possibilities of generalisation depend on the task and on the extent and purpose of the comparison. Tichy [2000] suggests that the use of student subjects is defensible when doing initial research on a subject. Berander [2004] builds on this notion and suggests that student subjects are suitable for i) pilot studies, ii) validating education, iii) worst case scenarios, and iv) identifying trends and behaviours.

We argue that Articles IV and V fulfil all of the above criteria to some extent. Since our study is, to our knowledge, the first to examine onboarding with mentoring in an OSS context, it has many commonalities with a pilot study: apart from its main finding, it develops a research design and shows how to obtain and analyse data to gain insight into the subject. Since mentoring can be seen as a form of education that can be offered also in a professional environment, the study can be seen as validating an educational approach. Most importantly, however, the study fulfils the last two points. Applying the definition and logic of worst case scenarios in Berander [2004] and Kuzniarz et al. [2003], the comparison between non-mentored and mentored subjects probably will show the same direction in findings for both student and professional subjects. This study does not address the extent to which mentoring is useful for professionals. The objective of the study is to show a general trend and demonstrate that it is possible to influence observable outcomes through interventions that are linked to developer experience. In the study, mentoring is a social and behavioural intervention that is shown to have an observable effect on outcomes. That this trend is found with student subjects in the treatment group is indicative of its relevance with professional subjects. Finally, the control group includes a mixture of both professional and non-professional subjects, and thus the comparison is not strictly between student groups but between students and a more diverse control group which may include professional developers. Our conclusion is that the main result – that mentoring can be useful – is not primarily threatened by the use of student subjects and can be transferred to professionals as long as the other limitations of this study are considered.

5.3.5 Reliability

Reliability is the extent to which data and analysis are dependent on specific researchers [Runeson and Höst, 2009; Wohlin et al., 2012]. In qualitative research, reliability is often understood as *consistency* – across participating researchers and different projects [Creswell, 2009; Gibbs, 2007]. In traditional empirical research, the notion of reliability is often based on an assumption of a single reality, the observation of which will always yield the same results [Creswell, 2009]; thus a study can be repeated, yielding the same results. In contrast, qualitative research, as used in this thesis, aims to capture the world as people experience it, which makes the research susceptible to multiple interpretations of events and limited possibilities of repeatability. The criterion for qualitative research is therefore "whether the results are consistent with the data collected" [Creswell, 2009; Merriam, 2009]. The question is not whether the same findings will be found again, but rather whether results make sense given the data collected. In each of the individual articles, we have carefully established a chain of evidence that allows tracing each result back to the data source. We argue that the articles that rely mainly on qualitative research have a high degree of consistency between data and results.

In the quantitative parts of the study, we may apply a slightly different interpretation of reliability. The question here is whether the operations of the study, such as data collection, are repeatable with the same results, minimising errors and biases [Yin, 2009]. A prerequisite for reliability is thorough documentation, which is provided in this thesis and in each of the individual articles. The research designs used in Articles III–V are fully documented and they can be replicated independently of the present author. We thus conclude that reliability is high, with the reservation that it is obviously not possible to return to the state in which the investigated case companies and Open Source projects were at the time of the studies. Also, the research design and theoretical outcome of this thesis overall may be considered a contribution that can be used to achieve consistency in other studies on the same subject.

5.4 Research Ethics

Ethical issues are important for individual research studies, but also for the health and continued credibility of a field of research [Runeson and Höst, 2009]. A lack of consideration of ethical issues in the research design may inadvertently cause unethical outcomes for researchers and participants. While definition and enforcement of ethical guidelines is an organisational

concern, the validity and reliability of a study depend largely on the ethics of the researcher [Merriam, 2009]. Ethical issues were considered throughout the conduct of this thesis, and particular attention was given to participants' informed consent and anonymity.

Informed consent means that research participants, be they individuals or organisations, must agree to participate in a study. Participants were given explanations of the purpose of the study and the methods used. Informed consent was obtained where possible. Since some participants in Articles IV and V remained anonymous even to the present author with no means of contact, informed consent was not possible to obtain. For Article II, explicit permission was obtained from the companies involved to publish the research. The anonymity of research participants was carefully preserved. In Article II, companies and individuals were given anonymous identifiers. In Article III, no companies or individuals were identified in the survey data. In Article IV and V, control group participants remained anonymous even to the present author, and treatment group participants' identities were hidden by the statistical analyses, which give only summary figures.

Feedback to participants is essential to maintain long-term trust and for the validity of the research. Participants should be able to give feedback and check the correctness of data concerning them, such as interview transcripts. To maintain participants' trust in the research, analyses should be presented to them. Although they may not agree with the analysis outcome, feedback is nevertheless important for the validity of studies. To the extent possible given the limitations of anonymity, feedback was given to the participants and they were able to communicate their opinions to the present author.

5.5 Future Work

This thesis represents a first attempt at understanding how software developers experience the activity of developing software and establishing a theoretical framework for directing further inquiry into the topic. It did not seek to exhaustively examine the research problem, but rather to build theory grounded in empirical observations. In this section, we consider some possible future research topics that emerge from the present research.

This thesis suggests a line of research where the software development activity is based on a behavioural and social science perspective. There is room for examining what this means for research methodology, whether this leads to limitations in the types of research questions that can be examined, how the research community should assess this type of research, and what new foundational knowledge and theory researchers need to conduct such research.

This suggests that the topic area "human factors in software development" is in need of a thorough mapping and review, both to identify the state of the art – what sub-topics have been examined, what methodologies have been used, what results have been obtained and what the strength of those results are – as well as to identify opportunities for expanding and deepening the field. This task is much more difficult than what may first be assumed: human factors may occur in studies where the researchers are unaware that they are actually examining a human factor and thus many relevant articles may not appear in metadata searches using a simple search phrase, such as "human factors". At least one attempt at a similarly monumental task already exists: Lenberg et al. [2015] selected 250 articles for analysis out of a screening set of more than 10 000 articles related to the subject area.

The relationship between developer experience and software development outcomes is open for further inquiry. That knowledge, skills, and motivation influence outcomes can be considered intuitive, but there are many open questions. First, there is room for research on how these and other individual factors influence outcomes. The mechanism of influence is largely unknown, and it is therefore difficult to make predictions or recommendations regarding these factors. Second, the interaction between factors is unexamined. For instance, does a high level of knowledge and skill lead to a high level of motivation or the other way around? Third, there is room to examine the affective and social dimensions in more detail. Since the social environment can influence the expression of individual traits, it would be important to examine the influence of group situations on individual factors.

Even when software development occurs in a virtual team environment, each developer is located in a physical space. Participants reported on the need for norms and behavioural signals to avoid the drawbacks of an open office [Article II]. Offices should be designed with developer experience in mind, e.g. by having zones for group work and individual work. Future research could examine how developers experience their physical work environment, especially with respect to its relationship to group work.

The link between software development paradigms and the experience of developers is open to inquiry. How, for instance, do developers alter their behaviour as a result of interpreting the principles of Lean and Agile software development or other approaches to software development?

Research on team performance management and software metrics could also take the developer experience aspect into account. What are the relationships between different stakeholders' evaluations of performance? For example, what aspects of good performance do developers and customers agree or disagree on? What is the link between customers' and develop-

ers' conceptions of good performance? These and other related questions could result in new understanding of performance and measurement in environments in which goals are very volatile.

Finally, existing research on motivation could be expanded to include social and affective aspects. Here, culture and values are of key interest because of the link between values, motivation, and affect. Values research could be a promising approach in understanding developer culture, which can be considered to be an aspect of developer experience on the large-group level.

Chapter 6

Conclusions

In this thesis, we examined the construct *software developer experience*. We asked how software developer experience can be characterised in Lean-Agile software development, how software developers experience the Lean-Agile value system, and how new developers can be supported in Open Source environments with virtual teams. We used a mixed-methods research design to examine developer experience through multiple case studies, some of which are qualitative and some of which are quantitative in nature. The final contribution, a theoretical framework for understanding developer experience, was obtained by interpreting the quantitative results in terms of the qualitative results. The results is based on a constructivist stance which aims to capture the meaning constructed by participants and the researcher.

We found that developer experience can be decomposed into seven aspects. The experience object is what is being experienced. Experience formation is how the experience is formed. Experience influencers are what influences the formation of the experience. Experience content is the content of the experience. Experience progression is how the experience changes over time as a result of prolonged exposure. Behaviour outcome is how the experience leads to or moderates behaviour, and object outcome is how the experience of resulting behaviour lead to or moderate changes in one or more external artefact or phenomenon. These outcomes may feed back into the experience.

We found that software developers experience team performance in Lean-Agile settings as Performance Alignment Work, a continuous process of negotiation, in which they become aware of performance concerns and definitions on multiple levels of an organisation, interpret performance to determine the desirable level to aspire to, and adapt their performance depending on their attitude to change. In terms of our results, a high-performing software development team is actively engaged in Performance

Alignment Work, interprets a desirable level of performance as transcending predefined objectives by becoming an active participant in their definition and assessment. Such a team is high-performing because it actively influences its environment.

We found that software developers experience the Lean-Agile value system as being a foundation for the methodology. We found 11 value dimensions along which software developers can be placed depending on how they interpret the partially ambiguous Lean-Agile approach. These value dimensions are related to, but not the same as universal human or national values, and have weak links to personality. This is significant, since it shows that Lean-Agile values form a distinct value system but that developers do not have a single shared understanding of it.

We found that software developers can be supported in Open Source environments that use virtual teams by providing deliberate onboarding support in the form of mentoring by an experienced project member. Supported developers surpassed the performance of non-supported developers, indicating that mentoring did, to some extent, improve the developer experience by transferring tacit knowledge in the project community to new developers. With some adaptation, these findings may be applicable in Lean-Agile environments with traits similar to Open Source environments.

The theoretical framework developed in this thesis may be used as a basis for further studies into the developer experience construct. The framework may be used to design inquiry into developer experience in other contexts. It can, for example, be extended by new case studies, or used to derive hypotheses to test in specific environments. The framework may also be used as a sense-making tool in practice. Improving developer experience requires common terminology and understanding, which are provided by the framework.

This thesis provides a first study of developer experience in specific contexts. It suggests a line of research that examines software development from the perspective of developers The importance of this perspective is visible in many contemporary software organisations, communities, and ecosystems. The thesis suggests a number of topics for further inquiry based on the results obtained.

References

Abrahamsson, P. (2001). Commitment development in software process improvement: critical misconceptions. In *International Conference on Software Engineering (ICSE 2001)*, pages 71–80.

Abrahamsson, P., Warsta, J., Siponen, M., and Ronkainen, J. (2003). New directions on agile methods: a comparative analysis. In *International Conference on Software Engineering*, pages 244–254.

Adolph, S., Kruchten, P., and Hall, W. (2012). Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software*, 85(6):1269–1286.

Agarwal, N. and Rathod, U. (2006). Defining 'success' for software projects: An exploratory revelation. *International Journal of Project Management*, 24(4):358–370.

Ahmad, M., Markkula, J., and Oivo, M. (2013). Kanban in software development: A systematic literature review. In *39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 9–16.

Alali, A., Kagdi, H., and Maletic, J. I. (2008). What's a Typical Commit? A Characterization of Open Source Software Repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 182–191, Washington, DC, USA. IEEE Computer Society.

Ale Ebrahim, N., Ahmed, S., and Taha, Z. (2009). Virtual R&D teams in small and medium enterprises: A literature review. *Scientific Research and Essays*, 4(13):1575–1590.

Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, Massachusetts.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Alexander, L. and Davis, A. (1991). Criteria for selecting software process models. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, COMPSAC '91, pages 521–528.

Amrit, C., Daneva, M., and Damian, D. (2014). Human factors in software development: On its underlying theories and the value of learning from related disciplines. A guest editorial introduction to the special issue. *Information and Software Technology*, 56(12):1537–1542. Special issue: Human Factors in Software Development.

Amrit, C. and van Hillegersberg, J. (2008). Detecting Coordination Problems in Collaborative Software Development Environments. *Information Systems Management*, 25(1):57–70.

Amrit, C. and van Hillegersberg, J. (2010). Exploring the impact of socio-technical core-periphery structures in open source software development. *Journal of Information Technology*, 25(2):216–229.

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

Baddoo, N., Hall, T., and Jagielska, D. (2006). Software developer motivation in a high maturity company: a case study. *Software Process: Improvement and Practice*, 11(3):219–228.

Baldwin, C. and Clark, K. (2006). Modularity in the Design of Complex Engineering Systems. In Braha, D., Minai, A. A., and Bar-Yam, Y., editors, *Complex Engineered Systems*, Understanding Complex Systems, pages 175–205. Springer Berlin Heidelberg.

Basili, V., Caldiera, G., McGarry, F., Pajerski, R., Page, G., and Waligora, S. (1992). The Software Engineering Laboratory: An Operational Software Experience Factory. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 370–381, New York, NY, USA. ACM.

Basili, V. and Turner, A. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396.

Basili, V. R. (1993). The Experience Factory and its relationship to other Improvement Paradigms. In Sommerville, I. and Paul, M., editors, *Software Engineering — ESEC '93*, volume 717 of *Lecture Notes in Computer Science*, pages 68–83. Springer Berlin Heidelberg.

Basili, V. R. (1996). The Role of Experimentation in Software Engineering: Past, Current, and Future. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 442–449, Washington, DC, USA. IEEE Computer Society.

Basili, V. R., Caldiera, G., and Rombach, H. D. (2002). *Experience Factory*. John Wiley & Sons, Inc.

Bauer, T. N. and Erdogan, B. (2011). Organizational socialization: The effective onboarding of new employees.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2 edition.

Beecham, S., Baddoo, N., Hall, T., Robinson, H., and Sharp, H. (2008). Motivation in Software Engineering: A systematic literature review. *Information and Software Technology*, 50(9–10):860–878.

Beecham, S., Sharp, H., Baddoo, N., Hall, T., and Robinson, H. (2007). Does the XP environment meet the motivational needs of the software developer? An empirical study. In *Agile Conference (AGILE), 2007*, pages 37–49.

Benington, H. D. (1983). Production of Large Computer Programs. *Annals of the History of Computing*, 5(4):350–361.

Berander, P. (2004). Using students as subjects in requirements prioritization. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 167–176.

Bergersen, G. and Gustafsson, J.-E. (2011). Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective. *Journal of Individual Differences*, 32(4):201–209.

Bergersen, G., Hannay, J., Sjøberg, D., Dybå, T., and Karahasanovic, A. (2011). Inferring Skill from Tests of Programming Performance: Combining Time and Quality. In *International Symposium on Empirical Software Engineering and Measurement*, pages 305–314.

Boehm, B. (1981). *Software Engineering Economics*. Prentice Hall, Upper Saddle River, New Jersey, USA.

Boehm, B. and Turner, R. (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley.

Bonaccorsi, A., Lorenzi, D., Merito, M., and Rossi, C. (2007). Business Firms' Engagement in Community Projects. Empirical Evidence and Further Developments of the Research. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, Washington, DC, USA. IEEE Computer Society.

Bond, M. H., Kwok, L., and Schwartz, S. (1992). Explaining Choices in Procedural and Distributive Justice Across Cultures. *International Journal of Psychology*, 27(2):211.

Bourque, P. and Fairley, R. E., editors (2014). *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 3 edition. Online: `http://www.swebok.org/`.

Brooks, F. P. (1975). *The Mythical Man-Month*. Addison-Wesley, Reading, MA.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554.

Calikli, G. and Bener, A. (2010). Empirical Analyses of the Factors Affecting Confirmation Bias and the Effects of Confirmation Bias on Software Developer/Tester Performance. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 10:1–10:11, New York, NY, USA. ACM.

Capiluppi, A. and Izquierdo-Cortázar, D. (2013). Effort estimation of FLOSS projects: a study of the Linux kernel. *Empirical Software Engineering*, 18(1):60–88.

Capiluppi, A., Lago, P., and Morisio, M. (2003). Characteristics of open source projects. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 317–327.

Carmel, E. and Agarwal, R. (2001). Tactical approaches for alleviating distance in global software development. *IEEE Software*, 18(2):22–29.

Cataldo, M. and Herbsleb, J. D. (2013). Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering*, 39(3):343–360.

Cataldo, M., Herbsleb, J. D., and Carley, K. M. (2008). Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 2–11.

Cedergren, S. and Larsson, S. (2014). Evaluating performance in the development of software-intensive products. *Information and Software Technology*, 56(5):516–526.

Clark, K. and Fujimoto, T. (1990). The power of product integrity. *Harvard Business Review*, 68(6):107–118.

CMMI Product Team (2010). CMMI® for Development, Version 1.3. Technical report, Pittsburgh, USA.

Cockburn, A. (2000). Selecting a project's methodology. *IEEE Software*, 17(4):64–71.

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

Coffey, D. (2006). *The Myth of Japanese Efficiency*. Edward Elgar Publishing Limited, Cornwall, UK.

Conboy, K. (2009). Agility from first principles: Reconstructing the concept of agility in information systems development. *Information Systems Research*, 20(3):329–354.

Conradi, R. and Fuggetta, A. (2002). Improving software process improvement. *IEEE Software*, 19(4):92–99.

Coram, M. and Bohner, S. (2005). The impact of agile methods on software project management. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05*, pages 363–370.

Corbet, J., Kroah-Hartman, G., and McPherson, A. (2015). Linux Kernel Development: How Fast is it Going, Who is Doing It, What Are They Doing and Who is Sponsoring the Work. Online: `http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015`. [Retrieved 2015-04-01.].

Corona, E. and Pani, F. E. (2013). A Review of Lean-Kanban Approaches in the Software Development. *WSEAS Transactions on Information Science and Applications*, 10(1):1–13.

Creswell, J. (2009). *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications Inc., 3 edition.

Crowston, K., Wei, K., Howison, J., and Wiggins, A. (2008). Free/Libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1–7:35.

Csikszentmihalyi, M. (1975). *Beyond Boredom and Anxiety*. Jossey-Bass, San Francisco.

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. Harper & Row, New York.

Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. (2005). Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465.

Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th International Conference on Software Engineering*, ICSE '84, pages 97–106, Piscataway, NJ, USA. IEEE Press.

Curtis, B., Hefley, W. E., and Miller, S. A. (2001). People Capability Maturity Model® (P–CMM®). Technical report, Pittsburgh, USA.

Curtis, B., Krasner, H., and Iscoe, N. (1988). A Field Study of the Software Design Process for Large Systems. *Commun. ACM*, 31(11):1268–1287.

Curtis, B. and Waltz, D. (1990). The Psychology of Programming in the Large: Team and Organizational Behaviour. In Hoc, J.-M., Green, T., Samurcay, R., and Gilmore, D. J., editors, *Psychology of Programming*, pages 253–270. European Association of Cognitive Ergonomics and Academic Press.

Cusumano, M. A. (1989). The software factory: a historical interpretation. *IEEE Software*, 6(2):23–30.

Dagenais, B., Ossher, H., Bellamy, R. K. E., Robillard, M. P., and de Vries, J. P. (2010). Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 275–284, New York, NY, USA. ACM.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press Ltd., London, UK.

DeMarco, T. and Lister, T. (2013). *Peopleware: Productive Projects and Teams*. Addison-Wesley, 3 edition.

Deming, W. E. (1975). On Probability As a Basis For Action. *The American Statistician*, 29(4):146–152.

Dingsøyr, T. and Dybå, T. (2012). Team effectiveness in software development: Human and cooperative aspects in team effectiveness models and priorities for future studies. In *5th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 27–29.

Dingsøyr, T., Nerur, S., Balijepally, V., and Moe, N. B. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, 85(6):1213–1221.

Dogan, H., Pilfold, S., and Henshaw, M. (2011). The role of human factors in addressing Systems of Systems complexity. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1244–1249.

Ducheneaut, N. (2005). Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Comput. Supported Coop. Work*, 14(4):323–368.

Dybå, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9–10):833–859.

Dybå, T. and Sharp, H. (2012). What's the Evidence for Lean? *Software, IEEE*, 29(5):19–21.

Earthy, J., Jones, B. S., and Bevan, N. (2001). The Improvement of Human-centred Processes – Facing the Challenge and Reaping the Benefit of ISO 13407. *International Journal of Human-Computer Studies*, 55(4):553–585.

Eisenhardt, K. M. (1989). Building Theories from Case Study Research. *The Academy of Management Review*, 14(4):pp. 532–550.

Eisenhardt, K. M. and Graebner, M. E. (2007). Theory Building From Cases: Opportunities And Challenges. *Academy of Management Journal*, 50(1):25–32.

Endres, A. and Rombach, D. (2003). *A Handbook of Software and Systems Engineering. Empirical Observations, Laws and Theories*. The Fraunhofer IESE Series on Software Engineering. Addison Wesley.

Espinosa, J. A., Slaughter, S. A., Kraut, R. E., and Herbsleb, J. D. (2007). Team Knowledge and Coordination in Geographically Distributed Software Development. *Journal of Management Information Systems*, 24(1):135–169.

Estrada, C. A., Isen, A. M., and Young, M. J. (1997). Positive Affect Facilitates Integration of Information and Decreases Anchoring in Reasoning among Physicians. *Organizational Behavior and Human Decision Processes*, 72(1):117–135.

Feather, N. (1995). Values, Valences, and Choice: The Influence of Values on the Perceived Attractiveness and Choice of Alternatives. *Journal of Personality and Social Psychology*, 68(6):1135–1151.

Feather, N. T. (1996). *Values, deservingness, and attitudes toward high achievers: Research on tall poppies*, pages 215–251. Lawrence Erlbaum Associates, Inc, Hillsdale, NJ.

Feldt, R., Torkar, R., Angelis, L., and Samuelsson, M. (2008). Towards Individualized Software Engineering: Empirical Studies Should Collect Psychometrics. In *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '08, pages 49–52, New York, NY, USA. ACM.

Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1994). Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578.

Floyd, C. (1992). Software Development as Reality Construction. In Floyd, C., Züllighoven, H., Budde, R., and Keil-Slawik, R., editors, *Software Development and Reality Construction*, pages 86–100. Springer Berlin Heidelberg.

Forlizzi, J. and Battarbee, K. (2004). Understanding experience in interactive systems. In *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '04, pages 261–268, New York, NY, USA. ACM.

Forlizzi, J. and Ford, S. (2000). The Building Blocks of Experience: An Early Framework for Interaction Designers. In *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, DIS '00, pages 419–423, New York, NY, USA. ACM.

Franca, A., Carneiro, D., and da Silva, F. (2012a). Towards an Explanatory Theory of Motivation in Software Engineering: A Qualitative Case Study of a Small Software Company. In *Brazilian Symposium on Software Engineering (SBES 2012)*, pages 61–70.

Franca, A. C. C., de L. C. Felix, A., and da Silva, F. Q. B. (2012b). Towards an explanatory theory of motivation in software engineering: A qualitative case study of a government organization. In *International Conference on Evaluation and Assessment in Software Engineering (EASE 2012)*, pages 72–81.

Franca, A. C. C., Gouveia, T. B., Santos, P. C. F., Santana, C. A., and da Silva, F. Q. B. (2011). Motivation in software engineering: A systematic review update. In *15th Annual Conference on Evaluation and Assessment in Software Engineering*, pages 154–163.

Freeman, M. and Beale, P. (1992). Measuring project success. *Project Management Journal*, 23(1):8–17.

Freeman, P. and Hart, D. (2004). A Science of Design for Software-intensive Systems. *Communications of the ACM*, 47(8):19–21.

Gallese, V., Keysers, C., and Rizzolatti, G. (2004). A unifying view of the basis of social cognition. *Trends in Cognitive Sciences*, 8(9):396–403.

Gibbs, G. (2007). *Analyzing Qualitative Data*. The SAGE Qualitative Research Kit. SAGE Publications Inc., London.

Gladstein, D. L. (1984). Groups in Context: A Model of Task Group Effectiveness. *Administrative Science Quarterly*, 29(4):499–517.

Glaser, B. and Strauss, A. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, Chicago.

Glass, R. (2006). *Software Creativity 2.0*. developer.* Books, Atlanta, Georgia.

Glass, R., Vessey, I., and Ramesh, V. (2002). Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491–506.

Grand, S., Von Krogh, G., Leonard, D., and Swap, W. (2004). Resource allocation beyond firm boundaries: A multi-level model for open source innovation. *Long Range Planning*, 37(6):591–610.

Graziotin, D., Wang, X., and Abrahamsson, P. (2013). Are happy developers more productive? The correlation of affective states of software developers and their self-assessed productivity. In *Proceedings of the 14th International Conference on Product-Focused Software Process Improvement*, volume 7983 LNCS of *Lecture Notes in Computer Science*, pages 50–64.

Graziotin, D., Wang, X., and Abrahamsson, P. (2014a). Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering. *Journal of Software: Evolution and Process*.

Graziotin, D., Wang, X., and Abrahamsson, P. (2014b). Happy software developers solve problems better: psychological measurements in empirical software engineering. *PeerJ*, 2.

Graziotin, D., Wang, X., and Abrahamsson, P. (2014c). Software Developers, Moods, Emotions, and Performance. *IEEE Software*, 31(4):24–27.

Graziotin, D., Wang, X., and Abrahamsson, P. (2015). Understanding the Affect of Developers: Theoretical Background and Guidelines for Psychoempirical Software Engineering. In *Proceedings of the 7th International Workshop on Social Software Engineering*, SSE 2015, pages 25–32, New York, NY, USA. ACM.

Guba, E. E. and Lincoln, Y. (1989). *Fourth generation evaluation*. SAGE Publications, Inc.

Hackman, J. (1987). The design of work teams. In Lorsch, J., editor, *Handbook of organizational behavior*, pages 315–324. Prentice Hall, Englewood Cliffs, NJ.

Hackman, J. (1998). Why teams don't work. In Tindale, R., Heath, L., and Edwards, J., editors, *Theory and research on small groups*, pages 245–267. Plenum, New York.

Hall, T., Jagielska, D., and Baddoo, N. (2007). Motivating developer performance to improve project outcomes in a high maturity organization. *Software Quality Journal*, 15(4):365–381.

Hannay, J. E., Sjøberg, D. I., and Dybå, T. (2007). A Systematic Review of Theory Use in Software Engineering Experiments. *IEEE Transactions on Software Engineering*, 33(2):87.

Hansen, M. E., Lumsdaine, A., and Goldstone, R. L. (2012). Cognitive architectures: a way forward for the psychology of programming. In

Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, Onward! '12, pages 27–38, New York, NY, USA. ACM.

Hassenzahl, M. (2004). Funology: From Usability to Enjoyment. Number 12, chapter The thing and I: understanding the relationship between user and product, pages 31–42. Kluwer Academic Publishers, Norwell, MA, USA.

Hazzan, O. and Dubinsky, Y. (2003). Bridging Cognitive and Social Chasms in Software Development Using Extreme Programming. In Marchesi, M. and Succi, G., editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 47–53. Springer Berlin Heidelberg.

Herbsleb, J. (2005). Beyond computer science. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 23–27, St. Louis, MO.

Herbsleb, J. D. and Mockus, A. (2003a). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494.

Herbsleb, J. D. and Mockus, A. (2003b). Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. In Inverardi, P., editor, *Proceedings of the Joint European Software Engineering Conference (ESEC) and SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)*, pages 138–147, Helsinki.

Herbsleb, J. D. and Moitra, D. (2001). Global software development. *IEEE Software*, 18(2):16–20.

Hertzum, M., Clemmensen, T., Hornbæk, K., Kumar, J., Shi, Q., and Yammiyavar, P. (2007). Usability Constructs: A Cross-Cultural Study of How Users and Developers Experience Their Use of Information Systems. In Aykin, N., editor, *Usability and Internationalization. HCI and Culture*, volume 4559 of *Lecture Notes in Computer Science*, pages 317–326. Springer Berlin Heidelberg.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105.

Hibbs, C., Jewett, S., and Sullivan, M. (2009). *The art of lean software development: a practical and incremental approach.* O'Reilly Media, Inc., CA.

Highsmith, J. (2001). History: The Agile Manifesto. Online: `http://agilemanifesto.org/history.html`. [Retrieved 2014-01-01.].

Hilgard, E. R. (1980). The trilogy of mind: Cognition, affection, and conation. *Journal of the History of the Behavioral Sciences*, 16(2):107–117.

Hines, P., Holwe, M., and Rich, N. (2004). Learning to evolve: A review of contemporary lean thinking. *International Journal of Operations & Production Management*, 24(9):994–1011.

Hoc, J.-M. (1988). Cognitive Psychology of Planning.

Hoc, J.-M., Green, T., Samurcay, R., and Gilmore, D., editors (1990). *Psychology of Programming*. Academic Press.

Holmstrom, H., Conchúir, E. Ó., Ågerfalk, P. J., and Fitzgerald, B. (2006). Global Software Development Challenges: A Case Study on Temporal, Geographical and Socio-Cultural Distance. In *International Conference on Global Software Engineering*, pages 3–11.

Holweg, M. and Pil, F. (2001). Successful Build-To-Order Strategies Start With the Customer. *MIT Sloan Management Review*, 43(1):74–83.

Höst, M., Regnell, B., and Wohlin, C. (2000). Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5(3):201–214.

Humphrey, W. (1988). Characterizing the software process: a maturity framework. *IEEE Software*, 5(2):73–79.

Humphrey, W. (1989). *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc.

Ikonen, M., Pirinen, E., Fagerholm, F., Kettunen, P., and Abrahamsson, P. (2011). On the Impact of Kanban on Software Project Work: An Empirical Case Study Investigation. In *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 305–314.

International Data Corporation (2013). 2014 Worldwide Software Developer and ICT-Skilled Worker Estimates. Technical report.

Isen, A. M. (2004). Positive affect facilitates thinking and problem solving. In Manstead, A. S. R., Frijda, N., and Fiscer, A., editors, *Feelings and emotions: the Amsterdam symposium*, pages 263–281. Cambridge University Press.

Isen, A. M. and Reeve, J. (2005). The Influence of Positive Affect on Intrinsic and Extrinsic Motivation: Facilitating Enjoyment of Play, Responsible Work Behavior, and Self-Control. *Motivation and Emotion*, 29(4):295–323.

Isen, A. M., Rosenzweig, A. S., and Young, M. J. (1991). The Influence of Positive Affect on Clinical Problem solving. *Medical Decision Making*, 11(3):221–227.

Jacobson, I. and Seidewitz, E. (2014). A New Software Engineering. *Commun. ACM*, 57(12):49–54.

Kahle, L. (1996). *Social values and consumer behavior: Research from the list of values*, pages 135–151. Lawrence Erlbaum Associates, Inc.

Kalfoglou, Y., Menzies, T., Althoff, K.-D., and Motta, E. (2000). Meta-knowledge in systems design: panacea. . . or undelivered promise? *The Knowledge Engineering Review*, 15(4):381–404.

Känsälä, M. and Tuomivaara, S. (2013). Do Agile Principles and Practices Support the Well-being at Work of Agile Team Members? In *Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA 2013)*, pages 364–367.

Katzenbach, J. R. and Smith, D. K. (1993). *The Wisdom of Teams: Creating the High-Performance Organization.* McKinsey & Company, Inc., New York, NY.

Kettunen, P. (2013). Bringing Total Quality in to Software Teams: A Frame for Higher Performance. In Fitzgerald, B., Conboy, K., Power, K., Valerdi, R., Morgan, L., and Stol, K.-J., editors, *Lean Enterprise Software and Systems*, volume 167 of *Lecture Notes in Business Information Processing*, pages 48–64. Springer Berlin Heidelberg.

Khan, I. A., Brinkman, W.-P., and Hierons, R. M. (2011). Do moods affect programmers' debug performance? *Cognition, Technology & Work*, 13(4):245–258.

Kittlaus, H.-B. and Clough, P. N. (2009). Software Products: Terms and Characteristics. In *Software Product Management and Pricing*, pages 5–15. Springer Berlin Heidelberg.

Knuth, D. (1997a). *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, 3 edition.

Knuth, D. (1997b). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2 edition.

Knuth, D. (1998). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 3 edition.

Knuth, D. (2011). *Combinatorial Algorithms, Part1*, volume 4a of *The Art of Computer Programming*. Addison-Wesley.

Krafcik, J. F. (1988). Comparative analysis of performance indicators at world auto assembly plants. Master's thesis, Sloan School of Management, Massachusetts Institute of Technology.

Kraut, R. E. and Streeter, L. A. (1995). Coordination in Software Development. *Commun. ACM*, 38(3):69–81.

Kroeger, T. A., Davidson, N. J., and Cook, S. C. (2014). Understanding the characteristics of quality for software engineering processes: A Grounded Theory investigation. *Information and Software Technology*, 56(2):252–271.

Kruchten, P., Obbink, H., and Stafford, J. (2006). The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30.

Kuzniarz, L., Staron, M., and Wohlin, C. (2003). Students as Study Subjects in Software Engineering Experimentation. In *Proceedings of the 3rd Conference on Software Engineering Research and Practice in Sweden*, pages 19–24, Lund, Sweden.

Laanti, M. (2013). Agile and Wellbeing – Stress, Empowerment, and Performance in Scrum and Kanban Teams. In *46th Hawaii International Conference on System Sciences (HICSS 2013)*, pages 4761–4770.

Laanti, M. (2014). Characteristics and Principles of Scaled Agile. In Dingsøyr, T., Moe, N., Tonelli, R., Counsell, S., Gencel, C., and Petersen, K., editors, *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation*, volume 199 of *Lecture Notes in Business Information Processing*, pages 9–20. Springer International Publishing.

Labuschagne, A. and Holmes, R. (2015). Do Onboarding Programs Work? In *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy. IEEE. To appear.

Ladas, C. (2008). *Scrumban: Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, Seattle, WA.

Lane, M., Fitzgerald, B., and Ågerfalk, P. (2012). Identifying lean software development values. In *Proceedings of European Conference on Information Systems (ECIS)*, Barcelona.

Larman, C. and Basili, V. (2003). Iterative and Incremental Developments: A Brief History. *IEEE Computer*, 36(6):47–56.

Law, A. and Charron, R. (2005). Effects of Agile Practices on Social Factors. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5.

Lawrence, C. and Rodriguez, P. (2012). The Interpretation and Legitimization of Values in Agile's Organizing Vision. *Proceedings of the European Conference on Information Systems (ECIS)*, pages 10–13.

Lehman, M. (1969). The Programming Process. Technical report, IBM Research Centre, Yorktown Heights, NY.

Lenberg, P., Feldt, R., and Wallgren, L.-G. (2014). Towards a Behavioral Software Engineering. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 48–55, New York, NY, USA. ACM.

Lenberg, P., Feldt, R., and Wallgren, L. G. (2015). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37.

Liang, T.-P., Liu, C.-C., Lin, T.-M., and Lin, B. (2007). Effect of team diversity on software project performance. *Industrial Management and Data Systems*, 107(5):636–653.

Liker, J. (2004). *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*. McGraw-Hill, New York.

Liker, J. and Hoseus, M. (2008). *Toyota Culture: The Hearth and Soul of the Toyota Way*. McGraw-Hill.

Lincoln, Y. and Guba, E. E. (1985). Naturalistic inquiry.

Lingard, R. and Berry, E. (2002). Teaching teamwork skills in software engineering based on an understanding of factors affecting group performance. In *Frontiers in Education, 2002*, volume 3, pages S3G–1–S3G–6.

Madachy, R. (2008). *Software Process Dynamics*. John Wiley & Sons, Inc., Hoboken, New Jersey.

Mannaro, K., Melis, M., and Marchesi, M. (2004). Empirical Analysis on the Satisfaction of IT Employees Comparing XP Practices with Other Software Development Methodologies. In Eckstein, J. and Baumeister, H., editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3092 of *Lecture Notes in Computer Science*, pages 166–174. Springer Berlin Heidelberg.

McCarthy, J. and Wright, P. (2004). Technology as experience. *Interactions*, 11(5):42–43.

Meijer, E. and Kapoor, V. (2014). The Responsive Enterprise: Embracing the Hacker Way. *Communications of the ACM*, 57(12):38–43.

Melnik, G. and Maurer, F. (2006). Comparative Analysis of Job Satisfaction in Agile and Non-agile Software Development Teams. In Abrahamsson, P., Marchesi, M., and Succi, G., editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 4044 of *Lecture Notes in Computer Science*, pages 32–42. Springer Berlin Heidelberg.

Merriam, S. (2009). *Qualitative research: a guide to design and implementation.* Jossey-Bass, 2 edition.

Michod, R. E. (1993). *Biology and the origin of values*, pages 261–272. Aldine de Gruyter, Hawthorne, NY.

Middleton, P. and Sutton, J. (2005). *Lean software strategies: proven techniques for managers and developers.* Productivity Press, New York.

Mills, J. A. (1985). A Pragmatic View of the System Architect. *Commun. ACM*, 28(7):708–717.

Mingers, J. and Walsham, G. (2010). Toward ethical information systems: The contribution of discourse ethics. *MIS Quarterly: Management Information Systems*, 34(4):855–870.

Misra, S. and Akman, I. (2014). A Cognitive Model for Meetings in the Software Development Process. *Human Factors and Ergonomics in Manufacturing & Service Industries*, 24(1):1–13.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

Mockus, A. and Herbsleb, J. (2001). Challenges of global software development. In *Proceedings of the Seventh International Software Metrics Symposium.*, pages 182–184.

Moe, N., Dingsøyr, T., and Dybå, T. (2008). Understanding Self-Organizing Teams in Agile Software Development. In *19th Australian Conference on Software Engineering (ASWEC 2008)*, pages 76–85.

Moran, T. P. (1996). *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates Inc., Hillsdale, NJ, USA.

Morgan, J. M. and Liker, J. K. (2006). *The Toyota Product Development System: Integrating People, Process, and Technology*. Productivity press, New York, NY, USA.

Mumford, M., Helton, W., Decker, B., Connelly, M., and Doorn, J. V. (2003). Values and Beliefs Related to Ethical Decisions. *Teaching Business Ethics*, 7(2):139–170.

Münch, J., Armbrust, O., Kowalczyk, M., and Soto, M. (2012). *Software Process Definition and Management*. Springer-Verlag.

Naylor, J. B., Naim, M. M., and Berry, D. (1999). Leagility: Integrating the lean and agile manufacturing paradigms in the total supply chain. *International Journal of Production Economics*, 62(1–2):107–118.

Nygaard, K. and Dahl, O.-J. (1978). The Development of the SIMULA Languages. *SIGPLAN Not.*, 13(8):245–272.

Oakland, J. S. (2008). *Statistical Process Control*. Butterworth-Heineman, Oxford, UK, 6 edition.

Ōno, T. (1988). *Toyota production system: beyond large-scale production*. Productivity press.

Osterweil, L. (1987). Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 2–13, Los Alamitos, CA, USA. IEEE Computer Society Press.

Oza, N., Fagerholm, F., and Münch, J. (2013). How Does Kanban Impact Communication and Collaboration in Software Engineering Teams? *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2013)*, pages 125–128.

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Berlin Heidelberg.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit.* Addison-Wesley Professional, Boston, MA, USA.

Powell, A., Piccoli, G., and Ives, B. (2004). Virtual Teams: A Review of Current Literature and Directions for Future Research. *SIGMIS Database*, 35(1):6–36.

Project Management Institute (2013). *A Guide to the Project Management Body of Knowledge (PMBOK® Guide).* Project Management Institute, Inc., Pennsylvania, USA, 5 edition.

Ralph, P., Johnson, P., and Jordan, H. (2013). Report on the First SEMAT Workshop on General Theory of Software Engineering. *SIGSOFT Software Engineering Notes*, 38(2):26–28.

Ralph, P. and Kelly, P. (2014). The Dimensions of Software Engineering Success. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 24–35, New York, NY, USA. ACM.

Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media, Inc.

Remus, W. (1989). Using students as subjects in experiments on decision support systems. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume 3, pages 176–180 vol.3.

Riemenschneider, C., Hardgrave, B., and Davis, F. (2002). Explaining software developer acceptance of methodologies: a comparison of five theoretical models. *IEEE Transactions on Software Engineering*, 28(12):1135–1145.

Ries, E. (2011). *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation To Create Radically Successful Businesses.* Crown Business.

Rittel, H. and Webber, M. (1973). Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169.

Robillard, P. N. (1999). The Role of Knowledge in Software Development. *Commun. ACM*, 42(1):87–92.

Robillard, P. N., d'Astous, P., Détienne, F., and Visser, W. (1998). Measuring Cognitive Activities in Software Engineering. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 292–299, Washington, DC, USA. IEEE Computer Society.

Rokeach, M. (1973). *The nature of human values.* Free press, New York.

Rokeach, M. (1979). *Understanding human values.* Free press, New York.

Rombach, D. (2011). Empirical Software Engineering Models: Can They Become the Equivalent of Physical Laws in Traditional Engineering? *International Journal of Software and Informatics*, 5(3):525–534.

Rosove, P. (1967). *Developing computer-based information systems.* Wiley, New York, NY, USA.

Roto, V., Law, E., Vermeeren, A., and Hoonhout, J., editors (2011). *User experience white paper: Bringing clarity to the concept of user experience.* Result from Dagstuhl Seminar on Demarcating User Experience, September 15-18, 2010.

Rouanet, J. and Gateau, V. (1967). *Le travail du programmeur de gestion: essai de description.* AFPA-CERP, Paris.

Royce, W. E. and Royce, W. (1991). Software Architecture: Integrating Process and Technology. *TRW Quest*, 14(1):2–15.

Royce, W. W. (1970). Managing the development of large software systems. Los Angeles, CA, USA.

Runeson, P. (2003). Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data. In *Proceedings 7th International Conference on Empirical Assessment & Evaluation in Software Engineering*, pages 95–102.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164.

Runeson, P., Höst, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples.* John Wiley & Sons.

Rus, I. and Lindvall, M. (2002). Knowledge management in software engineering. *IEEE Software*, 19(3):26–38.

Sabherwal, R. (1999). The Role of Trust in Outsourced IS Development Projects. *Communications of the ACM*, 42(2):80–86.

Sach, R. and Petre, M. (2012). Feedback: How does it impact software engineers? In *5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 129–131.

Salas, E., Sims, D. E., and Burke, C. S. (2005). Is there a "Big Five" in Teamwork? *Small Group Research*, 36(5):555–599.

Saruta, M. (2006). Toyota Production Systems: The 'Toyota Way' and Labour-Management Relations. *Asian Business & Management*, 5(4):487.

Sawyer, S. and Guinan, P. (1998). Software development: Processes and performance. *IBM Systems Journal*, 37(4):552–569.

Scaffidi, C., Shaw, M., and Myers, B. (2005). Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214.

Schön, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. Temple Smith, London.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall.

Schwartz, S. (1992). Universals in the content and structure of values: Theoretical advances and empirical tests in 20 countries. *Advances in experimental social psychology*, 25(1):1–65.

Schwartz, S. and Bilsky, W. (1987). Toward A Universal Psychological Structure of Human Values. *Journal of Personality and Social Psychology*, 53(3):550–562.

Schwartz, S. and Bilsky, W. (1990). Toward a Theory of the Universal Content and Structure of Values: Extensions and Cross-Cultural Replications. *Journal of Personality and Social Psychology*, 58(5):878–891.

Schwartz, S. H. (1999). A Theory of Cultural Values and Some Implications for Work. *Applied Psychology*, 48(1):23–47.

Seaman, C., Mendonca, M., Basili, V., and Kim, Y. (2003). User interface evaluation and empirically-based evolution of a prototype experience management tool. *IEEE Transactions on Software Engineering*, 29(9):838–850.

Sharp, H., Hall, T., Baddoo, N., and Beecham, S. (2007). Exploring motivational differences between software developers and project managers. In *Proceedings of the 6th Joint Meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, ESEC-FSE '07, pages 501–504, New York, NY, USA. ACM.

Shaw, M. (2003). Writing Good Software Engineering Research Papers: Minitutorial. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 726–736, Washington, DC, USA. IEEE Computer Society.

Shaw, T. (2004). The Emotions of Systems Developers: An Empirical Study of Affective Events Theory. In *Proceedings of the 2004 SIGMIS Conference on Computer Personnel Research: Careers, Culture, and Ethics in a Networked Environment*, SIGMIS CPR '04, pages 124–126, New York, NY, USA. ACM.

Shewhart, W. A. (1931). *Economic Control of Quality of Manufactured Product*. D. Van Nostrand Company, New York, NY, USA.

Shibuya, B. and Tamai, T. (2009). Understanding the process of participating in open source communities. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, pages 1–6, Washington, DC, USA. IEEE Computer Society.

Shimokawa, K. (2010). *Japan and the Global Automotive Industry*. Cambridge University Press, Cambridge, UK.

Simon, H. A. (1969). *The Sciences of the Artificial*. MIT Press, Cambridge, MA, USA, 3 edition.

Sjøberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanovic, A., Liborg, N.-K., and Rekdal, A. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753.

Sjøberg, D. I. K., Dybå, T., Anda, B. C. D., and Hannay, J. E. (2008). Building Theories in Software Engineering. In Shull, F., Singer, J., and Sjøberg, D. I. K., editors, *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer London.

Smith, P., Peterson, M., and Schwartz, S. (2002). Cultural Values, Sources of Guidance, and their Relevance to Managerial Behavior: A 47-Nation Study. *Journal of Cross-Cultural Psychology*, 33(2):188–208.

Sommerville, I. (2011). *Software Engineering and Computer Science*. Pearson Education Inc., Boston, USA, 9 edition.

Sowe, S., Stamelos, I., and Angelis, L. (2006). Identifying knowledge brokers that yield software engineering knowledge in OSS projects. *Information and Software Technology*, 48(11):1025–1033.

Sowe, S. K., Stamelos, I., and Angelis, L. (2008). Understanding knowledge sharing activities in free/open source software projects: An empirical study. *Journal of Systems and Software*, 81(3):431–446.

Stapleton, J. and Constable, P. (1997). DSDM: Dynamic Systems Development Method: The Method in Practice. Addison-Wesley Professional.

Stringer, E. (2014). *Action Research*. SAGE Publications Inc., Thousand Oaks, California, 4 edition.

Sudhakar, G. P., Farooq, A., and Patnaik, S. (2011). Soft factors affecting the performance of software development teams. *Team Performance Management*, 17(3):187–205.

Susman, G. I. and Evered, R. D. (1978). An Assessment of the Scientific Merits of Action Research. *Administrative Science Quarterly*, pages 582–603.

Sutherland, J. and Schwaber, K. (1995). SCRUM Development Process. Business Object Design and Implementation Workshop. In *Addendum to the Proceedings of the 10th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '95, pages 170–175, New York, NY, USA. ACM.

Svahnberg, M., Aurum, A., and Wohlin, C. (2008). Using Students As Subjects - an Empirical Evaluation. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 288–290, New York, NY, USA. ACM.

Swap, W., Leonard, D., Shields, M., and Abrahams, L. (2001). Using Mentoring and Storytelling to Transfer Knowledge in the Workplace. *J. Manage. Inf. Syst.*, 18(1):95–114.

Syed-Abdullah, S., Karn, J., Holcombe, M., Cowling, T., and Gheorge, M. (2005). The Positive Affect of the XP Methodology. In Baumeister, H., Marchesi, M., and Holcombe, M., editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 218–221. Springer Berlin Heidelberg.

The Agile Alliance (2001). The Agile Manifesto. Online: `http://www.agilemanifesto.org/`. [Retrieved 2014-01-01.].

Tichy, W. F. (2000). Hints for Reviewing Empirical Work in Software Engineering. *Empirical Software Engineering*, 5(4):309–312.

van Aken, J. E. (2004). Management Research Based on the Paradigm of the Design Sciences: The Quest for Field-Tested and Grounded Technological Rules. *Journal of Management Studies*, 41(2):219–246.

von Hippel, E. and von Krogh, G. (2003). Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science. *Organization science*, 14(2):209–223.

von Krogh, G. and Spaeth, S. (2007). The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems*, 16(3):236–253.

von Krogh, G., Spaeth, S., and Haefliger, S. (2005). Knowledge Reuse in Open Source Software: An Exploratory Study of 15 Open Source Projects. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 198b–198b.

Šmite, D., Wohlin, C., Gorschek, T., and Feldt, R. (2010). Empirical evidence in global software engineering: a systematic review. *Empirical Software Engineering*, 15(1):91–118.

Wallgren, L. G. and Hanse, J. J. (2007). Job characteristics, motivators and stress among information technology consultants: A structural equation modeling approach. *International Journal of Industrial Ergonomics*, 37(1):51–59.

Wang, Y. and Patel, S. (2009). Exploring the Cognitive Foundations of Software Engineering. *International Journal of Software Science and Computational Intelligence*, 1(2):1–19.

Warfield, D. (2010). IS/IT research: A research methodologies review. *Journal of Theoretical and Applied Information Technology*, 13(1):28–35.

Weeks, W. and Kahle, L. (1990). Social values and salespeople's effort. Entrepreneurial versus routine selling. *Journal of Business Research*, 20(2):183–190.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York, NY.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Berlin Heidelberg.

Womack, J. P., Jones, D., and Roos, D. (2007). *The Machine That Changed the World*. Simon & Schuster.

Wrobel, M. (2013). Emotions in the software development process. pages 518–523, Gdansk, Sopot.

Yin, R. (2009). *Case study research: design and methods*. SAGE Publications, Inc., 4 edition.

Zannier, C., Melnik, G., and Maurer, F. (2006). On the Success of Empirical Studies in the International Conference on Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 341–350, New York, NY, USA. ACM.