Pair Programming Interfaces and Research

Roland Doepke
RWTH Aachen
52056 Aachen, Germany
roland.doepke@rwth-aachen.de

Markus Soworka
RWTH Aachen
52056 Aachen, Germany
markus.soworka@rwth-aachen.de

ABSTRACT
Pair programming is an aspect of Extreme Programming where two software programmers work together at one computer on the same code. We give an overview of current research on pair programming and its benefits over individual programming.
We will see that pair programming is an effective way of programming while the role of the navigator stays unclear. As distance collaboration is mandatory due to globally distributed organizations we also consider distributed pair programming. However, this makes user interfaces necessary which permit that geographically distributed developers may work together over the internet as if they were using one computer.
We will present several attempts for this, namely Sangam, CollabVS, Facetop and CodeGraffiti. Sangam and CollabVS are IDE input-based solutions, whereas Facetop is a highly video dependent approach on distributed collaboration. The fourth approach, CodeGraffiti, enhances collocated pair programming with handwritten code comments and sketches.
We will observe that Facetop overloads the user with unnecessary information, whereas IDE Solutions lack extendability.

Author Keywords
Pair programming, distributed collaboration, user interfaces

ACM Classification Keywords
H.5.3 Information Interfaces and Presentation: Group and Organization Interfaces

INTRODUCTION

Explaining pair programming is simple: It is a style of programming where two programmers are simultaneously working on the same piece of code at the same computer. One of them, the *driver*, has full control over the integrated development environment (IDE), while the other, the *navigator*, observes the work. These roles frequently switch during the programming session, giving the navigator control over mouse and keyboard and vice versa. That's it.
But this raises questions as well:

- How efficient is the work of two programmers doing "the work of one" at the same time?
- How does the navigator contribute to progress?
- Why it is so successful?
- And lastly, as project teams are getting more and more spread around the globe: Does this work with partners who do not sit directly next to each other at one computer, as well?

We will try to answer these questions within the paper, but first of all we have to get more into detail about pair programming.
Pair programming is mainly used as a practice of Extreme Programming [3]. But as it turned out to be working well, it was also used in other development methodologies—while other aspects of Extreme Programming stay highly debated or simply can not efficiently be used alone. Extreme Programming is a style of programming where a project is divided into many small tasks, with each of them having at least one test case, that are written before a single line of program code is written. Commonly these tasks are written on index cards. The so-called user stories on the cards are translated into test cases—basically program code which automatically tests other code for correctness.
After this, the programmer begins to write code to handle these test cases—which then can be checked automatically. The project is done, if no test case fails any more.
As this is the approach for a single programmer, for multiple developers it gets vastly more difficult. Models and test cases have to be developed in a team which makes communication between all participants the main issue. And how could communication with someone be better than interacting with someone at all stages of development? This is the point where pair programming comes in: apart from meetings with the whole team, programmers work in teams of two—but switch their team partner within the team at regular intervals.
According to "Brook's Law", adding more people to a software project does not necessarily decrease its development time [4]. That is why one could think that pair programming does not speed up the development process and two programmers need twice as much effective time as one programmer. Only one of them is working and the other is wasting time watching him. But programmers who tried pair

programming felt that this is not the case and several studies substantiated this. We will focus on this in the second chapter.
Assuming that pair programming, as it is now, is an effective way of programming—integrated development environments (IDEs) usually are not intended to work with more than one user at once. You could suppose that IDEs solely made for this purpose would work even better. That is why we will look at some user interfaces which try to help programmer pairs with their work in the third chapter.
Lastly we will compare and judge these and give suggestions on improvements.

MAKING THE CASE FOR PAIR PROGRAMMING

Reading papers about pair programming means reading anecdotes about pair programming. Any paper we investigated contained own experiences where the authors proudly presented how positive their experiences with pair programming were or how well their test persons did with pair programming. As this might be not a very scientific way of investigation it shows at least how inspiring pair programming is.
According to Cockburn and Williams [7] we divide into eight aspects of software engineering and organizational effectiveness aligned with pair programming:

- *Economics.* To proof pair programming is not less effective than individual programming, Williams et al. [20] investigated a software engineering course at the University of Utah, where one part of the class coded class projects by themselves, as they had for years and the other part of the students completed their projects with a collaborative partner. At first they noticed an increase of 15% more coding time on the initial program. One of the advantages of extreme programming is that code can be automatically tested with test cases. Thus, making it easy to systematically check code for its correctness. With the course instructor's test cases it was shown that pair programmed code passed more tests than the code written by individuals. The resulting code had about 15% fewer defects, statistically significant. Other, earlier studies came to even better results for pair programming [17].
  Taking into account industry data about costs of software defects [13], it is almost obvious why a little more coding time on initial code with fewer errors is better than fixing these errors later on. Cockburn and Williams demonstrated this for a program of 50,000 lines of code [7]: It would use 15 times more time to program and to fix defects in a quality assurance department, given a conservative value of 10 hours per defect. Fixing these bugs "on the field" would take even 60 times more time—calculated with a conservative factor of 40 hours per defect.
  This doubtlessly shows, that pair programming does not double the time needed.

- *Satisfaction.* Another plus is the enjoyability of pair programming which leads to higher acceptance. Initially many programmers are skeptical, because "it takes the conditioned solitary programmer out of his comfort zone". But pair programming teams reported that they "enjoyed pair programming more"—and, additionally, that "they were more confident in their programs than when they programmed alone" [7].

- *Design quality.* Written code not only had fewer defects, they also had fewer lines of code with same functionality. While it would be an interesting challenge to show if shorter code means better design it is rather difficult to evaluate design quality with objective measurements. But fewer lines of code at least mean fewer lines of code to maintain. That is why pair programmed code seems to be of better quality and to have lower probability of errors.
  This is not surprising because different people bring different experiences and prefer different courses of action [9]. Exploring a larger number of alternatives than a single programmer alone "reduces the chances of selecting a bad plan".

- *Continuous Reviews.* Inspections are a cost-effective way of removing defects from software. As we explained earlier, fixing bugs on the field means much higher costs. However, inspections are found not enjoyable or satisfying by the majority of programmers. As a result they are often not done or are held with unprepared inspectors [7]. During pair programming a continuous review takes place. Additionally, the reviewer is prepared as good as possible. Mistakes are found as they entered, "saving the cost of compilation" and "coding standards are followed more accurately with the peer pressure to do so".

- *Problem solving.* Sooner or later programmers will come to the point where something does not work as intended. Pair programmers are able to solve these unexpected errors faster than individual programmers. Cockburn and Williams refer to "pair relaying" [7]: By "contributing their knowledge to the best of their abilities", programmers are able to share their knowledge and energy.

- *Learning.* Each programmer has different skills in programming or designing. In pair programming the two programmers work in one "line of sight" [14], so they learn form each other by watching and listening to the other person. This is especially useful if one of the programmers is a novice. By sharing the same workspace, the novice absorbs the experts knowledge. Furthermore, learning with a partner is easier and more effective.

- *Team Building and Communication.* Increased team togetherness is commonly—however an anecdotal evidence—observed in pair programming teams. People learn to work together. But they need to get used to work with a partner. Advising them to think out loud can do the trick: One of Cockburn's interview partners explained how a random collection of several software engineers grew together as a team solely with pair programming [7].

- *Staff and Project Management.* A key number to estimate project risk is the so called truck number: "How many or few people would have to be hit by a truck (or quit) before the project is incapacitated?" Here, the worst value is one. Spreading knowledge within the team by rotating

pair partners does reduce the risk. That is why the project team benefits from the increased learning due to pair programming.

Role of the Navigator

Complex software must be developed collaboratively, but it is shown that adding more heads to a project does not necessarily decrease its development time [4]. But working in teams of two does not seem to hurt.

To get behind the reasons why pair programming is that successful, it was observed how the navigator contributed to the code generation. The navigator seems to have a much more objective point of view. Bryant et al. give a wide overview of previous works in their paper "Pair Programming and the Mysterious Role of the Navigator" [5]. First of all, working in a pair encourages a programmer to talk. Talking about problems to oneself can help achieving a better understanding of a problem [6]. Anecdotally, many programmers stress they achieved an 'eureka' moment by explaining a difficult problem to somebody else. Ainsworth et al. refer to this effect as getting rid of 'cognitive off-load' [1]. By verbalization the programmer frees up 'working memory' which allows him to think further.

Another possible explanation would be the separation of a task into subtasks. This way, a programmer would have to think only about a subtask and thus being concerned only with a subset of complexity. But earlier studies showed that this happens only to a small extend.

However, the authors inspiring this chapter concentrated on the actual role of the navigator. Thereby they tried to categorize the navigator as a reviewer, catching syntax and spelling errors or as a foreman, working at higher levels of abstraction. The authors audio recorded pair programming sessions of commercial pair programmers, counted utterances and sorted these utterances of the navigators into classes. These were classified into six different levels of abstraction. According to the authors, syntax is the lowest level of abstraction, such as correcting punctuating marks. Links to the real world, where the programs discuss a real world problem connected to the program code, would be the highest level of abstraction. There is also an additional 'vague' classification for utterances where the level of abstraction could not be ascertained. Figure 1 shows the classification table.

The findings were analyzed with statistical methods and the comparison between driver and navigator surprises: At all levels of abstraction utterances are evenly distributed between the pair. This implies that both work at similar levels of abstraction and makes the pair programmers a 'Tag Team' which shares the additional cognitive load of typing—in contrast to earlier assumptions that the navigator would have the role of a 'reviewer' or a 'foreman'.

The study also reveals a proliferation of utterances at the 'program block' level, as figure 2 indicates. The 'program block' level describes utterances related to structures of the program such as case-blocks, databases, subsystems or libraries. Bryant et al. refer to future work but form hypotheses why this could be the case:

- The 'program block' level provides the driver with the information he needs to keep a clear head and stay in focus for the whole task.
- Pair programming keeps the navigator up to speed with the driver in preparation of a future role change. When the navigator becomes a driver—because the old driver needs a break or the former navigator is better informed in the current programming task—the role change can be as fluid as possible. That is why no time is wasted for explaining the status of the code. The 'program block' level provides a "missing level of information" which is not yet covered by the IDE and provides "the 'glue' that holds together the upper and lower levels of abstraction". Therefore it would help the navigator to get information what is happening at this part of the code "to take over in a fluid manner".

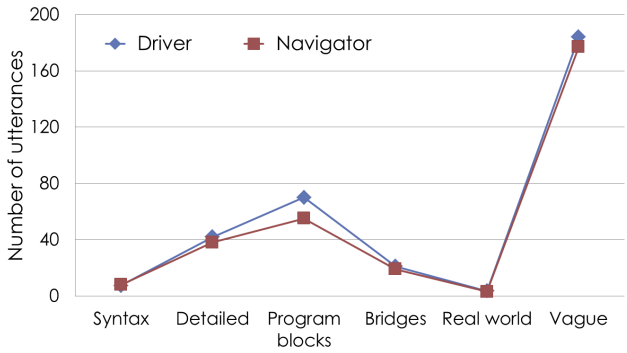

Figure 2. Average number of utterances of each level per session for each role [5]

However, studies on pair programming are limited in various ways. Bryant et al. already mentioned the most frequently voiced objections in their paper.

At first the availability of test persons is a problem. These can only be gathered within the academic and teaching sector or by voluntary companies. Students have a different pair programming behavior than commercial programmers due to less experience. Observing commercial programmers is purely opportunistic and the pairs observed are those who are happy to be watched or maybe those who want to get behind why it is not working.

Another limitation is the method of data collection. Not all information between a collocated pair can be gathered. People communicate in many "subtle means" which cannot be tracked by sound, video and screen information—and Bryant et al. were limited to audio only. On top of this, due to the sensitive nature of commercial projects, video and screen recording is often not possible. This leads to problems. In context of this paper this means the proper classification of utterances, which lead to having 'vague' classification contain by far most of the utterances. In our opinion this fact questions the findings of Bryant et al. : the number of 'vague' utterances include all the utterances where the authors were unable to classify the statement in one of the levels of abstraction. It is unclear in which proportion the 'vague' level contains utterances of agreement or disagreement, such as "ok", in contrast to utterances which possibly could fit into one of the other categories. Taking just the half

Code	Explanation	Examples
Syntax	Spelling or grammar of the program. Spelling is indicated in the transcription by single letter capitals. NOT semantics.	Spelling, dot, F9, 7.
Detailed	References to operations and variables in the program. A method, attribute or object which may or may not be referred to by name	That's not actually part of the array that we want. If we have this short description field.
Program Blocks	Blocks of the program. Including tests and abstract coding concepts. Also strategy relating to the program and its structure. General naming standards discussions etc. This could also include cases where the subject of the sentence refers to 'some of them' or 'they all' – i.e. a group of conditions. Anything to do with refactoring. Subsystems or libraries. Directories or paths, even if named.	We did the content test cases in a similar sort of way. I think that before we decommission the database we should take a snapshot of it.
Bridge	The statement bridges or jumps between the real world or problem domain and the programming domain. This may be where a case or condition exists in the code and the real world.	So we need to add a test condition here, to see if the the bank account is valid for this kind of transaction. How can the feed come in earlier than the trade?
Real World	Real world problem domain.	The user might genuinely put his phone down....he wanders away, has a bath, comes back... Just before the time that we set up we call and ask him "can i start now".
Vague	Including metacognitive statements and questions about progress or understanding. References to the development environment and/or navigating its menu structure. Utterances where the level of abstraction cannot be ascertained.	I know where you're coming from. Oh yeah, I see, that bit at the top.

Figure 1. Classification table based on Bryant et al. [5]

of the 'vague' classified utterances and classifying these to the smallest class, the 'real world' domain, would be enough to make it the biggest classification, which could mess up the results.

Distributed Pair Programming

Usually pair programming is performed by collocated programmers. On the contrary distributed collaboration is getting more important, teams consisting of people in Europe working with people in America or Asia become more common.

A study by Herbsleb et al. of the year 2000 gave out questionnaires over several years, to several sites of the Bell Labs—most of them located in Europe (UK and Germany) and some of them located in India. All sites are involved in software engineering in the same project. The questionnaire asked, among other things, how many days it took until a request to incorporate a specific functionality into the software would take until it is implemented, dependent on where it was developed. The shocking result of this was that it took about 5 days if the specific modification request was developed within the own site, and more than twice as long, namely 12.7 days, if developed across sites [11]. This presents evidence that "speed presents a challenge indeed in multi-site work. Diminished communication across distance and the loss of the subtle modes of face-to-face communication and coordination that collocated work affords, appear to have rather dramatic and unfortunate consequences". Another interesting finding of the study was that it was difficult to find the right people across sites.

To enable pair programming over distance, it is necessary to provide a workspace that offers tools for communication between driver and navigator. In the best case programmers can hear and see each other. Both programmers need to have a shared view on the code and at least the driver should have the possibility to edit the code. Under these conditions it is possible to do pair programming. But how effective is this technique of programming when the programmers are geographically distributed?

To achieve insight into this, Baheti et al. conducted an experiment in 2001 in a graduate class with 132 students who had to do programming projects [2]. 34 of these were distance learning students, who had a vested interest in doing distributed pair programming. The students were divided into teams of two to four people, each team attached to one of four groups: Collocated team without pairs, collocated team with pairs, distributed team without pairs and distributed team with pairs. The distributed teams with pairs worked on a shared desktop via VNC, communicating with a microphone and an instant messenger. This way they had to work simultaneously with the pair programming method. To compare the work between the different groups, the students used a web-based software-process analysis system. Over the course the research team supervised the student's recorded metrics to make sure that the recorded data was accurate. The analyzed metrics were productivity, measured in lines of code per hour, and quality, measured in the final grades the students achieved for the project. The results

show that the "collocated pairs for this experiment were not more productive (statistically) than distributed pairs". In addition it was found that distributed pairs were as efficient as collocated pairs regarding the quality of their work. After the project the students were asked to give feedback on the communication with their team and cooperation within their team. An interesting result is that the students cooperation was best rated in distributed paired teams, followed by the collocated teams with pairs. Even the communication is best in distributed teams with pairs, according to the survey.
The results show that distributed pair programming was not less effective than collocated pair programming in this case. Communication and teamwork do not necessarily have to be poor because of the geographical distribution—given there is direct communication possible.
Considering this insight, advantages of distributed pair programming over collocated pair programming become more valuable [19]:

- Visibility is much higher, because each programmer has his own screen. Thus having more space for interface elements one programmer uses and the other does not.
- The navigator has the possibility to use his PC to search the Web for resources while the other continues writing code.
- Time can be saved, because there is no need for traveling to meet the partner.
- With the right equipment, meetings are also possible when the persons are not at home, for example when they are on a business trip.
- All the work is done electronically instead of using paper or whiteboards. This takes more time as a downside, but, as everything is digital, copies and records are available for all versions of the work which might not been considered valuable enough to preserve at first sight.

On the other hand, distributed pair programming has some disadvantages as well:

- It is hard to point on the code, so the programmers have to describe where the problem is what takes a lot of time. Many tools for distributed collaboration still do not support this.
- If there is a problem with one computer, both have to stop working.
- While communicating the team members can not see facial expressions of the partner, because webcam videos are too small, have limited frame rates and are expensive in bandwidth.
- Other people may not know that the programmer is in a distributed pair programming session as it is not common sitting in front a PC but doing collaborative work. Other people could interrupt them with starting a conversation, distracting the programmer from working.
- Programmers have to get used to distributed collaboration to work efficient. They need to get used to the provided tools and get aware of their limitations. This is often the case in VNC based solutions. The driver might switch screens or scroll that fast that the navigator's video transmission cannot keep up.
- Caused by the physical distance of the programmers, they have to make time-consuming explanations that could be easy done by visual diagrams if the programmers were collocated.

To get rid of some of these issues there are some tools that try to solve the problems in distributed pair programming, enhancing simple desktop sharing applications. These need to fill the requirement of supporting all actions, which make collocated programming efficient and eliminate as many disadvantages of distributed pair programming as possible. In the next chapter we will present some of them.

USER INTERFACES FOR PAIR PROGRAMMING

The following tools are intended to be used for distributed collaboration: To begin with you have to distinguish between asynchronous and synchronous collaboration. Asynchronous collaboration means that the team works at different times and integrate their work after they have completed their task. This can be done by uploading with a version system like SVN[1] to a centralized server. Pair programming obviously belongs to the other category, synchronous collaboration. But to make synchronous collaboration possible, some efforts have to be done.
A basic approach would be streaming all the output of the driver to the navigator as image data such as possible with existing solutions like Teamviewer[2]. The good thing is that it supports all applications without any modification. However, this requires a fast network connection and wastes huge amounts of bandwidth for few actually used data and produces other difficulties such as changing roles. Strict firewall rules also often hinders the easy use. The other approach is to design the application specifically for collaboration. This enables the programmers to do asynchronous collaboration as well, but we are not going into detail in this paper.
We will investigate two IDE tools supporting distributed pair programming. One of them is Sangam [12, 8], a rather old tool, which is solely designed for distributed pair programming. Despite its age, we think it is particularly interesting what the authors thought about making a tool for this purpose. CollabVS [10] is a newer implementation of a collaborative programming IDE. It is not intended to be used for pair programming but comes along with several features which are handy for distributed pair programming.

Sangam

Sangam is a plugin for Eclipse to enable distributed pair programming. The plugin itself was developed by a distributed team in 2004. Eclipse qualifies as a base for this purpose because it is a widely accepted IDE, it is open source and a user can install an arbitrary number of plugins. The job of Sangam is to gather every action done on the driver's computer and send it to a server. On the navigator's computer

[1] http://subversion.apache.org/
[2] http://www.teamviewer.com

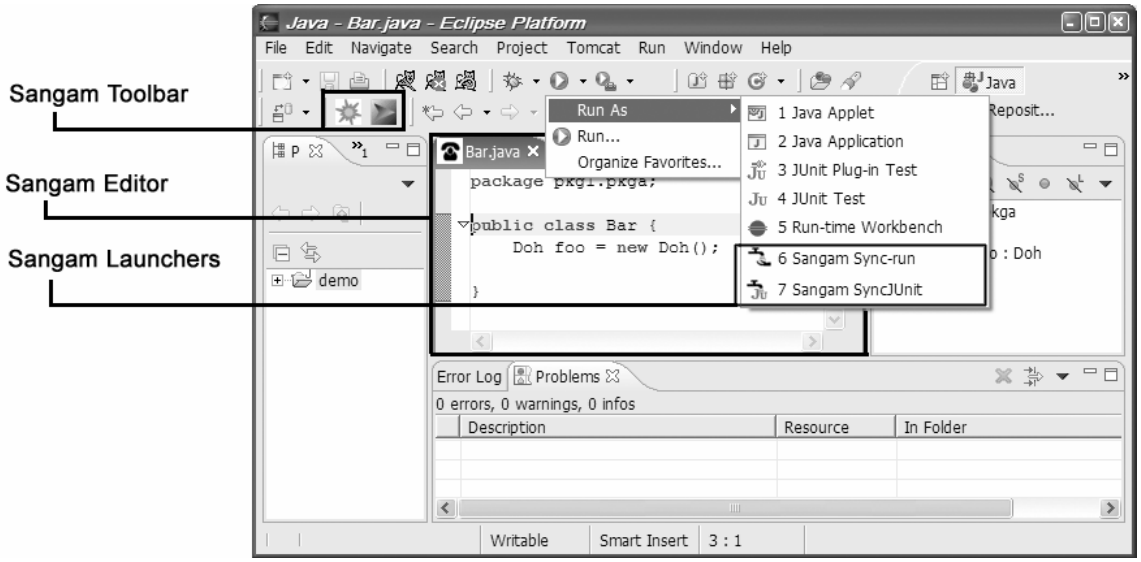

Figure 3. Sangam Eclipse Plugin [12]

Sangam reproduces the action.

Before each programming session all participants have to connect to a centralized server. There can be more than two developers but only one can be driver at a time, thus allowing for multiple navigators. After connecting to the server, the programming session can begin. Figure 3 shows an example view of Sangam running in the Eclipse IDE [12]. The Sangam Editor provides synchronous editing, whereas the Launcher enables to launch a Java application or JUnit test together. The toolbar enables the connection to the server as well as floor management. Floor management defines who has control over the input devices. Although the navigator still can use the mouse and keyboard, so remaining in control over his own PC, the information is not transferred to the driver's screen. However, the navigator can still type in the editor. This is considered as a bug by the authors, but we found it useful as it can be used for correcting syntax errors without bothering the driver. This requires a protocol between both programmers as a downside.

A further development of Sangam is Jazz Sangam [8] which is basically an implementation of Sangam's technologies within IBM's collaborative software development platform Jazz [3]. But since it is four years younger, and Jazz enables new features, it is a worthy enhancement of Sangam. One requirement of Sangam is that the source code needs to be consistent as it only reproduces changes. Distributed developers might have different versions of code on their workspace. Jazz Sangam solves this by bringing its own version control system, easing the project progress by not having to start several other programs at first in order to begin. This is why an instant message client got integrated within the development environment. This is a big advantage towards third party programs running in the background. The user has all necessary windows at sight without the problem of other programs hiding important elements.

[3]http://www.jazz.net

CollabVS

Third party tools, such as IM, version control, voice and video conferencing are all channels which can help with distributed collaboration. However, they "tend to be standalone systems that are not integrated with the programming environment, resulting in significant overhead in using these systems" [10]. This basically is what CollabVS tries to be: a "Collaboration-Centered User Interface" merging many different tools into one.

This tool by the Microsoft research team extends the Visual Studio programming environment, hence the name, and supports all functionality known from Jazz Sangam. To put this into a nutshell: synchronized editing windows, built-in text chat windows and version control come along with CollabVS as well. In addition, it offers audio and video chat and a new concept the authors refer as "real-time presence".

Figure 4 shows a screenshot of CollabVS, with as many streams running as possible. All these are optional and can be placed on different locations. The authors stress the optionality, as "the amount of communication and coding performance by programmer can vary based on their roles". As in Sangam, they are all build in within the IDE, enabling easy placement without hiding other windows.

Audio and video conversation (7) offers many positive effects. But mainly it eases communication between developers, since a direct conversation is always faster than typing. We will get more into detail about direction communication within the Facetop presentation.

The specialty of CollabVS is that it creates a presence stream for each developer (1 and 2). This feature contains the functionality of IM tools, showing if the user is online or offline, but adds more valuable programming context. 'Status' shows what the programmer is currently doing, e.g. if he is editing or just viewing the code and where he is doing this: you can see in which file he is currently active, in which class and even which method he is working at. Left clicking on these information opens a window where you can see the

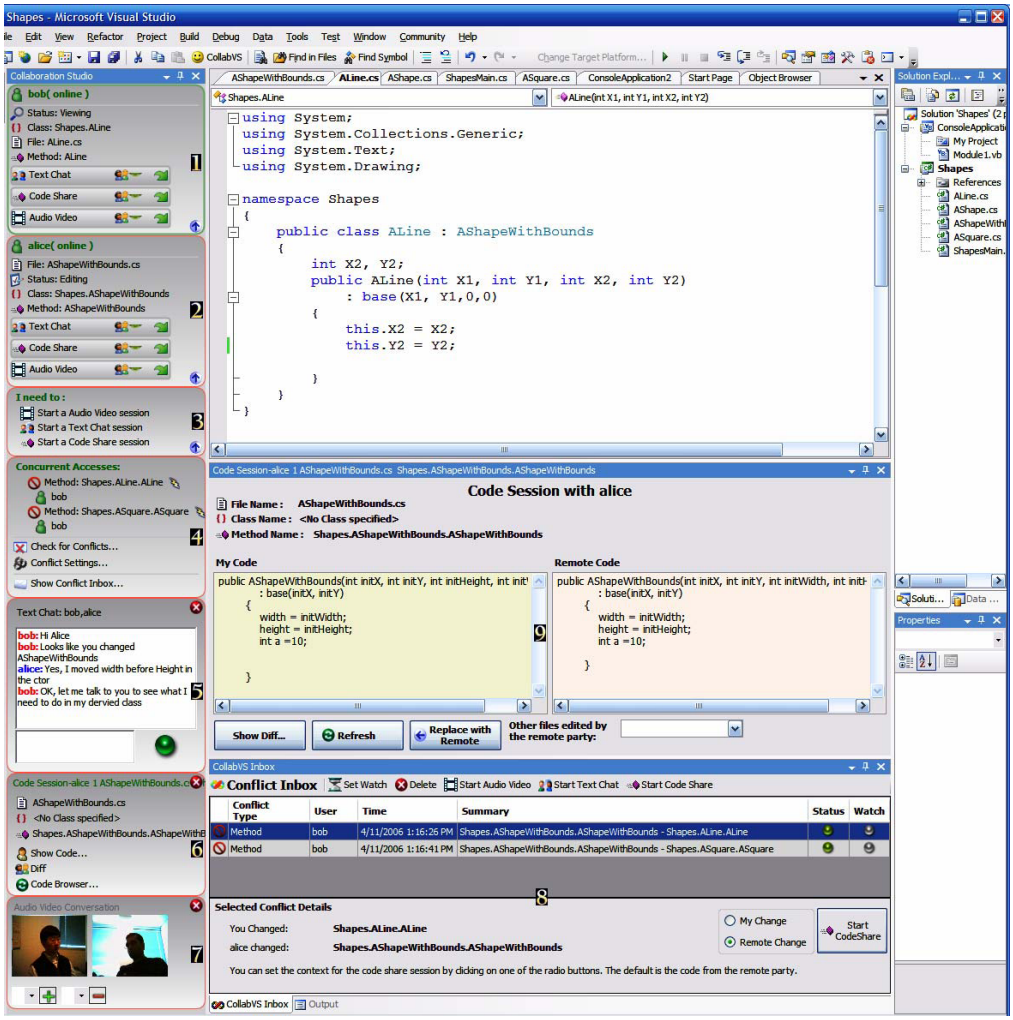

Figure 4. CollabVS Interface showing various features. Key features are identified with numbers [10]

current program code of this method.
A user test with CollabVS within the research lab showed that user like this feature and tend to continuously watch the presence stream to gain insight on which code fragments their partners work. This leads to awareness of others. But how can this asynchronous editing feature help with pair programming?—With finding a partner! As previously shown, pair programmers do not always work in pairs. With the presence streams you can wait for a good opportunity to interrupt another programmer, and when he is available to start a pair programming session. It helps with finding the best partner as well: The awareness of on which files, classes and methods team members work helps with finding the one who has the most experience with a given code artifact. To put the whole matter into a nutshell: Presence streams might work towards one of the reasons, why collocated teams perform better: physical proximity promotes that team members help each other more often and are more aware of what everyone is doing [11].

Nevertheless CollabVS and Sangam both do not fix the biggest disadvantage of IDE tools: their limited extendability. Both support only predefined events. Adding plugins to the IDE and using them in a pair programming context does not work by default as events produced by these are not mirrored. If such a plugin would for example change code, this would happen very fast. The navigator would not recognize this or would be confused on what happened.
Another important feature, namely pointing on code fragments or interface elements, is still not adressed with Sangam and CollabVS.

Facetop

Missing extensibility and pointing on code fragments makes video based solutions more acceptable.
While doing distributed pair programming, it is very important to keep up communication between the two programmers to maintain the advantages of pair programming. To do so, it is necessary to create a workspace that simulates collocated programming. That means you need to hear the other person commenting on the work and you need to see the person. In most cases a simple webcam for seeing the other person is installed, but in some cases this is not sufficient. The video window is much too small to gather

Figure 5. Facetop at semi-transparent level with finger tip tracking marked green. [18]

facial expressions. Simply resizing it would be no good solution as well as this would take even more space of the already messy IDE windows on the one hand. On the other hand a temporary resize would not fit the principle of pair programming with talking and coding at once—and both programmers need to interact with the code by pointing or gesturing on it. A temporary size adjustment of the video window would be not sufficient. There exist many tools to transmit the voice of other persons, but what about the other problem?

Facetop is trying to solve this by projecting the video of an ordinary webcam as a transparent image "behind" the desktop, so the user is seeing himself in the desktop and can naturally interact with other running applications via a fingertip-driven "virtual mouse". Therefore the user's fingertip is being tracked and the mouse can be driven from this tracker (see figure 5). If the tracker loses the fingertip, because of too fast movements or hiding the finger, it starts an entire image scan to retrieve it. Besides the user can adjust the transparency level of the video from opaque to transparent, so Facetop can be used while working and there is no need to hard-switch between the video and the workspace. In addition to simple finger tracking Facetop is also capable of interpreting finger gestures, for example to use for Mac Cocoa Gestures. Clicking on interface elements is achieved by another gesture, the 'pinching fist'. The user hides his finger and reveals it afterwards to do a click on an interface element. Another interaction method is interaction with voice commands, e.g. to adjust transparency of the different video layers.

As the fingertracking was a single user application possibility so far, Facetop also extends to a two head version for collaborative working: two users are visible on a shared desktop side by side, so they can work as if they were sitting side by side. Thus they can see each other in large scale and additionally they can interact with the IDE with help of the finger tracking (see figure 6).

However, the visual clutter brought by several semi-transparent images blend over each other seems to be an obvious problem. Stotts et al. provided several solutions to limit this issue. One of these is to have all participants sitting in front of a white wall. However, as this is often not possible they use algorithms to remove the background of the programmer by computing which pixels are moving and remove those which do not change. Another approach on this is to reduce video information to display only shadows instead of a rich detailed video. This helps with gesturing and pointing to focus on the important parts of the code whenever interpreting facial expressions is not needed.

The users pass the chalk, so to say who is in control over the interface, by hiding the mouse. As mentioned before, the tracker starts an entire image scan on each work station until a new fingertip is found. Once the tracker finds a fingertip, the mouse can be driven from it, but that also means that two users can not simultaneously interact with the IDE, because there is only one mouse pointer.

Additionally to the face-to-face video support, Facetop offers a whiteboard feature that enables a shared view on sketches. To realize that, an additional webcam is needed. This second webcam films a whiteboard and additionally merges this video with the standard Facetop video of the user.

A disadvantage might be that the permanent video of persons or moving objects in the background, such as in office situations, can be distracting while working and those can not be computed to hide till now.

Furthermore there is a problem with collaborative work in Facetop: Each user has access to his whiteboard as it is a physical artifact. If the first user writes something on the whiteboard the other user can see it. However, the second user cannot delete it and has to tell the first one to do that. That is why this solution is not capable of replacing a shared whiteboard, but rather merges two whiteboards into one. This approach eases communication and graphical explanations between both programmers with cheap and ubiquitous equipment.

Another downside is the high demand for bandwidth of the webcam that, if not sufficient, may cause artifacts or stagnant movements in the video—but as broadband connections are ubiquitous, this issues becomes trivial.

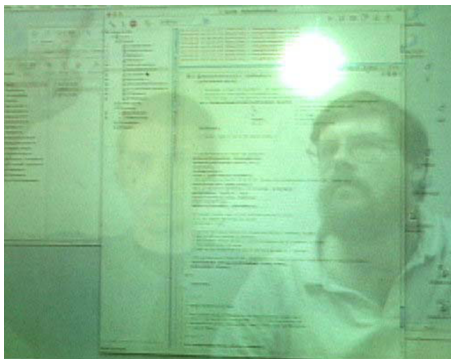

Figure 6. Facetop in dual-user mode [18]

In 2006 Navoraphan et al. did efforts to combine Sangam and Facetop [16], which turned out to be a non-trivial task. The idea behind this mashup is to enhance Sangam's event reproduction functionalities with gesturing. As the navigator was not able to see where the driver was pointing at before, he can now follow all his actions tracked by video.

Facetop is, in its two-user mode, only supported on the Macintosh platform. Sangam is an Eclipse plugin, and that is why the platform was fixed to Macintosh as well. This later turned out to be a problem as several bugs of Eclipse and Java AWT associated to the Mac platform hindered work in a unattended large extend.
The main task of the developers was to automatically synchronize display of the screens in every aspect. This involves resolution settings, each component needs to have the exact same size and the Eclipse window needs to take up the entire screen as well.
However, user studies are still pending.

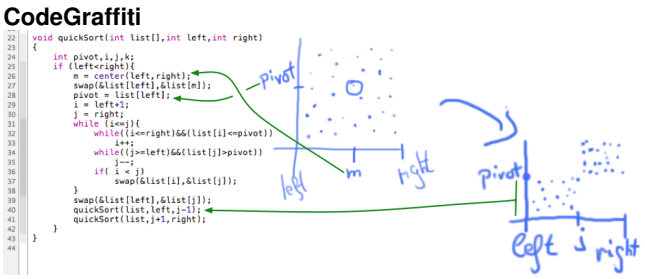

Figure 7. Sourcecode explained via a sketch with CodeGraffiti [15]

Using fast sketching things can often be explained more easily than with words. That is why whiteboards can be found in many workspaces. Developers utilize these not only to explain but also to keep hold of design decisions. Smaller teams, such as in pair programming, often use scratch paper for this purpose.
However, there are two main issues with this: First, whiteboards will be wiped out and paper will be thrown away if not considered valuable enough. Second, they are not digital—thus, making them hard to share over the internet for distributed collaboration. Of couse they can be scanned or copied into digital files, but this requires some effort and they most likely will be stored into a special folder, which is not linked to the actual code. Existing documentation tools like Doxygen[4] are not suited to capture quick annotations and do not support graphical annotations. Additionally, as textural comments grow in size, they make code hard to read.
CodeGraffiti aids with a direct link between Code and Sketches [15]. It gives the navigator the opportunity to sketch directly on the shared screen in line with the code, doing handwritten notes, drawing tables or documenting design decisions and besides having them directly digitalized (see figure 7). This is ensured by giving both driver and navigator the hardware they need for this task: the driver has access to mouse and keyboard. Thus he is in full control of the code and the IDE. The navigator uses a graphics device, such as a graphics tablet with a pen or a finger painting device like iPad and iPhone (see figure 8), to make freehand drawings or pointing to the code. To establish a connection, the navigator connects his device to the server, which runs as a plugin in the IDE.
Sketches done this way are stored in a companion file to each source file, containing information that links them to anchors in the code. These notes do not compile but are still sticked to the syntactical code: When the user removes a text anchor, the sticked sketch becomes uncoupled and can be dragged to other positions. This sticking technique ensures that when the user adds lines of code or dislocates the code, the sticked sketch moves with the code. Aditionally, one graphic can be sticked to several anchors. Thus making it possible to draw connections between several code fragments.
Beside annotating code the navigator can draw over interface widgets, reference documentation or other screen elements. Codegraffiti handles these drawings as 'ephemeral' notes which just fade out after a few seconds.
Supporting the intention of pair programming, driver and navigator always have the same view on the code, so no one can wander off and work on separate tasks or files. However, this might hinder workflows, as the the driver has to wait for the navigator to finish a drawing, until he can scroll down. CodeGraffiti solves this by sticking the view of the navigator until he finishes and then jumps to the drivers new view. The driver can go on with his work without beeing delayed.
With CodeGraffiti programmers are able to retain more sketches for documentation. Moreover, they can assign graphics to code very easy and efficient. All in all CodeGraffiti is a helpful tool. However, till now, it only supports Mac and does not provide integrability to existing major IDEs.

Figure 8. Various CodeGraffiti clients a) Apple iPad/iPhone, b) Wacom Intuos tablet, c) Wacom Cintiq tablet display [15]

CONCLUSION

We have given insight into current research on benefits and disadvantages and why pair programming is that successful. However, till today a final answer, especially on this problem, can not be given. Based on the level of abstraction, driver and navigator seem to contribute to the code in the same manner. On the other hand, there were still limitations which led to the 'vague' classification problem. Better methods of observation could lift this issue by considering the context of every utterance. It should be possible to distinguish between utterances of agreement, which would be an additional classification, and real 'vague' classifications.
Today's implementations of IDEs which are capable of logging activities can ease research and open new possibilities by mitigating the technical limitations. Investigating distributed pairs can be particularly interesting as all interactions between them need to be digitalized, thus making it easier to get access to this information. Additionally, different information channels, such as video, can be turned off to show their influence.

While these IDE based user interfaces can help with research, much work still can be done to ease programmer's every day life. CodeGraffiti's capabilities impress, but it does not allow integration in major IDEs like Eclipse.
The missing extensibility of existing IDE solutions, as mentioned one of their biggest problems, can not be addressed by

[4]http://www.stack.nl/ dimitri/doxygen/

small teams of scientists programming these pair programming IDE extensions within their research activities. They need to be commercially adopted or focused by the open source community. But as broadband internet has become ubiquitous, at least within the industry countries, video and VNC based solutions are outdoing pair programming specialized IDEs. Thus making them not interesting for commercial adaption.

Facetop's video approach with its unique way of interacting with the code, while being futuristic, however, for us seems not to be the way to go for future pair programming interfaces. The merging of real life video and IDE elements is very distracting. New image recognition technology possibly could lift this issue by reducing unnecessary image information—but pointing in the air to point on interface elements on the screen still seems unintuitive for us.

The upcoming ubiquitous availability of tablet PCs seems to be more promising: Being able to point directly on interface elements and do pen-type input could ease communication between distributed partners. We could think of a distributed version of CodeGraffiti with enhanced pointing and gesturing possibilities.

Further enhancing pair programming interfaces and still open questions regarding the role of the navigator are the reasons why pair programming stays an appealing field for research and interfaces.

REFERENCES

1. S. Ainsworth and A. Th Loizou. The effects of self-explaining when learning with text or diagrams. *Cognitive Science*, 27(4):669–681, 2003.
2. P. Baheti, E. Gehringer, and D. Stotts. Exploring the efficacy of distributed pair programming. *Extreme Programming and Agile MethodsXP/Agile Universe 2002*, pages 387–410, 2002.
3. K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.
4. F. Brooks Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
5. S. Bryant, P. Romero, and B. du Boulay. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, 66(7):519–529, 2008.
6. M. Chi, N. De Leeuw, M. Chiu, and C. LaVancher. Eliciting self-explanations improves understanding. *Cognitive science*, 18(3):439–477, 1994.
7. A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–248, 2001.
8. J. Devide, A. Meneely, C. Ho, L. Williams, and M. Devetsikiotis. Jazz Sangam: A Real-Time Tool for Distributed Pair Programming on a Team Development Platform. In *Workshop on Infrastructure for Research in Collaborative Software Engineering, Atlanta, GA*, 2008.
9. N. Flor and E. Hutchins. Analyzing distributed cognition in software teams: a case study of collaborative programming during adaptive software maintenance. In *Empirical Studies of Programmers: Fourth Workshop, Ablex, Norwood, NJ*, pages 36–64, 1992.
10. R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 178 –187, 2008.
11. J. Herbsleb, A. Mockus, T. Finholt, and R. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 319–328. ACM, 2000.
12. C.-W. Ho, S. Raha, E. Gehringer, and L. Williams. Sangam: a distributed pair programming plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, eclipse '04, pages 73–77, New York, NY, USA, 2004. ACM.
13. W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.
14. J. Lave and E. Wenger. *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.
15. L. Lichtschlag and J. Borchers. Codegraffiti: Communication by sketching for pair programming. In *UIST 2010 Extended Abstracts*, New York, NY, October 2010.
16. K. Navoraphan, E. Gehringer, J. Culp, K. Gyllstrom, and D. Stotts. Next-generation DPP with Sangam and Facetop. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 6–10. ACM, 2006.
17. J. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, 1998.
18. D. Stotts, J. Smith, and K. Gyllstrom. Support for distributed pair programming in the transparent video facetop. *Extreme Programming and Agile Methods-XP/Agile Universe 2004*, pages 150–192, 2004.
19. D. Stotts, L. Williams, N. Nagappan, P. Baheti, D. Jen, and A. Jackson. Virtual teaming: experiments and experiences with distributed pair programming. *Extreme Programming and Agile Methods-XP/Agile Universe 2003*, pages 129–141, 2003.
20. L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *Software, IEEE*, 17(4):19–25, 2002.