Chapter 19
Collaborative Software Engineering: Challenges and Prospects

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

Abstract Much work is presently ongoing in collaborative software engineering research. This work is beginning to make serious inroads into our ability to more effectively practice collaborative software engineering, with best practices, processes, tools, metrics, and other techniques becoming available for day-to-day use. However, we have not yet reached the point where the practice of collaborative software engineering is routine, without surprises, and generally as optimal as possible. This chapter summarizes the main findings of this book, draws some conclusions on these findings and looks at the prospects for software engineers in dealing with the challenges of collaborative software development. The chapter ends with prospects for collaborative software engineering.

19.1 Introduction

19.1.1 What We Know About Collaborative Software Engineering

Software engineering is naturally a team activity. Software engineers need to collaborate effectively in order to deliver a project on time, on budget and to an appropriate quality level [2]. Traditional software engineering projects have used primarily top-down approaches to team organization and project management, a homogeneous software process and toolset, are co-located enabling regular and proactive face-to-face meetings, and team members usually have the same language and work culture [2, 3, 8].

These projects still face daunting challenges around collaboration. Teams have to be formed and work appropriately delegated, tracked and managed. Specialists within teams or whole specialist teams need to exchange knowledge among themselves and across team boundaries. Evolving requirements and customer needs

I. Mistrík (✉)
Independent Consultant, 69120 Heidelberg, Germany
e-mail: i.j.mistrik@t-online.de

I. Mistrík et al. (eds.), *Collaborative Software Engineering*,
DOI 10.1007/978-3-642-10294-3_19,

require processes and collaboration support to enable these to be effectively managed [5, 26]. Traditional software tools usually provided limited collaboration support features. A toolset needs mechanisms to support collaboration (e.g., shared workspaces, file repositories, differencing and merging support, configuration, testing, design, process management), communication (email, messages, annotations, video/audio), and co-ordination (locking, versioning, hand-over, auditing) [20, 6]. Many studies of teams, processes, tools and real-world projects [5, 19, 27] have shown the value of appropriate process, project management, technique and tool selection and usage to enable effective and efficient collaboration.

Several recent trends in software engineering have greatly increased the challenges around collaboration on software projects. Agile processes enabling rapid requirements evolution and emergent architectures and documentation demand vastly different team organization, project management and communication strategies [19, 18, 8, 2]. Virtual software organizations with distributed teams, contractual obligations between constituent organizations, and highly distributed teams demand greater support for knowledge sharing, co-ordination and collaboration [13]. Communication may be complicated by time zone, culture and even language differences. Open source software projects exhibit similar challenges but are often characterized by a very wide range of participants, organizations and contributions from teams and individuals with very different motivations and needs. A trend to "global software development" similarly leads us to teams that span country, language, culture, organization, technical tool platform and ultimately software process [18].

In order to address many of these known issues – and discover new issues – in collaborative software engineering, a large number of research and practice projects are taking place. New software processes are being studied to gauge their impact on distributed software projects including open source, global software development and outsourcing projects [2, 8, 17, 24]. These aim to help organizations better understand such contexts for collaboration and formulate the most effective and complementary teams, processes, toolsets and techniques. Communication patterns in open source projects, requirements engineering, coding and testing projects, and projects in agile process, open source and virtual software development organizations, are being analyzed to enhance understanding of needs in these domains [24, 5, 2]. Many face challenges of distance including language, culture and work practice, as well as traditional communication and knowledge management issues. A wide range of new tool support approaches are being developed, deployed and evaluated in various domains. These include but are not limited to improved awareness support, software analysis, configuration management, co-ordination, communication and knowledge management [6, 9, 11, 19].

Sharing knowledge about software engineering projects continues to be a major challenge and best practices for knowledge management in many areas are still unclear [9, 11, 26]. Studies have shown benefits to collaboration of improved knowledge repositories and management practices in requirements, architecture, project management, and software process domains. Communicating rationale about decisions is critical at all levels of software engineering [12, 15, 16].

19.1.2 Objectives of This Chapter

This book makes a case for CoSE as a crucial part of research in software engineering (SE) and as an essential part of future software development and maintenance. In previous chapters, the book has explained what CoSE is, what its potential value is for SE, what its research challenges are and how these challenges might be met. The intention of this concluding chapter is to provide a summary of the previous chapters and a look at prospects meeting the challenges of future CoSE practice.

Section 19.1 summarizes a current status of CoSE. Section 19.2 presents a summary of the book. Section 19.3 reviews some of the present challenges facing collaborative software development and prospects for meeting them.

19.2 Summary of the Book

Software engineering collaboration has multiple goals and means spanning the entire lifecycle of development [27]. Chapters in this book are reporting on advances in achieving some of these goals by presenting their particular means and specific solutions.

Chapter 1 of the book introduces the concepts and tools for CoSE. Part I contains chapters that characterize CoSE. Part II contains chapters that examine various techniques and tool support issues in CoSE. Part III contains chapters addressing organizational issues in CoSE. Part IV contains chapters looking at a variety of related issues in CoSE. Finally, Chapter 19 concludes the book with a summary of the book, current challenges and prospects in CoSE.

As many organizations have discovered to their cost, implementing a global software engineering strategy is a complex and difficult task. Extensive research in this area has identified that this is due to a number of factors which include the nature and impact of geographical, temporal, cultural and linguistic distance. In addition, whether undertaken in a collocated or geographically distributed environment, team based software development is not simply a technical activity. It also has important human, social and cultural implications which need to be specifically addressed. While the technical aspects of software development cannot be underestimated, neither can the importance of establishing and facilitating the effective operation of these teams [18].

Requirements engineering (RE) is an area filled with challenges of a non-technical nature. RE involves activities such as negotiation, analysis and requirements management in subsequent phases of development. RE requires communication from the elicitation phase down to the analysis, implementation and test phases. As such, it involves collaboration among large, often geographically distributed cross-functional teams comprised of requirements analysts, software architects, developers, and testers. This collaboration is driven by coordination needs in software development and relies on communication and awareness. Coordination is a critical aspect in every activity related to a requirement's analysis, implementation or testing. Effective coordination, knowledge management and information sharing

among team members with diverse organizational and functional backgrounds is crucial. Collaboration across geographical distance (i.e., different time zones) and socio-cultural distance (i.e., language and culture) creates additional challenges in project members' communication and awareness in the development project [5].

Collaboration can be viewed as the most important lever for achieving high quality, efficient and effective software engineering practices and results in virtually any software developing organization. Although collaboration has been complicated, several trends increase the complexity of managing dependencies between software development teams and organizations. These trends include the increasing adoption of software product lines, the globalization of software engineering and the increasing use of and reliance on 3rd party developers in the context of software ecosystems. The trends share as a common characteristic that the coupling between the software assets as well as between the organizational units is increased. Consequently, decoupling mechanisms need to be introduced to address the increase in coupling [3].

Agile software development is a group of software engineering methodologies, e.g., eXtreme programming (XP), Scrum, Crystal, that became popular in the early 2000s. Agile advocates claim to increase overall software developer productivity, deliver working software on time, and minimise the risk of failure in software projects. While its effectiveness and applicability remain uncertain, it is attracting increasing interest from the software engineering community. The Agile Manifesto emphasises collaboration and interactions, and the reality of XP software development offers evidence that this emphasis is borne out in practice. Observing practice makes it clear to the researcher that the work of an XP team visibly and continually involves collaboration and communication – and that collaboration and communication are part of the technical business of creating working software. There are two key XP practices which illustrate the relationship between the social and technical: pairing and customer collaboration [19].

Ontology captures a shared understanding of a problem domain and is usually specified in a logical language by describing concepts, relationships and additional logical axioms. Knowledge included in ontology is designed for both humans and machines. It can be integrated in development infrastructures and in developed software to support various software project activities. Although ontologies have been around for many years, several factors promote their increasing adoption. First, with a number of W3C standards such as RDF and OWL issued in recent times, tools and methodologies for creating and managing ontologies have matured. Second, the success of the Web enables developers to collaborate in a richer and more dynamic way, instead of working in de facto isolation. Both factors contribute to a slow but growing number of semantic approaches addressing CSD issues. Applications of ontologies in software development can be manifold and so the resulting ontologies will differ in expressivity, scope and purpose [9].

A variety of novel tools have been created to allow software developers to collaborate with each other. There are many approaches how to classify them. One approach classifies them on whether they try to (a) make software developers feel they are co-located, or (b) provide features not found in co-located collaboration.

The result is an overview that relates concepts not linked together earlier, which include not only research tools but also studies that motivate/evaluate them. Each of the surveyed works is described by showing how it builds on or overcomes problems of other research addressed in this chapter. By focusing only on the differences among these works, the chapter covers a large variety of concepts, from over fifty papers. It is targeted mainly at the practitioner familiar with the state of the art, rather than the researcher working on improving current practices. Nonetheless, the interrelationships among the referenced works should be of interest to everyone. In particular, a new researcher in this area should be able to find holes in existing designs and evaluations [6].

In software development the need for coordination among developers generally arises because of the underlying technical dependencies among work artifacts; as well as the structure of the development process. Researchers in the software engineering as well as computer-supported cooperative work communities have recognized this problem and created a host of tools to improve team coordination. However, evaluating the usability and usefulness of such tools has proven to be extremely difficult. One possibility is to focus on different evaluation approaches that are applicable for coordination tools. There exists a diverse range of approaches to evaluating collaborative tools. Adopting a combination of empirical evaluation approaches is perceived as means to meet the challenges typically encountered. The diversity of existing tools and evaluation approaches reflect the many challenges of facilitating coordination in teams. Further, several evaluation frameworks have been proposed to support software tool evaluation [20].

Configuration Management is a discipline responsible for controlling the evolution of products. Since late 1960s, configuration management is considered to be one of the core supporting process to software development and a research field of software engineering. According to IEEE, there are five main functions of configuration management: configuration identification, configuration control, configuration status accounting, configuration evaluations and reviews, and release management and delivery. However, these five functions are traditionally supported by three main subsystems: issue tracking system, version control system, and build management system. Because the primary focus of configuration management is keeping the consistency of products, it is concerned with how people interact to develop and maintain these products. The complexity of software products led to the need of geographically distributed teams composed of a large number of developers with different background. These teams collaborate during software engineering activities, and configuration management can be considered as an enabling technology to allow this collaboration. Collaboration in the context of software engineering encloses different aspects, such as: implicit and explicit communication among developers, awareness regarding other developers' actions, coordination of development tasks to avoid rework and to achieve the project goals, keeping a shared memory with previous development actions history, and providing a shared space where the work made by a developer is available to other developers [15].

The advantages of using explicit software architecture include early interaction with stakeholders, its basis for establishing work breakdown structure and early

assessment of quality attributes. Although considerable progress has been made, we still lack techniques for capturing, representing, and maintaining knowledge about software architectures. While much attention has been given to documenting architectural solutions, the rationale for these solutions often remains implicit and is often exchanged in interpersonal, informal communication. The incomplete representation of the needed architectural knowledge leads to several problems that are generally recognized in any software engineering project, and that become just worse in distributed and global software development. When software engineering projects are distributed or global, the problems above are aggravated. Knowledge transfer is a communication process requiring strict interaction and agile information exchange. In local software development, it is already difficult to rationalize the type and amount of knowledge we need to exchange. If in addition exchanges occur remotely and via a technological infrastructure, we have to make this knowledge explicit, and we need to identify agile means to render this process as dynamic and powerful as possible [11].

Software development is in essence information-intensive collaborative knowledge activity. It is about using information, generating information, and making information artifacts. The wide acceptance of agile processes and the success of many open source projects provide strong evidence that human aspects do matter in software development; cognitive and social processes play essential roles in successful software projects in which individuals' creative thinking in using and generating information are nurtured. There is an argument that software engineering environments must be designed to foster such individuals' creative knowledge processes, and that collaboration must be supported in the context of individuals' development activities. Collaborative software development environments should be designed to facilitate and nurture individuals' creative knowledge processes. Collaboration takes place with or without explicit communication. On the one hand, software developers regularly engage in collaboration through artifacts without explicit communication (e.g., by writing comments in code to be read by others). On the other hand, explicit communication becomes necessary when developers must ask their peers for information that is otherwise not obtainable. Existing studies have provided ample evidence that both collocated and distributed software development teams frequently engage in communication to acquire necessary information from peer developers [16].

A common feature of many software analysis tools is that they focus on just a particular kind of analysis to produce the results wanted. If different analyses are required, an engineer needs to run several tools, each one specialized on a particular aspect, ranging from pure source code analysis, duplication analysis, co-change analysis, bug prediction, to bug fixing patterns and visualization. All these techniques have their own explicit or implicit meta-model which dictates how to represent the input and the output data. Thus the sharing of information between tools is only possible by means of a cumbersome export towards files complying with a specified exchange format. Also, if there are several analyses of the same kind (*e.g.,* code duplication analysis) there is hardly any way to compare the results or integrate them other than manual investigation. Tool interoperability is hampered

even more by their stand-alone nature as well as their platform and language dependence. As a consequence, distributed and collaborative software analysis scenarios are severely limited. The combination and integration of different software analysis tools is a challenging problem when we need to gain a deeper insight into a software system's evolution. For every required analysis a specialized tool, with its own explicit or implicit meta-model dictating how to represent the input and output, has to be installed, configured and executed. Even if different analyses of the same kind exist, the only way to compare them is to do it manually [8].

Communication and collaboration among team members are key success factors for large, complex software projects. In addition to industry, examples of such projects can be found in the Open Source Software (OSS) community, for example, the Mozilla, Apache, Eclipse projects. OSS projects are of particular interest for communication and collaboration research because their developers rarely or never meet face-to-face. Findings of previous research showed that OSS developers coordinate their work almost exclusively by three information spaces: the implementation space, the documentation space, and the discussion space. Typically, in OSS projects a versioning system, such as the concurrent versions system, provides the backend of the implementation space. It keeps track of changes made to projected related files and corresponding versions. The World Wide Web is used as the primary documentation space. Because of the distributed and informal nature of OSS projects, discussions between project members, project associates, and users are done and tracked in mailing lists and bug reporting systems. This results in a representative data set that enables communication and collaboration analysis [17].

For the past few years, Siemens has been experimenting with software development processes and practices for globally distributed projects using student-based development teams located at different universities around the world. The students who make up the Global Studio Project (GSP) simulate an industrial software development project using common practices for collaboration among distributed sites. Experiences with this project have been reported in a number of papers, and it has been documented as a case study (GSP 2005). The motivation for studying multisite software development processes is driven by the business needs. A number of questions were raised, and they are still being investigated [2].

Free/open source software development (FOSSD) is a way for building, deploying, and sustaining large software systems on a global basis, and differs in many interesting ways from the principles and practices traditionally advocated for software engineering. Hundreds of FOSS systems are now in use by thousands to millions of end-users, and some of these FOSS systems entail hundreds-of-thousands to millions of lines of source code. So what's going on here, and how are collaborative FOSSD processes used to build and sustain these projects, and how might differences with SE be employed to explain what's going on with FOSSD? One of the more significant features of FOSSD is the formation and enactment of collaborative software development practices and processes performed by loosely coordinated software developers and contributors. These people may volunteer their time and skill to such effort, and may only work at their personal discretion rather than as assigned and scheduled. Further, FOSS developers are generally expected

(or prefer) to provide their own computing resources (e.g., laptop computers on the go, or desktop computers at home), and bring their own software tools with them. FOSS developers often work on global software projects that do not typically have a corporate owner or management staff to organize, direct, monitor, and improve the software development processes being put into practice on such projects [24].

The outsourcing of software development implies that an organization wholly or partially contracts out software development to another organisation. If the partner organization is located abroad, this might be termed "an offshore outsourcing of software development". If the development takes place in physically far-flung locations, it is called "global software development" or "distributed software development". Whether domestic or foreign, outsourcing can be an uncertain undertaking. Nonetheless many companies use offshore outsourcing to reduce time-to-market, to tap global resources, to profit from round-the-clock development, and to reduce costs. The goal of "offshore outsourcing software development" is to uphold competitiveness in the global market. This goal should be promoted by the concise and purposeful employment of every resource – information technology, talent and competence to assure a thriving offshore outsourcing project. All of which helps the company maintain ongoing global penetration. However, global distribution of the development raises a number of knotty questions concerning accomplishment and implementation. Often there is a huge disparity between targets and the results attained [13].

According to a recent paradigm shift in the field of software architecture, the product of the architecting process is no longer only the models in the various architecture views, but the broader notion of Architectural Knowledge (AK): the architecture design as well as the design decisions, rationale, assumptions, context, and other factors that together determine architecture solutions. Architectural (design) decisions are an important type of AK, as they form the basis underlying software architecture. Other types of AK include concepts from architectural design (e.g., components, connectors), requirements engineering (e.g., risks, concerns, requirements), people (e.g., stakeholders, organization structures, roles), and the development process (e.g., activities). The entire set of AK needs to be iteratively produced, shared, and consumed during the whole architecture lifecycle by a number of different stakeholders as effectively as possible. The stakeholders in architecture may belong to the same or different organization and include roles such as: architects, requirements engineers, developers, maintainers, testers, end users, and managers etc. Each of the stakeholders has his/her own area of expertise and a set of concerns in a system being developed, maintained or evolved. The architect needs to facilitate the collaboration between the stakeholders, provide AK through a common language for communication and negotiation, and eventually make the necessary design decisions and trade-offs. However, in practice, there are several issues that hinder the effective stakeholder collaboration during the architecting process, which diminishes the quality of the resulting product. One of these problems is the lack of integration of the various architectural activities and their corresponding artifacts across the architecture lifecycle. The different stakeholders typically have different backgrounds, perform discrete architectural activities in a rather isolated

manner, and use their own AK domain models and suite of preferred tools. The result is a mosaic of activities and artifacts rather than a uniform process and a solid product [12].

Software product line engineering enables customization of products for various market-segments from an abstraction called a product line platform. The set of products are developed from a product line platform is termed as a software product line. Software product line engineering provides several advantages based on reuse; quicker time-to market, improved cost savings and defect rates. Using software product lines several companies have recorded success stories. A product line platform is made up of several assets. An asset could be a system model element (artifacts that are used in software development such as use cases, classes, test cases etc) or a variability model element, an abstraction for variability. Variability is introduced in a product line platform as an abstraction to allow customization and reuse of artifacts to address the needs of different market segments. Variability management involves several activities. Variability identification covers identification and representation of variability; product instantiation which deals with the resolution of variability for individual products of a product line; and variability evolution, which addresses the change of variability itself. Product line evolution includes the evolution of system model elements and variability model elements. Software product line engineering involves two activities, domain engineering and application engineering. Domain engineering is an activity in which assets of a product line platform are identified, implemented and maintained. Another activity, application engineering is responsible for instantiating products from a product line platform. In product line requirements engineering, the activities of variability management are to be performed based on collaboration of domain and application engineering. Therefore, supporting collaboration between domain and application engineering is critical. The communication problem between conflicting views exists from the level of single system requirements engineering. To address the collaboration between domain and application engineering, in this contribution, variability management is extended using rationale management in order to enable issue-based collaboration between domain and application engineers. The collaboration supported by a rhetorical model is termed as issue-based collaboration. Rationale is defined as the reasoning that leads to a system model. Rationale management is viewed as a special branch of collaborative software engineering [26].

19.3 Today's Challenges

As should be clear from the collected chapters in this book, much work is presently ongoing in collaborative software engineering research, work of a broad variety and often great amount of depth. This work is beginning to make serious inroads into our ability to more effectively practice collaborative software engineering, with best practices, processes, tools, metrics, and other techniques becoming available for day-to-day use. However, we have not yet reached the point where the practice

of collaborative software engineering is routine, without surprises, and generally as optimal as possible. Partly, this is unavoidable, as the fundamental tensions discussed in Section 1.7 make achieving the optimum very difficult, if not impossible. At the same time, we should acknowledge that, while the research has advanced greatly over the past decade, many difficult challenges still exist when it comes to understanding and practicing collaborative software engineering. In the below, we highlight several key such challenges that we believe are among the most pressing and at the same time most promising to address at this moment in time.

Building a theoretical understanding of COSE. In any research field, one of the keys to advancement is to build an understanding of its underlying truths and phenomena. So it is in software engineering, and in the case of this book, collaborative software engineering. We need to build an understanding of what factors influence collaborative work and how those factors together determine the overall effectiveness of a given collaborative effort. This not only requires identifying each of the factors at play, but also how those factors influence one another. As one example, the role of awareness has been recognized for some time now [6]. As another example, trust has recently come forward as a crucial factor in distributed projects [1]. While each of these factors must be studied in depth, they cannot be studied in isolation; they are closely interrelated and must be understood as a collective. The notion of congruence is appealing in this regard, having recently been proposed as foundational and theoretical approach to contextualizing coordination needs versus coordination capabilities [4, 21]. It remains to be seen whether all necessary data can be gathered, but the concept represents an intriguing look at collaborative work.

Designing assessment methods for specific situations. Having an overall understanding of the factors at work in collaborative software engineering is not sufficient. We should also be able to assess specific situations and circumstances in which collaborative individuals, teams, and organizations find themselves. Are there any coordination problems presently? Are there latent issues that may lead to future coordination problems? If there are issues, what are some potential solutions to them? How will those solutions affect other collaborative factors in the organization? These are key questions for which we do not have good answers at this time. Social-technical network analysis with respect to the presence or absence of communication with respect to pieces of code that depend on one another is an example of a promising direction of research in this regard [22], though even there it is still unproven whether it is actually the presence or absence of communication that indicates good collaboration. Advocates of "presence" argue that such communication indicates that people talk and presumably resolve issues. Advocates of "absence" argue that if every technical dependency had to give rise to communication between developers, excess communication would take place. Moreover, they argue that other strategies, such as properly partitioning and scheduling the work, should actually prevent communication from being needed. At this time, there is no clear answer, other than that both sides of the argument are right at different times, but that we have no way of distinguishing yet when those times are. Similarly open-ended question pertain to assessing given situations with respect to a whole host of different factors – the field has not matured sufficiently yet in this regard.

Implementing tool support. Many recent advances in collaborative software engineering have to do with the creation of new tools in support of particularly collaborative practices. A host of tools has emerged, with various purposes behind them. Mylyn focuses on providing task context [10], CollabVS [7] and Palantír [23] on mitigating risks of parallel work, and Expertise Browser [14] on finding experts on particular areas of the code base. Many others exist, as the survey by Dewan in Chapter 7 shows [6]. Some tools are designed to help the researchers themselves, in efforts to understand collaborative practices and situations. Social-technical network analysis tools such as Ariadne [25], for instance, serve this purpose. But today's tools have only brought us "so far"; as new situations are investigated and hypotheses formed, new tools can be developed. One could think of tools that explicitly represent and work with trust, tools that prevent to just direct conflicts but also indirect conflicts, tools that better help identify necessary communications across team or organizational boundaries, awareness tools that cross phases of the life cycle, and so on. Much work remains to be done.

Beyond these three overarching categories, several challenges of "smaller" scale are presently at the forefront of the community. That is, within and across the above three categories in-depth investigations are needed regarding a variety of subjects. We mention such questions as: How could closed-source development benefit from open-source practices, and vice versa? How can knowledge better be preserved as it arises from and spreads to various teams in a collaborative environment? How can wikis be streamlined to more effectively support collaborative work? How can cultural barriers be bridged more smoothly? What other forms of awareness can be supported with tools? How can we better predict future coordination needs, and bottlenecks? Answers to these and other questions like it stand to improve the practice of collaborative software engineering, but will require a broad and deep research effort for years to come.

19.4 Prospects

This book has emphasized how collaboration is an integral part of software engineering project work, making it seem that the problems of collaboration are eternal, a form of status quo. This couldn't be further from the truth, as software engineering collaboration is a clear example of tangible forward progress. Technologies such as wikis, software forges, discussion lists, web sites, social network sites, email, instant messaging, mobile phones (and many others) combined with improved conceptual understanding of the collaborative goals and practice have created a golden age for project collaboration.

Consider the difference between collaboration practice today and 20 years ago, just prior to the widespread adoption of the Internet. Today, open source projects routinely gather project participants from around the world, use project forges for project collaboration (including mailing lists, SCM repositories, bug tracking systems, project web pages, etc.) and gather bug reports from users of their software.

Twenty years ago there were open source projects, but it was very challenging to create the collaboration infrastructure needed (you typically needed to be in an academic environment), the number of people on the Internet was much smaller than today, and knowledge of how to use tools such as CVS was thinly spread.

Today, commercial projects often involve multiple groups, located at different geographic sites. Collaboration technologies, combined with an improving conceptual understanding of how to manage and foster collaboration across wide geographic and cultural distance make these wide-area collaborations work, with comparatively little impact on project speed and quality. Twenty years ago, such wide-area collaboration was rare, modularized at the level of system-components, and extremely expensive. It is unclear whether it was even possible to perform the kind of fine-grain global software engineering that is commonplace today.

Today, a project web site is a common tool for collecting project documents such as requirements, designs, test plans, user interface sketches, and so on. While simple, such web sites are a huge improvement in recording and finding project knowledge over 20 years ago, when finding and copying project documents was major challenge.

It is commonplace today for software to report back to the manufacturer when it experiences a crash. Web sites with end-user submitted questions, workarounds for problems, and suggestions for future features are now typical. Even the most obscure discussion forum can potentially be critically useful if it holds discussion relevant to a specific user's problem. Twenty years ago, users were able to exchange this type of knowledge via Netnews, if they were lucky enough to be on the Internet. Computer user groups, software magazines, and software retail outlets also helped, but the knowledge could not be easily stored and searched.

Finally, today computer games such as Little Big Planet allow players to create and contribute new game levels for others to play... over one million of them so far. This type of user generated content was just not feasible before the internet, combined with low-cost storage and servers.

Dramatic as the past 20 years have been, the future of collaboration in software engineering promises to be even brighter. For starters, the widespread integration of the internet into most facets of life is just beginning. Mobile internet access, now very expensive, will become less expensive over time, promoting the spread of networks out of the first world, making it possible to tap the potential of many billions more people. There are many smart people in the world with time on their hands. Some simply wish to find some way they can make a positive contribution, and thereby generate meaning and create community in their lives.

Collaboration tools will become more sophisticated. Following the trend of desktop applications migrating to the web, software development environments will increasingly be web-based, allowing all project documents to live in the cloud. This, in turn, makes it possible to add social network site capabilities to projects, which should make it easier to build collaborations. With project data in the cloud, it should become easier to combine together various types of software project models, thereby finding errors and inconsistencies, but also recording richer networks of interrelationships among the artifacts. Awareness of the work of others should also

be easier in web-based environments, where all work, down to the keystroke level, is available.

As the amount of code available on the web continues to grow, so does the potential for finding existing source code to use in an existing project. Once key issues in the formation of searches and adoption of found code are resolved, this kind of anonymous collaboration via code repositories could result in substantial improvement in coding productivity.

During the first phase of internet adoption (c. 1990–2010) advances in software project collaboration generally were the result of being able to communicate cheaply with people at a distance, and having a universal viewer for documents (the web). Future advances will be more sophisticated, explicitly modeling interpersonal and project relationships, providing deeper integration of software project data, leveraging deeper understanding of code structure and meaning, and combining collaboration services in unique configurations.

The many chapters in this volume speak to the broad array of potential futures in software engineering collaboration. Though not all of these ideas will be widely adopted, together they make a compelling case that the future of collaboration in software engineering is bright, with much potential for further unleashing the potential of software engineers working in teams.

References

1. Al-Ani B, Redmiles D (2009). In strangers we trust? Findings of an empirical study of distributed development. IEEE International Conference on Global Software Engineering, 13–16 July, Limerick, Ireland, 2009.
2. Avritzer A, Paulish DJ (2009) A comparison of commonly used processes for multi-site software development. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
3. Bosch J, Bosch-Sijtsema P (2009) Softwares product lines, global development and ecosystems: collaboration in software engineering. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
4. Cataldo M et al. (2006) Identification of coordination requirements: Implications for the design of collaboration and awareness tools. ACM Conference on Computer Supported Cooperative Work, pp. 353–362.
5. Damian D, Kwan I, Marczak S (2009) Requirements-driven collaboration: Leveraging the invisible relationships between requirements and people. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
6. Dewan P (2009) Towards and beyond being there in collaborative software development. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
7. Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. European Computer Supported Cooperative Work, pp. 159–178.
8. Ghezzi G, Gall HC (2009) Distributed and collaborative software analysis. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
9. Happel HJ, Maalej W, Seedorf S (2009) Applications of ontologies in collaborative software development. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.

10. Kersten M, Murphy GC (2006) Using task context to improve programmer productivity. Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Portland, OR, USA, 05–11 November 2006). SIGSOFT '06/FSE-14. ACM, New York, pp. 1–11.
11. Lago P, Farenhorst R, Avgeriou P, de Boer RC, Clerc V, Jansen A, van Vliet H (2009) The GRIFFIN collaborative virtual community for architectural knowledge management. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
12. Liang P, Jansen A, Avgeriou P (2009) Collaborative software architecting through knowledge sharing. In: Mistrík I, Grundy J, van der Hoek, Whitehead J (eds.) Collaborative Software Engineering. Springer.
13. Mäkio J, Betz S, Oberweis A (2009) OUTSHORE maturity model: Assistance for software offshore outsourcing decisions. In: Mistrík I, Grundy J, van der Hoek A, Whitehead (eds.) Collaborative Software Engineering. Springer.
14. Mockus A, Herbsleb J D (2002) Expertise browser: A quantitative approach to identifying expertise. Proceedings of the 24th international Conference on Software Engineering (Orlando, FL, 19–25 May 2002). ICSE '02. ACM, New York, pp. 503–512.
15. Murta LGP, Werner CML, Estublier J (2009) The configuration management role in collaborative software engineering. In: Mistrík I, Grundy J, van der Hoek, Whitehead J (eds.) Collaborative Software Engineering. Springer.
16. Nakakoji K, Ye Y, Yamamoto Y (2009) Supporting expertise communication in developer-centered collaborative software development environments. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
17. Pinzger M, Gall HC (2009) Dynamic analysis of communication and collaboration in OSS projects. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
18. Richardson I, Casey V, Burton J, McCaffery F (2009) Global software engineering: A software process approach. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
19. Robinson H, Sharp H (2009) Collaboration, communication and coordination in agile software development practice. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
20. Sarma A, Al-Ani B, Trainer E, Sila Filho RS, da Silva I, Redmiles D, van der Hoek A (2009) Continuous coordination tools and their evaluation. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
21. Sarma A, Herbsleb J, van der Hoek A (2008) Challenges in measuring, understanding, and achieving social-technical congruence. Technical Report CMU-ISR-08-106, Carnegie Mellon University, Institute for Software Research International, Pittsburg.
22. Sarma A, Maccherone L, Wagstrom P, Herbsleb J (2009) Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development, Proceedings of the Thirty-first International Conference on Software Engineering, Vancouver, Canada.
23. Sarma A, Bortis G, van der Hoek A (2007) Towards supporting awareness of indirect conflicts across software configuration management workspaces. Twenty-second IEEE/ACM International Conference on Automated Software Engineering, November 2007, pp. 94–103.
24. Scacchi W (2009) Collaborative practices and affordances in free/open source software development. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (eds.) Collaborative Software Engineering. Springer.
25. de Souza C, Quirk S, Trainer E, Redmiles DF (2007) Supporting collaborative software development through the visualization of socio-technical dependencies. 2007 International ACM SIGGROUP Conference on Supporting Group Work (Sanibel Island, FL), November 2007, pp. 147–156.

26. Thurimella AK (2009) Collaborative product line requirements engineering using rationale. In: Mistrík I, Grundy J, van der Hoek, Whitehead J (eds.) Collaborative. Software Engineering. Springer.
27. Whitehead EJ (2007) Collaboration in software engineering: a roadmap. Future of Software Engineering (FOSE 2007), 23–25 May 2007, Minneapolis, MN, pp. 214–225.