Journal of Universal Computer Science, vol. 15, no. 16 (2009), 3101-3125
submitted: 30/5/09, accepted: 29/9/09, appeared: 1/10/09 © *J.UCS*

Understanding Tools and Practices for Distributed Pair Programming

Till Schümmer
(FernUniversität in Hagen, Department for Mathematics and Computer Science, Cooperative Systems, Germany
till.schuemmer@fernuni-hagen.de)

Stephan Lukosch
(Delft University of Technology, Faculty of Technology, Policy, and Management, Systems Engineering Section, The Netherlands
s.g.lukosch@tudelft.nl)

Abstract: When considering the principles for eXtreme Programming, distributed eXtreme Programming, especially distributed pair programming, is a paradox predetermined to failure. However, global software development as well as the outsourcing of software development are integral parts of software projects. Hence, the support for distributed pair programming is still a challenging field for tool developers so that failure for distributed pair programming becomes less mandatory. In this paper, we analyse the social interaction in distributed pair programming and investigate how current technology supports this interaction. We present XPairtise, a plug-in for Eclipse that allows instant pair programming in distributed development teams. In addition, we report on experiences and findings when using XPairtise in a distributed software development setting.

Key Words: eXtreme Programming, distributed pair programming, patterns for computer-mediated interaction
Category: D.2.3, D.2.6, H.5.3

1 Introduction

Agile software development practices [Boehm and Turner, 2004], especially the eXtreme Programming [Beck, 1999] methodology, most importantly differ from other software development practices in the way how they address collaboration among participants. In the agile manifesto [Beck et al., 2001], the authors state 12 general principles that all highlight the importance of flexibility and collaboration. With respect to group interaction, principles 4, 5, 6, and 11 are most relevant:

> "(4) Business people and developers must work together daily throughout the project. (5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. (6) The most efficient and effective method of conveying

information to and within a development team is face-to-face conversation. (11) The best architectures, requirements, and designs emerge from self-organizing teams." [Beck et al., 2001]

Taking these principles seriously would imply that a distributed application of agile methods, especially the application of distributed eXtreme Programming (DXP), is a paradox predetermined to failure. In the same sense, global software development and outsourcing could not go together with agile approaches.

On the other hand, researchers have proposed several tools to better support distributed agile software development. The first notable publications that related distributed collaboration with agile methods were presented at the first international conference on eXtreme Programming. The *Team Streams* system [des Rivieres et al., 2001] provided support for asynchronous interaction in XP while the TUKAN system [Schümmer and Schümmer, 2001] had a focus on partner finding and synchronous interaction. Both of these tools mapped the social practices to groupware applications in order to improve the interaction between the participants. In the following years, additional tools were presented that again mapped social processes of XP to groupware solutions. These include tools for distributed pair programming and tools for better supporting the planning process in XP.

Eight years later, we still see the need for additional research on tools and processes for DXP, especially for solutions that extend the most obvious solution of providing a shared code editor. For that reason, we have revisited well-known assumptions for tool support in DXP [Schümmer and Schümmer, 2001] and extended these assumptions with novel interaction settings. These interaction settings focus on knowledge transfer and testing which are integral parts of most agile processes.

Our findings are presented in this paper: We first summarize the social practices of pair programming before we present XPairtise, yet another but different tool for distributed pair programming. We describe the interaction metaphors used in XPairtise and present first observations from a long term evaluation where two software development teams used XPairtise during a 6 month project. Our experiences show that XPairtise can be a valuable component in a DXP practitioner's tool suite and thus contribute to making DXP reality at the end.

2 The social practices of pair programming and its technology implications

In this section, we briefly summarize the interaction that takes place in pair programming, i.e. coordination, coding, communication, teaching, and testing. Our assumptions are based on findings reported in [Schümmer and Schümmer, 2001, Braithwaite and Joyce, 2006, Kircher et al., 2001]. As in [Schümmer and Schümmer, 2001], we take a look at the interaction between developers in a pair programming setting and discuss possible design alternatives for mapping this interaction to a computer-mediated setting by using a collaborative application. We make use of *design patterns for computer-mediated interaction (P4CMI)* [Schümmer and Lukosch, 2007] to describe the core design considerations. These patterns capture commonly used collaborative system design solutions and thus allow us to describe a hypothetical DXP system. We also use the patterns to compare existing solutions with the hypothetical solution by identifying the presence of patterns in the existing solutions. Summaries of the most important patterns will be given if the pattern is used in the design of XPairtise. More details on the individual patterns can be found on the CMI patterns repository web site at `http://www.cmi-patterns.org/`. Note that we will use Small Caps to identify a pattern name.

2.1 Coordination

The traditional setting: eXtreme Programming employs a lightweight planning metaphor using index cards as a main planning artefact. The use cases of the system under development are written down in the form of user stories on index cards (non-digital paper). In the planning game, these stories are discussed and sorted according to their importance. User stories are further decomposed to development tasks. Again, developers use index cards to store task details.

Before a pair programming session can start, a developer picks a task (from a set of shared task cards) and looks for a peer. In co-located settings, finding a partner is easy. The developer looks for other people who are currently finishing their tasks or work alone on other tasks. During daily planning meetings (the daily stand-up meetings), teams can be assigned for the day.

The computer-mediated setting: In distributed settings, both, the handling of story and task cards as well as the formation of pairs is much more challenging. Cards need to be stored in a light-weight planning environment. For group formation it is not as easy to detect the current status of remote users. Time shifts may make a stand-up meeting for coordination difficult or impossible. It is thus required that the developers become aware of one another, e.g., by having Activity Indicators, i.e., peripheral status views communicating the other users' current actions. Task cards, in addition, need to be available as shared objects, e.g., by organizing them in a Shared File Repository or by means of a planning wiki.

Activity Indicator
Problem: Users need time to perform a task but only the results are shared among them. In a collocated setting users are accustomed to perceive non-verbal signals such as movement or sounds when another user is active. If the users are distributed, these signals are missing. Users are therefore not aware of other users' activities, which can result in conflicting work or unnecessary delays.
Solution: Indicate other user's current activities in the user interface. To reduce interruptions, use a peripheral place or a visually unobtrusive indicator.

Shared File Repository
Problem: Users share intermediate results by passing files to one or more users. Ensuring that every user stays in the loop is error-prone.
Solution: Provide a Shared File Repository where users can place and retrieve files. Allow users to organize the files in folders.

2.2 Coding

The traditional setting: Once the team is formed, the team members sit together in front of the same screen and discuss a possible solution for the task. They maintain task awareness by placing the card next to the screen. One developer takes the role of a driver (the user having the keyboard) while the other user acts as a navigator. The navigator's task is to comment on the possible solutions for the task and check the quality and understandability of the created code.

The computer-mediated setting: In the distributed case, this should be supported with a Shared Editor. The editor should be part of the integrated development environment and automatically open for the navigator when the driver opens it on his screen (whenever they join the same Collaborative Session). The content of the driver's and the navigator's editor should be coupled so that both developers can see the same file at the same time (as described in the Shared Browsing pattern). The complexity of the shared editor can be reduced by following the Floor Control model proposed by XP. Since only one developer should have the keyboard at a time, this developer also controls the current scrolling and cursor position. Without Floor Control driver and navigator would be able to type at the same time (note that the users may then produce edit conflicts so that the solution would require mechanisms for Conflict Detection and means for resolving conflicting changes, e.g., using an Operational Transformation approach). However, synchronous editing blurs the Roles in the pair programming session, leading to two developers that lose a common focus.

> Shared Editor
> **Problem**: Users are sharing data for collaboration. The need to edit the shared data simultaneously emerges, but the shared single-user application (see Application Sharing) does not allow concurrent editing.
> **Solution**: Provide a shared editor in which users can manipulate the shared artifacts together. Ensure that state changes are instantly reflected in all other users' editors, and provide mechanisms that make users aware of each other.
>
> Collaborative Session
> **Problem**: Users need a shared context for synchronous collaboration. Computer-mediated environments are neither concrete nor visible, however. This makes it difficult to define a shared context and thereby plan synchronous collaboration.
> **Solution**: Model the context for synchronous collaboration as a shared session object. Visualize the session state and support users in starting, joining, leaving, and terminating the session. When users join a session, automatically start the necessary collaboration tools.
>
> Floor Control
> **Problem**: Synchronous interaction can lead to parallel and conflicting actions that confuse the interacting users and makes interaction difficult.
> **Solution**: Model the right to interact in the shared collaboration space by means of a token and only let the user holding the token modify or access the shared resources. Establish a fair group process for passing the token among interacting users.

One could think of using an Application Sharing approach for supporting this kind of interaction. The problem is that this takes reasonable bandwidth and that it is sometimes difficult to focus communication, which brings us to the next interaction in XP.

2.3 Communication

The traditional setting: Communication between developers and between developers and customers is the core of any agile method. By developing in pairs, communication naturally takes place between the developers. By changing partners frequently, knowledge is distributed epidemically. Communication is focussed by the shared display and gestures. Quick sketches on a sheet of paper can further support the communication. Having all developers in the same room allows other pairs to overhear conversations and thereby dynamically react to issues discussed in another pair.

The computer-mediated setting: For distributed interaction, communication poses the biggest problem in agile methods. On one hand, we would benefit from a media-rich communication channel, such as a video channel. On the other hand, the communication channel should not consume too much network bandwidth, be stable enough, and establishing connections needs to be quick and easy.

The simplest communication means would be an EMBEDDED CHAT. In addition, scribbles could be drawn in a synchronous graphical SHARED EDITOR and gestures could be conveyed in forms of REMOTE SELECTIONS in the source code that allow the navigator to point out relevant sections in the code. These kinds of communication have the advantage that they can be easily kept persistent and thereby enrich the comments of the discussed software artifacts.

> EMBEDDED CHAT
> **Problem**: Users need to communicate. They are used to sending electronic mail. But since e-mail is asynchronous by nature, it is often too slow to resolve issues that arise in synchronous collaboration.
> **Solution**: Integrate a tool for quick synchronous interaction into your cooperative application. Let users send short text messages, distribute these messages to all other group members immediately, and display these messages at each group member's site.

> REMOTE SELECTION
> **Problem**: Users select artifacts to start an action on the artifact. Selecting an artifact is considered as taking the artifact under personal control. Whenever two users select the same artifacts, this leads to coordination problems.
> **Solution**: Show remote users' selections to a local user. Make sure that other users who are interested in a specific artifact are aware of all distributed co-workers who have selected the object.

The disadvantage of textual communication for DXP is that the developers need their hands to produce code. Normally, coding and talking goes hand in hand. Thus, we expect that textual chats will not be the most important communication medium. As an alternative or addition to the communication functionality an audio channel can be embedded. An audio channel supports parallel communication and coding. However, audio channels are typically transient and even recorded communication is not easily accessible. The latter becomes important when communication logs should be used for teaching.

2.4 Teaching

The traditional setting: The intensive communication fosters peer-to-pear learning. By pairing a strong developer with a novice, the novice will learn best practices and gradually take more responsibilities. The expert will learn by making his knowledge explicit. We propose to extend this model for classroom education where an additional number of students can participate as spectators (following the Show Programming pattern of [Schmolitzky, 2008]).

The computer-mediated setting: Supporting the Mentor interaction in distributed settings may be much easier. Due to the fact of global distribution, the opportunity for finding an expert for a specific problem domain may increase. An almost unlimited number of Spectators [Lukosch and Schümmer, 2008] can be added to the application by distributing the actions of driver to all of them. Since learning becomes more effective when learners actively interact with the subject, we envision that Spectators become active learners who comment and analyze the activities of the driver and the navigator. The computer-mediated setting allows parallel communication channels for the audience and the pair of developers (multiple Embedded Chats). The developers can perform their pair interaction and communication and the audience can in parallel perform a meta discussion (a comparable interaction has been applied in several scientific conferences where the presentation was complemented with a textual chat channel, e.g. in [Rekimoto et al., 1998]).

Mentor
Problem: Newcomers do not know how community members normally act in specific situations. They are not used to practices that are frequently applied in the community.
Solution: Pair newcomers with experienced group members who act as mentors. Initially let newcomers observe their mentors, and gradually shift control to the newcomer.

Spectator
Problem: Users are interacting in a computer-mediated environment but are not familiar with the environment. These users perform activities which disturb the interaction and collaboration of other users.
Solution: Allow users to view and follow the interaction in an ongoing Collaborative Session as Spectator. Ensure that these Spectators cannot influence the interaction.

Capturing the interaction would further allow that students Replay interesting pair-programming episodes and thereby better understand the evolution of software artifacts (this is an argument why textual communication may be more suitable than audio in some cases).

Replay
Problem: When users join an ongoing collaboration as latecomers or when users rejoin a collaboration after a time of absence, it is hard for them to understand how the current state of the collaboration has been reached, or what has changed since their last participation, by only perceiving the current state of the collaboration.
Solution: Capture all changes to the shared objects used in the collaboration in an Activity Log. When users join or rejoin a collaboration, replay the captured changes to show them how the current state of the collaboration has been reached.

2.5 Testing

The traditional setting: eXtreme Programming advocates a test-first approach. This means that no code is written as long as no test fails. In a co-located setting, developers first think about how to test a feature that is requested in a specific task. They then create test code that tests the functionality of the feature. This test is executed by a test automation tool such as jUnit [Beck and Gamma, 2002]. Usually the test fails since the feature is not yet implemented. In a next step, the developers create code that fixes the broken test.

Driver and navigator use debugging tools that allow stepwise execution of the developed software. Additionally, they can inspect and modify variables and provide input values for the software.

The computer-mediated setting: This practice shows that it is not sufficient to have a Shared Editor when considering tool support for distributed XP. Instead, the developers need support for collaborative execution of tests. In a first approach, the system would create Distributed Commands for triggering unit tests. This would allow the developers to execute the tests locally at their machines and inspect the results. In a next step, the system would allow coupled debugging including collaborative inspectors of application data and collaborative stepwise execution of a program. To enable collaborative use of debugging tools, one could capture and replicate the commands performed by the driver to control the debugger (Distributed Command). Additionally, break points can be modeled as shared objects and views of the variables could be shared. The challenge with such an approach is to keep both client machines (or even more in a setting with spectators) synchronized.

> DISTRIBUTED COMMAND
> **Problem**: Clients can apply changes to replicated objects locally. When you distribute the new versions of locally changed replicated objects, you might distribute more information than is necessary to keep the other replicas consistent, especially if only a small part of the replicated object has changed. This increases the network load and the response time of your application unnecessarily.
> **Solution**: Capture the method calls that a client uses to manipulate their local replicas as COMMANDS [Gamma et al., 1995]. Distribute the captured COMMANDS to all other clients that also maintain replicas via the network. Let these clients re-execute the COMMANDS on their replica.

APPLICATION SHARING may ease the technical problems at this stage. However, the application will not be collaboration aware, which makes it again difficult to, e.g., point at specific artifacts.

3 Related work

As briefly mentioned in the introduction, existing approaches for supporting distributed pair programming either use an APPLICATION SHARING approach to enhance an existing tool suite or provide customized tools that include various groupware features such as SHARED EDITORS or SHARED BROWSING support. As Hanks [Hanks, 2005] pointed out, customized groupware tools often do "not provide all of the features used by a particular software developer" and thus "limit her ability to successfully accomplish her work." On the other hand, APPLICATION SHARING solutions lack process support and are thus not collaboration aware. Examples for systems with an APPLICATION SHARING APPROACH are JAZZ and MILOS.

The JAZZ system [Hupfer et al., 2004] is an extension of eclipse that supports the whole XP life cycle. Its main focus lies on supporting the workflows in asynchronous interaction. With respect to synchronous interaction, users can stay aware of co-workers and initiate chat sessions with co-workers who are logged in at the same time. Using an INTERACTIVE USER INFO, available users can also be invited to a synchronous pair programming session using an APPLICATION SHARING system. JAZZ can be complemented with a plugin for SHARED EDITING (e.g., the DocShare (`http://wiki.eclipse.org/DocShare_Plugin`) plugin that provides a synchronous shared code editor based on OPERATIONAL TRANSFORMATIONS). The problem with this plugin is that it is not integrated into the workflow of pair programming. Especially, it does not provide awareness and has no explicit notion of roles.

MILOS [Maurer, 2002] aims at supporting the coordination between software developers in an XP team. As in JAZZ, MILOS provides awareness of co-present users and allows users to initiate pair programming sessions using Application Sharing. Both JAZZ and MILOS make use of existing IDEs and thereby provide the full functionality that developers know from single-user development environments.

Some research prototypes have approached this gap by creating special purpose groupware for all phases of the eXtreme Programming process. Examples for such an approach are TUKAN and Moomba. Providing awareness is one of the central aspects of the TUKAN system [Schümmer and Schümmer, 2001]. The main focus lays on partner finding for pair programming. Developers working on related artifacts are identified and the system proposes them to create a pair for distributed pair programming. Shared Editors are provided for manipulating code together. Users can highlight important code using a Remote Selection. Unfortunately, TUKAN was built as an extension of a relatively unpopular development environment, namely ENVY for VisualWorks Smalltalk. This is one of the reasons why it has not gained high popularity.

Moomba [Reeves and Zhu, 2004] extends the awareness model of TUKAN and translates it to a Java environment. Developers are made aware of other developers who work on related tasks. Once they decide to join closer collaboration, they can launch a collaborative Java IDE where the developers can use a Shared Editor. Although Moomba supports Java development, it is still built as a proprietary tool and thereby can not provide the same domain-specific tool support as it is present in modern IDEs.

Solutions that combine the two approaches mentioned above extend professional IDEs with collaboration facilities so that they become collaboration aware. Examples for IDEs that are extended this way are TogetherJ and Eclipse. Cook [Cook, 2006] created the CAISE architecture to allow users of the Together Architect for Java to share different tools of the IDE. Unfortunately, this IDE does not propagate key-strokes to the CAISE system which has the effect that code changes can only be shared on a per save basis. Eclipse is a more open environment that allows closer coupling of the developers' IDEs. Thus, we will concentrate on plugins for making Eclipse collaboration aware in the remaining part of this section.

The Eclipse Communication Framework (ECF - `http://www.eclipse.org/ecf`) aims at integrating a collaboration infrastructure with the IDE. This allows users of the IDEs to collaboratively work on the same set of files in real time. All changes are combined by using Operational Transformations so that all collaborators keep a consistent state. However, both solutions do not address the collaboration process: Users can, e.g., type at the same point in time or view different documents. This weakens the strict separation of roles as it was present in XP.

Coordination work can be integrated into Eclipse in two ways: Traditional web-based systems can be shown inside the internal browser window or special-purpose planning plugins can be added to the Eclipse workbench. An example for the first approach is to integrate a planning Wiki (e.g., XP-Swiki [Pinna et al., 2003], which was also integrated into the IntelliJ IDE before). An example for a tighter integration is the use of the Mylyn plugin (`http://www.eclipse.org/mylyn/`). This plugin interacts with web-based project management systems such as SourceForge (`http://sourceforge.net/`) and adds views and editors for all planning artifacts from this external system to the Eclipse workbench.

In addition to ECF's Shared Editor, there are some additional collaborative code editors for Eclipse that integrate distributed editing in the context of the IDE. Besides the DocShare plugin mentioned above, we have analyzed three other solutions. The oldest plug-in that we are aware of is the Sangam system [Ho et al., 2004]. It allows developers to couple editors. Commands are replicated between the editors so that all connected developers can see and edit the same code. In our tests, we were able to create inconsistencies between the different editor instances. This means that the developers could end up with different data in their editors.

The Saros plugin [Djemili, 2006] supports driver-navigator interaction in Eclipse. Once users decide to start a distributed pair programming session, the system synchronizes the code base of both developers and provides awareness on files that are opened at the driver's site. A Shared Editor allows collaborative code creation and Remote Selections allow the navigator to point at relevant pieces of code. Saros is available under an open source licence at `http://dpp.sourceforge.net/`.

Finally, the XecliP plugin for Eclipse provides to a large extent a comparable functionality as the XPairtise system that we will present in the next chapter. The reason for this is that XecliP was developed in competition in another subteam of our research group. The developers had the same goals as those who developed XPairtise. However, there are slight differences with respect to project sharing, where XPairtise provided the simpler solution. This is the reason why we present XPairtise in this paper. More information on XecliP can be found on its project home page at SourceForge: `http://xeclip.sourceforge.net/`.

Debugging support is available in several distributed development tools. One of the oldest references is the FLECSE system, that supports users in stepwise execution of text-based software [Dewan and Riedl, 1993]. More recently, Moomba [Reeves and Zhu, 2004] allows developers to share the textual output of a program and to collaboratively execute jUnit tests using a Distributed Command approach. The Jazz system [Hupfer et al., 2004] uses an Application Sharing approach for supporting shared debugging. We are not aware of any system that allows loosely coupled interaction in the debugging context (e.g., independent exploration of variables). This is still an open research problem.

In summary, we can observe a trend to better integrate support for Shared Editing in professional IDEs. Not surprisingly, the Eclipse IDE becomes more popular for such developments. However, we still see the need for better support of the interaction, especially with respect to the roles in distributed pair programming. None of the tools explicitly addresses learner interaction. Saros seems to be one of the most promising plug-ins so far, but even this plug-in lacks a sophisticated role support.

4 Approach

In this section, we present XPairtise, our approach for supporting distributed pair programming. XPairtise is an Eclipse plugin that offers shared editing, project synchronization, shared program and test execution, user management, built-in chat communication, and a shared whiteboard.

Figure 1 shows two instances of the XPairtise plugin for Eclipse. In the following, we present the functional and user interface properties of XPairtise in more detail and relate them to the identified social practices.

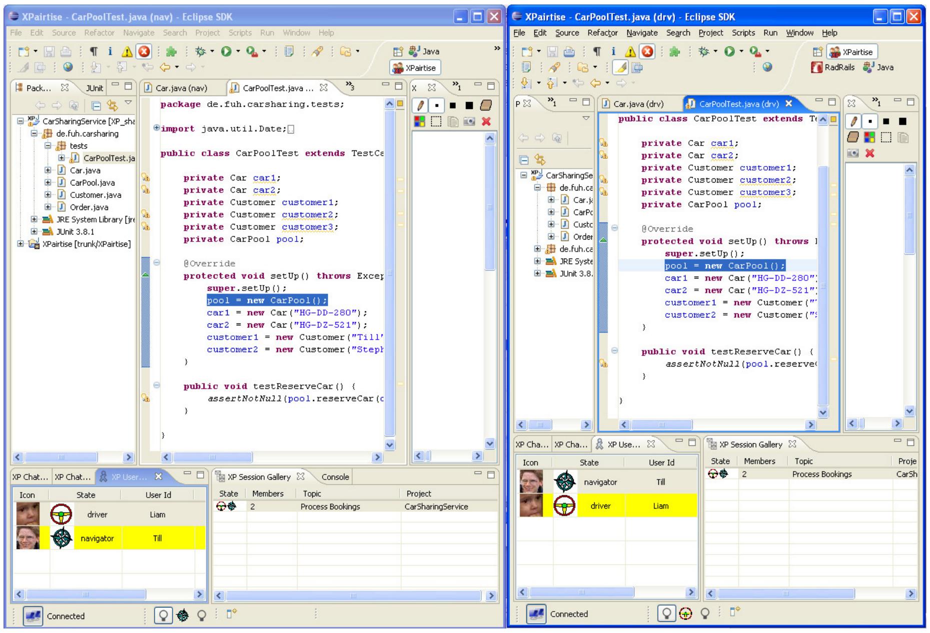

Figure 1: XPairtise

4.1 Coordination

Story card management is not part of the XPairtise system. The reason for that is that we decided to integrate a traditional web-based solution for capturing and editing the user stories and the task cards and use eclipse's embedded web browser to access the cards. In our setting, we used the CURE wiki [Haake et al., 2005] that provides templates for story cards and thus helps to ensure that all required information for a story card is present.

Concrete distributed pair programming sessions are modeled as Collaborative Sessions in XPairtise. When users feel the need for a pair programming session, they create a new collaborative session in Interaction Directory that is visible for all other XPairtise users (see Figure 2.1). Note that we decided to relax the rule of eXtreme Programming that said that every code should be created in pairs. This would be very problematic in distributed setting, especially in a global team. It is thus up to the developer to decide whether or not the task requires pair programming. If not, distributed developers can use the same tools and act as the driver. Others will still stay aware of such single-user sessions and can inspect changes afterwards as if it was a team session.

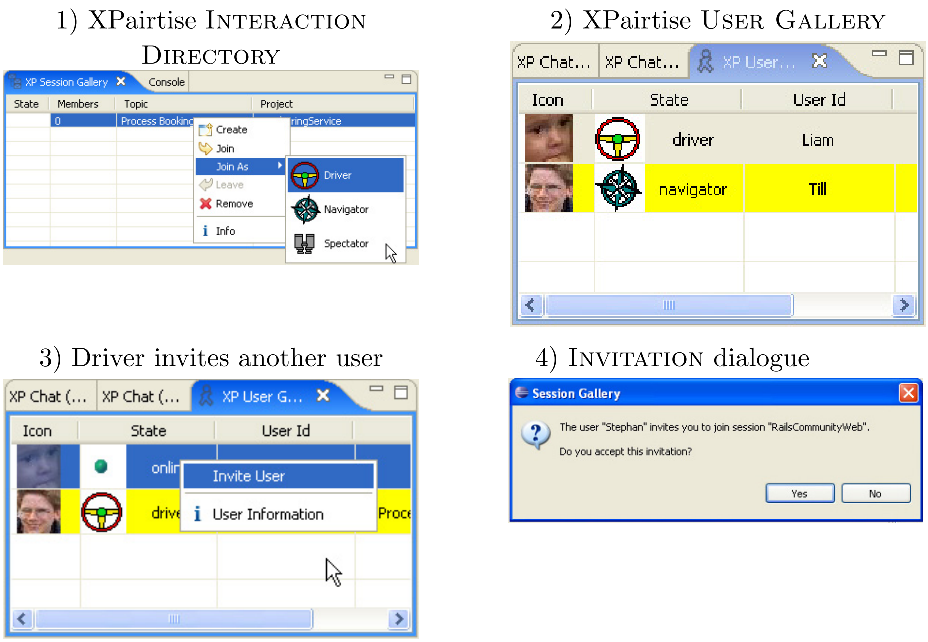

Figure 2: Setting up a distributed pair programming session with XPairtise

Collaborative Sessions have a name (typically the name of the task card) and an Eclipse project that is going to be shared via the proprietary XPairtise server. Once such a session has been created, it is listed in the Interaction Directory (see Figure 2-1). Each user who is currently connected to the XPairtise server can now join the session. When joining the session users can decide to join as navigator or driver. Of course, this is only possible when no other user has joined with the selected role so far. Thereby, users can decide to start a session in the role which from their perspective fits best to the task that as to be accomplished. Users might, e.g., decide to join in the role of a navigator and then invite someone else as driver because they are aware of the importance of the task but do not feel confident enough to take the lead to solve it. As shown in Figure 2, XPairtise offers the necessary means to create, invite and join sessions and no additionally support is necessary.

Users can browse for other users who are connected with the XPairtise server via the User Gallery (see Figure 2-2). This view includes the remote users' current Availability status and thereby eases the selection of an appropriate partner. By sending an Invitation they can invite another user to the pair programming session (see Figure 2-3). This opens a request at the invited user's site and the user can decide whether to join or not (see Figure 2-4).

The local workspace of a joining user is stored and the project for the distributed pair programming session is retrieved from the XPairtise server. This ensures that driver as well as navigator have synchronized workspaces when starting the session. Additionally, this approach makes XPairtise independent from code repositories like CVS or SubVersion and allows to establish ad-hoc distributed pair programming sessions.

4.2 Coding

Once a distributed pair programming session is established, driver and navigator can cooperate in a Shared Editor (see Figure 3). All actions of the driver are also performed at the navigator's site. This, e.g., includes opening source files, scrolling the window, marking text, moving the text cursor, highlighting lines, editing text, as well as refactoring source code. Thereby, the navigator is constantly aware of the changes that are performed by the driver.

Like in traditional XP settings, the navigator cannot change the code. To further support collaboration awareness, we integrated a Remote Cursor. This cursor shows the current working location of the driver. To foster collaboration and simplify coordination of driver and navigator, we additionally integrated a Remote Selection. The selections of the driver are also shown in the navigator's editor and vice versa. To avoid conflicting selections, the navigator's selection does not affect the clipboard content, as it only intended to highlight specific code fragment for coordination and communication purposes (see below). Such highlighting functionality avoids that driver and navigator talk about different code fragments.

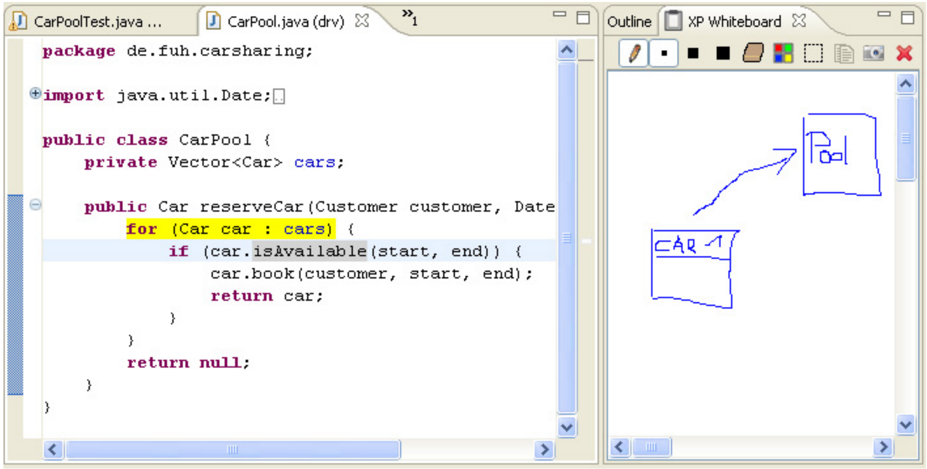

Figure 3: Shared Editor in XPairtise

In co-located settings, driver and navigator switch roles by passing the keyboard among each other. To reflect this in a distributed setting, XPairtise makes use of a Floor Control technique. Driver as well as navigator can request to switch roles (see Figure 4.1) by pressing a role change button. This request is highlighted in the user interface at the other user's site (see Figure 4.2). A role change cannot be forced. It only takes place when the other agrees by also pressing the role change button.

1) Role change request 2) Request notification

Figure 4: Changing roles in XPairtise

4.3 Communication

XPairtise supports multiple communication channels: driver and navigator can use the integrated shared whiteboard and a graphical Shared Editor (see right part of Figure 3) to exchange ideas. They can use the Embedded Chat for textual communication within a session as well as globally with all users currently logged in to the XPairtise server. The different chats can be accessed via different tabs (cf. Figure 5). Furthermore, XPairtise offers an integrated Skype control to establish audio connections. As mentioned above, the shared editor also supports REMOTE SELECTIONS. This allows the navigator to raise the driver's attention to specific parts of the source code and thereby start and focus communication.

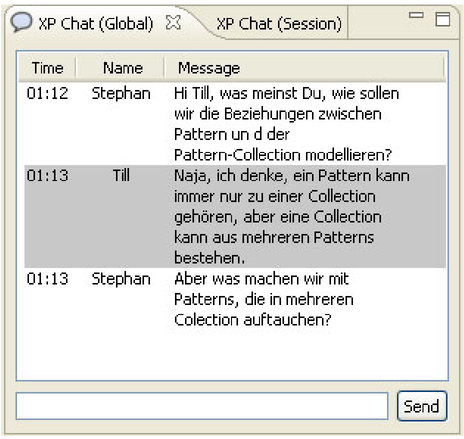

Figure 5: Embedded Chat in XPairtise

4.4 Teaching

With the above functionality, XPairtise already enables distributed pair programming. As it is also possible to create ad-hoc distributed sessions via the XPairtise server without the need of project-specific code repositories, a novice can easily invite an expert to a pair programming session. The expert can then act as a MENTOR and teach the novice on how to solve current problems.

To widen the audience of a pair programming session, XPairtise furthermore supports the additional role of a SPECTATOR. Users who join an ongoing pair programming session as SPECTATOR can watch the interaction among the driver and the navigator. For that purpose, XPairtise also retrieves the project of the session from the XPairtise server. When a SPECTATOR joins as a latecomer and the driver already performed some changes, XPairtise still ensures that the workspaces are synchronized.

SPECTATORS cannot change the code nor can they select or highlight anything in the editor. Still, they are allowed to participate in the session chat and thereby can, e.g., ask questions for clarification or highlight possible problems in the shown code. Like in traditional settings, the driver and navigator have to decide when and how they are going to address the posted messages. Thereby, XPairtise can easily be used to teach a group of novices in a specific problem domain when the driver is an expert in that domain. Additionally, this also allows to teach distributed pair programming, when novices join an ongoing pair programming session among two experts in eXtreme Programming.

4.5 Testing

When the driver performs run actions or starts tests, these are started at the navigator's site as well. Thereby, XPairtise enables basic collaborative testing.

However, since testing is more than the execution of JUnit tests, we have recently added an XPairtise extension that supports collaborative debugging. Break points are modelled as shared objects as well allowing all participants to stop the program under test at the same place. In the same way as editor inputs were shared among the members of a collaborative session, the use of the Eclipse debugger can also be coupled: Eclipse commands at the driver's client such as stepwise execution of code or inspection of variables are monitored by the XPairtise plugin and distributed to all other clients. Floor Control is an important issue here again since it needs to be ensured that only one client at a time is able to continue the execution of the program under test.

The main reason why our debugging support for XPairtise is not yet part of the official open-source release is that we are still working on synchronizing the effects of external influences on the execution of the program under test. This includes that all input (e.g., files, streams, mouse movements or hardware signals like timer values) used by the tested program needs to be identical to ensure the same execution. To what extent this can be solved is still an open issue.

4.6 Implementation of XPairtise

XPairtise makes use of a client server architecture. On a technical level, XPairtise clients communicate with the XPairtise server using a JMS infrastructure (Java Messaging Service), namely the ActiveMQ messaging server (see Figure 6).

During the bootstrap phase of the XPairtise infrastructure, the XPairtise server connects to the message bus and creates a message channel that allows clients to request information on shared objects stored in the HSQL database that acts as XPairtise's object repository (Centralized Objects).

Whenever clients register by sending a registration message to the server's message queue, the XPairtise server creates a message queue for the client. The client subscribes to this queue and from then on receives updates on changes to

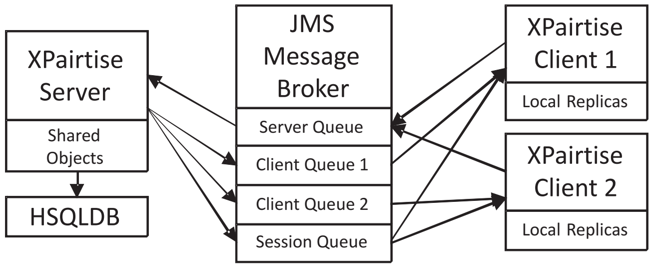

Figure 6: XPairtise conceptual system architecture

shared objects (Remote Subscription). To accommodate latecomers, XPairtise uses a State Transfer approach, i.e. the initial state is transferred to the latecomer and later on only updates have to be applied.

Collaborative Sessions are also modelled as shared objects. In addition, each collaborative session has a message queue to which the server can add updates needed by all participants in the session. These updates are either sent as state updates (e.g., when users add drawings to the shared whiteboard) or as Distributed Commands. When a client, e.g., changes the selection in the Shared Editor of Eclipse, the XPairtise plugin captures the selection command and sends this to the server. The server in turn adds the command to the session's message queue so that it is received by all members of the Collaborative Session. Each client executes the selection command locally with the effect that all clients can see the Remote Selection.

Due to the XPairtise system architecture and the used approach for latecomer accommodation, XPairtise can easily deal with connection losses. Users that loose connection simply re-join a session and thereby receive the most recent session state.

5 Experiences

XPairtise was developed in our 2006/07 lab course on cooperative system development. As our university is a distance teaching university, the team members only met at the beginning and the end of the lab course. In the meantime, the team members collaborate at a distance. Once the team had a first running prototype, the team used XPairtise for distributed pair programming. This allowed the team to identify problems early in the development cycle and address such problems directly. At the end of the lab course, all team members reported that XPairtise simplified their collaboration a lot. Since then, XPairtise is available at sourceforge.net.

5.1 Hypotheses and setting

While the first informal evaluation gave us anecdotal evidence that XPairtise can be used for distributed pair programming, a more detailed evaluation was needed to better understand the tool's contribution to the pair programming interaction, especially with respect to the individual users' contributions. Thus, a second evaluation was performed in our 2007/08 lab course. Besides general observations of distributed pair programming, we wanted to test the following hypotheses:

1. Students with comparable skills will make equal contributions to the pair programming sessions, which means that they will have frequent role switches.

2. In a pair with different experience levels, the experienced partner will use XPairtise for training less experienced students. The experienced partner will keep the driver role throughout the session.

3. In both constellations, driver and navigator will frequently use the REMOTE SELECTION to focus the discussion on a specific position in the source file.

We observed two teams of 5-6 students for a period of 18 weeks. During that time, we recorded the JMS messages exchanged in 52 XPairtise sessions. This allowed us to make a detailed analysis on actions performed in the sessions. In addition, we recorded the audio communication in seven (randomly selected) sessions where XPairtise was used. In order to relate the level of participation in XPairtise to the individual's participation in the software project, we also logged the interaction with the version management system (CVS). We correlated log entries of the CVS system with activities performed in XPairtise Sessions which allowed us to better understand the role of collaboration in distributed eXtreme Programming. Again, the results from the CVS analysis are not statistically significant but they can still point our analysis to interesting differences between the two teams. Finally, we conducted semi-structured interviews in order to get feedback on the perceived usefulness of XPairtise.

5.2 Results

Based on this observation, we report first anecdotal results in relation to the social practices presented in the previous sections. In total, we logged about 80 hours in which XPairtise was used. Users performed 80.000 commands that led to change notifications to other clients. They created 5514 versions of classes (CVS check-ins). The two groups differed significantly in their tool use: The first group used XPairtise five times as long as the second group. They created ten times as many editor events as the second group. On the opposite, the second group performed 31% more check-ins in the CVS system. The check-ins of the second group were in most cases the result of isolated work without XPairtise (98% of all check-ins). The first group showed a different behaviour: 28% of the check-ins were performed during a pair-programming session. These differences suggest that the first group interacted in a very synchronous way while the second group distributed tasks and solved the tasks asynchronously without much pair programming.

5.2.1 Coordination

The observed groups created between 120 and 250 planning cards. The cards were stored in the group's project wiki. Interestingly, the group members made heavy use of the cards in the early planning phases of the project but did not maintain the cards in later implementation phases. This raised the problem that some group members lost track of the project's progress due to the lack of awareness information (CHANGE WARNINGS). Shortly before the end of the project, both groups revisited the planning cards and marked them as finished.

The existence of the sessions is already an indication that users were able to meet in COLLABORATIVE SESSIONS or join existing sessions as spectators (in 17 of 51 sessions).

5.2.2 Coding

In all 52 observed sessions, code was changed. The shared editor was used as expected. Surprisingly, we could observe a less agile interaction between driver and navigator. While in co-located pair programming sessions, role switches are expected to happen every 20 minutes [Hanks, 2004], we could observe only 21 of 52 sessions where a role change took place at all. And even where a role change took place, there were only 4 sessions where the navigator was active for more than 30% of the time. An ideal pair programming session would have frequent role switches and lead to an equal participation of both partners. This could not be confirmed in our observations. Even the first group that showed a very cooperative work style did not change roles often (on average 2.2 role switches per session). One reason for this was that in many cases experienced developers interacted with novice navigators (see section on teaching below). Even when considering only those sessions where both students had comparable background knowledge, we can not observe the same frequency of role changes as in co-located settings [Hanks, 2004]. Only one out of 17 sessions showed a relatively large number of role switches (8 role switches in 150 minutes). This gives us at least an indication that our first hypothesis was not confirmed in our observational setting.

5.2.3 Communication

When designing XPairtise, we expected that audio communication would be the most prominent communication channel but that the Embedded Chat would also be used frequently. However, there was almost no use of the text-based chat. Only 9 of 52 sessions had any chat entries in the session chat. Even fewer sessions had entries in the global chat that was intended as a meta communication channel. Using chat logs for augmenting code comments would thus not be possible in the observed groups.

Only 4 sessions utilized the whiteboard. These sessions were not used for pair programming but for creating sketches for the final project presentation. The interviews did not provide any further clues why the whiteboard was not used more frequently. Actually, students reported that they liked the feature of the whiteboard, which is in contrast to the log data that shows that the students did not use the whiteboard frequently.

The analysis of dialogues in the audio communication channel gave us more information regarding situations where the navigator did not perform actions in XPairtise. The developers frequently searched the web for missing information. At the same time, the driver continued coding. Once the navigator found the required information, he informed the driver who then used the information and modified the code.

The Remote Selection was used in all sessions. In all but 9 sessions, the navigator also selected code to focus the communication on a specific part of the code. However, there were much fewer occasions of remote selections than expected. In average, the ratio between driver selections and navigator selections was 92 to 8. The interviews on the other hand indicated that the users perceived the remote selections as a very important feature of XPairtise. The third hypothesis thus could be confirmed for the driver but not for the navigator. Although the navigator considered the remote selection as useful for him, he did not use it to focus the driver's attention on a specific part of the code.

5.2.4 Teaching

To our surprise, there were fewer than expected sessions with Spectators (17 of 52 sessions). In most of these cases, there was exactly one Spectator (12 cases) who joined the XPairtise session for a short time. From the developers' feedback, we can say that in some cases, the guest was an expert who joined the session with the goal of explaining a specific part of the code. Instead of the driver educating the Spectators, these were directing the driver in this case.

In addition to these sessions, we could observe teaching to a large extent: Many pair-programming sessions brought together expert developers as drivers with novices as navigators (approx. 40% of the sessions). In these settings, there were almost no role switches (only 36% of all observed sessions included a role switch). The driver kept his role throughout the session (between 1 and 2 hours). In many cases (especially in the second group), the navigator stayed passive throughout the whole session. This confirms our second hypothesis that assumed that the novice will not take the role of the driver.

Looking at the audio logs of these sessions, we could observe that the driver was speaking much more than the navigator (in four of the seven observed sessions, the driver talked more than 85% of the time). All these observations indicate that the driver was presenting his code to a rather passive navigator. However, especially the inexperienced developers who participated as navigators in these sessions reported that observing the expert was very helpful for them.

5.2.5 Testing

Since the observed version of XPairtise only provided collaborative unit test execution, we cannot provide proofs on the usability of further testing support. Interestingly, many developers reported that they would prefer testing alone. But some developers also highlighted the expectations that they would have for a stable testing support. One participant mentioned that debugging

> "would be an excellent feature because you would learn a lot. Since you would know that your partner reads the same code, you know that he will see bugs that you simply miss because you read the code in a different way. In your head, you have an idea of what code should be present although it isn't there. Your partner does not have the same image in his head and he may see it that you are writing rubbish."

This quote shows that at least some of the students were able to imagine the benefits of collaborative test execution.

5.3 Summary

The above evaluation shows that XPairtise supports the social practices for distributed pair programming. However, compared to pair programming in a traditional setting, our observations highlight some interesting differences. For coding, it is interesting to note that role switches did not occur as often as expected from a traditional setting (in contrast to hypothesis one. When considering communication, the EMBEDDED CHAT was used less than expected and almost all communication was handled via an audio channel. It is interesting to note that the whiteboard as well as the REMOTE SELECTION feature were rarely used but still considered as an important feature by the users which makes us confident that the REMOTE SELECTION should be an important feature in all distributed pair-programming tools.

Concerning teaching, we also expected a different behavior. Instead of laymen in the role of the SPECTATOR, experts were using this role. Still, this feature enabled the transfer of knowledge which is the basic task of teaching. Teaching did – as expected in hypothesis two – not press the learning navigator into the more active role of the driver. Finally, when considering testing, the basic support was not used at all. But, this is mainly due to the fact that developers preferred testing on their own as well as developers not being experienced in testing at all.

6 Conclusions

In this paper, we have discussed the main social practices for distributed pair programming, i.e. coordination, coding, communication, teaching, and testing. We analyzed the technology implications when transferring these practices to distributed settings and provided guidance for developing technology support. We have discussed to what extent existing systems support these social practices and presented a tool that integrates support for the practices in Eclipse.

First experiences, when using our tool during its development and during two long term development projects indicated that XPairtise supports the social practices for distributed pair programming. The evaluation revealed some interesting aspects on the tool usage. The evaluation of XPairtise during the lab course has of course a different context from the real. Students in a lab course that uses a blended setting cannot be exactly compared to professionals developing a product which has to be delivered to a real customer with scheduling and deadlines to be followed. Still, the contexts are similar and this initial evaluation provides insights on how XPairtise supports distributed pair programming. Nevertheless, we plan to continue the evaluation of XPairtise in further lab courses and especially in commercial distributed development projects. In these evaluations, we aim to better understand the impact of the plugin on distributed team performance and to identify additional functionality to further support distributed pair programming.

Independent from these findings, we will in the next future improve the functionality for pair formation. For that purpose, we will include ACTIVITY INDICATORS or EXPERT FINDER mechanisms as they were, e.g., present in TUKAN [Schümmer and Schümmer, 2001] that allow to retrieve experts from the registered users for specific problem domains. This retrieval could, e.g., be based on source code analysis or activity analysis of pair programming session. For improving the teaching functionality, we plan to include a REPLAY mechanism which records pair programming session and allows to review them afterwards.

Acknowledgements

Special thanks are due to the members of the lab group who developed the initial version of XPairtise (in alphabetical order): Karsten Juschus, Timo Kanera, Manfred Krapf, Nicolas Lavit-Justen, Robert Mischke, Heiko Schlenker, Roland Wielnig. The implementation of the debugging support for XPairtise was provided by Manfred Krapf. Additional repository support was added by Timo Kanera. Dominik Kröger substantially contributed to the evaluation of XPairtise. Last but not least, we thank all students who used XPairtise during our 2007/08 lab course and thereby helped us to study the application of XPairtise in distributed development projects.

References

[Beck, 1999] Beck, K. (1999). *eXtreme Programming Explained.* Addison Wesley, Reading, MA, USA.

[Beck et al., 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development.

[Beck and Gamma, 2002] Beck, K. and Gamma, E. (2002). Junit cookbook.

[Boehm and Turner, 2004] Boehm, B. and Turner, R. (2004). *Balancing Agility and Discipline – A Guide for the Perplexed.* Addison Wesley, Boston, MA.

[Braithwaite and Joyce, 2006] Braithwaite, K. and Joyce, T. (2006). Xp expanded: Patterns for distributed extreme programming. In Longshaw, A. and Zdun, U., editors, *Proceedings of the 10th European Conference on Pattern Languages of Programs, EuroPLoP'05*, pages 337–345, Konstanz, Germany. UVK.

[Cook, 2006] Cook, C. (2006). *Towards Computer-Supported Collaborative Software Engineering.* PhD thesis, University of Canterbury, Christchurch, New Zealand.

[des Rivieres et al., 2001] des Rivieres, J., Gamma, E., Mätzel, K.-U., Moore, I., Weinand, A., and Wiegand, J. (2001). *Extreme Programming Examined*, chapter Team Streams: Extreme Team Support, pages 333–353. Addison Wesley.

[Dewan and Riedl, 1993] Dewan, P. and Riedl, J. (1993). Toward computer-supported concurrent software engineering. *IEEE Computer*, 26(1):17–27.

[Djemili, 2006] Djemili, R. (2006). Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung. Master's thesis, Freie Universität Berlin.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[Haake et al., 2005] Haake, A., Lukosch, S., and Schümmer, T. (2005). Wikitemplates: adding structure support to wikis on demand. In *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, pages 41–51, New York, NY, USA. ACM Press.

[Hanks, 2004] Hanks, B. (2004). Tool support for distributed pair programming: An empirical study. In *Proceedings of XP/Agile Universe 2004: 4th Conference on Extreme Programming and Agile Methods*, Calgary, Canada.

[Hanks, 2005] Hanks, B. F. (2005). *Empirical Studies of Distributed Pair-Programming.* Dissertation, University of California Santa Cruz.

[Ho et al., 2004] Ho, C.-W., Raha, S., Gehringer, E., and Williams, L. (2004). Sangam: a distributed pair programming plug-in for Eclipse. In *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 73–77, New York, NY, USA. ACM Press.

[Hupfer et al., 2004] Hupfer, S., Cheng, L.-T., Ross, S., and Patterson, J. (2004). Introducing collaboration into an application development environment. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 21–24, New York, NY, USA. ACM Press.

[Kircher et al., 2001] Kircher, M., Jain, P., Corsaro, A., and Levine, D. (2001). Distributed extreme programming. In *Proceedings of XP2001 - eXtreme Programming and Flexible Processes in Software Engineering*, pages http://www.kircher–schwanninger.de/michael/publications/xp2001.pdf, Villasimius, Sardinia, Italy.

[Lukosch and Schümmer, 2008] Lukosch, S. and Schümmer, T. (2008). The role of roles in collaborative interaction. In *13th European Conference on Pattern Languages and Programs (EuroPLoP'08)*.

[Maurer, 2002] Maurer, F. (2002). Supporting distributed extreme programming. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 13–22, London, UK. Springer-Verlag.

[Pinna et al., 2003] Pinna, S., Mauri, S., Lorrai, P., Marchesi, M., and Serra, N. (2003). XPSwiki: An agile tool supporting the planning game. In Marchesi, M. and Succi, G., editors, *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003*, number LNCS 2675, pages 104–113. Springer-Verlag Berlin Heidelberg.

[Reeves and Zhu, 2004] Reeves, M. and Zhu, J. (2004). Moomba – a collaborative environment for supporting distributed extreme programming in global software development. In *Lecture Notes in Computer Science : Extreme Programming and Agile Processes in Software Engineering*, pages 38–50. Springer.

[Rekimoto et al., 1998] Rekimoto, J., Ayatsuka, Y., Uoi, H., and Arai, T. (1998). Adding another communication channel to reality: an experience with a chat-augmented conference. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pages 271–272, New York, NY, USA. ACM.

[Schmolitzky, 2008] Schmolitzky, A. (2008). Patterns for teaching software in classroom. In Hvatum, L. and Schümmer, T., editors, *Proceedings of EuroPLoP'08*, pages 37–54, Konstanz. UVK.

[Schümmer and Lukosch, 2007] Schümmer, T. and Lukosch, S. (2007). *Patterns for Computer-Mediated Interaction*. John Wiley & Sons, Ltd.

[Schümmer and Schümmer, 2001] Schümmer, T. and Schümmer, J. (2001). Support for distributed teams in extreme programming. In Succi, G. and Marchesi, M., editors, *eXtreme Programming Examined*. Addison Wesley.