Collaborative Software Engineering

Ivan Mistrík · John Grundy · André van der Hoek ·
Jim Whitehead
Editors

Collaborative Software Engineering

Springer

Editors
Ivan Mistrík
Independent Consultant
Werderstr. 45
69120 Heidelberg
Germany
i.j.mistrik@t-online.de

John Grundy
Centre for Complex Software
Systems & Services
Swinburne University of Technology
Faculty of Information and
Communication Technologies
PO Box 218
Hawthorn, Victoria
Australia 3122
jgrundy@swin.edu.au

André van der Hoek
University of California, Irvine
Donald Bren School of
Information & Computer Sciences
5029 Donald Bren Hall
Irvine CA 92697-3440
USA
andre@ics.uci.edu

Jim Whitehead
University of California, Santa Cruz
Dept. Computer Science
Santa Cruz CA 95064
USA
ejw@cs.ucsc.edu

ISBN 978-3-642-10293-6 e-ISBN 978-3-642-10294-3
DOI 10.1007/978-3-642-10294-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2010921110

ACM Computing Classification (1998): D.2, K.6

Cover design: KuenkelLopka GmbH, Heidelberg

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

While many empirical studies over the years have shown that software development skills and aptitude vary between individuals, the reality is that the size, complexity and longevity of software development projects and artefacts far exceed what any individual software developer can manage on her own. Collaboration among individuals – from users to developers – is therefore central to modern day software engineering. Collaboration takes many forms: joint activity to solve common problems, complementary activity to solve diverse problems, and both social and technical perspectives impacting all software development activity.

The difficulties of collaboration are also well documented. For example, when managerial instinct in dealing with a problematic software project was to add more developers to the development team, Fred Brooks observed and argued in his classic book *The Mythical Man Month* (Addison-Wesley, 1975) that such additions impaired rather than speeded up development. Reflecting on Brooks' observation, one could argue that it is not the addition of developers *per se* that is problematic, but the lack of effective means by which they are able to *collaborate effectively* that is crucial. Indeed the grand challenge of effective collaboration is not only to ensure that developers in a team deliver effectively as individuals, but that the whole team delivers more than the sum of its parts.

Enabling effective collaboration of course is easier said than done. As this book shows, there are many dimensions of collaboration, and many different development contexts in which different forms of collaboration are necessary and effective. The many tools and techniques that work in one context may not work in another. Collaborative software engineering therefore provides a fertile ground for empirical research on collaborative practices and collaboration tools, for technology research on developing tools and techniques for supporting collaboration, and operational research to understand organisational structures, processes, and experiences that impact, or are impacted by collaboration. This book is a welcome contribution to the research discourse in all these areas of study.

As a doctoral student some 20 years ago, I was very interested in understanding and supporting multiple software development stakeholders, as they articulated their differing perspectives of software problems and solutions, developed some shared understanding of their problem and solution worlds, and crucially important in my view, as they agreed to disagree about the parts of the world where their perspectives differed. Acknowledging, understanding and tackling disagreements head on was and is, in my view, fundamental to effective collaboration, and remains at the heart of collaborative software engineering research. While much progress has been made in the area of conflict management research, I believe that it remains a key area for tackling the challenges of supporting effective collaborative software engineering.

The editors of this book have assembled an impressive selection of authors, who have contributed authoritative body of work tackling a wide range of issues in the field of collaborative software engineering. The book will be of tremendous value to practitioners grappling with managing multi-person software development activity, as well as researchers and students interested in the state-of-the-art and the many research directions in this area. The volume is not simply a collection of papers, but a thoughtful assembly of contributions, suitably structured and introduced by the editorial team. Many of the chapters reflect on a body of research and practice that spans many years gone past, while other chapters pose research questions and describe research problems that are fundamental and long-standing. The result is a reference book, a research resource, and a pleasurable read.

Milton Keynes, UK Bashar Nuseibeh
June 2009

Preface

Software engineering is almost always a collaborative activity. This book brings together a number of recent contributions to the domain of Collaborative Software Engineering (CoSE) from a range of research groups and practitioners. These range from tools and techniques for managing discrete, low-level activities developers engage in when developing parts of software systems; knowledge, project and process management for large scale collaborative software engineering enterprises; and new ways of organizing software teams including outsourcing, open sourcing, highly distributed virtual teams and global software engineering. We believe that all practitioners engaging in or managing collaborative software engineering practices, researchers contributing to advancement of our understanding and support for collaborative software engineering, and students wishing to gain a deeper appreciation of the underpinning theories, issues and practices within this domain will benefit from most if not all of these contributions.

Introduction

Ever since people began to create software there has been a need for collaborative software engineering. At some point people need to share their code and designs with others. Software frequently grows large and complex, thus requiring a team of multi-talented experts to work together to tackle the project. Such a team must adopt suitable processes and project management to ensure the myriad of tasks are completed; to keep track of what each other is doing; and to ensure the project advances on-time, on-budget and with the software meeting appropriate quality levels. The team must share both low-level artifacts and higher-level knowledge in controlled, consistent ways, be proactively informed of changes others make, and co-ordinate their work "in the small" as well as "in the large". Various studies have demonstrated that peer review of designs and code improve them, leading to collaborative testing and quality assurance practices. Recent trends have moved software across organizational and country boundaries, including virtual software teams and open source software development. Agile methods have brought bottom-up, human-oriented processes and techniques to bear that are very different from traditional, centralized and hierarchical development practices.

Our understanding of and support for collaborative software engineering has advanced tremendously over the past forty years. We understand that team formation and management is not a straightforward task. However we are still learning about formation, management and evolution in domains such as agile teams, projects with substantive outsourcing, open source software, virtual software teams and global software engineering domains. Knowledge management is critical in software engineering and we have developed as a community many approaches to representing knowledge about software as well as tools to facilitate its capture. However, shared, evolving knowledge and appropriate tools and techniques to support this is less well-understood from both theoretic and practical standpoints. How do we best represent and collaboratively manage knowledge about requirements, architecture, designs, quality assurance measures and software processes themselves? Social influences on software engineering and teams have become more important as have organizational implications. How do team members relate to one other and how to we build effective team relationships for communication, co-ordination and collaboration? How do we set up a successful multi-site software project? A successful open source project? A successful outsourcing project?

The actual act of collaborative software creation has received much attention over many years. But what are the right sets of tools and work practices to deploy on a collaborative software engineering project to best-support engineers and ensure quality? What are the unsolved issues around co-ordination especially in large or highly distributed teams? Configuration management remains one of the most challenging activities in collaborative software engineering.

Book Overview

We have divided this book into four parts, with a general editorial chapter providing a more detailed review of the domain of collaborative software engineering. We received a large number of submissions in response to our call for papers and invitations for this edited book from many leading research groups and well-known practitioners of leading collaborative software engineering techniques. After a rigorous review process 17 submissions were accepted for this publication. We begin by a review of the concept of collaborative software engineering including a brief review of its history, key fundamental challenges, conceptual models for reasoning about collaboration in software engineering, technical, social and managerial considerations, and define the main issues in collaborative software engineering.

Part I contains five chapters that characterize collaborative software engineering. This includes characterizing global software engineering via a process-centric approach, requirements-driven collaboration using requirements/people relationships, decoupling in collaborative software engineering, agile software development and co-ordination, communication and collaboration, and applying the concept of ontologies to collaborative software engineering.

Part II contains five chapters that examine various techniques and tool support issues in collaborative software engineering. This includes an analysis of awareness support in collaborative software development teams, an overview of several approaches and tools to supporting continuous co-ordination, a maturity model for outsourcing offshore, an architectural knowledge management platform, and a set of design principles for collaborative software engineering environments.

Part III contains three chapters addressing the issue of organizational issues in collaborative software engineering. This includes supporting the concept of collaborative software analysis and making analysis tools widely accessible, open source software project communication and collaboration analysis and visualization support, and a review and critique of multi-site software development practices.

Part IV contains four chapters looking at a variety of related issues in the collaborative software engineering domain. These include key open source/free software development collaboration issues, configuration management and collaborative development, knowledge sharing to support collaborative software architecting, and rationale management to enhance collaborative requirements engineering. We conclude with a summary of current challenges and future directions in collaborative software engineering.

What Is Collaborative Software Engineering?

Collaboration has been a necessity ever since software engineering began. The early days of software engineering saw very limited process, technique and tool support for collaboration. Early efforts to support collaboration were limited to structured, waterfall-based processes, early version control tools, rigid team role specialization, and centralization of software activities. The advent of Computer-Aided Software Engineering tools and Integrated Development Environments introduced a wider, more accessible range of collaboration support mechanisms including awareness support, collaborative analysis and reviews and iterative, rapid applications development processes. More recently has seen the growth of distributed teams, outsourcing, open source software projects, global software engineering processes and highly decentralized team support tools.

Fundamental challenges in collaborative software engineering remain the same: the need to share artifacts, communicate and co-ordinate work. These occur across a spectrum of low-level to high-level. Low-level challenges include making shared artifacts like code, tests and designs accessible in a timely manner to team members while controlling access, ownership, integrity and quality. Large software projects require effective version control and configuration management techniques and tools. Knowledge management is fundamental especially around design rationale, architecture and processes. Software development has changed dramatically over the past 10 years. This is evidenced by new organizational and team dynamics including open source software, software outsourcing, distributed teams, and global software engineering. Choice of processes, project management, tools and evolution of software in these domains is still an emerging field of research and practice.

Key technical considerations in collaborative software engineering revolve around process, project management, knowledge and configuration management and tool platform selection and operation. A software process and project management regime must be chosen that supports collaboration appropriate to the team, project and organizational circumstances. These range from small, single-site/single-project teams, to large team/multi-project/multi-site domains. The later may include outsourcing and open source components. Complex software systems require effective knowledge management approaches and support tools. They also require scalable configuration management tools. Tool platforms and collaboration-supporting components have become very diverse. These range from small-team, homogeneous IDEs with awareness and collaboration plug-ins to highly diverse platforms where software engineering is part of a larger systems engineering activity. Communication support between engineers often becomes a crucial component of the team support infrastructure.

Being an inter-personal and–often–inter-organizational activity, collaborative software engineering introduces a number of social and managerial challenges. Teams may be homogeneous or highly diverse in terms of culture, language and location. This introduces many challenges to supporting collaboration at high levels (process, project management) and low-levels (artifact sharing, consistency). Teams may be comprised of many generalist's e.g., agile methods or highly specialized individuals or sub-teams whose efforts must be coordinated. An organization needs to ensure appropriate management of teams and between teams. In particular, global software engineering domains introduce very new and challenging problems, such as in contracting and quality control in outsourcing, ownership and "group dynamics" in multi-site projects, and overall project direction and co-ordination in open source software projects.

Part I – Characterizing Collaborative Software Engineering

The five papers in this section identify a range of themes around the characteristics of collaborative software engineering. There has been a dramatic increase in interest in the concept of "global software engineering" over the past 10 years. This has included the increasing number of distributed, multi-site software engineering teams; outsourcing of software engineering activities, often in search of cost savings and capacity limits, and open source software development. Each of these trends brings with it added complexity to the engineering process—software engineers are no longer co-located, are no longer in regular face-to-face contact (if at all), and different time zones, cultures and languages enter the mix.

A number of studies have been undertaken to better-understand the issues of collaboration challenges in such "virtual" software team environments. A key aim is to understand factors that adversely impact on collaboration practices and factors that support communication, co-ordination and collaboration in such domains. Studies have focuses on a range of organizations, projects and team sizes. One area of particularly detailed study has been requirements engineering. A distributed team develops and shares a set of requirements and a crucial factor impacting quality of these is communication strategies.

Knowledge engineering has become important in collaborative software engineering. One aspect is the development of ontologies, or shared semantic meanings, of software artifacts and processes. These enable co-ordination of activities along with improved communication about shared concepts in domains ranging from requirements engineering to software architecture.

Agile methods have become popular in many domains of software engineering. A characteristic is their focus on people-centric aspects of software engineering tasks, including communication and co-ordination. Pairing is one aspect of several agile methods that offers a tangible way to encourage improved collaboration outcomes.

Part II – Tools and Techniques

Software engineering requires a number of complex, interleaved activities to be carried out. These must be organized into logically correct teamwork and be supported by appropriate tools. Because of the challenges of supporting collaborating in an already complex engineering process, a multitude of techniques and tools have been developed to support almost all activities of collaborative software engineering.

Traditionally software engineering had been a co-located activity where team members could expect some degree of face-to-face communication and collaboration and co-ordination were important activities but discrete and compartmentalized. Outsourcing parts of a software engineering project and highly iterative agile processes have led to an increased interest in how to best support virtual, distributed collaboration and communication and co-ordination for team activities that repeat in days rather than months.

A range of support mechanisms and associated tool support have appeared in recent years to address concerns in both traditional but more particularly these newer domains of collaborative software engineering. Social networking-style support such as tagging, shared knowledge repositories and communication support have become popular. New search-based support and associated visualization support have become more important as developers are less familiar with large tracts of software systems. These include mining of software repositories and context-aware filtering mechanisms in IDEs. Event-based support mechanisms have always been popular in collaborative support environments. These have been explored further in the context of both same-place and distance-located teams to support proactive notification and various levels of group awareness.

Developer-centric software engineering tools are crucial and this includes support for collaboration. Areas of particular interest in these tools are knowledge management and expertise communication. Knowledge management requires use of shared ontologies and supporting authoring tools, but as importantly the development of true “virual communities” where informal knowledge sharing is supported and encouraged. Expertise communication is one aspect where the collaboration environment allows increasingly geographically dispersed team members to better communicate both knowledge and expertise relating to knowledge and tasks.

Part III – Organizational Experiences

Multi-site, or geographically distributed software development, has introduced a range of unknowns into software engineering practice and research. Of particular note is the lack of guidance around process selection. When running a multi-site, geographically distributed software project, what is the "best" software process to choose to organize this activity, quite apart from tool, project management and team selection issues? How can organizations make process choices, in particular, to best exploit multiple time zones, team expertise, out-sourced and open-sourced parts of a product, and ensure quality, cost and timeliness thresholds? Two fundamental ways of organizing a distributed project are centralized control of overall process and distribution of scoped design/code/test, compared with distributing different phases e.g. requirements team, design and build team, testing team in different locations.

Open source software projects are an increasingly common model of distributed, virtual software teams. Many studies have looked at collaboration aspects of such projects, in particular the evolution of the code base and team communication and co-ordination patterns. Recovering such information is challenging–often via bug reports, detailed code analysis and informal interviews of key team members. It is still an unsolved research problem how to best set up an open source project to achieve high quality communication and co-ordination.

Software artifact analysis has been used extensively for many years. This includes static analysis of source code, tests, designs and requirements and dynamic analysis of execution traces, side-effects and formal models of code. Collaboration around analysis has often been informal and poorly structured. Given the increasing complexity of code and analysis tools and techniques, an open challenge is how to share analysis processes and techniques, and also the tools supporting these, particularly across organizations.

Part IV – Related Issues

A number of socio-technical issues arise in collaborative software engineering. In free and open source software development projects these are particularly challenging. Key issues include overall project ownership and co-ordination, task de-composition, trust, accountability, commitment and social networking. Collaboration affordances in the individual and group development ecosystem must support both the range of collaboration activities but take into account the free and open source domain of work.

Knowledge sharing is crucial in all domains of software engineering. Particular domains of interest include requirements engineering and software architecture where commissioner, engineer, manager and end user constraints intersect and often must be balanced. Knowledge sharing in collaborative software architecting supports better decision making, surfacing of assumptions, and reasoning about design decisions. In product line engineering, variability management is a key challenge, particularly when faced with multi-site software teams. Rationale management can be used to augment the variability management process to improve collaboration support in this context.

Configuration management has long been a challenge in software engineering particularly as systems have grown enormously in size and complexity. As configuration management requires integrating many software artifacts and ultimately impacts all phases of proceeding development, configuration management support systems have been an early contributor to collaborative software engineering infrastructure. They provide a shared space, awareness support, record and enable tracing of team actions, and support both knowledge sharing and communication. Many outstanding research and practice issues exist in each of these areas of configuration management systems support, however, leading to next generation collaborative software engineering tools.

Current Challenges and Future Directions

Collaborative software engineering has been a very heavily researched area and almost all practicing software teams will need to engage in it. However, many challenges still present both in terms of adopting collaboration practices, processes and tools and improving the state-of-the-art. Many of these challenges are long standing, and hence are fundamental to the act of working together to engineer shared artifacts. These include assembling teams, dividing work, social networking within and between teams, choosing best-practice processes, techniques and supporting tools, and effective project management. Others have arisen due to new organizational practices and technical advances, including open-sourced, out-sourced, multi-site and agile software engineering contexts. We still do not know the ideal way to share knowledge, facilitate the most effective communication, co-ordinate massively distributed work, and design and deploy support tools for these activities.

Auckland, New Zealand — John Grundy
Heidelberg, Germany — Ivan Mistrík
Irvine, CA, USA — André van der Hoek
Santa Cruz, CA, USA — Jim Whitehead

Acknowledgements

The editors would like to sincerely thank the many authors who contributed their works to this collection. The international team of anonymous reviewers gave detailed feedback on early versions of chapters and helped us to improve both the presentation and accessibility of the work. Finally we would like to thank the Springer management and editorial teams for the opportunity to produce this unique collection of articles covering the very wide range of areas related to collaborative software engineering.

Contents

Contributors

Ban Al-Ani Department of Informatics, University of California, Irvine, CA 92697-3440, USA, balani@ics.uci.edu

Paris Avgeriou Department of Mathematics and Computing Science, University of Groningen, 9747 AG Groningen, Netherlands, paris@cs.rug.nl

Alberto Avritzer Siemens Corporate Research, Inc., Princeton, NJ, 8540, USA, alberto.avritzer@siemens.com

Stefanie Betz Institute of Applied Informatics and Formal Description Methods, Universität of Karlsruhe, 76128 Karlsruhe, Germany, stefanie.betz@aifb.uni-karlsruhe.de

Jan Bosch Intuit Inc., Mountain View, CA 94043, USA, jan@janbosch.com

Petra M. Bosch-Sijtsema Helsinki University of Technology and Stanford University, Stanford, CA 94305, USA, Pbosch@stanford.edu; petra@petrabosch.com

John Burton Vitalograph Limited, Ennis, Co Clare, Ireland, john.burton@vitalograph.ie

Valentine Casey Software Systems Research Centre, Bournemouth University, Poole, Dorset, UK, vcasey@bournemouth.ac.uk

Viktor Clerc VU University Amsterdam, 1081 HV Amsterdam, Netherlands, viktor@cs.vu.nl

Isabella A. da Silva Department of Informatics, University of California, Irvine, CA 92697-3440, USA, bellinha@gmail.com

Daniela Damian Software Engineering Global interAction Lab, University, of Victoria, Victoria, BC, Canada, danielad@cs.uvic.ca

Remco C. de Boer VU University Amsterdam, 1081 HV Amsterdam, Netherlands, remco@cs.vu.nl

Prasun Dewan Department of Computer Science CB 3175, University of North Carolina, Chapel Hill, NC 27599-3175, USA, dewan@cs.unc.edu

Jacky Estublier Université de Grenoble (LIG/CNRS), 38400 Saint Martin, France, Jacky.Estublier@imag.fr

Rik Farenhorst VU University Amsterdam, 1081 HV Amsterdam, Netherlands, rik@cs.vu.nl

Harald C. Gall Software Evolution and Architecture Lab, Department of Informatics, University of Zurich, 8050 Zürich, Switzerland, gall@ifi.uzh.ch

Giacomo Ghezzi Software Evolution and Architecture Lab, Department of Informatics, University of Zurich, 8050 Zürich, Switzerland, ghezzi@ifi.uzh.ch

John Grundy Electrical & Computer Engineering, University of Auckland, Auckland 1142, New Zealand, j.grundy@auckland.ac.nz

Hans-Jörg Happel FZI Research Center for Information Technology, 76131 Karlsruhe, Germany, happel@fzi.de

Anton Jansen Department of Mathematics and Computing Science University of Groningen, 9747 AG Groningen, Netherlands, a.g.j.jansen@cs.rug.nl

Irwin Kwan Software Engineering Global interAction Lab, University of Victoria, Victoria, BC, Canada, irwink@cs.uvic.ca

Patricia Lago VU University Amsterdam, 1081 HV Amsterdam, Netherlands, patricia@cs.vu.nl

Peng Liang Departmentt of Mathematics and Computing Science, University of Groningen, 9747 AG Groningen, Netherlands, liangp@cs.rug.nl

Walid Maalej Technische Universität München, 85748 Garching, Germany, maalejw@in.tum.de

Juho Mäkiö Forschungszentrum Informatik FZI, 76131 Karlsruhe, Germany, juho.maekioe@gmx.de

Sabrina Marczak Software Engineering Global interAction Lab, University of Victoria, Victoria, BC, Canada, smarczak@cs.uvic.ca

Fergal McCaffery Stokes Lecturer, Dundalk Institute of Technology, Dundalk, Co Louth, Ireland, fergal.mccaffery@dkit.ie

Ivan Mistrík Independent Consultant, 69120 Heidelberg, Germany, i.j.mistrik@t-online.de

Leonardo Gresta P. Murta Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ 24210-240, Brazil, leomurta@ic.uff.br

Kumiyo Nakakoji Research Centre for Advanced Science and Technology, University of Tokyo, Japan; SRA Key Technology Laboratory Inc., Japan, kumiyo@kid.rcast.u-tokyo.ac.jp

Bashar Nuseibeh Department of Computing, The Open University, Milton Keynes MK7 6AA, UK, B.Nuseibeh@open.ac.uk

Andreas Oberweis Institute of Applied Informatics and Formal, Description Methods, Universität Karlsruhe, 76128 Karlsruhe, Germany, andreas.oberweis@aifb.uni-karlsruhe.de

Daniel J. Paulish Siemens Corporate Research, Inc., Princeton, NJ 08540, USA, daniel.paulish@siemens.com

Martin Pinzger Software Engineering Research Group, Delft University of Technology, Netherlands, M.Pinzger@tudelft.nl

David Redmiles Department of Informatics, University of California, Irvine, CA 92697-3440, USA, redmiles@ics.uci.edu

Ita Richardson Department of Computer Science & Information Systems, Lero – the Irish Software Engineering, Research Centre, University of Limerick, Ireland, ita.richardson@ul.ie

Hugh Robinson Centre for Research in Computing, The Open University, Milton Keynes, MK7 6AA, UK, h.m.robinson@open.ac.uk

Anita Sarma Department of Computer Science & Engineering, University of Nebraska, Lincoln, NE 68588-0115, USA, asarma@cse.unl.edu

Walt Scacchi Donald Bren School of Information and Computer Sciences, Institute for Software Research, University of California, Irvine, CA 92697-3455, USA, Wscacchi@ics.uci.edu

Stefan Seedorf University of Mannheim, 68131 Mannheim, Germany, seedorf@uni-mannheim.de

Helen Sharp Centre for Research in Computing, The Open University, Milton Keynes, MK7 6AA, UK, H.C.Sharp@open.ac.uk

Roberto S. Silva Filho Department of Informatics, University of California, Irvine, CA 92697-3440, USA, rsilvafi@ics.uci.edu

Anil K. Thurimella Harman/Becker Automotive Systems, 76307 Karlsbad, Germany, anil_98ee601@yahoo.com

Erik Trainer Department of Informatics, University of California, Irvine, CA 92697-3440, USA, etrainer@ics.uci.edu

André van der Hoek Department of Informatics, University of California, Irvine, CA 92697-3440, USA, andre@ics.uci.edu

Hans van Vliet VU University Amsterdam, 1081 HV Amsterdam, Netherlands, hans@cs.vu.nl

Claudia Maria L. Werner Programa de Engenharia, de Sistemas e Computação, COPPE – Universidade Federal do Rio de Janeiro, Rio de Janeiro 21941-972, Brazil, werner@cos.ufrj.br

Jim Whitehead Department of Computer Science, Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA, ejw@soe.ucsc.edu

Yasuhiro Yamamoto Research Centre for Advanced Science and Technology, University of Tokyo, Tokyo, Japan, yxy@kid.rcast.u-tokyo.ac.jp

Yunwen Ye SRA Key Technology Laboratory, Inc., Shinjuku, Tokyo 160-0004, Japan, ye@sra.co.jp

Chapter 1
Collaborative Software Engineering: Concepts and Techniques

Jim Whitehead, Ivan Mistrík, John Grundy, and André van der Hoek

Abstract Collaboration is a central activity in software engineering, as all but the most trivial projects involve multiple engineers working together. Hence, understanding software engineering collaboration is important for both engineers and researchers. This chapter presents a framework for understanding software engineering collaboration, focused on three key insights: (1) software engineering collaboration is model-based, centered on the creation and negotiation of shared meaning within the project artifacts that contain the models that describe the final working system; (2) software project management is a cross-cutting concern that creates the organizational structures under which collaboration is fostered (or dampened); and (3) global software engineering introduces many forms of distance – spatial, temporal, socio-cultural – into existing pathways of collaboration. Analysis of future trends highlight several ways engineers will be able to improve project collaboration, specifically, software development environments will shift to being totally Web-based, thereby opening the potential for social network site integration, greater participation by end-users in project development, and greater ease in global software engineering. Just as collaboration is inherent in software engineering, so are the fundamental tensions inherent in fostering collaboration; the chapter ends with these.

1.1 Introduction

Software projects are inherently co-operative, requiring many software engineers to co-ordinate their efforts to produce a large software system. Integral to this effort is developing shared understanding surrounding multiple artifacts, each artifact embodying its own model, over the entire development process [97].

J. Whitehead (✉)
Department of Computer Science, Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA
e-mail: ejw@soe.ucsc.edu

I. Mistrík et al. (eds.), *Collaborative Software Engineering*,
DOI 10.1007/978-3-642-10294-3_1,

Software engineers have adopted a wide range of communication and collaboration technologies to assist in the co-ordination of project work. Every mainstream communication technology has been adopted by software engineers for project use, including telephone, teleconferences, email, voice mail, discussion lists, the Web, instant messaging, voice over IP, and videoconferences. These communication paths are useful at every stage in a project's lifecycle, and support a wide range of unstructured natural language communication. Additionally, software engineers hold meetings in conference rooms, and conduct informal conversations in hallways, doorways, and offices. While these discussions concern the development of a formal system, a piece of software, the conversations themselves are not formally structured (exceptions being automated email messages generated by SCM systems and bug tracking systems).

In contrast to the unstructured nature of conversation, much collaboration in software engineering is relative to various formal and semi-formal artifacts. Software engineers collaborate on requirements specifications, architecture diagrams, UML diagrams, source code, and bug reports. Each is a different model of the ongoing project. Software engineering collaboration can thus be understood as artifact-based or model-based collaboration, where the focus of activity is on the production of new models, the creation of shared meaning around the models, and elimination of error and ambiguity within the models.

This model orientation to software engineering collaboration is important due to its structuring effect. The models provide a shared meaning that engineers use when co-ordinating their work, as when engineers working together consult a requirements specification to determine how to design a portion of the system. Engineers also use the models to create new shared meaning, as when engineers discuss a UML diagram, and thereby better understand its meaning and implications for ongoing work. The models also surface ambiguity by making it possible for one engineer to clearly describe their understanding of the system; when this is confusing or unclear to others, ambiguity is present. Without the structure and semantics provided by models, it would be more difficult to recognize differences in understanding among collaborators.

These twin threads – the appropriation of novel communications technologies for project work, and the model-centric nature of collaboration – are what give the study of software engineering collaboration its unique character. Focusing just on communication, the low cost and global reach of email, web, and instant messaging technologies created the potential for global, multi-site software engineering teams. This made it less expensive to globally distribute closed source projects, and created the technological conditions that supported the emergence of open sourceopen source software. In turn, understanding how best to structure and support this communication-afforded collaboration within distributed software engineering has been the focus of sustained study. Much traditional collaborative work research has focused on the use of novel communication technologies in a variety of work settings, viewing them as artifact-neutral co-ordination technologies. What distinguishes the study of collaboration within software engineering from this more general study of collaboration is its focus on model creation. Software engineers are not just collaborating in the abstract – they are collaborating over the creation of a series of artifacts that, together, provide a multi-faceted view of the behavior of a complex system.

1.2 Defining Collaborative Software Engineering

Collaboration is pervasive throughout software engineering. Almost all non-trivial software projects require the effort and talent of multiple people to bring it to conclusion. Once there are two or more people on a software project, they must work together, that is, they must collaborate. Thus, a simple ground truth is that *any software project with more than one person is created through a process of collaborative software engineering*.

There is an old story, running through many cultures, about six blind men and an elephant. One man touches the elephant's trunk, and says the elephant is a rope. Another touches a leg, and says the elephant is a tree trunk. The remaining four describe the elephant as a snake (tail), spear (tusk), wall (body), or brush (end of tail). A large software system is like the elephant in the story, with each software engineer having their own view and understanding of the overall system. Unlike the story, a software system under development lacks the physical fixedness of the elephant; one cannot simply step back and see the shape of the entire software system. Instead, a software system is shaped by the intersecting activities and perspectives of the engineers working on it. Software is thought-stuff, the highly malleable conversion of abstractions, algorithms, and ideas into tangible running code. Hence software engineers shape the system under construction while developing their understanding of it.

Human minds are enormously flexible, approaching problems from unique experiential, cultural, educational, and biochemical conditions; developers have widely varying backgrounds and experiences, come from different cultures, have different types of educational backgrounds, and have varying body chemistry. Somehow, through the imperfect instrument of language, the vast pool of variable outcomes inherent in any software system needs to be reduced to a single coherent system. In this view, *software engineering collaboration is the mediation of the multiple conflicting mental conceptions of the system held by human developers*.

Collaboration takes the form of tools to structure communication and lead to consensus, as in the case of requirements elicitation tools. Other tools mediate conflicts among differing views of the system, as in the case of configuration management tools both preventing conflicting viewpoints from being realized as incompatible code changes, and providing a process for handling conflicts when they occur (merge tools). Tools for representing design and architecture diagrams also help to mediate conflicts by making internal mental models explicit, thereby allowing other actors to identify points of departure from their own views of the system.

Since software is so abstract and malleable, and is created via a process of negotiating multiple viewpoints on the system, it is inevitable that software will have errors. *Consequently, software engineering collaboration also involves the joint identification and removal of error*. This can be seen in software inspections, where multiple engineers bring their unique perspectives to the task of finding latent errors. It is also visible in test teams, where many engineers work together to write system test suites, and use bug tracking software to co-ordinate bug fixing effort.

People have a hard time working together effectively. To work well together, engineers need to understand near-term and long-term goals, be clustered into teams, and understand their personal responsibilities. Engineers also need to be motivated, and receive appropriate reward for their work. Hence, *software engineering collaboration is about creating the organizational structures, reward structures, and work breakdown structures that afford effective work towards goal*. As a consequence, software engineering management and leadership is an integral part of software engineering collaboration.

1.3 Historical Trends in Collaborative Software Engineering

Software engineers have developed a wide range of model-oriented technologies to support collaborative work on their projects. These technologies span the entire lifecycle, including collaborative requirements tools [5, 39], collaborative UML diagram creation, software configuration management systems and bug tracking systems [11]. Process modeling and enactment systems have been created to help manage the entire lifecycle, supporting managers and developers in assignment of work, monitoring current progress, and improving processes [7, 57]. In the commercial sphere, there are many examples of project management software, including Microsoft Project [69] and Rational Method Composer [42]. Several efforts have created standard interfaces or repositories for software project artifacts, including WebDAV/DeltaV [24, 98] and PCTE [96]. Web-based integrated development environments serve to integrate a range of model-based (SCM, bug tracking systems) and unstructured (discussion list, web pages) collaboration technologies.

Tool support developed specifically to support collaboration in software engineering falls into four broad categories. *Model-based* collaboration tools allow engineers to collaborate in the context of a specific representation of the software, such as a UML diagram. *Process support* tools represent all or part of a software development process. Systems using explicit process representations permit software process modelling and enactment. In contrast, tools using an implicit representation of software process embed a specific tool-centric work process, such as the checkout, edit, checking process of most SCM tools. *Awareness tools* do not support a specific task, and instead aim to inform developers about the ongoing work of others, in part to avoid conflicts. *Collaboration infrastructure* has been developed to improve interoperability among collaboration tools, and focuses primarily on their data and control integration. Below, we give a brief overview of previous work in these areas, to provide context for our recommendations for future areas of research on software collaboration technologies.

1.3.1 Model-Based Collaboration Tools

Software engineering involves the creation of multiple artifacts. These artifacts include the end product, code, but also incorporate requirements specifications, architecture description, design models, testing plans, and so on. Each type of artifact has its own semantics, ranging from free form natural language, to the semi-formal semantics of UML, or the formal semantics of a programming language. Hence, the creation of these artifacts is the creation of models.

Creating each of these artifacts is an inherently collaborative activity. Multiple software engineers contribute to each of these artifacts, working to understand what each other has done, eliminate errors, and add their contributions. Especially with requirements and testing, engineers work with customers to ensure the artifacts accurately reflect their needs. Hence, the collaborative work to create software artifacts is the collaborative work to create models of the software system. Systems designed to support the collaborative creation and editing of specific artifacts are really supporting the creation of specific models, and hence support model-based collaboration. Collaboration tools exist to support the creation of every kind of model found in typical software engineering practice.

Figure 1.1 provides an overview of model-oriented collaboration across a software project lifecycle. In the figure, rows represent different types of actors or models, while columns represent different phases in the development of a software system. Overlaps between bubbles for types of people represent collaboration. So, for example, the overlap of stakeholders and requirements engineers in the requirements column represents their collaboration to create the requirements documentation for the system to be built. Project management cuts across all project phases and impacts all types of software engineer, hence it is represented as a horizontal bar. Remote collaboration occurs when the set of people within a bubble is distributed across multiple sites, or when each bubble in a collaboration is at a different site.

Overlap between model type bubbles indicates dependencies between the models. For example, determining a system's software architecture often requires negotiation with the customer over the implications of requirements, and may require an understanding of the fine-grained design of certain system functions. For simplicity, the figure is drawn using a waterfall-type process model. Other process models modify this picture. Spiral development would involve additional negotiation around the importance of various types of risk, and what constitutes acceptable levels of risk. An evolutionary prototyping model would add collaboration between stakeholders and developers in the coding phase, representing the negotiation that takes place after a demonstration of the evolving system prototype to the customer.

In the sections below, we provide an overview of the collaboration that takes place during each project phase, and active areas of research within these phases.

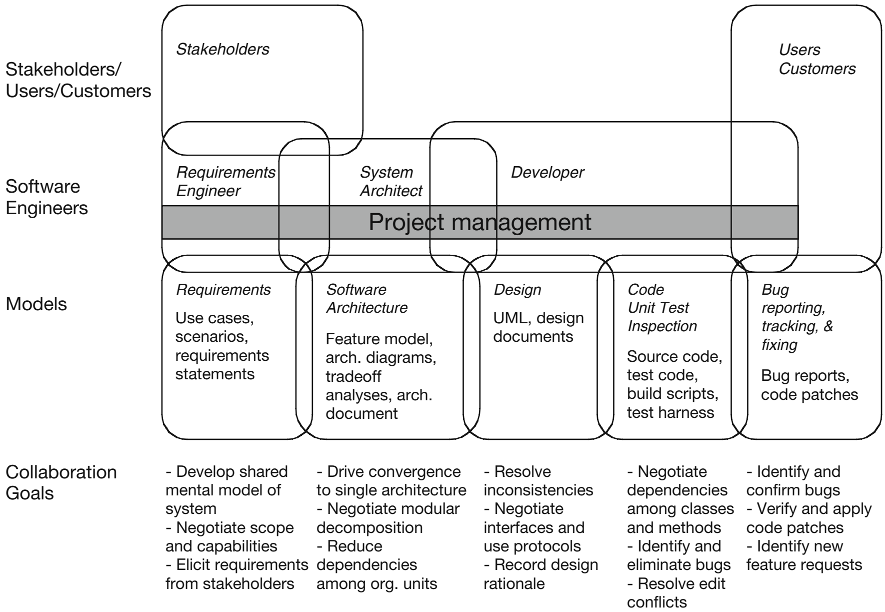

Fig. 1.1 Overview of model driven collaboration

1.3.1.1 Requirement Centered Collaboration

In the requirements phase, there are many existing commercial tools that support collaborative development of requirements, including Rational's RequisitePro [43] and DOORS [41] products, and Borland's CaliberRM [8] (a more exhaustive list can be found at [60]). These tools allow multiple engineers to describe project use cases and requirements using natural language text, record dependencies among and between requirements and use cases, and perform change impact analyses. Integration with design and testing tools permits dependencies between requirements, UML models, and test cases to be explicitly represented.

Collaboration features vary across tools. Within RequisitePro, requirements are stored in a per-project requirements database, and can be edited via a Web-based interface by editing a Word document that interacts with the database via a plugin, or by direct entry using the RequisitePro user interface. Multiple engineers can edit the requirements simultaneously via these interfaces. While cross-organization interaction is possible via the Web-based interface, the tool is primarily designed for within-organization use. RAVEN [79] supports collaboration via a built-in checkout/checkin process on individual requirements. While most requirements tools are desktop applications, Gatherspace [29] and eRequirements [29] are web-based collaborative requirements tools, with capabilities only accessible via a Web browser.

Research on collaborative requirements tools has focused on supporting negotiation among stakeholders, use of new requirements engineering processes, and