Supporting distributed software development by modes of collaboration

Till Schümmer, Jörg M. Haake
GMD – German National Research Center for Information Technology
IPS1 – Integrated Publication and Information Systems Institute
{Till.Schuemmer|Joerg.Haake}@darmstadt.gmd.de

Abstract. Work processes in team based software development need to be structured to minimise and resolve conflicting or divergent work. Current software development methodologies propose ways for dividing the whole task of software development between team members. This paper suggests a different way of working by introducing modes of collaboration (MoCs), which support concurrent and collaborative work. A MoC defines how tight two people can work together and how much the rest of the group can demand to know about a programmer. Different MoCs are ordered in a spectrum from single user's offline usage up to concurrent editing of the same source code. Special emphasis is put on balancing gains and efforts that are related to a specific MoC. The second part of the paper presents how MoCs are implemented in the distributed co-operative software development environment TUKAN. TUKAN includes synchronous co-operative tools and awareness widgets, which operate on a spatial representation of the software under construction. TUKAN provides tools for each MoC and allows programmers to switch between MoCs.

Introduction

Nowadays, software development is usually carried out in teams. Many modern software development methodologies emphasise this fact by introducing special forms of collaboration. For instance, the eXtreme Programming methodology (XP) (Beck, 1999) introduces pair programming sessions, where two program-
mers share one computer and solve the programming task together. The adaptive software development process (ASD) (Highsmith, 1999) is another methodology that focuses on collaboration within the team.

Anyhow, besides the collaborative aspects, programming is implicitly an activity performed by (many) individual users, as writing a book or composing a piece of music (Weinberg, 1971). The discrepancy between isolated work and group work is therefore inherent to software development. Even within XP’s pair programming sessions the participants frequently select one of two possible roles: One is coding (driving) and has the keyboard while the other person observes, comments and corrects the programming activity of the first. Environments that want to support programmers in their job of programming should therefore provide different modes of collaboration matching the roles and phases within the software development process and should ease the transitions between them.

When programming in medium sized to large teams, these teams are frequently distributed across many locations. Even small teams are often composed of experts who work at different locations. This introduces new challenges to the organisation of the programming work. Version management tools and conference systems can be used in these settings; but these tools fail to provide awareness on each other’s work. Conflicting changes and the fact of solving the same problem over and over again are consequences of this lack of awareness.

To assist programmers in programming as a team we propose a new tool called TUKAN. TUKAN provides different modes of collaboration and awareness, which meet the different needs of the programming team at different phases of the collaboration.

In a previous publication on TUKAN, we first identified different points of collaboration during the process of software development (Schümmer and Schümmer, 2001). These points of collaboration (PoCs) will serve as a basis to define different modes of collaboration (MoCs) for software development in this paper. After this, the latter part of the paper describes transitions between the different MoCs and how these transitions and the MoCs were implemented in TUKAN. We will then present some experiences that we gained from first experiments with the usage of TUKAN in programming groups. A section on related and future work concludes this paper.

Different Modes of Collaboration

Based on the PoCs that we found in Schümmer and Schümmer (2001), we identified different collaboration modes that are used during distributed team programming. Following the abbreviation PoC for “point of collaboration”, we call the mode of collaboration MoC. A MoC is a lightweight mode, which defines possible collaborative activities. Changing MoCs can therefore be considered as a lightweight activity (comparable to the effort of changing different modes of op-
eration by selecting different windows). MoCs range from single user's offline usage up to concurrent editing of the same source code. In fact, MoCs can be ordered in a spectrum from isolated work to completely (tightly coupled) collaborative work (cf. cooperative modes defined by Haake and Wilson (1992)). We will now discuss each MoC in a separate paragraph following the ordering of the spectrum.

**Offline mode.** When programmers have to solve hard problems, they often demand to be undisturbed. They want to work on the specific problem either alone or in a co-located two person team, without being interrupted. Thus, it is important that the system provides isolation for a single or a pair of programmers during the session. If the user's work raises questions, he can use asynchronous communication tools (e.g. e-mail) to send these questions to his colleagues. He may always be contacted by e-mail, but the point of time, where he reads his mail is self-determined (and therefore, also the time for receiving the answer is unpredictable). In the case where synchronous communication is indispensable, he may decide to change his MoC to a mode that allows synchronous communication (manually or automatically, as we will describe it in the next main section).

**Process level mode.** Planning the software project is essential to every software development process. XP introduces a lightweight planning strategy (the planning game (Beck, 1999)): the XP project is planned by stories that are constructed together with the customer. Stories describe the behaviour of the system under development (one might thus see the stories as kind of use cases). Each story is rated by the customer concerning the stories' importance, which allows the team to implement the most important stories first.

Stories are realised by a set of tasks. Tasks transform the content of a story into a mission for the programmer. Every programming activity is related to a task and a task should be fulfilled during one session (lasting about half a day).

When collaborating in process-level mode, the programmers interact with a planning tool and take responsibilities for their current task. Other programmers can see the current task of the programmer. This helps them to get at least a feeling of what the programmer is doing. They can retrieve meta-information, but they do not know which concrete artefacts the programmer currently manipulates.

**Change level mode.** If a programmer works in the change-level mode, the system logs all his manipulations of the source code. Logged changes can help to understand the work afterwards, or to detect conflicts. If the programmer wants to be informed about other programmers' changes, he may change to the change aware mode.

**Change aware mode.** Concurrent changes are always sources for possible conflicts. If for instance two programmers change the same unit of source code (e.g. a "method" in the object-oriented programming paradigm), these changes have to be integrated by including the intentions of both programmers in the source code. The change aware mode helps to avoid parallel changes at related or
same artefacts of the software project by telling all other programmers that the artefact has been changed. Unlike traditional version management systems (e.g. CVS (Price, 2000)), this notification is done at the time of change and not at the time of reintegration. In the change aware mode, programmers are thus aware of the artefacts that are currently modified by other developers.

**Presence level mode.** The change level mode records only the modifications that programmers do to the project’s artefacts. In addition to this, the presence level mode also tracks activities that do not modify artefacts. Viewing artefacts is such an activity that can be very important for understanding the programmer’s changes, because it reveals what knowledge led her to the change.

**Presence aware mode.** If the non-modifying activities are recorded, it is possible to display this information to all other programmers using a small presence indicator in front of the artefact (if the artefact is visible to them). The presence indicator tells the programmer that there is someone else around and thus helps to find a colleague with a comparable focus on the software system. By focus we mean the set of artefacts, which a programmer is currently inspecting or modifying. If another programmer is viewing the same or a related artefact (i.e. he has the same or a comparable focus), it is likely that he tries to solve a related problem. When detecting another person, there are two possible reactions: The programmer who detected another developer nearby moves away from this person so that he can act alone, or the two people start tighter collaboration by switching to communication mode.

**Communication mode.** Tighter collaboration starts mainly by discussing the circumstances that brought the participants together. In this phase, each of the participants tells the group about their aims and the group tries to formulate a common goal. Without a common goal the group will hardly be able to act as a group. So, this is an important phase when switching to a tighter mode of collaboration. Communication tools, such as a chat tool or electronic mail, assist the participants in this phase. If a common goal is found, or if the group had a common goal even without discussing it (because it was discussed in a previous session), the group might switch to the tightly-coupled collaboration mode.

**Tightly-coupled collaboration mode.** When programmers work in the tightly-coupled collaboration mode, they allow others to share their workspaces. One example is a collaborative class browser, where programmers can browse the code and write new units of source code together, as if they would sit in front of the same screen.

The reader should note that only the information indicated above is recorded in a specific MoC. Information that is not captured in a MoC is therefore not available for future usage. On the other hand, additional monitoring can be performed by underlying tools (such as a version management system) to ensure that results can be merged after periods of work in the offline mode or the process level mode.
Transitions between Modes of Collaboration

With each of the MoCs, as they were described in the former section, the programmer reveals a part of her privacy and she has to provide information to the system. This causes additional work-load for the programmer or her computer system. If seen from this perspective, revealing information provides no direct benefit for the programmer. Allowing others to interrupt the own work is also regrettable, if no personal advantage can be achieved with this action.

Evaluations of other groupware systems (e.g. Grudin, 1988) have shown that one important factor for the acceptance of a groupware system is the balance between efforts and gains for each individual user that are implied by the system.

TUKAN tries to address this problem by introducing rules that define how much a user has to contribute to the system, if he wants to benefit from it. Simply spoken, a user only gets as much information from the system as he is willing to provide to others. On the other hand, the user may decide to reveal more information than he is consuming. If he, for instance, works in the presence level mode, he decided to inform all other users about his present activities and about the artefacts he is currently interested in. But this does not necessarily mean that he is interested in the focus of his colleagues or the changes that they performed.

Figure 1 illustrates the rules. The middle part shows the possible MoCs. Each MoC (except the change aware, the presence aware, and the tightly-coupled collaboration mode) serves as a prerequisite for another MoC, which is indicated by the thin headed arrows in figure 1. If a user works in a specific MoC, he automatically provides all the information of all its prerequisite MoCs. Transitions
between MoCs usually take place following the prerequisite chain. If a user for instance works in the presence level MoC and he wants to initiate a tightly coupled collaboration, he switches to this MoC passing the communication MoC (and thus fulfilling all the duties for this MoC). Anyhow, the process of passing a MoC can be invisible to the user. The change aware MoC and the presence aware MoC do not serve as a prerequisite for another MoC, as they do not state any duties for the local user. Even though, the user may change his MoC from the change aware MoC to the presence level MoC and he may change from the presence aware MoC to the communication MoC (shown by arrows with filled heads in fig. 1).

The left part of the figure states the duties, which are linked to the MoCs. Duties imply a fair contribution to the system before a benefit is granted. When the duty is fulfilled it is ensured that all other users may get the same kind of information about the requesting user, as the information that this user is interested in.

The right side of figure 1 shows the allowed activities and the accessible information. These are for instance the provision of information about possible conflicts (in the change aware mode) or the right to initiate collaborative programming sessions (when working in the tightly-coupled collaboration mode).

If a user, for instance, wants to be informed about other users’ presence, he selects to work in the presence aware mode. This implies that he has to provide all information demanded by the presence level mode to the system. He thus has to inform the system about his current focus, the artefacts that he is modifying, and the task, which is the context for his activities. He may also browse other user’s tasks or activate the change aware mode, although the change aware mode is not a prerequisite for the activation of the presence aware mode.

While all MoCs up to the presence level mode (resp. the presence aware mode) primarily provide awareness on other team members’ activities, the communication mode and the tightly-coupled collaboration mode provide means for negotiation future changes and resolving conflicts or exploiting synergies. This improves the collaborative processes since coordination is facilitated. The reader should note that the MoCs do not prescribe any negotiation or coordination procedure.

It is possible that different users select different MoCs. For instance, one user can work in the presence level mode, while a second user works in the tightly-coupled collaboration mode. If the second user detects the presence of the first, he might want to initiate communication or tightly-coupled collaboration with the other user. But this is not possible, since the other user decided not to communicate or collaborate in a tightly-coupled way. The system tries to meet both users needs by detecting the highest possible degree of collaboration between both users. In the case of the above example, there is no synchronous communication channel available in the first user’s profile. Thus, the only way to get in contact would be asynchronous e-mail communication. If the second user, who works in the tightly-coupled collaboration mode, tries to initiate a chat session, the system
Figure 2: The user dialog

states that this MoC is not available for the first user and comes up with a mail client interface, where the second user can enter his question.

Switching between MoCs can be done automatically or manually. The first alternative eases tool usage because it does not require any explicit user action, whereas the second possibility ensures that the user always knows his current MoC. In this case, he explicitly controls the way that he wants to collaborate.

**Changing MoCs automatically:** Within each MoC user interface elements exist that allow a strengthening of the collaboration. For example, if a programmer works in the presence aware mode, she can easily establish a chat connection to other users. Therefore, she activates a collaboration info dialog that informs her about other the other users who work next to her. She then selects the other user and presses a ‘chat’-button.

**Changing MoCs manually:** The user may adjust her MoC using her user dialog (as it is shown in figure 2). The slider control on the left side of the window is used to adjust the amount of information provided by this user (they correspond to the elements that appeared in the left column of figure 1). The amount of information that is displayed about other user’s activities can be controlled by the two checkboxes right to the slider. Other users can always inspect their colleagues state by looking at the colleague’s profile in the user dialog.

If the user decides to change her MoC from a tight mode to a more loose mode, the system will close all tools that are not allowed in the new mode. One user might for example change from communication mode to presence aware mode. All open chat connections would then be closed automatically.
TUKAN and the Software Space

We implemented the different collaboration modes in the co-operative programming environment TUKAN. TUKAN was built using the open source groupware framework COAST (OpenCoast, 2001; Schuckmann et al., 1996) and the version management system ENVY for VisualWorks Smalltalk (Cincom, 2001).

Depending on the desired strength of collaboration, TUKAN provides different tools to the user. All tools share the metaphor of software space. In this section, we will first give a brief explanation of the software space (a more detailed description is given by Schümmer (2001)) followed by a description of how the software space can be turned into a shared workspace.

The Software Space

We interpret the artefacts (such as classes or methods of an object-oriented program), which are produced in the software project as semantic networks, which form a hyperspace. Each artefact is mapped to a node of a graph. Whenever a programmer creates an artefact, this is added to the graph and the system scans the artefact for possible relations to other artefacts, which are already known in the graph. The relations are then represented as weighted edges in the graph. A spatial layout is determined by the presence of an edge with a specific weight between the two nodes, which controls the distance in the space. Two semantically related artefacts are thus nearer together in the software space, as it would be the case if they were not related.

The Collaborative Software Space

During software development, the programmers work with a set of artefacts. This set forms the programmer's current focus. In a collaborative scenario, the focus can be of importance for other users and overlapping foci are important awareness clues for possible tightly-coupled collaboration.

Benford and Fahlen (1993) have analysed the application of awareness in 3D environments. In the Massive system, these ideas were realised and refined Greenhalgh and Benford (1997), where worlds form spaces for communication. For each observing object (each user), they define a focus as the set of the object's current interests. The nimbus "represents an observed object's interests to be seen in a given medium." (Greenhalgh and Benford (1997)) In general, the nimbus contains all objects within a geometric area around the focused objects. By combining two user's foci and nimbi, MASSIVE calculates the awareness strength that they have of one another.

Rodden (1996) has proposed a generalisation of this awareness model for the work on artefacts, which are arranged in a spatial graph structure. He defined the
focus as the set of currently focused nodes and calculated the nimbus as the combination of all focused nodes’ adjacent nodes within a well-defined distance.

The software space is such a graph. We define the focus as the set of artefacts (the methods, classes or applications) that a programmer is currently looking at. The nimbus consists of all artefacts in the software space, which are semantically related to the focused artefacts (i.e. they are nearby).

If the user works in the presence aware mode and the artefact, which is shown on his screen, is part of another user’s nimbus, the system displays a presence indicator in front of this artefact. It is a small coloured figure with a colour ranging from red to green. The nearer the artefact is to the user’s focus, the more red colour is used to display the figure.

The left part of figure 3 shows an example for the calculation of presence indicators. Another user is working on the method \textit{hash}. The method \textit{year} is directly related to \textit{hash}, since it is used by \textit{hash}. On the other hand, the method < uses \textit{year}, \textit{month}, and \textit{day}. These semantic relations form the basis for the layout of the software space. The calculation of the colours for the presence indicators leads to the example shown on the left side of figure 3 (note that the figure only shows a small part of the software space and its relations).

Besides the presence aware mode, TUKAN has a second MoC where awareness information is calculated using the concepts of focus and nimbus: the change aware mode. We introduced a second definition of focus for this mode. The \textit{change focus} is defined by the set of artefacts for which a newer version was created by another user. Artefacts, which are part of a change focus, are interpreted as sources for possible conflicts. The nimbus of a change focus is calculated in the same way as it was presented for the calculation of the presence indicators. If an artefact is part of the nimbus of a change focus, this artefact may be affected by the change.

An example for the calculation of conflict indicators is shown in the right part of figure 3. We use a weather metaphor to indicate possible conflicts. In the example, another user has modified the method \textit{hash}. A heavy lightning symbol is used to indicate this fact. The methods \textit{year} and < are very near to this changed artefact, thus a lightning symbol is shown in front of them. The further away arte-
facts are from a possible conflict, the better is the weather indicated by the symbol shown in front of it.

Working in the Collaborative Software Space

After we presented the general metaphor of TUKAN’s collaborative software space, we will now use a scenario to describe how the different tools help working in the software space. For reasons of simplicity, we will describe all uses of collaborative tools for the case of two users. Of course, the tools work also for larger groups.

Assume that there are three users, who are currently working with the system: Alice, Bob and Charlie. Alice and Bob are working on an accounting application, which has two stories to solve: it has to be adapted to the new European currency (the Euro), and it has to be checked against any bugs that might occur at the beginning of the new millennium (Y3K-bugs). The project manager has identified these two stories and entered them in the planning tool.

Charlie has decided to do some optimisations of the hash functions in the class library. This is a task, which is not planned by the management. Thus, he decides to work in the offline mode.

Planning the Work

Alice and Bob both logged on to the system and look at the stories in the planning tool (Alice’s display is shown in figure 4). They start their work in the process level mode. Alice modifies the tasks, which are associated with the Euro-story. At one point, she does not know how to name a new subtask. She thus decides to invite Bob to the planning tool. Therefore, she changes her MoC to the tightly-coupled collaboration mode and invites Bob. But unfortunately, Bob is currently working in the planning level mode and the system proposes to invite Bob by e-mail.

After Bob received the mail, he changes his MoC to the tightly-coupled mode. The system detects that Bob is now willing to co-operate and adds Bob to the user’s of Alice’s planning tool. They now share the selection of the planned tasks and can modify the tasks and stories co-operatively (cf. figure 4, showing Bob and Alice as co-operating users).

After they finished describing the tasks, they end their tightly coupled session (Bob closes the shared browser and returns to his private browser again). They have a look at the tasks and both select the task they want to work on. This is expressed by adding their user representation to the task (they press the ‘add user’-button and select their name). When they start actual work, they have to choose one of the tasks, where they contribute and do the work to reach the task’s goal. They launch this task by pressing the ‘start task’-button.
If the user works in the change level mode, all accesses to artefacts of the software space are recorded in this task (they appear in the activities list in figure 4). The name of the selected task is also displayed in the active browsers (cf. figure 5). Programmers are thus always aware of their own task, which helps them to keep their work focused.

Alice chooses the task ‘Redesign interface’, and Bob decides to work on the task ‘Currency conversion’. Since Alice and Bob noticed in their discussion that they work on related tasks, they decide to do their work in the presence aware mode. This keeps them informed, if they do some work with mutual consequences.

Doing the Work

Alice and Bob select the co-operation aware class browser to do the implementation, as it is shown in figure 5. This browser is a slightly adapted REFACTORING BROWSER (Roberts, 1999). In addition to the programming source code, it shows the local user’s current task and awareness icons in front of the method list, which provide information about other user’s activities (depending on the MoC).

The first awareness information is the visualisation of possible conflicts (provided by the change aware mode). The calculation follows the algorithm as it was explained in the section ‘The Collaborative Software Space’. In the screenshot,
Figure 5: The co-operation aware class browser.

there is a lightning symbol in front of the method `year`. This demands caution when using or modifying this method, because another user changed a method, which is semantically related to the method `year`. Any changes on `year` might thus be incompatible with the changes that other programmers applied to the method `hash` (actually, this was the method changed by Bob).

Alice does not like this uncertainty and decides to load the newer version of the method `hash` into her workspace. This has the effect that all related methods are now up to date and only sun symbols are shown in front of the methods.

Whenever Alice selects a method, the system monitors a change of focus. This change of focus triggers a re-computation of the awareness indicators. In figure 5, Bob is working at the method `hash`. Therefore, `year`, `month`, and `day` are shown with little figures in front of it indicating that there is another user nearby.

When Alice notices that there is another user nearby, she may continue working, as if nobody was nearby. She may also feel the need to coordinate her activities and therefore she may decide to communicate with Bob about her concerns using the communication level MoC. As a result they may decide to switch to a tighter collaboration mode and meet in a tightly-coupled collaborative class browser. The next section will provide more details about how they actually meet.

Switching to the tightly-coupled mode of collaboration

The artefact that will form the focus of the collaborative session may differ according to the Alice’s intention. If she is more interested in the work of her colleague, she will prefer to join him in his current work context (open a browser on the method `hash`). On the other hand, if she wants to inform Bob about her current work, she will prefer to invite him to her browser. Bob will then receive an invitation stating the person, who invited him (Alice), her current task, and the artefact, on which the collaborative session will focus.
Figure 6: The collaborative class browser.

When accepting the invitation, Alice and Bob can choose how to reach one another. The fastest way is to directly meet at the artefact. Because of the rapid movement, we call this possibility warping to the collaboration. This will open a collaborative class browser (cf. figure 6), where the participants share the selection of the current artefact and can manipulate this artefact using a collaborative text editor widget. Each user has a personal cursor, which is used to manipulate the source code. The personalised text cursors can also be used to point at specific parts of the text and thus serve as a telecursor (Hayne et al., 1993) to focus communication.

The second way of reaching the artefact for collaboration is by following the relations of the software space from the current focus to the new focus. In contrast to warping, we call this movement walking to the collaboration. Walking is done by first opening a browser on the currently focused artefact. A small window (which is not shown in Figure 6) informs the user that this is not yet the artefact of collaboration, but an artefact, which is on the way from the current focus to the desired focus. This window contains a control, which navigates the browser one step further on the path to the desired artefact. Upon reaching the artefact for collaboration, the other user joins his browser and they start to collaborate.

The idea of walking to the collaboration helps the person who was currently working on a different focus to understand how the two focuses are related and what they have in common. Both users will then share a common focus, not just in the browser, but also (hopefully) in their minds.

Understanding previous work

Walking to the artefact of collaboration lets the programmer know the shared context between the two programmer’s work. Another kind of walkthrough can help a programmer to understand work, which took place during his absence.
In the scenario, Charlie decides to catch up with the group again, after he finished his work in offline mode. He therefore switches to the presence level mode and activates the planning tool, where he can select a story and get a guided tour through this story (cf. figure 4). A guide shows the stories and tasks in their logical relationship, and for each task all touched artefacts are shown. In addition to the artefacts, the guide shows some context information, such as the name of the programmer that did the activity and the structural context of the artefact itself (e.g. the class for a changed method).

By pressing forward and back buttons, Charlie can easily navigate through the tasks. There are several filters that can make the tour even more interesting: He can focus just on the changed artefacts or include the viewing of artefacts as well. He could also let the guide show him just those activities that took place during his absence.

Compared to other modes of collaboration in figure 1, the guided tour is positioned between the presence level mode and the presence aware mode. Anyhow, it has a different quality, since it provide asynchronous history awareness to the programmers. The guide provides as much information on the tour as it can find. Thus it accumulates all available information, which was provided by users working in the process level mode, the change level mode, or the presence level mode. A guided tour therefore needs at least one user who did some work in the process level mode.

Experiences

Up to now, we used TUKAN in two settings for a period of one week: in our research group for the development of groupware prototypes, and in a company that develops groupware for knowledge management. In this section, we will present the settings and our first findings of the usage of TUKAN in an anecdotal way.

The group in our institute consisted of two full time researchers and one student, who is currently learning how to enhance the TUKAN environment. They used the planning tool for the specification of tasks (about 30 different tasks grouped into 5 stories). The full time researchers did the specifications of the tasks, while the student browsed the tasks and looked for one in which he was interested. At this time, they all worked at the process level mode. When the student decided what task he would like to solve, he switched to the tightly coupled MoC and invited one of the researchers. They discussed the suitability of the selected task and other tasks. At the end of the planning session they had sorted the tasks according their importance and the student started doing the work.

By convention, they decided all to work at least in the presence level MoC. They were working at very diverse parts of the project. This led them to the decision to switch the awareness icons off. At one point of work the student did not know how to get on with the code of a specific method. Therefore, he contacted
one of his colleagues and asked him if he wanted to work in a tightly-coupled session. They both switched their MoCs to the tightly-coupled collaboration mode and started to browse through the class of the method. They discussed different alternatives and made heavy use of telepointers and communication tools (actually, they used a separate audio-connection). After the problem was solved, the student started to work alone again.

Meanwhile, the second researcher finished his task and chose a new one, which was related to the task of the other researcher. The researchers thus decided to turn the awareness indicators on. After turning on the indicators, one of the programmers saw immediately that there were some newer versions of the methods, which he was going to work on. He therefore integrated these methods and used the new versions as the basis of his work. After about 20 minutes of work they reached related points in the software space and decided to continue in tightly-coupled co-operation mode. They figured out that the two problems had the same reasons and found a solution together. This group found TUKAN very useful (especially the student liked the way of learning by contacting experts).

The second setting involved a company that develops knowledge management groupware. They tested TUKAN during the development of a large commercial application. They used it for small stories within the overall development process. The company used the planning game before and they had created a large set of story and task cards. For their use of TUKAN, they transferred only the card’s title to the planning tool. The observed subgroup consisted of 4 people. They chose to work on a set of tasks that were highly related. They divided themselves into pairs that were each responsible for one task.

For programming, they worked in the presence aware mode. Each pair worked together in front of one screen using one user’s name (doing traditional pair programming). There were frequent occurrences of overlapping foci, which were indicated by the presence indicators. In most of the cases of overlapping foci, the two pairs decided to join for a short session and did some programming together (leading to quartet programming).

Most frequently, they used the conflict indicators. Whenever a conflict was detected, the system helped them to detect the conflict and they compared their version with the newer version. In fact, the group was used to search for conflicts manually before, thus they appreciated this feature very much.

Finally, they discovered the guide and used it to reason in front of the other programmers about the work that they did. In their setting, every programmer was sitting in front of his screen (the screens were arranged in a way that allows eye contact between all the programmers) and one programmer controlled the guide. With every displayed change, he provided some comments why he did this. This usually started a vivid discussion.

Both groups scathed the system’s performance, regarding its speed. Since the calculation of focus and nimbus was done on the fly, it took about 3 seconds until
the whole awareness information was refreshed. We have analysed the performance lacks and found some ways to significantly speed up the system. The resolution was to reduce the size of the user's nimbus and add some filters that remove some of the most ambiguous relations from the software space (e.g. the usage of the methods 'new' or '+' is spread around the system, but does not necessarily imply semantic nearness of clients and suppliers). An evaluation of this faster prototype will be part of our future work.

Both groups felt that their performance was improved by the usage of TUKAN and they could imagine to use it in future projects.

As for all systems, which trace users' activities, the fear of being monitored is a crucial point for TUKAN's acceptance. A social protocol and trust needs to be established to exploit the benefits of TUKAN. In our observations, the groups were working together for a long time. Thus, these social protocols already existed and trust was no problem for them.

Related Work

Related work includes general support for collaboration in shared workspace systems and group awareness as well as more software development specific support. We will first cover the groupware specific issues. The latter part of this section will compare TUKAN with synchronous software development tools.

Group awareness (Dourish and Bellotti, 1992) facilitates the assessment of the present state (who is doing what) in a shared workspace. It is usually a concept that is applied to help synchronously co-operating users to co-ordinate their activities. It also supports – to some degree – making informed decisions about what to do next (based on knowledge about who is working in which part of the shared workspace, thus showing some opportunities for synergy or conflict). However, it does not support finding out about past activities (i.e. the history of the work), and it does not explicitly address current tasks and plans of the group. Most systems in this area provide what is called local awareness by Haake (1999). Local awareness relates to information about ongoing activities in the current workspace. Examples for local awareness are applications implemented in GROUPKit (Roseman and Greenberg, 1992) and SUITE (Dewan and Choudhary, 1991) as well as SEPIA (Haake and Wilson 1992; Streitz et al., 1992), TEAMROOMS (Roseman and Greenberg 96). Some other systems also provided tools for global awareness (i.e. giving information about the general activities in a complete shared workspace). Examples include task lists as in workflow management systems, radar views as in SEPIA, or history logs as in NOTECARDS (Trigg et al., 1986). BSCW (Bentley et al., 1997) provides workspace awareness through a user presence and activity monitor since version 3.2. ORBIT (Mansfield et al., 1997) organizes a shared workspace using the locales concept (as a means to communicate tailored awareness) and supports global awareness on other locales through a "navigator". An-
other interesting approach is to provide activity awareness between different individual workspaces as in the INTERLOCUS system (Nomura et al., 1998). Here, notifications and awareness functions provide asynchronous workspace awareness. However, synchronous awareness and assessment of future activities are not supported. In summary it can be said that with the exception of SEPIA no shared workspace system explicitly supported different modes of collaboration and transitions among them. Even in the SEPIA system, only three modes of collaboration were explicitly supported (and these were aimed on collaborative writing). Most systems do offer different forms of awareness, but none aims explicitly at providing awareness tailored to software development. In addition, asynchronous and synchronous collaboration are rarely equally well supported.

Coordination systems help a group in the definition and enactment of the group process. For instance, XCHIPS (Wang et al., 2001) is a shared workspace for creation and execution of emerging work process descriptions. Several programming environments followed the basic approach of guiding the programmers through the process. Weinreich and Altmann (1997) presented an environment, in which the programmers can define work packages and see the related software artefacts. It provides communication tools and change notifications. SPADE (Bandinelli et al., 1996) helps the programmers to model the software process and define when synchronous co-operation should take place. The actual co-operation took place using simple shared editing and communication tools. While all the process centred environments focus on the explicit definition process, TUKAN handles the process in a more implicit way, because collaboration is not process driven, but artefact driven.

There are some systems, which bring the artefacts and the process together in a virtual environment. Promo (Doppke et al., 1998) maps tasks of software development to rooms of a MOO (Curtis, 1997). The rooms contain the artefacts that have to be manipulated in the task. Rooms can be connected and thus guide the programmers in the execution of the software development process. The mentioned ORBIT system allows a programmer to have more than one presence position in different 'locales'. Artefacts can be placed in (possibly multiple) locales, where the locales' inhabitants can edit them. Awareness clues show what artefacts other users currently work on.

Christensen (1998) presented the RAGNAROK environment that maps software design to landmarks of shared design landscapes. Users interact with the artefacts by opening them on the design landscape. A colour coding approach shows where other users are working. Within CHIME (Dossik and Kaiser 1999), programmers can walk through a graphical virtual environment that consists out of the projects artefacts and meet other programmers to communicate via a chat tool. Users can specify how the artefacts are related. CHIME places the software artefacts in a three-dimensional environment depending on the relations between them. All discussed systems that are based on the metaphor of virtual worlds provide no or
just synchronous awareness. None of the systems has a conflict awareness mode, or explicit control of the mode of collaboration. Except of the virtual worlds in CHIME, all the virtual worlds require the user to design their layout manually.

Marshall and Irish (1989) presented the idea of guided tours as a means for guiding the reader through a hypermedia graph. These tours had to be created manually and were static at the time. Riedl (2001) presented the TRAILGUIDE system, which monitors browsing activities and allows authors to publish the recorded trails as guided tours. An avatar accompanies the user and provides additional comments on the visited pages. We do not know of a system that generates guided tours based on the navigation activities of other users, as was presented in this paper to aid comprehension of activities of other developers.

Conclusions and Future Work

In this paper, we presented a set of different modes of collaboration (MoCs) that support the cooperative development of software systems. We introduced constraints and transitions between the MoCs that ensure a fair balance between efforts and gains that are implied by the system.

TUKAN is our first environment that supports the different MoCs and eases the transition between them. It combines different aspects of shared workspaces with a spatial awareness model and provides collaboration tools for each MoC. Collaboration ranges from offline (asynchronous) collaboration up to tightly coupled code editing. First users rated TUKAN to be very supportive for their software development.

First experiences encouraged us to enhance the system regarding its performance and the automatic transition between MoCs. Another issue of future work is the generalisation of the concept of MoCs. One could for instance interpret a hypertext as a graph in the sense of TUKAN. Browsing the hypertext would then be matched to activities in the graph. We will examine if and how the concept can be applied to other artefact based collaborative activities (e.g. collaborative shopping in the WWW or collaborative work in virtual enterprises).

Acknowledgments

We thank our colleagues at CONCERT, especially the COAST development team. Special thanks are due to Alejandro Fernandez, Shirley Holst, Holger Kleinsorgen, Christian Schuckmann, and Jan Schümmer for the discussion of TUKAN's concepts and comments on this paper. Torsten Holmer, Ruediger Pfister, and Martin Wessner came up with the concept of a PoC. We also thank the reviewers for providing constructive comments, which helped to improve the paper.

The DFG supported the work of Till Schummer as part of the PhD program „Enabling Technologies for Electronic Commerce“ at Darmstadt University of Technology.
Bibliography


