Chapter 7
Towards and Beyond Being There in Collaborative Software Development

Prasun Dewan

Abstract Research has shown that the productivity of the members of a software team depends on the degree to which they are co-located. In this chapter, we present distributed tools that both (a) try to virtually support these forms of collaboration, and (b) go beyond co-located software development by automatically offering modes of collaboration not directly supported by it.

7.1 Introduction

A variety of novel tools have been created to allow software developers to collaborate with each other. This chapter classifies them based 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.

Naturally, not all aspects of all research in collaborative software development are covered, or all viewpoints taken. By focusing on the "being there" and beyond themes, this discussion concentrates on the nature of the collaboration rather than the form of software engineering such as design and inspection. It addresses tools

P. Dewan (✉)
Department of Computer Science CB 3175, University of North Carolina, Chapel Hill, NC 27599-3175, USA
e-mail: dewan@cs.unc.edu

and related studies, rather than collaboration theories, cultural issues, organizational structures, studies that have not yet informed tool design, and other aspects of collaborative software development. Finally, it is intended to be a broad overview of the area, identifying relationships among diverse classes of research, rather than among different approaches within a particular class such as expert finders.

It begins by identifying the various degrees of physical co-location that have had an impact on software productivity. It then presents virtual channels that allow distributed developers to simulate these forms of co-location, and go beyond.

7.2 Productivity vs. Co-Location Degrees

Complex software must be developed collaboratively. However, Brooks [4] observed that adding more people to a software team can result in disproportionate increase in coordination cost, thereby reducing the productivity of the individual programmer. Surveys have found that, on an average, 50–80% of software developers' time is taken by communication [2, 44] and they are interrupted every three minutes [24].

These results seem unintuitive for two reasons. First, modular decomposition of software products should isolate software developers. Second, documentation should reduce the need for direct communication. However, studies have found that the approaches of documenting and partitioning are far from a panacea. Curtis et al. [10] found documentation is problematic for several reasons. Requirements, designs and other collaborative information keep changing, making it hard to keep their documentation consistent. After finishing an activity, software developers often choose to proceed to the next task rather than document the results of what they have done. People may deliberately hide information for career advancement. Sometimes there is conflicting information from different stakeholders that needs to be resolved through meetings. For example, for a defense project, the following stakeholders may provide different requirements: the champions responsible for getting the project approved the procurement office responsible for setting and monitoring the goals, the commanders, and the actual operators of the software to be created by the project.

Perry et al. [43] found that modularizing a project into multiple files does not isolate programmers. They studied Lucent's 5ESS system and found a high level of concurrency in the project – for example, they found hundreds of files that were manipulated concurrently by more than twenty programmers in a single day. Often the programmers edited adjacent or same lines in a file. They found that the more a file is accessed concurrently, the more the numbers of defects in it, despite the fact that state-of-the-practice versions control tools were used. There are many possible reasons for this correlation. After checking-in a file, a programmer may remember that some necessary change was not made, and to correct this mistake, may change the file in-place without creating a new version [26]. Programmers concurrently working on different private spaces (created from the same base) often race to finish first to avoid having to (a) deal with merging problems [26] and/or (b) re-run test suites on the merges [13, 14]. Programmers may not look at the documentation of previous versions to understand the code they are modifying [27]. Indirect conflicts on related files are not caught by differencing or file-based locking. Few people have a sense of the overall picture or the broad architecture [10, 27] which is required to reduce indirect conflicts.

All of the studies above assumed that team members are co-located in a single building and work from separate cubicles. If coordination/communication is really an issue, as these studies indicate, then distributing the team should further aggravate this problem and radically co-locating it, that is, requiring all team members to work in a single war-room, should reduce it. Two independent studies have found that this is indeed the case – the productivity of distributed teams was lower than that of co-located teams [32] and the productivity of radically co-located teams was higher than that of co-located ones [53].

The study comparing co-location and distribution [32] found that in distributed team development, it was harder to find people, get work-related information through casual conversation, get access to information shared with co-located co-workers, get timely information about plan changes, have clearly formed plans, agree about plans, be clear about assigned tasks, and have co-workers assist with heavy workloads, beyond what they are required to do. Interestingly most people thought that they gave help equally to local and remote collaborators but received more help from local collaborators. The study found that the perception of received help was the only factor that correlated with productivity.

The study on radical co-location [53] found two main factors that made it work better. First, there was continuous face-to-face communication among team members. Second, they were able to overhear and see each other's activity, which allowed them to solve their problems and interject commentary, clarifications and corrections. On the other hand, the study found that people sometimes wanted private spaces, and there was concern about distraction and getting individual recognition for work.

In radical co-location, even though the members of the team work in one room, they use different workstations. Higher physical coupling is achieved in pair programming, wherein two programmers sit next to each other, sharing a workstation and working on a single task, with only one programmer providing input to the workstation at one time. One study comparing pair and individual programming produced several interesting findings. It found that in the pair programming case (a) 80% of programmers felt higher satisfaction, (b) more alternatives were explored and fewer lines written, and (c) there was more team building as programmers were involved with each other and enjoyed celebrating project-completion together. Even more interesting, it found that pair programming took more person hours but resulted in fewer bugs [55]. Assuming certain times for fixing and detecting bugs, the study established that pair programming actually increases the productivity of an individual programmer. This result seems to contradict Brooks's law [4] which says that adding more programmers to a late project makes it later. The two results are not, in fact, contradictory, because Brooks assumed programmers were co-located but in separate cubicles. Thus, he did not consider radical co-location or pair programming.

The above studies, together, show that (a) communication and coordination are problems in team software development, (b) the more the physical coupling between members of a team, the less the severity of these problems. The moral, then, seems to be to increase the co-location degree among team members to the maximum degree possible.

Other work has shown that this conclusion is not necessarily correct. Nawrocki and Wojciehowski [39] found pair programming often took about twice as many person hours, though the pair-programming times showed less variance. Ratcliffe and Robertson [45] found that programmers with high (self-reported) skills did not like being paired with those with low skills.

More interesting, recent work has proposed a variation of pair programming, called side-by-side programming, wherein two programmers, sitting next to each other and using different workstations, work together on the same task [7]. A study showed that, in comparison to pair programming, side-by-side programming offers significantly lower completion times [40] while slightly reducing the understanding developers have of code written by their partners. It also found that developers who liked working together on a single task preferred side-by-side programming to pair programming.

The more complicated argument, then, seems to be that there are both benefits and drawbacks of tight physical coupling. Its strength is that multiple programmers can communicate with each other about a problem and possibly discuss it. Its weakness is that it reduces concurrency even when communication/discussion would be useless. More important, tight coupling may not always be preferred or even possible. For team members who are geographically dispersed, a closer physical coupling is not an option. Even when a team is co-located, because of lack of war-rooms in the workplace and the concerns mentioned above regarding radical co-location, team members may work in different rooms/cubicles. Pair programming is not widely practiced currently, and not always the most preferred or productive coupling, and even if it were, different pairs would have looser physical couplings. Thus, the communication/coordination problems of these couplings remain.

One way to address these problems is to provide virtual channels that simulate a variety of physical couplings, making the team members feel that they are together in a single building or room, or sitting in adjacent seats, or sharing a single workstation. This is consistent with the idea of taking steps towards virtually "being there." A complementary solution is to support virtual channels that reduce collaboration problems existing in all forms of co-location. This is consistent with the idea of virtually going "beyond being there" [34].

Examples of both kinds of channels exist in traditional – that is, state-of-the-practice – tools. For example, IM systems provide the "towards being there" functionality of synchronously chatting, and version control systems provide the "beyond being there" functionality of asynchronous merging. The fact that, despite the pervasiveness of these tools, communication/coordination is still a major issue in team software development seems to indicate that there are opportunities to significantly improve existing collaboration channels. Several research efforts have explored such opportunities. The remainder of this chapter surveys some of the concepts identified by these research efforts, and experience with these concepts.

7.3 Towards Being There

7.3.1 Virtual Co-location and Radical Co-location

Co-location, especially radical co-location, allows developers to easily communicate to the whole team events of shared interest. When the team is distributed, several approaches have been devised and used for conveying this information. A version control system provides a way for distributed programmers to formally communicate some of this information through check-in comments. Grudin and Poltrock [28] advocate the use of project Blogs to informally communicate with co-developers. Gutwin et al. [29] found that email can be a practical alternative for announcing important, infrequent events such as the starting and termination of tasks.

For supporting continuous "stream of consciousness babbling" [31] of the kind that can be expected in radical co-location, lighter weight tools have been developed. Elvin [22] is an example of such a tool. Messages posted by a team continuously scroll in a ticker tape. A tool with similar goals is RVM (Rear View Mirror) [31] so named because it is intended as an unobtrusive background "rear view mirror" for the members of the team as they performed their tasks. User studies yielded several counter-intuitive results about desired features. Originally, the tool showed users only the last few hours of those messages that were exchanged when they were logged on. Based on user feedback it was changed to support all of the conversation. Also previously, an explicit permission had to be given to each person viewing presence information. Based on user feedback, the system was changed to allow each member of a team to see the presence information of all the other members. Presence information was liked more than chat. In fact, managers exchanged only two chat messages during the study!

A potential problem with the tools above is that a developer interacting with a programming environment must switch to a separate tool to see the presence information of and interact with co-developers. Jazz [6] and CollabVS [30] provide these facilities, in-place, within the programming environment. A study of CollabVS found that programmers preferred in-place presence and communication [30].

7.3.2 Distributed Pair Programming

The channels above simulate physical channels in radical and regular co-location. Let us next consider concepts supporting the higher physical coupling provided by pair programming.

The easiest way to support such coupling is to use a generic desktop-sharing system, which traps window level input events and window or frame-buffer level screen updates, and transmits them to a remote collaborator. An alternative is to couples the edit buffers and other components of the semantic state of the programming environment of the developers [20, Schummer #1092]. The former is slower and requires use of a special, potentially unfamiliar system for sharing. On the other hand, the latter requires the developers to manually synchronize their views. A hybrid approach, taken in Jazz [6] and CollabVS [30] is to add commands to the user-interface of a programming environment to invoke a desktop sharing system [6, 30].

A study comparing distributed and co-located serial pair programming found that physical distance does not matter [1]. This is an interesting result because, as mentioned above, studies of individual programming have found that distance reduces productivity [32].

7.3.3 Distributed Side-by-Side Programming

As mentioned above, a variation of pair programming is side-by-side programming, wherein two programmers sit next to each other working on the same task. It offers (potentially) looser coupling than pair programming, as the developers can work concurrently on different aspects of the task; and tighter coupling than radical co-location, as they are required to work on a single task that has not been decomposed for them; and more important, are able to see all actions of their partners.

Dewan et al. [17] have devised a distributed analog of this idea. Each developer in the pair interacts with two computers – one primary computer to act as the driver of his subtask, and an awareness computer to act as the navigator for the partner's subtask. In other words, each programmer interacts with the windows displayed on his primary computer, and each awareness computer shows the screen of the partner's primary computer. The developers use the phone to talk to each other. No video channel is established between them in this set-up.

A desktop sharing system is used to ensure that each awareness computer shows the screen of the partner's desktop. In addition a model-sharing system such as a file system or a Web server is used to synchronously share edits to code made concurrently on the two primary computers. Thus, the same input is shared at multiple levels of abstraction – at the window level by the desktop sharing system and at the semantic level by the model sharing system.

In this architecture, local response is not affected by the network delays, as is the case in single-computer (desktop-sharing based) distributed pair-programming implementations. Thus, the two-computer solution offers good response times for even pure pair programming.

A study of distributed side-by-side programming showed that developers used its ability to dynamically switch between pair programming, independent programming, and several other intermediate synchronous programming modes such as concurrent searching/programming [17].

7.3.4 Distributed Synchronous Design and Inspection

Distributed pair programming is only one form of distributed synchronous software engineering. It is particularly interesting because, traditionally, programmers collaborate asynchronously on different parts of a project rather than synchronously on every line of code. On the other hand, activities that precede and succeed the coding phase – design and inspection – are typically carried out in synchronous, face-to-face meetings. Therefore, tools have been built and effectively used for distributed synchronous design [41] and inspection [38]. A study of distributed synchronous inspection has shown that it is as effective as face to face inspection in terms of faults found, but developers preferred face-to-face inspection [38].

7.3.5 Other "Towards Being There" Mechanisms

There are a variety of other kinds of distributed tools such as connected kitchens [35] video walls [25] and media spaces [37] which provide elements of being there in the same building or room. However, as there have been no studies of their use in team software development, we ignore them in this chapter. See [15] for a survey of these and other tools that have not been targeted at software development.

7.4 Beyond Being There

Software tools that go beyond being there automate various aspects of collaboration, and are thus useful for both (radically) co-located and distributed teams.

7.4.1 File System Events

Traditional version control systems provide an important form of collaboration automation. When users check-out or check-in files from a version control repository, interested users are automatically notified about these events. O'Reilly et al. [42] point out that it is also useful to monitor operations at the file-system level, for several reasons. Sometimes users manually change the permissions of files to make them writeable instead of checking them out from the repository. A new project file is not known to the repository until it is checked in. A repository tracks events at the user level – sometime a user takes multiple personas, creating multiple different private workspaces from the same base. While working on one of these workspaces, it is not possible for him to be notified about actions he took in another workspace.

Therefore they extend the repository events above with the following additional events: (a) Added/removed: A file known to the repository has been added to/removed from project working directory pending commit. (b) Updated: A file in the repository has been updated in the working directory. (c) Needs checkout: A file in the working directory has been updated in the repository. (d) Needs merge: A file has been updated both in the working directory and the repository. (e) Unknown added/removed/updated: A file in the working directory not known to the repository has been added/removed/updated.

7.4.2 Persistent Awareness vs. Notifications

An alternate to notifying interested developers about operations of their collaborators on files is to update a persistent view of the file status in the user-interface of a programming environment or a separate tool. For example, the Jazz [6] and CollabVS [30] programming environments continuously indicate to developers which files have been checked-out or are being edited by their team members. In FASTDash [3] a separate tool provides this facility. Thus, programmers interested in knowing, for example, if a file is being currently edited by a collaborator need only look at the persistent view rather than mine through the event history to determine this information. On the other hand, changes to the awareness information may go unnoticed. For example, if two developers start editing the same file, neither of them may notice the change to the view of the file status. Thus, both persistent awareness of and notifications about collaborators' operations on files/versions are useful.

7.4.3 Programming Environment Events

Operations on objects maintained by a programming environment that are not known to the file or version control system may also be of interest to collaborators. These include starting/stopping of the editing of a particular program construct such as method or class [19, 50, 51] and concurrent editing of the same or dependent program constructs [19, 46, 50]. Awareness of this information can be provided through notifications or updates of persistent status views. For example, in CollabVS, concurrent editing of dependent program constructs results in both notifications and updates of awareness views [19].

Three studies have shown the usefulness of providing awareness of programming environment events. A study of Tukan found that when programmers found themselves editing the same program construct, they transformed their individual coding sessions into a joint pair programming session [51]. Two studies, of CollabVS and Palantír, respectively, have found that programmers used information about concurrent editing of dependent constructs to prevent direct and indirect conflicts [19, 48]. The comments from the CollabVS study [19] also showed that programmers liked having information about programming environment events even when these events had no apparent benefit such as conflict prevention. Hegde and Dewan [30] give several scenarios in which awareness of programming environment events may be useful. For example, if Alice sees Bob taking an undue amount of time editing a method, she can offer to help him with the task.

7.4.4 Shared Version with Multiple Views

Suppose Alice does wish to help Bob finish his task. One approach to do so is to use distributed pair or side-by-side programming. As mentioned above, distributed pair programming requires her to work lock step with Bob, not allowing concurrent work on the task. The scheme for distributed side-by-side programming described above addresses this issue, but suffers from two related drawbacks in its attempt to faithfully mimic co-located side-by-side programming. First, it requires each programmer to view a separate display to observe his partner's incremental updates. Second, to receive these edits, the developer must manually pull them from the file system or web server. These updates are not automatically pushed to him.

Some software development systems have addressed these two problems using a variation of distributed side-by-side programming. In these systems, as in side-by-side pair programming, the developers work on the same version of the code-base. The difference is that they can edit it concurrently using different views of it that are updated automatically or manually. This is a special case of the general idea of editing the same model using multiple views [18]. Changes to the model can be pulled and pushed at various time and space granularities depending on the coupling between the views [18].

An early system supporting this approach was Flecse [20] which provided tools that allow programmers to do synchronous concurrent editing, debugging, testing and inspection. As motivation for such tools, the paper on Flecse [20] provides the following hypothetical scenario. Three users have finished creating different procedures of a matrix multiplication program. One of them finds an error in the output. Two of them use the Flecse collaborative debugger to jointly work with another to find the bug. The two users find that the bug can be fixed by changing the semantics of one, of two procedures and cannot agree on which, one of these should be changed. They use the Flecse multi-user inspection tool to hold a more formal code-review meeting involving all three users to make the decision. The tool allows them to make their annotations privately before discussing them in public. The code review session suggests changing both procedures to eliminate other related errors.

A follow-up to this work was CAIS [38] an inspection tool supporting both asynchronous and synchronous inspection. User studies with this tool [38] found that people preferred to perform software inspection asynchronously, until the discussion became controversial, when they switched to synchronous discussion.

Several other tools have been built based on these ideas. CollabVS [30] allows developers to asynchronously share the contents of their edit buffers before checking them to the version control system. SubEthaEdit and Sun's JSE 7 allow synchronous editing of the same file in different views. JSE 7 also supports synchronous collaborative inspection by allowing code to be sent through the chat tool, which correctly formats it. Users can independently scroll the shared code and user comments about it in the chat window, thereby seeing different views of the inspection data.

Unlike the scheme for distributed side-by-side programming, none of these systems require a special awareness screen. In these systems, when developers edit the same model using different views, they can lose track of the activities of their collaborators. As a separate screen for showing these activities is not guaranteed, more space efficient and thus higher level mechanisms are needed for allowing the team members to be aware of each others' views. These mechanisms are different from those we saw above that allow developers to be aware of the semantic or model changes of their collaborators. For example, a multi-user scrollbar in SubEthaEdit, which shows the scrollbars of the collaborators, provides view awareness, while awareness about the methods being edited by collaborators, provided by CollabVS [30] provides model awareness.

Few studies have been performed in which software developers concurrently interact with different views of a shared version without special awareness screens. Two exceptions are [30] and [8] which were targeted mainly at determining if such a mechanism could reduce conflicts, and found that this is indeed the case.

7.4.5 Searching and Mining

Allowing developers to easily search for project-related information is another form of automation that can be supported by a beyond being there tool. Microsoft's Team Visual Studio allows developers to easily track information related to work items. It associates a work item with status information indicating whether it is active, pending, resolved, or closed. A check-in can be linked to the work item implemented by it. In addition, if the work item is a bug correction request, it can be linked to the build and test suites that identified the bug. These links allow the system to search for various kinds of project information – in particular the status of work items, the users assigned to a work item, and duplicate work items.

Hipikat [9] extends the above concept by linking additional kinds of information, deriving some of these links automatically based on similarity of documents, and providing sorted recommendations in response to requests for similar documents. These queries are made from the programming environment by asking the system to provide documents similar to the one that is selected. The user can then recursively look for documents similar to the recommendations.

To determine how well these features worked, two user studies were performed, involving an "easy" and "difficult" task. In the easy task, programmers were asked to extend Eclipse's hover capability. Given the task description, Hipikat automatically found a very similar past task as the highest recommendation. As a result, novice programmers who used Hipikat were as successful as expert programmers who did not. In the hard task, programmers were asked to extend Eclipse's version system integration. Expert programmers who did not use Hipikat missed some subtle issues while some novice programmers who used Hipikat addressed them because the system provided a recommendation that identified these issues. On the other hand, some of the Hipikat users also missed these issues as the relevant recommendation was not the highest ranked one. Studying each recommendation was a difficult heavyweight task – hence when users found a relevant recommendation, they did not look for lower ranked recommendations that provided additional information. Thus, the recommendations were used shallowly to understand how to start a task, rather than deeply to understand the system architecture.

The general idea of mining the software artifacts created by groups of software developers has several other useful manifestations. Document similarity can be used to trace different versions of some software artifact, and thereby gain a better understanding of the project [36]. SpotWeb [54] finds reuses of the classes of a framework, and classifies the reused classes as hotspots (coldspots) if there are several (few) reuses of the classes. Hotspots (coldspots) can be expected to more (less) tested and hence reliable than the average classes. More important, from the point of team software development, new developers in a team can, instead of consulting older members, look at hotspots and associated reuses to understand how to use the framework. Zou and Godfrey [57] mine interaction histories to automatically separate newcomers and experts – the latter tend to focus on a smaller set of artifacts. This information can be used to find experts, and also to pair developers in pair programming.

Cataldo et al. [5] have found that mining the version-control logs provides a method for finding useful dependencies, which complements the static analysis used in CollabVS and Palantír. For example, files that are committed together have been found to be closely related to each other. They classify communication among developers as "good" or "bad" based on whether or not the programmers are modifying dependent artifacts. Xiang et al. [56] build on this idea by automatically recommending communication among developers working on dependent files. Schroter et al. [49] support a variation of this idea on in which the communication is recommended only on failed builds.

7.4.6 Visualization

Visualization is an alternative to query-based searches. Instead of specifying a query to find some aspect of data, users locate it in a visualization of the data. Tools have been developed to provide visualization (a) in-the-large of the entire software engineering project, (b) in-the-medium of sets of files, and (c) in the small of components of a file.

Doppke et al. [21] visualize and enforce the software process by mapping it into MUD abstractions [11]. Each task is mapped to a MUD virtual room containing representations of the artifacts manipulated to perform the task. For example the testing activity is mapped to a room containing the executable being tested, the inputs fed to it, and the output produced by it. A human enters a room with artifacts to perform the associated activity, and cannot leave until the activity is finished. On leaving, the human is directed to the next activity in the workflow.

Doppke et al. found that software processes could not be mapped completely too traditional MUD spaces, for several reasons. (a) In a traditional MUD environment, a person can be in a single room at a time, while in a software process, a human can be in multiple activities simultaneously. Therefore, they defined the abstraction of a persona, which is a person's activity thread. A persona is always at a particular stage in the activity. When a person enters a room taking on the role of the persona in the room, he carries on work from that stage onwards. A persona rather than person is mapped to a room. They defined several typical software engineering personas such as generic developer and programmer. (b) A software process is associated with constraints – therefore they extended MUDs to support programmer-defined constraints for entering and leaving a room. (c) A software task can have several subtasks, which in turn can have their own subtasks. This was modeled by creating sub-buildings within rooms.

Instead of or in addition to displaying the current state of the formal process associated with a project, it is also possible to visualize the informal collaboration describing its state. Jazz [6] creates such a visualization, called "Team Jam," which is a persistent virtual place that includes a discussion board, links to transcripts of chats, and notifications of the kind of events we saw earlier such as check-in and check-out of code.

Instead of seeing the complete communication regarding a project, as in Team Jam, it may be useful to understand the impact various aspects of the communication has on a project. Sarma et al. [47] have recently developed a browser/visualize, called Tesseract, that allows programmers to relate artifact dependencies, communication patterns, and features/bug fixes. For instance, given a bug-fix, a user can see a visualization of all files and developers involved with the bug, and which of these developers communicated with each other. Similarly, it visualizes the relationship between the amount of communication among developers working on related artifacts and the number of bugs.

Tesseract, Team Jam, and the MUD-like process visualization address visualization in the large of the entire project. Let us consider next techniques for visualizing in the medium and small.

Palantír [48] provides visualization of concurrent accesses to hierarchic objects checked-in by a user to a private workspace. Each object checked out by the user is associated with a stack of tiled boxes. Different tiles in the stack correspond to parallel checked out versions. Each tile can contain sub-stacks corresponding to subobjects. The stacks are sorted by severity of divergence among the tiles in the stack. Palantír allows the application to calculate the severity, and proposes some simple measures including changed vs. not changed, lines changed/total lines, and number of interface changes. Thus, Palantír provides visualization in the medium of sets of files as sorted, nested collections of file stacks.

Several examples of visualization in the small exist. One of them is based on the notion of physical wear, which is a useful concept in the physical world – by following worn paths, we can find our way in an unknown terrain; and by looking for worn pages in a recipe book, we can find the popular recipes. Hill and Hollan [33] create a virtual concept out of physical wear. They associate a line of text with edit and read wear. Edit wear counts the number of edits made to the line. Edits can be differentiated based, for instance, on time and author, to create different categories of wear. Read wear measures how long the line was viewed before it was scrolled out or the viewing user became inactive. Edit/read wear can be used to determine, for instance, which sections are currently changing the most or of most interest.

In the visualization provided by Hill and Hollan [33] collaboration-related information about a file is shown in-place by widths of lines in the scrollbar of a window displaying the file. Froehlich and Dourish [23] provide an alternative approach wherein collaboration information is displayed by coloring a miniature of the file displayed in a separate tool. Each line of text in a program is represented by a graphical line consisting of three parts. The first two parts are of fixed length, while the length of the third part is proportional to the length of the associated text line. All three parts are colored to indicate collaboration attributes of the text, which include author, age (time of last edit), and structure (method, comment, import, variable declaration).

Froehlich and Dourish deployed this system and found that people liked the fact that they could see project growth over time. Users reported discovering notable aspects of the team development such as finding (a) from the drastic changes to a file one day that re-factoring happened that day, (b) up to 15 authors for some files, (c) files with unusual growth patterns, (d) different indentation and import styles, (e) changes made by others to files they thought were owned by them, (f) heavily indented files that were candidates for re-factoring, (g) structures of large functions without scrolling.

7.4.7 Context-Based Automatic Filtering

In both searching and visualization, a software developer must explicitly find information of interest. An alternative is to automatically show information relevant to the current task of the developer that is based on the activities of the whole team. An example of this approach is supported by Team Tracks [12]. It allows developers to identify those classes of the current project that are often visited by the team. (These are different from hotspots which are classes that are often reused but not necessarily often visited). In addition, if the developer is currently viewing some program construct, Team Tracks shows a list of related items that are often visited before or after the construct by the team.

A lab study of Team Tracks showed that the participants used and liked these features and were able to use them to better understand code. A field study shows ways in which it could be improved. Code that was often visited to fix bugs in it was not of interest to people not responsible for fixing these bugs. Moreover, programmers also wanted to explicitly filter related items by person and time.

7.4.8 Tagging

TagSEA [52] shows how the above limitations of Team Tracks can be addressed. Like Team Tracks, it can be used by a developer to identify important locations in a route through the program, which may be a "maintenance pattern" [52] so that other developers can easily take the same route. Thus, instead of trying to automatically deduce interesting routes, it requires developers to manually specify them. Developers can tag any construct using a shared structured tag name and description, which essentially identifies the route. TagSEA supports both the search and visualization approaches to finding tagged constructs. A developer can ask the system to show all constructs matching a tag/route. In addition, when a file is opened for editing, all tagged constructs are highlighted.

The general lesson to be learnt from TagSEA is that developers interested in finding some information can be helped not only by tools but also other developers. It would be useful to integrate the Team Tracks and TagSEA approaches by supporting semi-automatic identification of routes. For instance, a system could automatically tag constructs that are visited before or after a construct with the same name, and allow developers to later edit these tags.

7.5 Summary

We have taken above a tour of several novel collaborative software development concepts. The tour provides a high-level overview of the rationale and nature of these concepts. More important, it classifies these concepts based on several criteria, thereby providing an efficient taxonomy for describing the large range of research tools in which these concepts are implemented.

The "towards being there" virtual channels simulate physical channels available in face-to-face collaboration. These include light-weight communication channels such as ticker-tape, which support distributed "stream of consciousness babbling"; desktop sharing and multi-user programming environments, which supports distributed pair programming; and multi-user inspection/design tools, which support distributed synchronous inspection and design.

The "beyond being there" features offer automation that is useful even in face-to-face collaboration. Some of these make collaborators aware of events that would otherwise have to be communicated manually. Others allow them to share a single version using multiple flexibly coupled views. The last form of computer automation discussed here consists of helping developers locate some information of interest.

This taxonomy is a relatively superficial/high-level classification of collaborate software development concepts. It is possible to provider more detailed taxonomies such as the one given in [16] for conflict management. It would be useful to create detailed taxonomies for other features presented here such as view and model awareness and information visualization.

This chapter provides a basis for creating some of these more detailed taxonomies.

Acknowledgements This work was funded in part by NSF grants IIS 0312328, IIS 0712794, and IIS-0810861, and DARPA/RDECOM Contract N61339-04-C-0043. The comments of the reviewers improved the chapter.

References

1. Baheti P, Gehringer EF, Stotts PD (2002) Exploring the efficacy of distributed pair programming. Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods – XP/Agile Universe 2002 Springer-Verlag, pp. 208–220.
2. Barstow D (1987) Artificial intelligence and software engineering. Proceedings of the 9th International Conference on Software Engineering.
3. Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: A visual dashboard for fostering awareness in software teams. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.
4. Brooks F (1974) The mythical man-month. Datamation 20(12): 44–52.
5. Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM (2006) Identification of co-ordination requirements: Implications for the Design of collaboration and awareness tools. Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work ACM. Banff, Alberta, Canada, pp. 353–362.
6. Cheng LT, Hupfer S, Ross S, Patterson J (2003) Jazzing up eclipse with collaborative tools. Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange.
7. Cockburn A (2005) Crystal Clear: A Human-Powered Methodology for Small Teams. Boston, MA: Addison-Wesley.
8. Cook C, Irwin W, Churcher N (2005) A user evaluation of synchronous collaborative software engineering tools. Proceedings of the 12th Asia–Pacific Software Engineering Conference (APSEC'05), pp. 705–710.
9. Cubranic D, Murphy GC, Singer J, Booth KS (2004) Learning from project history: A case study for software development. Proceedings of the 2004 ACM Conference on Computer Supported Co-operative Work.
10. Curtis B, Krasner H, Iscoe N (1988) A field study of the software design process for large systems. Communications of the ACM 31(11): 1268–1287.
11. Curtis P (1992) Mudding: Social Phenomena in Text-Based Virtual Reality. Palo Alto, CA: Xerox Palo Alto Research Center.
12. DeLine R, Czerwinsky M, Robertson G (2005) Easing program comprehension by sharing navigation data. IEEE Symposium on Visual Languages and Human-Centric Computing.
13. de Souza, CRB, Redmiles D, Dourish P (2003) "Breaking the code," moving between private and public work in collaborative software development. Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, Sanibel Island, FL, USA.
14. de Souza, CRB, Redmiles D, Mark G, Penix J, Sierhuis M (2003) Management of interdependencies in collaborative software development. International Symposium on Empirical Software Engineering, ISESE'03.
15. Dewan P (2004) Collaborative applications. In: Singh M (Ed.) The Practical Handbook of Internet Computing, Vol. 5. London: Chapman & Hall, pp. 1–26.
16. Dewan P (2008) Dimensions of tools for detecting software conflicts. Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering ACM. Atlanta, Georgia, pp. 21–25.
17. Dewan P, Agarwal P, Shroff G, Hegde R (2009) Distributed side-by-side programming. ICSE Workshop on Co-operative and Human Aspects of Software Engineering (CHASE), IEEE, Vancouver.
18. Dewan P, Choudhary R (1995) Coupling the user interfaces of a multiuser program. ACM Transactions on Computer Human Interaction 2(1): 1–39.
19. Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. Proceedings of the 2007 Tenth European Conference on Computer-Supported Co-operative Work.
20. Dewan P, Riedl J (1993) Toward computer-supported concurrent software engineering. IEEE Computer 26(1): 17–27.

21. Doppke J, Heimbigner CD, Wolf AL (1998) Software process modeling and execution within virtual environments. ACM Transactions on Software Engineering and Methodology 7(1): 1–40.
22. Fitzpatrick G, Mansfield T, Kaplan S, Arnold D, Phelps T, Segall B (1999) Instrumenting and augmenting the workaday world with a generic notification service called Elvin. Proceedings of the Sixth European Conference on Computer Supported Co-operative Work.
23. Froehlich J, Dourish P (2004) Unifying artifacts and activities in a visual tool for distributed software development teams. International Conference on Software Engineering.
24. Gonzalez VM, Mark G (2004) "Constant, constant, multi-tasking craziness": Managing multiple working spheres. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.
25. Goodman GO, Abel MJ (1987) Communication and collaboration: Facilitating co-operative work through communication. Information Technology & People 3(2): 129–145.
26. Grinter RE (1995) Using a Configuration Management Tool to Co-ordinate Software Development. Proceedings of Conference on Organizational Computing Systems (COOCS'95).
27. Grinter RE (1998) Recomposition: Putting it all back together again. Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW'98).
28. Grudin J, Poltrock S (2005) Enterprise knowledge management and emerging technologies. Unpublished.
29. Gutwin AC, Penner R, Schneider K (2004) Group awareness in distributed software development. Proceedings of the 2004 ACM Conference on Computer Supported Co-operative Work.
30. Hegde R, Dewan P (2008) Connecting Programming Environments to Support Ad-Hoc Collaboration. Proceedings of the 23rd ACM/IEEE Conference on Automated Software Engineering.
31. Herbsleb JD, Atkins DL, Boyer DG, Handel M, Finholt TA (2002) Introducing instant messaging and chat in the workplace. Proceedings of the SIGCHI conference on Human factors in computing systems.
32. Herbsleb JD, Mockus A, Finholt TA, Grinter RE (2000) Distance, dependencies, and delay in a global collaboration. Proceedings of the ACM Conference on Computer Supported Cooperative Workshop.
33. Hill WC, Hollan JD (1992) Edit wear and read wear. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.
34. Hollan J, Stornetta S (1992) Beyond being there. Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems.
35. Jancke G, Venolia GD, Grudin J, Cadiz J, Gupta A (2001) Linking public spaces: Technical and social issues. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.
36. Lucia AD, Oliveto R, Tortora G (2008) IR-based traceability recovery processes: An empirical comparison of "one-shot" and incremental processes. Proceedings of 23rd IEEE/ACM Conference on Automated Software Engineering (ASE).
37. Mantei M, Backer RM, Sellen AJ, Buxton WAS, Milligan T, Wellman B (1991) Experiences in the use of a media space. Proceedings of CHI'91.
38. Mashayekhi V, Feulner C, Riedl J (1994) CAIS: Collaborative Software Inspection of Software. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.
39. Nawrocki J, Wojciehowski A (2001) Experimental evaluation of pair programming. European Software Control and Metrics, London.
40. Nawrocki JR, Jasinski M, Olek L, Lange B (2005) Pair programming vs. side-by-side programming. Software Process Improvement, Springer, Berlin/Heidelberg, pp. 28–38.

41. Olson JS, Olson GM, Storrosten M, Carter M (1992) How a group editor changes the character of a design meeting as well as its outcome. Proceedings of the ACM Conference on Computer Supported Co-operative Work.
42. O'Reilly C, Morrow P, Bustard D (2003) Improving conflict detection in optimistic concurrency control models. 11th International Workshop on Software Configuration Management.
43. Perry DE, Siy HP, Votta LG (2001) Parallel changes in large-scale software development: An observational case study. ACM TOSEM 10(3): 308–337.
44. Perry DE, Staudenmayer N, Votta LG (1994) People, organization, and process improvement. IEEE Software 11(4): 36–45.
45. Ratcliffe TL, Robertson A (2003) Code warriors and code-a-phobes: A study in attitude and pair programming. Proceedings of the SIGCSE Technical Symposium on Computer Science Education.
46. Sarma A, Bortis G, Hoek Avd (2007) Towards supporting awareness of indirect conflicts across software configuration management workspaces. Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, Georgia.
47. Sarma A, Maccherone L, Wagstrom P, Herbsleb J (2009) Tesseract: Interactive visual exploration of socio-technical relationships in software development. ICSE 09, Vancouver.
48. Sarma A, Noroozi Z, Hoek Avd (2003) Palantír: Raising awareness among configuration management workspaces. Proceedings of the 25th International Conference on Software Engineering.
49. Schroter A, Kwan I, Panjer LD, Damian D (2008) Chat to succeed. Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, ACM. Atlanta, Georgia, pp. 43–44.
50. Schummer T (2001) Lost and found in software space. Proceedings of the 34th Annual Hawaii International Conference on System Sciences, Hawaii.
51. Schummer T, Haake JM (2001) Supporting distributed software development by modes of collaboration. Proceedings of the Seventh European Conference on Computer Supported Co-operative Work.
52. Storey MA, Cheng LT, Bull I, Rigby P (2006) Shared waypoints and social tagging to support collaboration in software development. Proceedings of the 2006 20th Anniversary Conference on Computer Supported Co-operative Work ACM. Banff, Alberta, Canada, pp. 195–198.
53. Teasley S, Covi L, Krishnan MS, Olson JS (2000) How does radical collocation help a team succeed? Proceedings of ACM Conference on Computer Supported Cooperative Work.
54. Thummalapenta S, Xie T (2008) SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. Proceedings of 23rd IEEE/ACM Conference on Automated Software Engineering (ASE).
55. Williams L, McDowell C, Nagappan N, Fernald J, Werner L (2003) Building pair programming knowledge through a family of experiment. Proceedings of the International Symposium on Empirical Software Engineering, p. 143.
56. Xiang PF, Ying ATT, Cheng P, Dang YB, Ehrlich K, Helander ME, Matchen PM, Empere A, Tarr PL, Williams C, Yang SX (2008) Ensemble: A recommendation tool for promoting communication in software teams. Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering ACM. Atlanta, Georgia, pp. 1–1.
57. Zou L, Godfrey MW (2008) Understanding interaction differences between newcomer and expert programmers. Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering ACM. Atlanta, Georgia, pp. 26–29.