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

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.

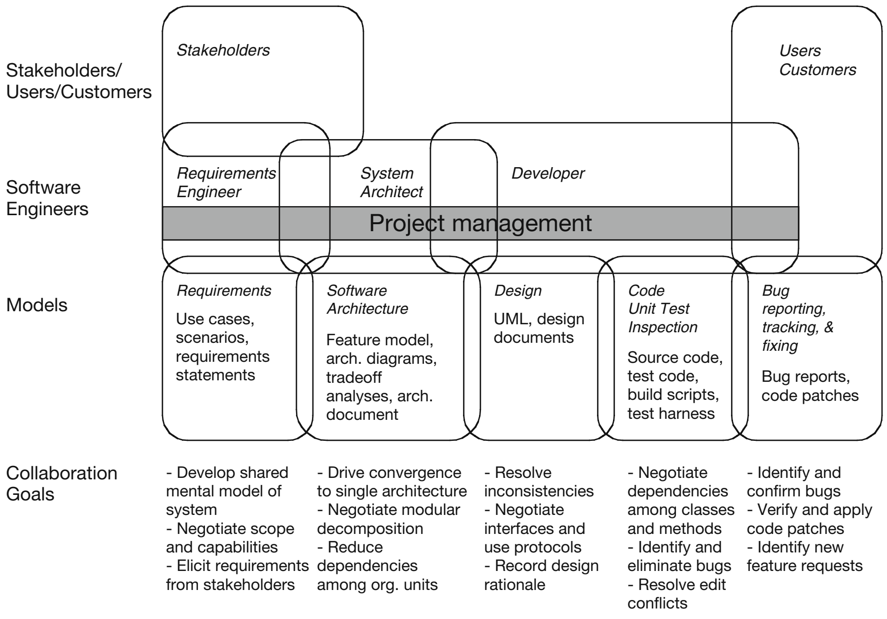
Stakeholders/ Users/Customers
Software Engineers
Models
Collaboration Goals
Stakeholders
Users Customers
Requirements Engineer
System Architect
Developer
Project management
Requirements
Use cases, scenarios, requirements statements
Software Architecture
Feature model, arch. diagrams, tradeoff analyses, arch. document
Design
UML, design documents
Code Unit Test Inspection
Source code, test code, build scripts, test harness
Bug reporting, tracking, & fixing
Bug reports, code patches
- Develop shared mental model of system - Negotiate scope and capabilities - Elicit requirements from stakeholders
- Drive convergence to single architecture - Negotiate modular decomposition - Reduce dependencies among org. units
- Resolve inconsistencies - Negotiate interfaces and use protocols - Record design rationale
- Negotiate dependencies among classes and methods - Identify and eliminate bugs - Resolve edit conflicts
- Identify and confirm bugs - Verify and apply code patches - Identify new feature requests

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 exploration of new media and platforms. Win-Win was designed to support a requirements engineering process that made negotiation processes explicit in the interface of the tool, with an underlying structure that encouraged resolution of conflicts, creating “win–win” conditions for involved stakeholders [5]. ART-SCENE supports a requirements elicitation approach in which a potentially distributed team writes use cases using a series of structured templates accessible via a Web-based interface. These are then used to automatically generate scenarios that describe normal and alternative situations, which can then be evaluated by requirements analysts [63]. Follow-on work has examined the use of a mobile, PDA-based interface for ART-SCENE, taking advantage of the mobility of the interface to show use cases to customer stakeholder in-situ [64]. The Software Cinema project examined the use of video for recording dialog between engineers and stakeholders, allowing these conversations to be recorded and analyzed in depth [17].

1.3.1.2 Architecture Centered Collaboration

Though the creation of final software architecture for a project is a collaborative and political activity, much of this collaboration takes place outside architecture-focused tools. Rational Software Architect is an UML modelling tool focused on software architecture. Engineers can browse an existing component library and work collaboratively on diagrams with other engineers, with collaboration mediated via the configuration management system. Research systems, such as ArchStudio [18, 95] and ACMEStudio [53] typically support collaborative authoring by versioning architecture description files, allowing a turn-taking authoring model. The MolhadoArch system is more tightly integrated with an underlying fine-grain version control system, and hence affords collaboration at the level of individual model elements [73]. Supporting an explicitly web-based style of collaboration, Maheshwari and Teoh [62], describes a web-based tool that supports the ATAM architecture evaluation methodology.

1.3.1.3 Design Centered Collaboration

Today, due to the strong adoption of the Unified Modelling Language (UML), mainstream software design tools are synonymous with UML editors, and include Rational Rose [44], ArgoUML [78], Borland Together [9], and Altova UModel [2] (a more complete list is at [102]). Collaboration features of UML authoring tools mostly depend on the capabilities of the underlying software configuration management system. For example, ArgoUML provides no built-in collaboration features, instead relying on the user to subdivide their UML models into multiple files, which are then individually managed by the SCM system. The Rosetta UML editor [32] was the first to explore Web-based collaborative editing of UML diagrams, using a Java applet diagram editor. Recently, Gliffy [30] and iDungu [45] have web-based diagram editors that support UML diagrams. Gliffy uses linear versioning to record document changes, and can inform other collaborators via email when a diagram has changed. SUMLOW supports same-time, same-place collaborative UML diagram creation via a shared electronic whiteboard [16].

1.3.1.4 Collaboration Around Testing and Inspections

Like requirements, testing often involves substantial collaboration between an engineering team and customers. Testing interactions vary substantially across projects and organizations. Application software developers often make use of public beta tests in which potential users gain advance access to software, and report bugs back to the development team. As well, best practices for usability testing involves multiple people performing specific tasks under observation, another form of testing based collaboration. Adversarial interactions are also possible, as is the case with a formal acceptance test, where the customer is actively looking for lack of conformance to a requirements specification.

Within an engineering organization, testing typically involves collaboration between a testing group and a development team. The key collaborative tool used to manage the interface between testers (including public beta testers) and developers is the bug tracking (or issue management) tool [90]. Long a staple of software development projects, bug tracking tools permit the recording of an initial error report, prioritization, addition of follow-on comments and error data, linking together similar reports, and assignment to a developer who will repair the software. Once a bug has been fixed, this can be recorded in the bug tracking system. Search facilities permit a wide range of error reporting. A comparison of multiple issue tracking and bug tracking systems can be found at [101].

Software inspections involve multiple engineers reviewing a specific software artifact. As a result, software inspection tools have a long history of being collaborative. Hedberg [34] divides this history into early tools, distributed tools, asynchronous tools, and web-based tools. Early tools (circa 1990) were designed to support engineers holding a face-to-face meeting, while distributed tools (1992–1993) permitted remote engineers to participate in an inspection meeting. Asynchronous tools (1994–1997) relaxed the requirement for the inspection participants to all meet at the same time, and Web-based tools supported inspection processes on the Web (1997–onwards). MacDonald and Miller [61] also survey software inspection support systems as of 1999. More recently, Meyer describes a distributed software inspection process using only off-the-shelf communication technologies, including voice over IP, Google Docs (web-based collaborative document authoring), and Wiki. These technologies were found to be sufficient to conduct effective reviews; no specialized review software was necessary [68].

1.3.1.5 Traceability and Consistency

While ensuring traceability from requirements to code and tests is not inherently a collaborative activity, once a project has multiple engineers, creating traceability links and ensuring their consistency is a major task. XLinkit performs automated consistency checks across a project [71], while [65] describes an approach for automatically inferring documentation to source code links using information retrieval techniques. Inconsistencies identified by these approaches can then form the starting point for examining whether there are mismatches between the artifacts created by different collaborators.

1.3.2 Process Centered Collaboration

Engineers working together to develop a large software project can benefit from having a predefined structure for the sequence of steps to be performed, the roles engineers must fulfill, and the artifacts that must be created. This predefined structure takes the form of a software process model, and serves to reduce the amount of co-ordination required to initiate a project. By having the typical sequence of steps, roles, and artifacts defined, engineers can more quickly tackle the project at hand, rather than renegotiating the entire project structure. Over time, engineers within an organization develop experience with a specific process structure. The net effect is to reduce the amount of co-ordination work required within a project by regularizing points of collaboration, as well as to increase predictability of future activity.

To the extent that software processes are predictable, software environments can mediate the collaborative work within a project. Process centered software development environments have facilities for writing software process models in a process modelling language (see [74] for a retrospective on this literature), then executing these models in the context of the environment. While a process model lies at the core of process centered environments, this process guides the collaborative activity of engineers working on other artifacts, and is not itself the focus of their collaboration. Hence, for example, the environment can manage the assignment of tasks to engineers, monitor their completion, and automatically invoke appropriate tools. A far-from-exhaustive list of such systems includes Arcadia [49], Oz [3], Marvel [4], Conversation Builder [51], and Endeavors [7]. One challenge faced by such systems is the need to handle exceptions to an ongoing process, an issue addressed by [50].

1.3.3 Collaboration Awareness

Software configuration management systems are the primary technology co-ordinating file-based collaboration among software engineers. The primary collaborative mechanism supported by SCM systems is the workspace. Typically each developer has their own workspace, and uses a checkout, edit, checkin cycle to modify a project artifact. Workspaces provide isolation from the work of other developers, and hence while an artifact is checked out, no other engineer can see its current state. Many SCM systems permit parallel work on artifacts, in which multiple engineers edit the same artifact at the same time, using merge tools to resolve inconsistencies [67]. Workspaces allow engineers to work more efficiently by reducing the co-ordination burden among engineers, and avoiding turn-taking for editing artifacts. They raise several issues, however, including the inability to know which developers are working on a specific artifact. Palantir addresses this problem by providing engineers with workspace awareness, information about the current activities of other engineers [85]. By increasing awareness of the activities of other engineers, they are able to perform co-ordination activities sooner, and potentially avoid conflicts. Augur is another example of an awareness tool [28]. It provides a visualization of several aspects of the development history of a project, extracted from an SCM repository, thereby allowing members of a distributed project to be more aware of ongoing and historical activity.

1.3.4 Collaboration Infrastructure

Various infrastructure technologies make it possible for engineers to work collaboratively. Software tool integration technologies make it possible for software tools (and the engineers operating them) to co-ordinate their work. Major forms of tool integration include data integration, ensuring that tools can exchange data, and control integration, ensuring that tools are aware of the activities of other tools, and can take action based on that knowledge. For example, in the Marvel environment, once an engineer finished editing their source code, it was stored in a central repository (data integration), and then a compiler was automatically called by Marvel (control integration) [4].

The Portable Common Tool Environment (PCTE) was developed from 1983 to 1989 to create a broad range of interoperability standards for tool integration spanning data, control, and user interface integration [96]. Its greatest success was in defining a data model and interface for data integration. The WebDAV effort (1996–2006) aimed to give the Web open interfaces for writing content, thereby affording data integration among software engineering tools, as well as a range of other content authoring tools [24, 98]. Today, the data integration needs of software environments are predominantly met by SCM systems managing files via isolated workspaces. However, the world of data integration standards and SCM meet in tools like Subversion [75] that use WebDAV as the data integration technology in their implementation.

For control integration there are two main approaches, direct tool invocation, and event notification services. In direct tool integration, a primary tool in an environment (e.g., an integrated development environment, like Eclipse) directly calls another tool to perform some work. When multiple tools need to be coordinated, a message passing approach works better. In this case, tools exchange event notification messages via some form of event transport. The Field environment introduced the notion of a message bus (an event notification middleware service) in development environments [81], with the Sienna system exemplifying more recent work in this space [13].

Ahmadi et al. suggest that future collaboration support for software projects should build upon a foundation of technologies that can be used to create social networking web sites, what they term Social Network Services [1].

1.3.5 Project Management

Software project management is intimately concerned with collaboration, since it structures the effort of the project via the creation of teams, subdivision of work to teams, schedules, and budget. These organizational, task, and cost structures drive the co-ordination and collaboration needs of a project.

Software project management is a subdiscipline of project management, and emerged as a separate concern within software engineering in the 1970s. During this decade, organizations made increasing use of computer-based information technology, leading to a demand for more, and larger software systems. The most influential early project management book is Brook's *Mythical Man Month* (1975) [10]. In 1981 Boehm defined the entire field of software economics in his landmark book of the same name [6] introducing COCOMO, the Constructive Cost Model for software. A January, 1984 edition of IEEE Trans. on Software Engineering [93] portrayed the state of the practice in software project management, and looked into its future. The year 1987 saw the release of DeMarco and Lister's *Peopleware: Productive Projects and Teams*, which emphasizes the importance of team collaboration [19]. A recent book in a similar vein was written in 1997 by McConnell, who proposed a list of Ten Essentials for software projects, based on "hard-won experience" [66].

The past 20 years have seen multiple efforts to capture and codify the knowledge and key practices required to perform effective project management. Watts Humphrey wrote *Managing the Software Process* in 1989, which first introduced the capability maturity model (fully completed in 1993) [38]. This model is significant for providing a multi-stage evolutionary roadmap by which an organization can improve its ability to manage and construct software systems. The IEEE Software Engineering Standards [47] capture many of the fundamental "best practices" of the software engineering project management. The Project Management Book of Knowledge (PMBOK), (1987, with four revisions since) documents and standardizes well-known project management knowledge and practices across a wide range of project types, including software projects [76]. The second edition of Thayer's *Software Engineering Project Management* [92] provides a framework for project management activities based on the planning, organizing, staffing, directing, and controlling model. The ISO 10006 "Quality management – Guidelines to quality to project management" [48], claims to provide "guidance on quality system elements, concepts and practices for which the implementation is important to, and has an impact on, the achievement of quality in project management".

In 2005 Pyster and Thayer decided to revisit software project management and assemble a set of articles that reflect how it has advanced over the past 20 years [77].

1.4 Global and Multi-Site Collaboration

In today's global economy, increasing numbers of software engineers are expected to work in a distributed environment. For many organizations, globally-distributed projects are rapidly becoming the norm [35]. Organizations construct global teams so as to leverage highly skilled engineers and site-specific expertise, better address the needs of users and other stakeholders, spread project knowledge throughout the organization, exploit advantages of specific labor markets, accommodate workers who wish to telecommute, and reduce costs. Mergers and alliances among organizations also create the need for distributed projects. While providing many advantages, global distribution also makes it harder for project members to collaborate effectively.

Global teams find it much harder to develop shared understanding around the evolving software artifact, as the distribution involved makes every aspect of communication more difficult. Team members at different sites lose the ability to have ad-hoc, informal communication due to spontaneous face-to-face interactions. Different sites often involve different national and organizational cultures, creating what Holmstrom et al. call socio-cultural distance [36]. As this distance increases, there is an increase in the challenge of interpreting the meaning of project communication. Engineers spread across many time zones reduce communication windows [33]. In reaction to these challenges, a core set of developers tends to emerge that acts as the key liaisons, or gatekeepers, between teams in different geographical locations. This team not only performs key co-ordination activities, but also contains the most technically productive team members [14].

Research on globally distributed software projects tends to focus on either characterizing their behavior (e.g. [33, 36]), or developing tools and techniques to mitigate the negative aspects of global distribution, so as to leverage its benefits. An example of the latter is the *global software development handbook*, which documents a wide range of issues and techniques for managing a global software project [82]. Lanubile provides a recent overview of tools for communication and co-ordination in distributed software projects [56]. In a hopeful sign that advanced tool support can overcome some of the drawbacks of global distribution, Wolf et al. report on a study of the development of the IBM's Jazz project [103]. This study shows that the Jazz team did not experience a significant decrease in project communication due to the distance between project sites.

Herbsleb presents a thorough survey of research on distributed software engineering in [35], along with thoughts on future research challenges. Herbsleb views the main challenge of distributed software engineering as the management of dependencies (that is, co-ordination) over a distance. We share this view, though this chapter also emphasizes the challenges inherent in creating shared meaning around (and identifying defects in) the many model-oriented artifacts in a software project.

1.5 Social Considerations

1.5.1 Software Teams

All engineering domains have a mix of technical and social aspects. For software engineering, such technical aspects include: software processes used to organise the life-cycle of software development; project management to co-ordinate teams working on software projects; requirements engineering, to capture key user needs of software systems and to specify – formally and/or informally – these needs; design, to identify the approaches via which the software systems will be realised; implementation, constructing executable systems; quality assurance, ensuring developed systems meet user requirements to acceptable thresholds; and deployment, making and keeping software systems available In addition, software very often must be modified over time and "maintained".

All of these technical activities must be carried out – in almost all cases – by a team of software engineers and related personnel. Such a "software team" is responsible for all of these technical aspects of engineering the software system and must be formed, organized, managed, evolved and ultimately disbanded. Team formation may be top-down or bottom-up [12, 99]. Recently team formation has had to take into account a trend to global software engineering including outsourcing, open sourcing and virtual teams [82].

1.5.2 Team Organization

Teams may be organised in a variety of ways [99]. "Tayloristic" teams have specialists filling specific roles, such as a requirements team, design team, testing team, coding team etc. These tend to be specialized, role-specific, task-focused and top-down directed units. "Agile" teams adopt a very different approach [88]. In these teams members tend to be generalists, the team people-focused rather than task-focused, and management bottom-up. Each of these teams brings very different social interaction protocols to bear on software development. Traditional, Tayloristic teams tend to be hierarchical and more centralized which suits some development projects and personalities. Agile teams tend to be more customer-driven, democratic and flexible. While this suits some developer personalities and problem domains it can be problematic. Each style of team organizationteam organization tends to utilize different collaboration approaches, project management strategies and sometimes tool support.

More recent trends have seen the rise of virtual software teams, outsourced software and open source communities. From a social perspective virtual teams need to overcome the challenges of distance, cultural and language differences and often different time zones [12]. Language barriers can mean it is difficult for team members to exchange information, co-ordinate work and communicate without mediation.

Cultural barriers can impact team dynamics in terms of co-ordination strategies, timeliness of work, and task allocation and monitoring. Different time zones delay communication sometimes leading to incorrect actions or incorrect assumptions about software artefacts and processes.

Outsourcing usually requires strong contractual relationships between teams [22]. Two common approaches are to divide an overall team into units of specialisation e.g. requirements, code, test etc., or to divide up the team vertically according to software function, e.g., the payments team, the on-line transaction processing team, the integration team. Collaboration challenges arise on the team boundaries, within teams as per other co-located models as well as for overall project management.

A very interesting set of social dynamics occur in the open source/voluntary software arena [21]. Often effort is either donated or contributed out of a sense of community belonging or mutual interest, in contrast to most other software development endeavours. This can lead to issues of ownership, or lack thereof, co-ordination challenges when available time of "team members" is unknown or opaque, and usually voluntary team membership for most or all members. Opt-in and opt-out to particular parts of a development project or software can often occur.

1.5.3 Team Composition

Team composition has a strong bearing on the social dynamics of both a single team and others its members may need to interact with. Some teams may be composed of a set of specialists while others mainly generalists. Traditional approaches to software team organisation often assume teams of specialists [99] and many outsourcing and virtual team models have also adopted this approach [12, 22]. Specialisation has advantages of clearer division of responsibility among members and ability to leverage particular skill bases. However it has major disadvantages when particular skills are rare or become unavailable for a time; and can lead to team conflict around divisions of work. Generalist teams are often favoured in agile projects [88] and are often a characteristic of many open source "teams" [21] by virtue of opt-in/opt-out driven by particular areas of interest or need.

Some teams include end users, or "customers", of the software product as a matter of course [88] whereas others isolate many team members from these customers [99]. Each has advantages and disadvantages in terms of collaboration support and project co-ordination from a social perspective. Customers generally have a very different perspective on the software project to developers and co-location greatly enhances communication and collaboration. However customers are often driven by self-interest and localised perspectives which may result in limited communication in particular areas.

Team membership can be whole-of-project, short-lived, or periodic. Some teams are created for the lifetime of a project in order to ensure available skill base and to enable deep understanding not only of the project but other team member's skills, abilities and awareness of work. Outsourced projects will typically leverage a remote team for the lifetime of the outsourced activity. Traditional teams may be sensitive to particular skill loss and agile teams try to mitigate this by a stronger emphasis on generalists [88].

Many teams are shared across projects. This is particularly common in virtual and out-sourced domains where specialised teams may be working on several projects at once. This greatly complicates inter-team communication and collaboration. Open source projects are often characterised by some team members participating for the whole duration of a project; some leaving early or joining later; and some participating on and off as their interest and time allows. Sometimes a team or members of a team may be contributing simultaneously to software development in different organisations. Again, virtual teams and particularly open source and outsourced projects may show this characteristic. These situations make building up a "corporate memory" around software a real challenge.

1.5.4 Knowledge Sharing

Knowledge sharing in software development has always been a challenge. The trend to global software engineering – common in virtual teams, outsourcing projects and open source projects – exacerbates this. Working in different time zones means that co-ordination of activities will typically be coarser-grained than possible with co-located teams.

Information may be written in different languages or from very different perspectives. Different emphases may be put on information depending on the cultural background of team members. Approaches to managerial aspects of teams, task division and reporting may need to take careful account and respect of cultural differences to ensure team harmony and effectiveness [55]. Language difference is probably the most obvious – and most challenging – issue when sharing knowledge across teams. However, cultural differences and the impact of different time zones and lack of face-to-face collaboration and co-ordination can also be significant issues [35, 55].

It is common to encounter significant differences in work culture, habits, approach to management and self-organization in cross-cultural teams. Again, open source projects, outsourcing projects and distributed software teams commonly exhibit the need to manage software engineering knowledge in cross-cultural, cross-language and cross-time zone environments.

1.6 Managerial Considerations

Software project management (SPM) includes the knowledge, techniques, and tools necessary to manage the development of software products. In more detail, SPM includes the inception, estimation, and planning of software projects along with tracking, controlling, and co-ordinating the execution of the software project. The goal of SPM is to tackle an optimal balance between planning and execution.

1.6.1 Software Project Management

The Project Management Institute defines project management as "the application of knowledge, skills, tools, and techniques to project activities in order to meet or exceed stakeholder's needs and expectations from a project" [76].

The intent of project management is to drive a project forward through a series of periods, phases and stages tailored to the specific project and its particular development and implementation strategy. These time intervals should be reflective of the product and its environment. Driving a project forward means steering it through these intervals separated by "gates" as a means of ensuring control and continued support by all of the partners involved [100].

Software engineering management can be defined as application of management activities – planning, co-ordinating, measuring, monitoring, controlling, and reporting – to ensure that the development and *maintenance* of software is systematic, disciplined, and quantified [46].

The key issue in Software Project Management (SPM) is decision making. Many of the decisions that drive software engineering are about how the software engineering process should take place, not just what software supposed to do or how it will do it, i.e., the project management has to be viewed in relation with product development and engineering processes.

1.6.2 SPM for Collaborative Software Engineering

There are four management areas that are particularly important in collaborative software engineering: (1) supporting communications in the project; (2) reconciling different stakeholder's viewpoints; (3) improving the process; (4) rapidly constructing the knowledge [25].

1.6.2.1 Supporting Communications in the Project

It is known that large organizations are associated with large communication overhead [6, 10, 54, 86]. For example, it is typical for an engineer in mid to large organizations to spend between half and three quarters of their time on communication, leaving only a fraction of their time for engineering work [86].

While the cost of communication has been noted for a long time, it is becoming increasingly worse. Communication overhead has a broad number of causes: *number of counterparts; differences in backgrounds, notations, and conventions; effectiveness of communication tools; distribution of organizations*. In general, the worse the communication overhead associated with the transmission of information, the less effective and responsive an organization becomes.

1.6.2.2 To Reconcile Conflicting Success Criteria in the Project

One of the problems in software development is to elicit and satisfy the success criteria of multiple stakeholders. Users, clients, developers, and maintainers are involved in different aspects of the development and operating of the software system, and have different and conflicting views on the system [26]. The role of the project manager is to elicit, negotiate, satisfy, and trade-off multiple criteria originating from the key stakeholders so that each stakeholder “wins” to ensure the success and sustainability of the product.

Often, the issue of dealing with conflicting success criteria is not only to reconcile conflicting views, but to identify the key stakeholders of the system and to clarify their success criteria. Once these criteria are known to all, it is much easier to identify conflicts and to resolve them by negotiating compromise alternatives.

To address these issues, there is a need for negotiation techniques and support early in system development, while changes in requirements and technology are possible and cost effective.

1.6.2.3 Improving the Process in the Project

Software engineering literature has provided many models, called life cycle models, of how software development occurs. In practice, software engineering tends to follow a more complex pattern, similar to problem solving in other human activities, which creative, opportunistic, involving, incremental building is followed by radical reorganizations sparked by sudden insights [72]. Moreover, the occurrence and frequency of the radical reorganization depend on the organization and the project context.

The field of software process improvements has gained ground in recent years, in supporting managers and organization in modelling and measuring software development processes. While software process improvement practices lead to more repeatable and more predictable processes, they usually do not deal with creative processes such as requirements engineering and do not support managers in dealing with radical reorganizations.

1.6.2.4 Rapidly Construct the Knowledge in the Project

A knowledge management approach should focus on the informal communication helping navigate and update digital repositories and digital repositories helping to identify key experts and stakeholders. Such a knowledge management approach would also enable stakeholders to create, organize, and capture informal or formal knowledge, in real time. This approach is called rapid knowledge construction [89].

Rapid knowledge construction is often needed when common knowledge needs to be elicited and merged from a number of groups, possibly distributed in the organization. Rapid knowledge construction includes the following challenges: *adaptable to context; real-time capture; enable reuse*.

Knowledge management and rapid knowledge construction are not management activities in the traditional sense (organizing work and resources). However, knowledge management is essentially cross-functional, and hence, requires the participation and facilitation of many levels, including project and program management.

1.7 Future Trends

As our understanding of software engineering collaboration deepens and the range of easily adoptable collaboration technologies expands, opportunities are created for improving collaborative project work. This section outlines several future trends in software collaboration research.

1.7.1 IDEs Shift to the Web

One clear trend in collaboration tools is the existence of web-based tools in every phase of software development. This mirrors the broader trend of many applications moving to the web, afforded by the greater interactivity of AJAX (asynchronous JavaScript and XML), more uniformity in JavaScript capabilities across browsers, and increasing processing power in the browser. Web-based applications have the benefit of centralized tool administration, and straightforward deployment of new system capabilities. They also make it possible to collect highly detailed usage metrics, allowing rapid identification and repair of observed problems. Web application variants can also be evaluated quickly by giving a small percentage of the users a slightly modified version, then comparing results with the baseline. The advantages of web-based applications are compelling, and create substantial motivation to move capability off of desktops and into the web.

Traditionally, the most significant drawback to web-based applications has been the lack of user interface interactivity, and so graphics or editing intensive applications were traditionally not viewed as being suitable for the web. In the realm of software engineering, this meant that UML diagram editing and source code editing were relegated to desktop only applications. Google Maps smashed the low interactivity stereotype in early 2005, and is now viewed as the vanguard of the loosely defined "Web 2.0" movement that began in 2004. Web 2.0 applications tend to have desktop-like user interface interactivity within a web browser, as well as facilities for other sites to integrate their data into the application, or integrate the site's data into another application.

The pathway is now clear for the creation of a completely web-based integrated development environment. The Bespin code editor supports highly interactive, feature-rich source code editing within a browser [70], with direct back-end integration with source code management systems. Due to the high degree of interactivity required, source code editing is the most thorny problem of moving to a totally web-based environment. Bespin demonstrates that completely web-based code editing is possible. With the source code editor in place, editors for other models in the software engineering lifecycle can be integrated. For example, the Gliffy drawing tool supports browser-based UML diagram editing [30]. Web-based requirements and bug tracking tools can also be tied in, along with web-based word processing and spreadsheets, such as Google Docs [31], Zoho Writer [104], and the Glide suite [94]. Web-based project build technologies such as Hudson [37] make it possible to remotely build and unit test software, removing the last threads that bind software development to the desktop.

The technical hurdle of bolting together multiple existing web-based tools into a single environment should be straightforward to overcome. What comes next are the fundamental research questions. To achieve close integration among tools, some form of data integration will be necessary. This then leads to the hard problem of developing data interchange standards among pluggable tools in various parts of the development lifecycle.

The ability to gather finely detailed information about the work practices of software engineers can allow rapid tuning and improvement of web-based environments. It also opens the possibility of a flowering of research in empirical software engineering, as large amounts of software project activity data are gathered across many open source software projects. This, in turn, raises the issue of just what degree of project monitoring is acceptable to developers, and who should have access to collected data.

A web-based environment opens the possibility for integration with other web-based collaboration technologies, such as social networking sites. This leads to our next future direction.

1.7.2 Social Networking

Social networking sites such as MySpace, Facebook, and LinkedIn have, in the space of a few short years, emerged as major hubs of social interaction. By providing awareness of the actions of friends and the ability to build closer social ties, these sites act as a kind of social glue, knitting together communities. These sites are also becoming major software development platforms, leading to the rapid rise of social gaming companies such as Zynga and Playdom.

It is an open question how best to integrate social networking sites into software development teams. The simplest approach is to have all team members use a single social networking site, and use it for non-project oriented socializing. Sites like Advogato [58] and Github [59] provide developer profiles. Advogato provides the ability for developers to rate each others' technical proficiency, creating a trust network. Each user also has a weblog. Github provides automated status update messages shown on a developer's profile page based on activity in Github managed software projects, and project-specific news feeds.

At present, sites like Advogato and Github only have affordances for the identity of each participant as a software engineer. This can be contrasted with sites like Facebook and MySpace, where a broader range of tools make possible the integration and presentation of multiple identities for each participant, though with a bias towards non-work identities. LinkedIn is another choice, clearly focused on business networking and job seeking. Clearly there is a potential for tight integration of software development activities with social networking sites. But how? One possibility is integration with Facebook. However, it seems a bit counter to the site's focus to have successful build and code checkin messages appearing in someone's wall. On the other hand, since sites like Github and Advogato have fewer social affordances, they feel less interesting than Facebook. Even for the most hardcore developers, there is more to life than code alone.

1.7.3 Broader Participation in Design

Many forms of software have high costs for acquiring and learning the software, leading to lock-in for its users. This is especially true for enterprise software applications, where there can be substantial customization of the software for each location. This leads to customer organizations having a need to deeply understand product architecture and design, and to have some influence over specific aspects of software evolution to accommodate their evolving needs. In current practice, customers are consulted about requirements needs, which are then integrated into a final set of requirements that drive the development of the next version of the software. Customers are also usually participants in the testing process via the preliminary use and examination of various beta releases. In the current model, customers are engaged during requirements elicitation, but then become disengaged for the requirements analysis, design, and coding phases, only to reconnect again for the final phase of testing. This can be seen in Fig. 1.1 (earlier in this chapter), where the stakeholders/users/customers row has engagement in requirements, and then again in test.

Broadened participation by customers in the requirements, design, coding and early testing phases would keep customers engaged during these middle stages, allowing them to more actively ensure their direct needs are met. While open source software development can be viewed as an extreme of what is being suggested here, in many contexts broadening participation need not mean going all the way to open source. Development organizations can have proprietary closed-source models in which they still have substantial fine-grain engagement with customers in which customers are directly engaged in the requirements, design, coding, and testing process. Additionally, broadening participation does not necessarily mean that customers would be given access to all source code, or input on all decisions. Nevertheless, by increasing the participation of the direct end users of software in its development, software engineers can reduce the risk that the final software does not meet the needs of customer organizations. As in open source software, a more broadly participative model can allow customers to fix those bugs that mostly directly affect them, even if, from a global perspective, they are of low priority, and hence unlikely to be fixed in traditional development. A participatory development model could also permits customers to add new features, thereby better tailoring the software to their needs.

A completely web-based software development environment would make it easier to broaden participation. In such an environment, it would be possible to give outsiders direct access to limited parts of the source code (and other project artifacts). With direct web-based access, external sites would not need to take source code offsite in order to build and test it, reducing the risk of proprietary information release.

1.7.4 Capturing Rationale Argumentation

An important part of a software project's documentation is a record of the rationale behind major decisions concerning its architecture and design. As new team members join a project over its multi-year evolution, an understanding of project rationale makes it less likely that design assumptions and choices will be accidentally violated. This, in turn, should result in less code decay. A recent study [91] shows that engineers recognize the utility of documenting design rationale, but that better tool support is needed to capture design choices and the reasons for making them.

Technical design choices are often portrayed as being the outcome of a rational decision making process in which an engineer carefully teases out the variables of interest, gathers information, and then makes a reasoned tradeoff. What this model does not reflect is the potential for disagreement among many experienced software engineers on how to assess the importance of factors affecting a given design. One of the strongest design criteria used in software engineering is design for change, which inherently involves making predictions about the future. Clearly we do not yet have a perfect crystal ball for peering into the future, and hence experienced engineers naturally have differing opinions on which changes are likely to occur, and how to accommodate them. As well, architectural choices often involve decisions concerning which technical platform to choose (e.g., J2EE, Ruby on Rails, PHP, etc.), requiring assessments about their present and future qualities. As a result, the design process is not just an engineer making rational decisions from a set of facts, but instead is a predictive process in which multiple engineers argue over current facts and future potentials. Architecture and design are *argumentative* processes in which engineers resolve differences of prediction and interpretation to develop models of the software system's structure. Since only one vision of a system's structure will prevail, the process of architecture and design is simultaneously cooperative and competitive.

Effective recording of a project's rationale requires capturing the argumentation structure used by engineers in their debates concerning the final system structure. Outside of software engineering, there is growing interest in visual languages and software systems that model the structure of arguments [52]. While models vary, argumentation support systems generally record the question or point that is being contested (argued about), statements that support or contest the main point, as well as evidence that substantiates a particular statement. Argumentation structures are generally hierarchical, permitting pro and con arguments to be made about individual supporting statements under the main point. For example, a "con" argument concerning the use of solar panels as the energy source for a project might state that solar electric power is currently not competitive with existing coal-fired power plants. A counter to that argument might state that while this is true of wholesale costs, solar energy is competitive with peak retail electric costs in many markets.

Providing collaborative tools to support software engineers in the recording and visualization of architecture and design argumentation structures would do a better job of capturing the nuances and tradeoffs involved in creating large systems. They would also better convey the assumptions that went into a particular decision, making it easier for succeeding engineers to know when they can safely change a system's design. A persistent challenge in rationale management in software engineering is keeping arguments consistently linked with the artifacts the affect (a form of traceability management). A completely web-based development environment, by providing centralized control over development artifacts, can ease this problem by making it possible to reliably perform link fix-up actions when an argument, or linked artifact, are changed.

1.7.5 Using 3D Virtual Worlds

Software engineers have a long track record of integrating new communication technologies into their development processes. Email, instant messaging, and web-based applications are very commonly used in today's projects to coordinate work and be aware of whether other developers are currently active (present). As a result, engineers would be expected to adopt emerging communication and presence technologies if they offer advantages over current tools.

Networked collaborative 3D game worlds are one such emerging technology. The past few years have witnessed the emergence of massively-multiplayer online (MMO) games, the most popular being World of Warcraft (WoW). These games support thousands of simultaneous players who interact in a shared virtual world. Each player controls an avatar, a graphic representation of the player in the world. Communication features supported by games include instant messaging, voice chat, email-like message services, and presence information (seeing another active player's avatar).

Steve Dossick's PhD dissertation [23] describes early work on the use of 3D game environments to create a "Software Immersion Environment" in which project artifacts are arranged in a physical 3D space, a form of virtual memory palace. Only recently have MMOs like Second Life emerged that are not explicitly role-playing game worlds, and hence are framed in a way that makes them potentially usable for professional work. While Second Life's focus on leisure activities makes it unpalatable for all but the most adventurous of early adopters, these environments still hint at their potential for engineering collaboration. IBM's Bluegrass project [40] is a 3D virtual world explicitly designed to support software project work. Goals of the work include improved awareness of the current status and ongoing work of a project, and project brainstorming. The work exposes many research issues in use of 3D virtual worlds for software project collaboration. Representation of software artifacts in the 3D world is a thorny problem, as there is no canonical way of spatially representing software. One possibility is to have the virtual space represent the organization of the various software project artifacts including requirements, designs, code, test cases, and so on. Alternately, the virtual space could be a form of idealized work environment, where everyone has a nice, large office with window. Combinations of the two are also possible, given the lack of real-world constraints. Virtual worlds typically have avatars that walk about in the world, a slower way of navigating project artifacts than a traditional directory hierarchy. The explicit representation of a developer avatar raises issues of appropriate representation of identity in the virtual space, an issue not nearly so prevalent in email, instant messaging, and other text-based communication technologies.

The utility of adopting a 3D virtual world needs careful examination, as the benefits of the technology need to clearly exceed the costs. It is currently unclear whether this is true.

1.8 Fundamental Tensions

Underneath many of the situations present and advances made in collaborative software engineering lie fundamental tensions that must be acknowledged. Optimizing towards one aspect of collaboration support often involves tradeoffs with respect to other aspects [84]. It is currently an open question as to where the theoretically optimal level of support lies for a given situation, a state some have labeled congruence [15]. Below, we identify some of the key tensions that exist.

What is good for the group may not be good for the individual. For an organization to effectively operate, certain individuals may be required to perform work that is not optimal from their personal perspective. Ultimately, of course, collaborative work must be optimized from an organization's perspective. However, if such optimization goes at the expense of the individuals, it is unlikely that a productive process is achieved. Some kind of balance must be found in which individuals' satisfaction with their work is respected, yet at the same time organizational needs are met. An example of when both can be achieved in parallel lies in the use of awareness technologies with configuration management workspaces [20, 83], where individuals are spared the merge problem, and organizations benefit from a higher quality code base.

What is good in the long term may not be good right now. Ultimately, the goal is to optimize the collaborative process as it plays out over time. This means that, at times, work performed right now is suboptimal in the short term, but crucial to later efficiencies. For instance, it is well-known that it is important to leave sufficient information along with the artifacts produced for later re-interpretation and re-consideration. However, such documentation is not always produced because it is seen as superfluous work, and even when it is produced, keeping it in sync with an ever-evolving code base is a tedious and arduous job.

Co-ordination needs are highly dynamic, but processes and tools in use tend to be largely static. Because of the ever changing nature of software and its underlying requirements, exactly what co-ordination needs exist that give rise to actual collaborations fluctuate [15]. But the processes and tools in use tend to be static in nature, chosen once at the beginning of the project and rarely adjusted after. Some tools have recognized this and provide different modes of collaboration e.g. [87], but in general serious mismatches can emerge between co-ordination needs and affordances.

Tools can, and should, only automate or support so much of collaborative practice. Ultimately, tools formalize and standardize work. Developers rely on tools every day, but it has been observed that they also establish informal practices surrounding the formally supported processes [80]. These informal practices are a crucial part of any effective development project. The tension, then, is how much to automate of the "standard" practices and how much to leave in the developers hands to enable them to own part of the process and flexibly be able to perform their work.

Sharing is good, but too much sharing is not. Much work must be performed in isolated workspaces of sorts to protect ongoing efforts from other ongoing efforts. The canonical example is each developer making their own changes in their own workspace, so they can test their changes in isolation and without interference by changes from other developers that may still be partial in nature. To overcome the issue of insulation becoming isolation, information about work must be shared with others. Such sharing can be beneficial, but must be carefully weighted with the fact that too much sharing leads to information overload, causing developers to ignore the information brought to them. Once again, a balance must be struck.

Record keeping is good, but it could be misused. The canonical example is the manager judging performance via lines of code contributed to a code base; this is a fundamentally flawed metric. With a broad set of new collaborative tools relying on and visualizing key data regarding individuals' practices, choices, and results, misuse of such data could lead to serious problems.

The above represents some of the key considerations that must be kept in mind when one attempts to interpret collaborative software engineering or provide novel solutions. In this book, we will see these tensions come back repeatedly, sometimes explicitly recognized as such, at other times providing implicit motivations and design constraints. These tensions will persist for the time and ages, and always govern how we approach collaboration.

1.9 Conclusions

After 35 years of research and tool making to foster collaboration in software engineering, we now have useful collections of tools, work practises, and understandings to guide multi-person software development activity. Indeed, internet-based collaboration tools and practices directly led to the creation of a globally distributed, open source software ecosystem over the past 20 years, accelerating in the last 10. Clearly, progress has been made in supporting collaborative software development.

Despite this progress, our understanding of collaboration in software engineering is still imperfect, and there is room for improvement in many arenas. A fundamental stumbling block is the lack of established metrics for quantitatively assessing collaboration in software projects. This, in turn, makes it challenging to know when a new collaboration tool has made an improvement, or when a new tool will make a difference. For example, it was only in hindsight that SourceForge (and similar web-based "forge" systems) was viewed as a major advance in software collaboration infrastructure, and not simply the integration of several pre-existing tools.

There are many current challenges in collaborative software engineering research. These include:

- *Understanding how to adapt new communications media for collaboration.* The computer is a rich nursery for new types of media. Social networking sites and 3D virtual worlds are two kinds of computational media that show potential for improving software project collaboration.
- *Reducing the effects of distance on remote collaboration.* Adding distance between people makes it harder to collaborate – is it possible to remove the negative effects of distance with superior tool support?
- *Improve shared understanding of artifacts.* Much work in software projects surrounds the removal of ambiguity in natural language and semi-formal artifacts. Improved collaboration support could assist this process of identifying ambiguity and developing shared understanding. Additionally, there is still room for improvement in the ways developers become aware of the work being performed by others.
- *Improved techniques for leveraging the expertise of others.* A persistent challenge in software engineering collaboration is identifying people within an organization that have expertise relevant to a current problem or task [27].
- *Improved ways of finding and removing errors.* Improving the collaboration between and among users and developers in identifying and fixing errors could help reduce software bugs, and improve the experience of using software.
- *Better understanding of how to motivate people to work together effectively.* As is mentioned in the previous section, there is a tension between individual and group goals. Providing sufficient rewards to encourage project collaboration is important, and not well understood.
- *Improve and integrate software project management, software product development, and software engineering processes.* This goal is often hampered by a great variety of methods and tools in the individual disciplines and limited integration methodologies between project management, product development, and engineering processes. An effective collaborative environment must inject basic elements of project management, including activity awareness, task allocation, and risk management, directly into the software engineering process.

The chapters in this volume address these issues, and more. In so doing, they deepen our understanding of collaboration in software engineering, and highlight the potential for new tools, and new ways of working together to create software projects, large and small.

References

1. Ahmadi N, Jazayeri M, Lelli F, Nesic S (2008) A survey on social software engineering. First International Workshop on Social Software Engineering and Applications (ASE 2008 Workshops), L'Aquila, Italy.
2. Altova (2009) Altova UModel – UML Tool for Software Modeling and Application Development.

3. Ben-Shaul IZ (1994) Oz: A decentralized process centered environment. PhD Thesis, Department of Computer Science, Columbia University.
4. Ben-Shaul IZ, Kaiser GE, Heineman GT (1992) An architecture for multi-user software development environments. ACM SIGSOFT'92: 5th Symposium on Software Development Environments, Tyson's Corner, VA, USA, pp. 149–158.
5. Boehm B, Egyed A (1998) Software requirements neogotiation: Some lessons learned. International Conference on Software Engineering (ICSE'98), Kyoto, Japan, pp. 503–507.
6. Boehm BW (1981) Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall.
7. Bolcer GA, Taylor RN (1996) Endeavors: A process system integration architecture. 4th International Conference on the Software Process (ICSP'96), Brighton, UK, pp. 76–89.
8. Borland (2009) CaliberRM – Enterprise Software Requirements Management System. http://www.borland.com/us/products/caliber/index.html.
9. Borland Software Corp. (2009) Borland Together.
10. Brooks FP Jr. (1975) The Mythical Man-Month: Essays on Software Engineering. Reading, MA: Addison-Wesley.
11. Bugzilla Team (2009) The Bugzilla Guide – 3.5 Development Release. http://www.bugzilla.org/docs/tip/en/html/.
12. Carmel E (1999) Global Software Teams. Upper Saddle River, NJ: Prentice-Hall.
13. Carzaniga A, Rosenblum DS, Wolf AL (2001) Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19(3): 332–383.
14. Cataldo M, Herbsleb JD (2008) Communication networks in geographically distributed software development. CSCW'08, San Diego, CA, USA.
15. Cataldo M, Wagstrom P, Herbsleb JD, Carley KM (2006) Identification of coordination requirements: Implications for the design of collaboration and awareness tools. CSCW 2006, Banff, Alberta, Canada, November, pp. 353–362.
16. Chen Q, Grundy J, Hosking J (2003) An e-whiteboard application to support early design-stage sketching of UML diagrams. IEEE Symposium on Human Centric Computing Languages and Environments, Auckland, New Zealand, pp. 219–226.
17. Creighton O, Ott M, Bruegge B (2006) Software cinema-video-based requirements engineering. 14th International Requirements Engineering Conference (RE'06), pp. 106–115.
18. Dashofy EM (2007) Supporting stakeholder-driven, multi-view software architecture modeling. PhD, Department of Informatics, School of Information and Computer Science, University of California, Irvine.
19. Demarco T, Lister T (1987) Peopleware: Productive Projects and Teams. New York: Dorset House Publishing.
20. Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. European Computer Supported Cooperative Work (ECSCW'07), pp. 159–178.
21. DiBona C, Ockman S (1999) Open Sources: Voices from the Open Source Revolution. Sebastopol, CA: O'Reilly.
22. Dominguez L (2006) The Manager's Step-by-Step Guide to Outsourcing. New Delhi: McGraw-Hill.
23. Dossick SE (2000) A virtual environment framework for software engineering. PhD, Department of Computer Science, Columbia University.
24. Dusseault L (2003) WebDAV: Next Generation Collaborative Web Authoring. Indianapolis, IN: Prentice Hall PTR.
25. Dutoit AH, McCall R, Mistrík I, Paech B (2006) Rationale management in software engineering: Concepts and techniques. In: Dutoit AH, McCall R, Mistrík I, Paech B (Eds.) Rationale Management in Software Engineering. Heidelberg: Springer-Verlag, 1–48.
26. Dutoit AH, Paech B (2001) Rationale management in software engineering. In: Chang SK (Ed.) Handbook of Software Engineering and Knowledge Engineering, Vol. 1. Singapore: World Scientific.

27. Ehrlich K, Shami NS (2008) Searching for Expertise. CHI 2008, pp. 1093–1096.
28. Froehlich J, Dourish P (2004) Unifying artifacts and activities in a visual tool for distributed software development teams. 26th International Conference on Software Engineering (ICSE'04), Edinburgh, Scotland, UK, pp. 387–396.
29. Gatherspace (2009) Agile Project Management, Requirements Management – Gatherspace.com. http://www.gatherspace.com/.
30. Gliffy Inc. (2009) Gliffy. http://www.gliffy.com/.
31. Google (2009) Google Docs: Create and Share Your Work Online. http://docs.google.com/.
32. Graham TCN, Ryman AG, Rasouli R (1999) A world-wide-web architecture for collaborative software design. Software Technology and Engineering Practice (STEP'99), Pittsburgh, PA, pp. 22–29.
33. Grinter RE, Herbsleb JD, Perry DE (1999) The geography of co-ordination: Dealing with distance in R&D work. GROUP 1999, Phoenix, AZ, USA, pp. 306–315.
34. Hedberg H (2004) Introducing the next generation of software inspection tools. Product Focused Software Process Improvement (LNCS 3009), pp. 234–247.
35. Herbsleb JD (2007) Global software engineering: The future of socio-technical co-ordination. Future of Software Engineering (FOSE'07), Minneapolis, MN, USA.
36. Holmstrom H, Conchúir EO, Ågerfalk PJ, Fitzgerald B (2006) Global software development challenges: A case study on temporal, geographical and socio-cultural distance. IEEE International Conference on Global Software Engineering (ICGSE'06), Princeton, NJ, USA, August.
37. Hudson Team (2009) Hudson: An Extensible Continuous Integration Engine. https://hudson.dev.java.net/.
38. Humphrey W (1989) Managing the Software Process. Reading, MA: Addison-Wesley.
39. IBM (2009) Getting Started with Rational DOORS. http://publib.boulder.ibm.com/infoceter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/doors_getting_started.pdf.
40. IBM (2009) Project Bluegrass: Virtual Worlds for Business. http://domino.watson.ibm.com/cambridge/research.nsf/99751d8eb5a20c1f852568db004efc90/1b1ea54cac0c8af1852573d1005dbd0c?OpenDocument.
41. IBM Rational (2009) Rational DOORS. http://www.ibm.com/software/awdtools/doors/.
42. IBM Rational (2009) Rational Method Composer. http://www.ibm.com/software/awdtools/rmc/.
43. IBM Rational (2009) Rational RequisitePro. http://www.ibm.com/software/awdtools/reqpro/.
44. IBM Rational (2009) Rational Rose. http://www.ibm.com/software/awdtools/developer/rose/.
45. iDungu.com (2009) iDungu.com – Enterprise Architect Web-Based. http://www.idungu.com/.
46. IEEE (1990). IEEE Std. 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology.
47. IEEE (1998). IEEE Std 1058–1998, IEEE Standard for Software Project Management Plans.
48. International Standards Organization (ISO) (2003). Quality Management Systems: Guidelines for Quality Management in Projects (ISO Std. 10006).
49. Kadia R (1992) Issues encountered in building a flexible software development environment. ACM SIGSOFT'92: 5th Symposium on Software Development Environments, Tyson's Corner, VA, USA, pp. 169–180.
50. Kammer PJ, Bolcer GA, Taylor RN, Hitomi AS, Bergman M (2000) Techniques for supporting dynamic and adaptive workflow. Computer Supported Cooperative Work (CSCW) 9(3/4): 269–292.
51. Kaplan SM, Tolone WJ, Carroll AM, Bogia DP, Bignoli C (1992) Supporting collaborative software development with conversation builder. ACM SIGSOFT'92: 5th Symposium on Software Development Environments, Tyson's Corner, VA, USA, pp. 11–20.
52. Kirschner PA, Buckingham-Shum S, Carr CS (2003) Visualizing Argumentation: Software Tools for Collaborative and Educational Sense-Making. London: Springer-Verlag.

53. Kompanek A (1998) Modeling a System with ACME. http://www.cs.cmu.edu/~acme/html/WORKING%20Modeling%20a%20System%20with%20Acme.html.
54. Kraut RE, Streeter LA (1995) Coordination in software development. Communications of the ACM 38(3): 69–81.
55. Krishna S, Sahay S, Walsham G (2004) Managing cross-cultural issues in global software outsourcing. Communications of the ACM 47(4): 62–66.
56. Lanubile F (2009) Collaboration in distributed software development. Software Engineering: International Summer Schools, ISSSE 2006–2008 (LNCS 5413), Salerno, Italy.
57. Lerner BS, Osterweil LJ, Sutton SM Jr., Wise A (1998) Programming process coordination in little-JIL toward the harmonious functioning of parts for effective results. European Workshop on Software Process Technology.
58. Levien R (2009) Advogato. http://www.advogato.org/.
59. Logical Awesome (2009) Secure Source Code Hosting and Collaborative Development – GitHub. http://github.com/.
60. Ludwig Consulting Services (2009) Requirements Management Tools. http://www.jiludwig.com/Requirements_Management_Tools.html.
61. Macdonald F, Miller J (1999) A comparison of computer support systems for software inspection. Automated Software Engineering 6(3): 291–313.
62. Maheshwari P, Teoh A (2005) Supporting ATAM with a collaborative web-based software architecture evaluation tool. Science of Computer Programming 57(1): 109–128.
63. Maiden N (2004) Discovering requirements with scenarios: The ART-SCENE solution. ERCIM News 58, July 2004.
64. Maiden N, Seyff N, Grunbacher P, Otojare O, Mitteregger K (2006) Making mobile requirements engineering tools usable and useful. 14th International Requirements Engineering Conference (RE'06), pp. 26–35.
65. Marcus A, Maletic JI (2003) recovering documentation-to-source-code traceability links using latent semantic indexing. 25th International Conference on Software Engineering (ICSE'03), Portland, OR, USA, pp. 125–135.
66. McConnell S (1997) Software Project Survival Guide. Redmond, WA: Microsoft Press.
67. Mens T (2002) A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5): 449–462.
68. Meyer B (2008) Design and code reviews in the age of the internet. Communications of the ACM 51(9): 67–71.
69. Microsoft Corporation (2009) Project Home Page – Microsoft Office Online. http://office.microsoft.com/en-us/project/default.aspx.
70. Mozilla Labs (2009) Bespin: Code in the Cloud. https://bespin.mozilla.com/.
71. Nentwich C, Capra L, Emmerich W, Finkelstein A (2002) xlinkit: A consistency checking and smart link generation service. ACM Transactions on Internet Technology (TOIT) 2(2): 151–185.
72. Nguyen L, Swatman PA (2001) Managing the requirements engineering process. 7th International Workshop on Requirements Engineering: Foundation for Software Quality, Interlaken, Switzerland.
73. Nguyen TN, Munson EV (2005) Object-oriented configuration management technology can improve software architectural traceability. 3rd ACIS International Conference on Software Engineering Research, Management and Applications (SERA'05), Mount Pleasant, MI, USA, pp. 86–93.
74. Osterweil L (1987) Software Processes are Software Too. International Conference on Software Engineering, Monterey, CA, USA, pp. 2–13.
75. Pilato CM, Collins-Sussman B, Fitzpatrick BW (2008) Version Control with Subversion (2nd Ed). Sebastopol, CA: O'Reilly.
76. Project Management Institute Standards Committee (2003). A guide to the project management body of knowledge (IEEE Std 1490–2003).
77. Pyster AB, Thayer RH (2005) software engineering project management 20 years later. IEEE Software 22(5): 18–21.

78. Ramirez A, Vanpeperstraete P, Rueckert A, Odutola K, Bennett J, Tolke L, Wulp M (2009) ArgoUML User Manual – A tutorial and reference description http://argouml-stats.tigris.org/documentation/manual-0.28/.
79. Ravenflow (2009) RAVEN for Rapid Requirements Elicitation and Definition. http://www.ravenflow.com/products/index.php.
80. Redmiles D, Hoek A, Al-Ani B (2007) Continuous coordination: A new paradigm to support globally distributed software development projects. Wirtschaftsinformatik 49: S28–S38.
81. Reiss SP (1995) The Field Programming Environment: A Friendly Integrated Environment for Learning and Development. Norwell, MA: Kluwer.
82. Sangwan R, Bass M, Mullick N, Paulish D, Kazmeier J (2006) Global Software Development Handbook. Boca Raton, FL: Auerbach Publications.
83. Sarma A, Bortis G, Hoek A (2007) Towards supporting awareness of indirect conflicts across software configuration management workspaces. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 94–103.
84. Sarma A, Herbsleb J, 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, Pittsburgh, PA, USA.
85. Sarma A, Noroozi Z, Hoek A (2003) Palantir: Raising awareness among configuration management workspaces. 25th International Conference on Software Engineering, Portland, OR, USA, May, pp. 444–454.
86. Scacchi W (1984) Managing software engineering projects: A social analysis. IEEE Transactions on Software Engineering 10(1): 49–59.
87. Schümmer T, Haake JM (2001) Supporting distributed software development by modes of collaboration. 7th European Computer Supported Cooperative Work (ECSCW'01), pp. 79–98.
88. Schwaber K (2004) Agile Project Management with Scrum. Redmond, WA: Microsoft Press.
89. Selvin A, Buckingham-Shum SJ (2000) Rapid knowledge construction: A case study in corporate contingency planning using collaborative hypermedia. KMAC 2000: Knowledge Management Beyond the Hype, Birmingham, UK, July.
90. Shukla SV, Redmiles DF (1996) Collaborative learning in a software bug-tracking scenario. Workshop on Approaches for Distributed Learning through Computer Supported Collaborative Learning, Boston, MA.
91. Tang A, Babar MA, Gorton I, Han J (2005) A survey of the use and documentation of architecture design rationale. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), Pittsburgh, PA.
92. Thayer RH (2001) Software Engineering Project Management, 2nd edn. Los Alamitos, CA: Wiley-IEEE Computer Society Press.
93. Thayer RH, Pyster AB (1984) Editorial: Software engineering project management. IEEE Transactions on Software Engineering 10(1): 2–3.
94. Transmedia Corp. (2009) Glide OS 3.0 – The First Complete Online Operating System. http://www.glidedigital.com/.
95. UCI Software Architecture Research Group (2009) ArchStudio 4 – Software and Systems Architecture Development Environment. http://www.isr.uci.edu/projects/archstudio/.
96. Wakeman L, Jowett J (1993) PCTE: The Standard for Open Repositories. Englewood Cliffs, NJ: Prentice-Hall.
97. Whitehead J (2007) Collaboration in software engineering: A roadmap. Future of Software Engineering (FOSE 2007), Minneapolis, MN, USA.
98. Whitehead EJ Jr., Goland YY (1999) WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web. 6th European Conference on Computer Supported Cooperative Work (ECSCW'99), Copenhagen, Denmark, pp. 291–310.
99. Whitehead R (2001) Leading Software Development Teams. London: Addison-Wesley.
100. Wideman RM (2009) Wideman Comparative Glossary of Project Management Terms (v. 5.0).

101. Wikimedia Foundation (2009) Wikipedia – Comparison of issue tracking systems. http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems.
102. Wikimedia Foundation (2009) Wikipedia – List of UML Tools. http://en.wikipedia.org/wiki/List_of_UML_tools.
103. Wolf T, Nguyen T, Damien D (2008) Does distance still matter? Software Process Improvement and Practice 13: 493–510.
104. Zoho Corp. (2009) Online Word Processor – Zoho Writer. http://writer.zoho.com/.