In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking: Understand-Improve-Apply (pages 197-218)
Springer 2011 (doi:10.1007/978-3-642-13757-0_12)

Agile Software Development in Virtual Collaboration Environments

Principal Investigator: Robert Hirschfeld
Authors: Robert Hirschfeld, Bastian Steinert, and Jens Lincke

Abstract Agile processes are gaining popularity in the software engineering community. We investigate how selected design practices and the mind-set they are based on can be integrated into Agile software development processes to make them even stronger. In a first step, we compared Agile methodologies with interaction and product design methodologies and discovered that both fields have much in common with respect to their underlying principles and values. Based on our findings and by applying both methodologies, we improved collaboration support for geographically-dispersed software development teams. We designed and implemented ProjectTalk and CodeTalk as part of our XP-Forums platform. Independently of their geographical location, team members can create and maintain user stories with ProjectTalk. CodeTalk enables team members to efficiently communicate their concerns regarding development artifacts in an informal manner.

1 Introduction

Agile software development processes are increasingly followed in software development projects that deal with complex domains and require continuous interaction among developers and with customers and prospective users. Agile approaches such as Extreme Programming [3] or Scrum [13] are people- and code-centric. Based on a high-quality code base throughout the entire project, developers can respond almost instantly to customer needs and requests. Teams can quickly make progress in providing the desired technical solution due to short development cycles and incremental explorations.

Design Thinking [12, 6] as a process has interesting aspects to offer—not only to designers, but also to software engineers. In our project, we will extend agile

Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
e-mail: firstname.lastname@hpi.uni-potsdam.de

development processes with elements from the Design Thinking approach to make them even stronger. Our enhancements will explicitly support both developers and customers to explore divergent alternatives and to converge on a decision or solution whenever necessary and possible.

There is also the trend that project teams tend to disperse around the world. Distributed development is getting more common, requiring team members to resort to means other than face-to-face communication to organize themselves, to collaborate, and to keep in touch regardless of geographical location.

Teams following agile software development processes or employing Design Thinking methodologies are usually small compared to the ones adhering to more traditional approaches. Team members collaborate closely via continuous and informal interactions rather than via large formal documents and schedules planned far ahead. This kind of collaboration is difficult to achieve in distributed settings, for example, when trying to gather expertise from team members. We will use and improve our extended agile software development process to design and implement better ways of communication for efficient and effective information exchange in distributed teams regardless of their geographical distribution, allowing them to collaboratively immerse in their tasks.

We both improve the tools we have developed so far, such as ProjectTalk for managing user stories and planning activities collaboratively, and expand our tool suite as necessary and desirable for improved interaction. We aim for a solution that allows a seamless transition between asynchronous and synchronous collaboration styles and which provides support for user-specific views at different levels of detail. We will focus on communication that is essential for keeping distributed teams in sync and for allowing a high degree of transparency on their core development activities.

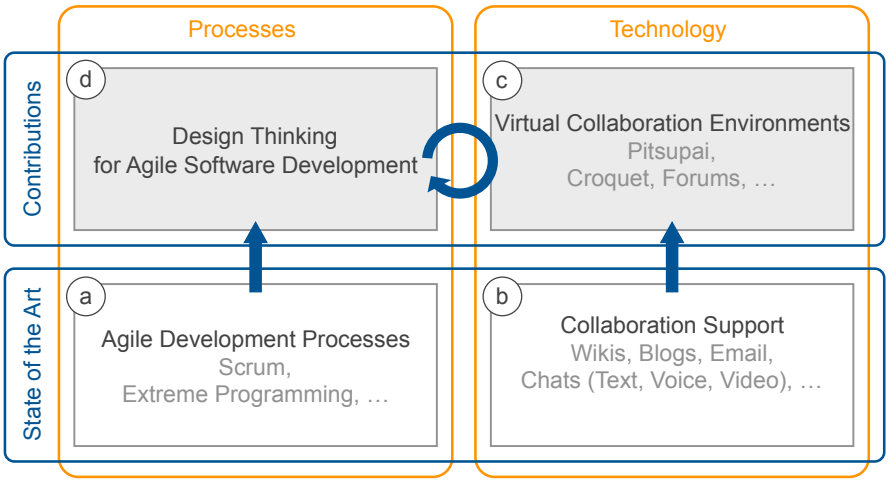

Fig. 1 Conceptual Project Map

Our approach is twofold (Fig. 1): First, we extend state-of-the-art agile development processes with elements of Design Thinking to allow software developers to benefit from the mind-set of design. Second, we employ our extended software development process to design and implement tools supporting this process keeping everyone collaboratively involved.

In the following, we outline our motivation to extend agile development processes and to provide appropriate tool support within a virtual collaboration environment. We then describe current findings regarding desired extensions to agile methodology. Thereafter we present our results gained with respect to tool support for distributed development teams—an application and an extension to a programming environment have been developed.

2 Motivation and State of the Art

Agile software development processes are iterative and incremental, embracing change and evolution, and promoting design simplicity and high software quality. In this section, we first describe important aspects of agile methodologies and then discuss our objectives to enrich these methodologies with elements from industrial and interaction design methodologies. After that, we discuss the increased need for collaboration support as design and development teams tend to disperse around the world.

2.1 Design Thinking for Agile Software Development

Most agile software development processes [1] are people- and code-centric in that they foster interaction between project participants, grounded on short and many iterations, each of which resembling a full development cycle including planning, analyzing and prioritizing requirements, designing, and testing. Risk is minimized by producing a running system in every such iteration in a short period of time.

The most popular representatives of such processes are Scrum [13] and Extreme Programming (XP) [3]. Scrum is a high-level process framework that defines roles and practices. The Scrum process skeleton (Fig. 2a) has two main cycles: The long cycle (30 days) represents a development activity that leads to an increment of the product to be built, based on the requirements and the budget allocated for their implementation. It groups several short cycles (24 hours each) that cluster daily activities of the team members inspecting each other's activities, proposing next steps, and suggesting corrective actions if necessary.

Compared to Scrum, XP is a more disciplined method. It focuses on the strict application of programming techniques representing best practices, on clear communication, and on teamwork. XP assumes short development cycles that allow for

early feedback based on actual code. Automated test suites represent an executable specification of the system to be built, which is co-evolved with the system itself.

We regard agile processes as solution-oriented since they encourage developers to advance only in small increments that are all based on sound technical decisions only, without enough opportunities to approach the same problem from different perspectives. Unfortunately, this does not leave much room for exploring both problem and design space.

In our project, we investigate how elements from Design Thinking such as divergent and convergent thinking (Fig. 2b), need-finding, brainstorming and sketching, and the preservation of ambiguity can be integrated into agile processes like Scrum and XP.

2.2 Collaboration Support for Distributed Development Teams

Both Design Thinking and agile software development projects rely on small teams working closely together. Informal direct communication and physical tools and artifacts such as whiteboards, sticky notes, and story cards are preferred means of expression and interaction. For that to work, team members need to be co-located to take full advantage of the benefits offered by these tools and artifacts.

Due to organizational structures and economical concerns of modern organizations, distributed development is getting more popular, requiring geographically dispersed project teams to collaborate across space and time. Geographically distributed teams have difficulties to apply the tools and artifacts preferred or required when following Design Thinking and agile development methods. This requires teams to resort to means other than face-to-face communication to organize themselves, to collaborate, and to keep in touch and sync.

We argue that there is a need for virtual collaboration environments as a shared place for project participants to meet, to work, and to collaborate as informal as they are used to. One key challenge addressed in our project is the computerization of these informal but important tools and artifacts without either loosing their advan-

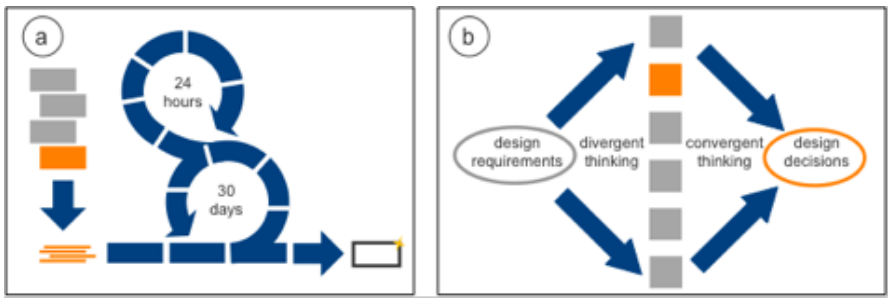

Fig. 2 (a) Scrum backlog and process skeleton (a); (b) Divergent and convergent thinking

tageous properties, or by compensation for their potential loss. We will compensate for such losses with advantages offered by the new media of virtual environments, using their power and nearly unlimited space to go beyond the possibilities and constraints of the physical world.

Examples of tool support to be provided by our virtual collaboration environment include whiteboards that are unbound in screen estate, persistent, and searchable even after some of the design phases are finished. This way, early design decisions are available to the program maintainers when needed. Furthermore, we want important relationships between design and development artifacts to be made explicit and preserved for continuing efforts and future reference. Code editors, for example, can be annotated with the alternatives considered in previous convergent/divergent design phases.

3 Design Thinking for Agile Software Development

Relying on a natural approach to learning, that is comparing the new with facts and knowledge already understood and internalized, we examined design-related topics on innovation, need finding, interaction design, and creativity techniques from a software engineering perspective, allowing us to better understand and integrate new interesting elements into agile development processes. First results of this comparison between design methodologies and agile software engineering topics reveal many commonalities.

In this section, we describe these commonalities concerning the underlying values. We then present two different approaches to combine design activities with development activities and discuss pros and cons with respect to the principles of both fields.

3.1 Common Values

Recognizing similarities between agile and design methodologies has been our motivation for investigating the combination of XP and elements from Design Thinking. We studied literature on Design Thinking and interaction design from a software engineering perspective, and identified many commonalities with respect to the values of respective methodologies; values referring to underlying principles of the methodologies, the principles upon which concrete techniques are based.

- **Wicked Problems.** Software development projects are confronted with *Wicked Problems* [5]. Originally described in [21], the term *Wicked Problems* refers to problems that are not well understood and thus difficult to describe. The problem becomes, however, clearer as one moves ahead to the solution of the problem. The closer one gets to the solution, the more one can understand and describe what the actual problem is. In the field of design, it is reported that design teams

usually face this kind of problem [4]. Moreover, solving a problem that is understood well may not be referred to as a design activity.

- **Close Interaction.** People interacting closely with each other exchange a lot of knowledge and opinions, which in turn supports making progress. For this particular and other reasons, agile processes such as XP strongly suggests a close interaction amongst all team members as well as with the customer. The value of close interaction results in recommendations of concrete practices such as on-site boards, pair programming, collective planning sessions, collective code ownership, or small but regular releases [3, 13]. Close interaction is also a key aspect of design activities. Many designers work with their customers using different methods to elaborate their understanding of the domain from multiple perspectives. As another example, the collaboration of team members having different areas of expertise and experience further supports the exploration of the problem and solution space; it eases the creation of a multitude of divergent ideas and supports their connection and composition.
- **Go for Feedback.** Iterative and incremental development is the foundation of all agile methodologies [14]. In each iteration a next running version of the system is created and delivered, bringing value to the customer and allowing for feedback on this running, executable prototype that is used in real work settings. Being close to design processes in this respect, XP recommends to have actual users on-site; this enables early and direct feedback during the workout and implementation of all details of the higher-level concepts and ideas. Programming is also an activity resulting in feedback. Developers get feedback on their understanding of the program domain and about the quality of their implementation [3]. For similar reasons, designers are encouraged to create many prototypes and to work with them, getting feedback on the forms and materials, for example, or feasibility constraints. Early prototypes should further be tried out by users of the target group in target scenarios, striving for valuable feedback on aspects such as usability.

3.2 Approaches to Combine Design and Development Activities

The development of a software product involves a multitude of different activities. While some activities may assigned more to design than to development and vice versa, they cannot be clearly separated. This would require precise definitions of both design and software engineering methodologies that are not available. Moreover, design in the broadest sense can be considered as the entire process of creating a new product from understanding the needs over multiple prototypes to the final product; software development usually refers also to the entire process including aspects such as requirements engineering and user interface concepts [20, 23]. For that reason, we also examine interaction design as one specific candidate of design methodologies that deals with understanding user needs and elaborating interaction concepts to meet these needs. Interaction Design involves activities of the following

categories: inquiry (the study of the existing), exploration (the study of the possible), composition of the existing and the potential, assessment, and coordination [15].

While agile methodologies value user feedback, they are usually not very specific about useful techniques for understanding the users' needs and developing respective user interface concepts. However, the design of the interface to the users gets more important and software vendors have started to attache more value to it [17, 19] This was a main reason for researchers to conduct case studies investigating how companies integrate design activities into the overall software development process [9, 27, 7]. Basically, there are two different approaches to combining the work of interaction designers with the work of software engineers; both are depicted in Fig. 3.

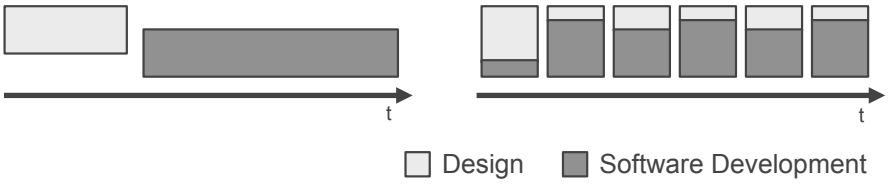

Fig. 3 Approaches to combine design and development activities; a waterfall-like approach, and an iterative and incremental approach.

The left side of Fig. 3 shows a waterfall-like approach: conducting design activities first and handing over the resulting concept to a development phase. It is the task of the designers to understand the problem domain of the users. Based on that, they develop several concepts addressing the users' needs, work the concepts out, and align them to each other. The overall and finalized design represents the requirements on the system that is to be realized by the software developers in a subsequent stage.

The right side of Fig. 3 depicts an alternative approach that is based on the notion of iterative and incremental development. Over multiple iterations, designers and software developers simultaneously work on the design concepts and the code base respectively. This approach is built on the observations that getting all requirements right first is rarely possible, that requirements can and will change during the project lifecycle, and that the understanding of the problems evolves as we get closer to their solution.

The iterative and incremental approach implicates that the overall design is neither final nor complete until the end of the project and can be changed at any time during the course of the project. The problem domain and the corresponding set of requirements has to be separated early in the project so that designers and developers can work on a prioritized subset of requirements and problems in each iteration. On the one hand, changes to the design result in additional development effort. On the other hand, the iterative approach allows for taking advantage of new insights gained during the ongoing project development. With that, this approach allows both designers and developers to embrace change in most if not all all different aspects

relevant to the project. It is based on the notion that learning is a natural consequence of making progress and reflecting on it.

The executable or running systems delivered after each iteration bring value to the customer and thus form trust based on their early return of investment. Furthermore they provide the opportunity to obtain and incorporate feedback from real work usage settings. Getting feedback early and often is an important aspect of both Design Thinking and agile development methodologies, and prototyping techniques such as sketching and paper prototyping support the exploration of alternatives and eases getting insights and making progress. The main goal of a prototype is to reveal misconceptions and to improve the understanding of the problem domain. In this sense, each version of the software system delivered after an iteration can be considered as another kind of prototype. In contrast to a sketch, for example, it has a higher resolution, but it allows for getting different aspects of feedback, in particular the adequateness of the current solution in the target settings, when real users work with the application in real work situations.

4 Virtual Collaboration

Close collaboration and communication is vital in XP projects—amongst team members and also with customers. Development teams tend, however, to disperse around the globe and thus have to resist to means other than face-to-face communication. In our project, we integrate and develop tool support for distributed development teams to allow for informal communication and efficient collaboration despite geographical dispersion. We describe the results of our efforts during the last year in this section. Amongst others we have developed ProjectTalk an application that supports collaborative planning activities in distributed teams. ProjectTalk's design allows for working with story cards in a similar way as it is possible with physical artifacts, by still providing the advantages of a digital solution (4.1). Co-present users can interact with ProjectTalk simultaneously without synchronizing on an input device, for example. All users are further enabled to act on their own behalf (described in 4.2). This functionality represents a contribution to the collaboration community and is described in detail in [24]. We also have developed CodeTalk [25], an extension to an development environment that enables distributed developers to have conversations about source in an informal and efficient manner (described in 4.3). Along with other tools, such as ProjectTalk, CodeTalk was used in several development projects and showed its usefulness. We have written and submitted a research paper on the approach of CodeTalk to informal conversations about source code.

4.1 Bringing Physical Artifacts to Digital Environments

XP similar to design processes heavily relies on co-location of all teams members and on physical tools for communication and organization such as index cards and whiteboards. *User Stories* are the central artifacts in XP teams. They form a concise description of the customer's requirements written in everyday language. *User Stories* are elaborated in concerted planning session with the customer and persisted on index cards. These cards are usually managed by pinning and moving them on a whiteboard, being visible for the team and indicating progress of the project.

Bringing all these information from the physical whiteboard into the digital world promises a multitude of new possibilities, such as having unbounded space or support for full text search. In addition, having these artifacts digitalized provides a good basis for supporting and encouraging close collaboration in teams working geographically dispersed. Prospective collaboration software should thereby incorporate as many strengths of physical setups as possible.

Learning from others about the flaws and strengths in this and other respects, we extensively benchmarked existing software solutions supporting agile processes. As one important result, it turns out that a main challenge is the design for interaction with huge amounts of information within limited dimensions of a computer screen. Many solutions have decided for tabular representation of *User Stories* actually causing a feeling of information overload. All solutions distinguish between viewing and editing information and usually offer a number of forms to alter contents of *User Stories*. This design does not harmonize well with the card metaphor and requires a decent number of clicks for simple usage scenario.

By trying out available solutions and actively working with them, we became aware of that the design of the tools influenced the project team in working with user stories; depending on available space, team members cut descriptions down to single bullet-points or fill out many different fields of the forms making stories too formal and complex.

Getting closer to a solution candidate meeting the needs, we continuously designed user interface concepts, created various prototypes, implemented the prototype concepts, and used the implemented version during our daily work. The left column of Fig. 4 depicts prototypes of different concepts developed in the course of our project. The right column shows corresponding versions of implementation called *ProjectTalk*. The last picture at the bottom left shows a new prototype for ProjectTalk that is currently implemented. This current version includes some innovative user interface concepts for easily browsing large amounts of project artifacts.

We argued in Section 3 that the early use of the application, which is to be enhanced over the project time, is important for getting feedback from actual users already working with the application during their regular activities. Applying this theory, we used ProjectTalk from early on for planning the next version. This continuous work with ProjectTalk help us to understand important aspects regarding its collaborative use.

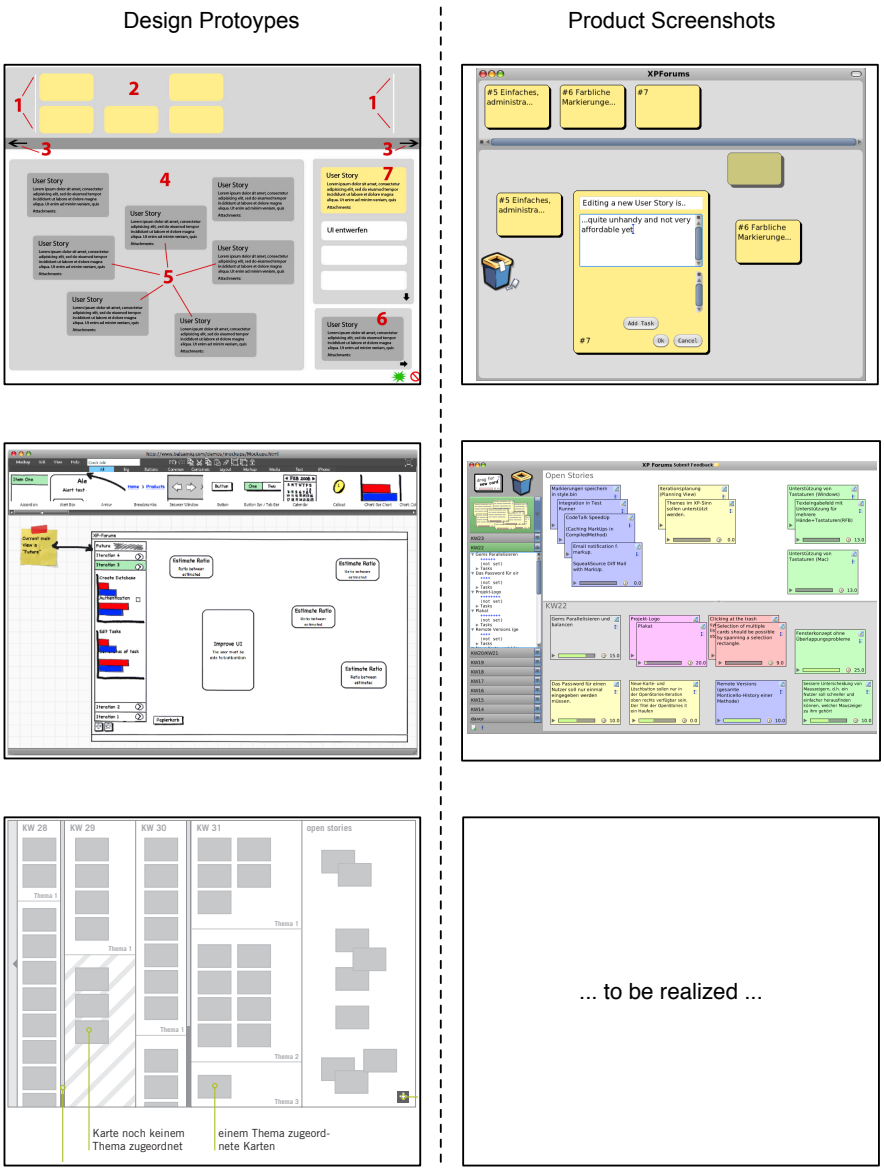

Fig. 4 User interface concepts (left column) and screenshots of corresponding applications.

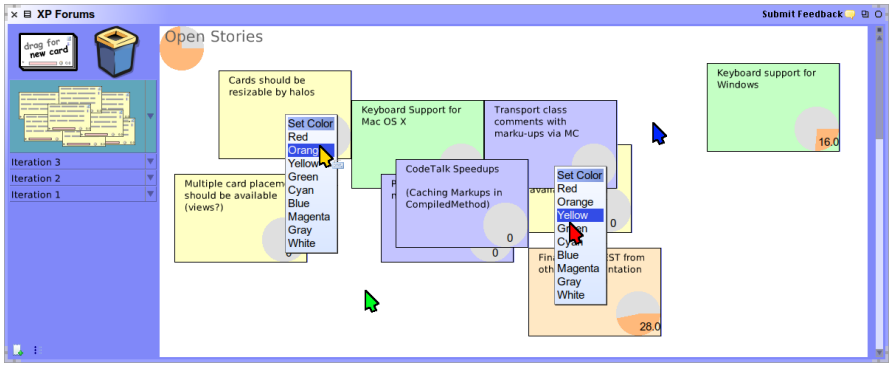

Fig. 5 Screenshot of the application: Multiple users, represented on the screen by colored mouse cursors, interact with a virtual whiteboard. Every user may open its own context menu.

4.2 Multi-user Multi-account Single-screen Interaction

Integrating Single Display Groupware (SDG) concepts [26] with more traditional groupware, such as Wikis or project management software, requires re-considering the way of interacting with and designing for users. ProjectTalk integrates SDG concepts, enabling co-present team members to collaborate using the same application instance. Users are provided with separate input channels allowing them to contribute with the need to synchronize. While users often have an account and interact with the application in a way specific to their account, traditional SDG concepts do not allow users to act on their on behalf. In this subsection, we describe the issues resulting from this limitation and present our approach to handle them, which was implemented in ProjectTalk.

4.2.1 Merging Characteristics of Asynchronous Groupware and Single Display Groupware

By employing SDG concepts, we provide XP teams with interaction characteristics similar to working with physical tools (Fig. 5). XP teams traditionally rely on these physical tools such as index cards and whiteboards for communication and organization purposes. Bringing all these information from the physical whiteboard into the digital world would enable a multitude of new possibilities, such as having unbounded space or support for full text search. Additionally it enables remote collaboration, because virtual whiteboards, unlike physical ones, can be shared over computer networks.

Using physical tools such as whiteboards and index cards, team members are able to act independently of one another without the need to synchronize on pens, for example. Traditional applications, however, only support interaction with one user at a time. The need to synchronize on application control impedes spontaneous

interaction and reduces social dynamic in comparison to a physical whiteboard. Therefore we enable multiple users to interact with the application independently. To further increase the dynamic of a session, users are able to join or leave a session at any time. We also incorporated screen sharing technology to support distributed XP teams. Remote team members can share planning sessions, for example, and interact with the same shared screen.

When multiple users interact with a single screen, traditional applications are unable to distinguish acting users. These restrictions of current concepts lead to issues concerning authorization and traceability. Fig. 6 depicts a typical multi-user single-screen scenario. In this scenario, the application is unable to make a reasonable decision whether the user is privileged to perform the desired action as the application does not know who the the currently acting user is. By using current approaches, all users actually act on behalf of a host user, the one who logged in before. This gives all acting users the same privileges in the described scenario. The inability to distinguish multiple acting users does not only lead to undesired modifications, but also to unintended restrictions of users. When, for example, users want access to previous projects for analysis purposes, they might be unable to open the projects as the host user is not privileged for accessing this information. As another consequence of application actions not being linked to the acting user, tracing data of user action become unreliable. It is impossible to find out who modified certain important information.

The examples described above show that it is necessary to distinguish users concerning their security context and to execute every action users want to perform in their respective context. In particular, the following questions come up:

- How might an application be designed to allow multiple interacting users logging in and logging out?
- How should the user credentials be managed?
- How might UI events be distinguished by acting users?
- How might the application make use of this distinction and link application actions to users?
- How should applications be designed to deal with multiple interacting users having different privileges?

4.2.2 Platform Support for Multi-user Multi-Account Interaction

An application featuring multi-user single-screen interaction requires special support in the application's platform such as handling the events from multiple, similar input devices independently from each other. In addition, if multiple remote users should be able to work in same way as local users, and if UI actions should be linked to the users, an adequate concept representing the users and their actions is needed.

The concept of *Hands* has shown as a meaningful approach to represent and manage user interactions. *HandMorphs*, or *Hands* for short, are part of the object-oriented GUI framework Morphic [16]. Hand objects obtain their event data from

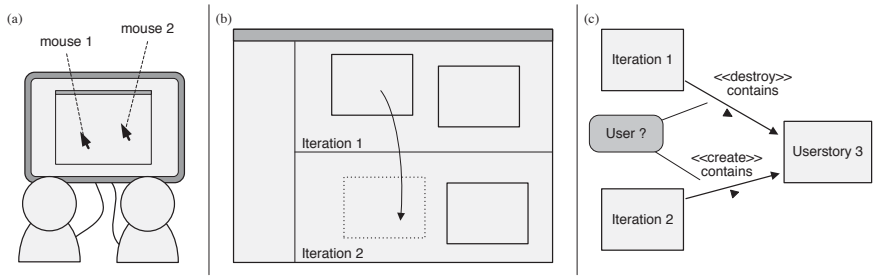

Fig. 6 A participant moves a user story from one iteration to another. Due to the missing user context, the application cannot determine the user who initiated the action, and is unable to check whether the acting user is privileged to perform the desired action.

the corresponding input stream autonomously. By using this concept, additional input streams can be integrated; multiple hands, that is, mouse pointers and cursors, can be controlled by different input sources. Current operating systems, however, support only one system cursor and input events from different devices are merged into one input event stream. To bypass the operation system's behavior [30, 28], we developed special support for Squeak's virtual machine (VM). The design of our VM extensions allows attaching and detaching input devices during the application's run-time. Hand objects further abstract from concrete input sources and provide a defined interface to applications. Thus, it is transparent to the application, whether the hands are controlled from a local device or via a VNC connection.

Finally enabling applications to link users to actions, we extended the concept of *HandMorphs* and allow for impersonation. Our extensions to *HandMorphs* manages required user information and provide an interface to applications. Application objects have access to the currently active HandMorph and can ask this HandMorph for credentials of the currently acting users. If the HandMorph is not yet associated with an user, it opens a dialog asking the user to provide username and password. Based on that information, the HandMorph is then associated with the corresponding user.

4.2.3 Application Support for Multi-user Multi-account Interaction

Our extensions to the application platform, described above, enable applications to link users to actions. This in turn allows all users for acting on their on behalf. We describe how ProjectTalk makes use of this functionality here. The integration of SDG concepts further requires application developers to handle the upcoming additional issues regarding authorization. Additionally, content and functionality that is specific to certain users or roles must be offered in new ways as the users interacting with the shared display may have different privileges. Both kinds of user, user-specific content and authorization, have been addressed in the design of ProjectTalk, and is presented in this subsection.

ProjectTalk realizes multi-user multi-account single-display interaction by accessing the information of the user that triggered the current event processing and links this user to HTTP actions. It consists of a client and server component that synchronize on shared data using the HTTP protocol. A user interface action that involves modification of data in the client results in one or more HTTP request to the server. The application platform provides access to that particular *HandMorph* object that represent the input the channel of the user that initiated the current event processing. This *HandMorph* object is also called *ActiveHand*. The *ActiveHand* is accessed by the client HTTP-communication layer, retrieving the associated user and using corresponding credentials for the HTTP requests to the server. With that, ProjectTalk links HTTP request, which form the primary action of the client, to users enabling them to perform all actions on their own behalf.

Supporting our approach to multi-user multi-account single-display interaction, application developers have to consider additional authorization issues. A groupware that provide role-specific behavior is often designed in a way that users see only the functionality they have access to. In a project management software, for example, only administrative users are able to create new projects; users without these privileges cannot see this functionality. If multiple users interact with a single screen at the same time, the different users might have different privileges. It has to be respected that some amongst all co-present users sharing one display are not allowed to perform actions that are offered.

This mismatch can be handled in three different ways. One way is displaying the collective set of functionality all users have access to. Unfortunately, users would be limited and could not use all features they are allowed to. Another approach is presenting every feature available to at least one user. As a result, users can activate actions they are not allowed to perform. Applications have to handle denied access explicitly and be able to recover from this error. The last possibility is to (re-)design the application so that, for instance, users are provided with menus specific to their privileges.

The application design of ProjectTalk combines the second and the last option. For example, some users will not be allowed to modify user stories or move them between iterations. Still, all users have read access to these user interface components; users unprivileged to perform a modification will experience a failure and receive a corresponding message. The menu for opening projects exemplifies the last option. For each user accessing the menu, it provides specific content—only projects the user is a member of.

The handling of denied access gained special attention in ProjectTalk, so that this concern is not scattered over the entire application code. A typical user interface action results in one or more HTTP request to the server. If the acting user is not authorized to read or write specified resources, the server will return with an unauthorized error. An HTTP error is responded and converted into an application specific *NotAuthorized* exception. The exception is handled in the implementation of the model proxy objects. The proxy objects provide application specific interfaces that are implemented generically. The proxy objects store object properties and synchronize modifications with the server. If the server processes the request success-

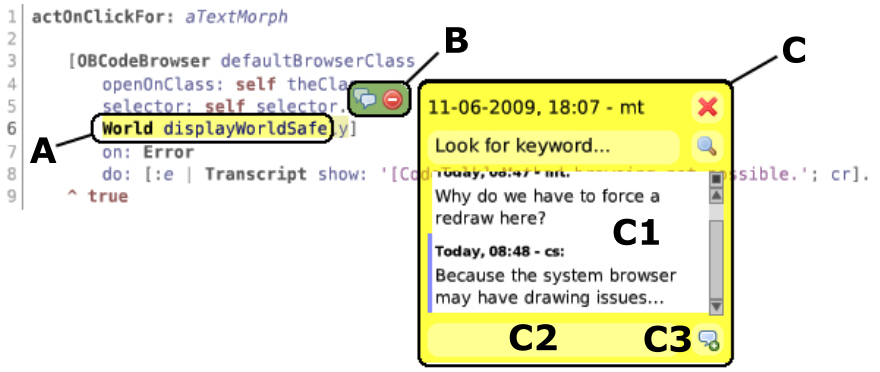

Fig. 7 Right: Code with markup (A), inline morphs (B) and the chat (C); Left: marked methods

fully, the modification will be applied to the corresponding description property of the user story's proxy object, and bound views will get notified about the change. Otherwise, the modifications are discarded, the stories will show the old description, and the user will receive an error message.

The implementation of both handling denied access and linking actions to users is integrated well in the design of ProjectTalk. Both concerns are well-separated from application specifics and the extensions can be integrated easily into other applications.

4.3 CodeTalk—Conversations About Code

This subsection describes CodeTalk, our approach to enable efficient informal communication about source code. CodeTalk, which was implemented in Squeak Smalltalk [11]. CodeTalk allows developers to mark and annotate single expressions, whole lines, or entire methods in the source code. It works similar to text processing applications and tools that are capable of adding comments to PDF files. The markup and annotations are shared along with the source code using regular source code management support.

4.3.1 The Need for Communicating About Code

Developers often talk about the source code of the system to be developed and extended. The source code itself is the most important artifact during the development process, in particular in agile development processes. Developers usually care much about it and prefer, for example, simple and elegant solutions over complex ones that are more difficult to understand and maintain [3]. The system naturally evolves and is extended; so, software developers spend much time reading code.

However, parts of the system may be difficult to comprehend raising the need to request support from the originators; an algorithm might be very complex or the intended run-time behavior might be difficult to infer [29]. Programs can also be written in different styles making them more or less easy to understand [18]. This leads to another kind of communication amongst developers having the source code itself as the topic. During code reading developers also often discover source code that needs to be revisited and improved; for example, variable or selector names can be too general and thus not very meaningful [18]. Developers might further have ideas to simplify the system's design [3, 8] or even detect potential failures in algorithms. While these issues are often discovered during regular coding activities, developers may not have enough time or background knowledge [22] to refactor the respective parts of the system or to validate their theory of a failure and fix it if necessary. Also, developers rather might to continue working on their primary task at hand [10]. So, an efficient mechanism is needed to make the discovered issues explicit and share their insight with peers.

4.3.2 Informal Communication via Markups

Current approaches to communicate about source code include *source code comments* and external communication tools and protocols such as email and instant messaging. However, both have limitations and do not provide adequate means to support informal spontaneous communication about source code. But this kind of communication is important; it helps to ensure a high code quality and helps developers to become better in their profession. This has been our motivation to design and develop a new approach to support this informal ad-hoc communication.

Developers might, for example, discover a message send calling an expensive operation. Fig. 7 shows an example method in a typical code browser in Squeak. The statement selected in the figure enforces a full redraw of the entire scene graph, which can be a quite time-consuming operation. Developers might be skeptic about the necessity and mark the selected code as critical using a context menu or a keyboard shortcut. This will highlight the statement with a red background color. To additionally describe their opinion and thoughts, developers add then a note in the dialog that will be displayed next to the marked section, as shown in Fig. 7.

This new annotation functionality CodeTalk was integrated into the standard development environment, in particular into the tools for browsing and editing the code. So, developers can informally annotate a piece code whenever necessary during their regular code activities. The region of interest in the source code can directly be marked and annotated with an explanation.

Annotations are an integral part of the source code and as thus they are exchanged along with the source code itself. When developers commit modifications applied to their working copy, they will also submit all annotations currently in the code base to the source code repository, as depicted in Fig 8. The critical question about the statement that force a complete redraw is now part of the newly created source code revision.

When team members update their working later, they will retrieve the newly added annotation along with source code modifications. They will notice the question regarding the redraw statement, and the authors of that code might either remember a reason for forcing the redraw or they might not. In the latter case, they might consider removing the statement, test the application to validate the assumption, and commit the modification. As the annotations are connected to the source code they reference, the annotations would be removed together with the referenced statement in the described scenario.

If in the other case, enforcing the redraw is well-founded, developers can change the type of annotation, from critical to normal, and answer the previous question. Our extensions to the code browser enables developers to directly reply to questions or remarks in annotations so that a chat can evolve (Fig. 9).

Talking about source code often involves other sources located outside the currently discussed context: Sometimes developers come across methods that seem to be very similar, but they do not have the time or knowledge to perform the necessary refactoring. CodeTalk allows developers to mark that issue and to reference the other method in their comment. For example, the chat in Fig. 10 replaces the occurrence of "`String >> #findTokens:`" automatically with a hyperlink that browses to the method "`findToken:`" in the class "`String`". The link below points to a method "split" that does not exist and is therefore drawn in red.

The primary concept of CodeTalk is to separate the discussions about the source code from the source code itself, while keeping the connection to each other. This separation allows for individual support for the different concerns; specially designed tools can ease the creation and exchange of annotations and can provide a better awareness of these issues. The direct connection between source code and annotations indicates that they belong together and, thus, encourages developers to

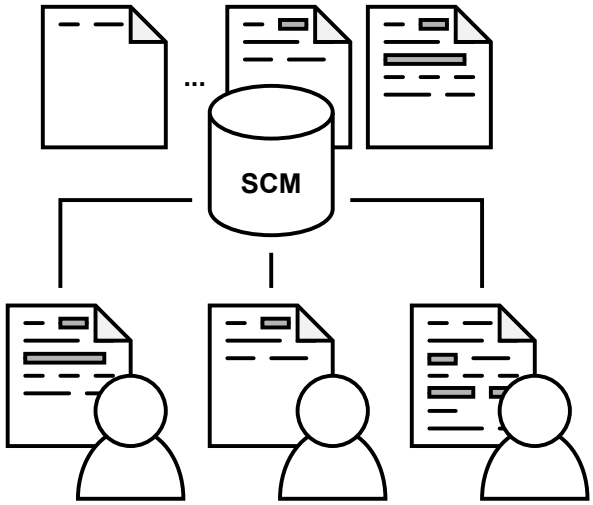

Fig. 8 CodeTalk's markups (gray) are shared through the SCM

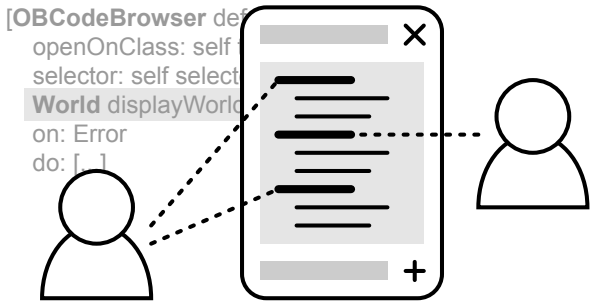

Fig. 9 A new conversation about code evolves

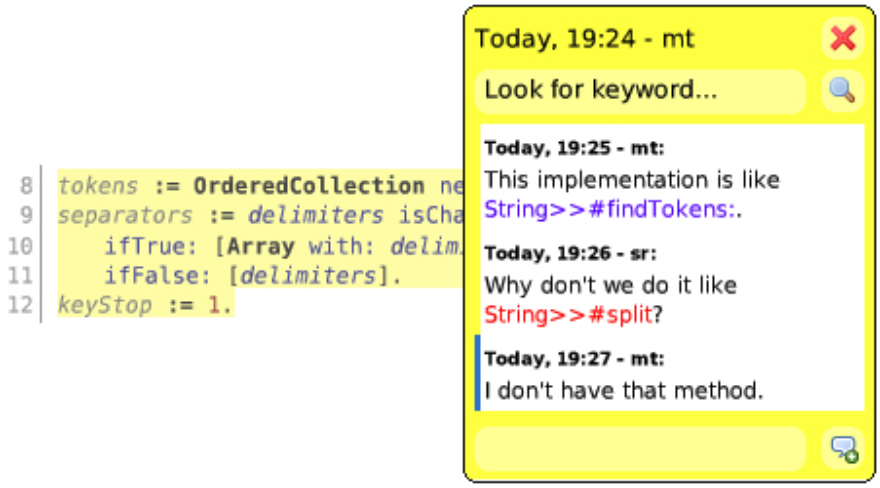

Fig. 10 Method links enable convenient source code navigation

keep both in sync. This may prevent the problem, that occurs when the code gets updated with accidentally ignoring the corresponding comment.

4.3.3 Case Study

CodeTalk has been used by several development during a case studies that was carried with 80 students in our *Software Engineering I* lecture. Students formed 16 different teams, ˜5 each, that were asked to develop applications in Squeak. The teams used an agile software development process such as *Extreme Programming* [2]. The project's time frame was about three months. After the end of the projects, we analyzed the source code of all revisions of all groups for markups. As shown in Fig. 11, the analyzed projects are of similar size consisting of about 600 to 800 methods. While one team created 20 annotations, other teams created up to 100 annotations.

		Team 1	Team 2	Team 3	Team 4
Number of Methods		745	605	700	828
Number of Revisions		178	174	246	335
Number of Markups	All Over Time	103	25	82	43
	Maximum	50	9	22	17
	At Project End	0	1	2	3
Average Lifetime		33	18	27	63

Fig. 11 Summary of markup usage from selected teams

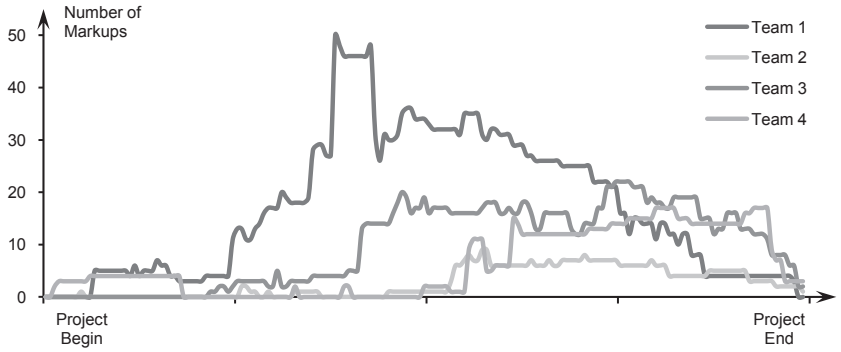

Fig. 12 Absolute number of markups in the source code over whole project development time

Fig. 12 and Fig. 13 indicate a continuous use of CodeTalk during the course of the project. At the end of the projects development teams cleaned up all markups, hopefully handling the described issues before. Note that the source was inspected by teachers at the end of the project. The average lifetime of annotations was 20 to 60 revisions, approximately a fifth of all revisions created during the project time.

Annotations created during the projects include the following examples:

- "That is somehow totally crap. The instance variable *separatorMap* seemed to be good for defining the place of these separating things for each category ..." (from German: "Das ist irgendwie total Mist. Die Instanzvariable *separatorMap* dacht ich wär gut, um für jede Rubrik festzulegen, wie die Trenndinger stehen müssen ...")
- "Looks paradoxical! ..." (from German: "Irgendwie paradox! ...")
- "Where should the layout code be included, this seems not to be a good place?" (from German: "Wo soll das Layout stehen? Hier ist vielleicht nicht der beste Platz.")
- "onClick + callback =>nonsense"

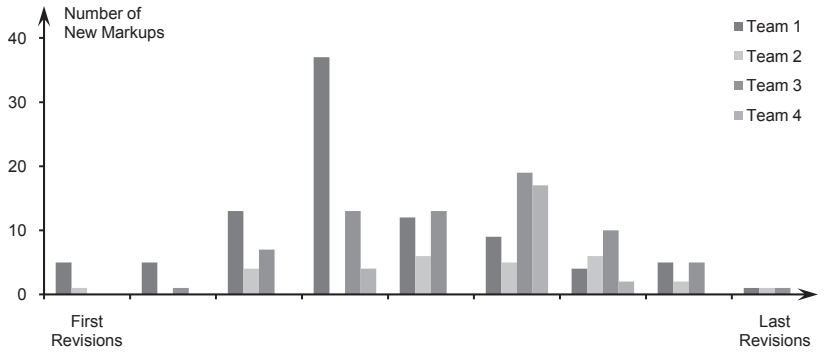

Fig. 13 Number of new markups in several development parts

- "Yes, there is a better way to do this :-)"

Developers started to use CodeTalk occasionally in the beginning of the projects and used it more often later on (Fig. 13). It seems that the need for conversations increases with the size of the code base. We further think that developers regard annotations as part of the source code and understand critical annotations as an indicator for insufficient code quality. All remaining annotations were addressed to the end of a project, to make it ready for release. The example annotations listed above show that developers like to use a colloquial style for communicating about source code related issues. While we have no evidence whether the teams would have discussed a similar amount of issues without CodeTalk or not, we think CodeTalk actually encourages this kind of conversations, which is important to bring all flaws to light.

Gathering additional personal opinions, we also conducted interviews with two teams. The teams reported that markups were used to write down tasks. This included planned refactorings of bad source code and new features that needed to be implemented. Additionally, the *critical* markup was occasionally used to point out bad coding style. Markups were also used for personal notes, especially as ToDo-items. Those teams that made heavy use of CodeTalk actually had a strong need for asynchronous communication, as many team members contributed from many different location and at different times for several reasons. The students argued that they used CodeTalk mainly due to convenience; it allows for staying in the current environment and context instead of switching tools.

5 Summary and Outlook

In this report, we present the objectives and first results of our project *Agile Development in Virtual Collaborative Environments*. We first describe and argue for our twofold research approach, investigating both design and agile software de-

velopment processes and accompanying tools to support geographically dispersed teams in applying these processes. We then present commonalities between Design Thinking and agile development methodologies with respect to their underlying principles. Two different approaches to integrate design activities in agile development processes are discussed. Thereupon we describe efforts and findings regarding the technological support for distributed development teams; the application *ProjectTalk* and the development environment extension *CodeTalk* are presented. ProjectTalk has been designed and developed for collaboratively managing user stories and planning activities; it particularly supports co-present team members by allowing them to act on their own behalf when simultaneously interacting with shared tools. CodeTalk realizes an informal yet efficient approach to communicate about source code collaboratively and over time.

With our first insights being very encouraging as they support our original hypothesis that agile software development will benefit from elements of Design Thinking, our next involve introducing other elements and values of Design Thinking and balancing them with respect to the elements and values of the original agile software development practices are still very challenging. Therefore, we will reflect on our experiences, revisit our design decisions, redefine our theory and then apply it in subsequent projects in several iterations. Design Thinking and agile development seem to have much in common, also because both rely on skilled, motivated, and professional individuals and teams working creatively together. We will elaborate models of both methodologies to improve our understanding of both their similarities and their differences.

We plan to use and improve our extended software development process to design and implement new ways of communication support that enable distributed teams to collaboratively immerse in their tasks and that encourages efficient and effective information exchange regardless of the team members' geographical location. We will both improve and extend the tools we have developed so far aiming for a solution that allows a seamless transition between asynchronous and synchronous collaboration styles and which provides support for user-specific views at different levels of detail. We will focus on communication that is essential for keeping distributed teams in sync and for allowing a high degree of transparency on their core development activities.

References

1. Agile Alliance. Manifesto for Agile Software Development. http://agilemanifesto.org/, 2001.
2. K. Beck. *Extreme Programming Explained: Embrace Change. ISBN 0201616416*. Addison-Wesley, 1999.
3. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, 2nd edition, 2004.
4. R. Buchanan. Wicked Problems in Design Thinking. *Design Issues*, 8(2):5–21, 1992.
5. Peter DeGrace and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions*. Yourdon Press, Upper Saddle River, NJ, USA, 1990.

6. C.L. Dym, A.M. Agogino, O. Eris, D.D. Frey, and L.J. Leifer. Engineering Design Thinking, Teaching, and Learning. *IEEE Engineering Management Review*, 34(1):65–92, 2006.
7. J. Ferreira, J. Noble, and R. Biddle. Agile Development Iterations and UI Design. In *AGILE '07: Proceedings of the AGILE 2007*, pages 50–58, Washington, DC, USA, 2007. IEEE Computer Society.
8. M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
9. D. Fox, J. Sillito, and F. Maurer. Agile Methods and User-Centered Design: How These Two Methodologies are Being Successfully Integrated in Industry. In *AGILE '08: Proceedings of the Agile 2008*, pages 63–72, Washington, DC, USA, 2008. IEEE Computer Society.
10. E. Horvitz, C. Kadie, T. Paek, and D. Hovel. Models of Attention in Computing and Communication: From Principles to Applications. *Commun. ACM*, 46(3):52–59, 2003.
11. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM.
12. T. Kelley and J. Littman. *The art of innovation*. HarperCollinsBusiness, 2001.
13. Schwaber Ken. *Agile Program Management with Scrum*. Microsoft Press, 2004.
14. C. Larman and V.R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, 2003.
15. J. Löwgren and E. Stolterman. *Thoughtful Interaction Design*. MIT Press, 2004.
16. J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, New York, NY, USA, 1995. ACM.
17. G. Meszaros and J. Aston. Adding Usability Testing to an Agile Project. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 289–294, Washington, DC, USA, 2006. IEEE Computer Society.
18. P.W. Oman and C.R. Cook. Typographic Style is More Than Cosmetic. *Commun. ACM*, 33(5):506–520, 1990.
19. J. Patton. Hitting the Target: Adding Interaction Design to Agile Software Development. In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, pages 1–ff, New York, NY, USA, 2002. ACM.
20. K. Pohl. *Requirements Engineering: Grundlagen, Prinzipien, Techniken*. dpunkt-Verl., 2007.
21. H.W.J. Rittel and M.M. Webber. Dilemmas in a General Theory of Planning. *Policy sciences*, 4(2):155–169, 1973.
22. T.M. Shaft and I. Vessey. The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process. *Journal of Management Information Systems*, 15(1):78, 1998.
23. I. Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.
24. B. Steinert, M. Grünewald, St. Richter, J. Lincke, and R. Hirschfeld. Multi-user Multi-account Interaction in Groupware Supporting Single-display Collaboration. In *Proceedings of the Fifth International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2009)*. IEEE Computer Society, 2009.
25. B. Steinert, M. Taeumel, J. Lincke, T. Pape, and R. Hirschfeld. CodeTalk—Conversations about Code. In *Proceedings of the Eigth International Conference on Creating, Connecting and Collaborating through Computing (C5 2010)*, La Jolla CA, USA, January 2010. IEEE.
26. J. Stewart, B. B. Bederson, and A. Druin. Single Display Groupware: A Model for Co-present Collaboration. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 286–293, New York, NY, USA, 1999. ACM.
27. D. Sy. Adapting Usability Investigations for Agile User-centered Design. *Journal of usability Studies*, 2(3):112–132, 2007.
28. E. Tse and S. Greenberg. Rapidly Prototyping Single Display Groupware Through the SDGToolkit. In *AUIC '04: Proceedings of the fifth conference on Australasian user interface*, pages 101–110, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

29. A. Von Mayrhauser and AM Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.
30. G. Wallace, P. Bi, K. Li, and O. Anshus. A Multi-cursor X Window Manager Supporting Control Room Collaboration. Technical report, Princeton University, Computer Science, Technical Report TR-707-04, 2004.