Types of Cooperation Episodes in Side-by-Side Programming

Lutz Prechelt, Ulrich Stärk, and Stephan Salinger

Freie Universität Berlin, Institut für Informatik, Berlin, Germany
prechelt,ustaerk,salinger@inf.fu-berlin.de

Abstract. In side-by-side programming, two programmers (typically working on related aspects of a project) move their computers so close to one another that they can effortlessly change between working alone and working together, where working alone is the primary mode. The technique was proposed in order to obtain some of the advantages of pair programming at lower total effort. As a first step towards understanding how and when to use side-by-side programming, the present study aims at describing when and for what purpose side-by-side programmers get together to cooperate. The main result is a classification of the cooperation episodes by purpose and content into different types: Exchange project details, Exchange general knowledge, Discuss strategy, Discuss step, Debug work product, Integrate work products, and Make remark. These types were derived via the Grounded Theory method and are described conceptually in terms of the types of events of which they consist. All concepts used in these descriptions are grounded in actual observations.

1 Introduction

In the last few years, the popularity of agile development methodologies in general [13] and eXtreme Programming (XP) in particular [1, 2] has led to a lot of interest in pair programming, which is one of XP's core practices.

Studying pair programming is relevant because the practice is such a provocative idea: On the one hand, it claims a number of benefits such as better designs, fewer defects, high productivity (for a number of different reasons), mutual learning, more people being closely famililar with the resulting code, and others (see for example [31]). On the other hand, investing two people to solve a task traditionally solved by one alone might obviously be costly.

So far, the empirical work on pair programming has mostly focussed on the claim of high productivity by comparing pairs to solo programmers in controlled experiments; see for instance [15, 21, 30]. The results are somewhat mixed and also hard to interpret, because the short-term setups used in the experiments do not allow to observe the longer-term benefits that would manifest only much later in a real software development context.

Side-by-side programming [8, Section 3.T8] was suggested as a compromise between solo programming and pair programming. It can be described and interpreted for instance as follows. Side-by-side programming is like solo programming in that the two programmers involved each use their own computer and normally work alone on a (sub)task. However, side-by-side programming resembles pair programming in that the programmers can switch to a pair mode at any moment: their two computers are located very closely side-by-side to one another. The idea is that they will be cooperating directly for a while whenever this appears particularly useful.

Side-by-side programming aims at getting at least some of the benefits of pair programming (in particular those related to having more knowledge at hand) at essentially no additional cost. Obviously, this is not a guaranteed win: The partners may cooperate too much or too little, may interrupt each other's train of thought badly, etc.

Our aim is first understanding how side-by-side programming works and then formulating advice regarding how to optimize the use of side-by-side programming. This involves three questions:

1. Understand when and why and for what purpose side-by-side programmers cooperate directly.

2. Understand how they work while cooperating directly.
3. Evaluate which of these behaviors are helpful, which are problematic, and which may be missing entirely.

We have started studying question 2 in the context of normal pair programming a few years ago [24, 25]. The present study concerns question 1. To be practical, work on questions 1 and 2 obviously requires the use of qualitative (rather than quantitative) research methods.

1.1 Related work

As far as we are aware, only two studies of side-by-side programming have been published to date. The first, by Nawrocki et al., is in the tradition of previous controlled experiments that compare the raw productivity of solo programmers to that of pair programmers [7, 12, 15–20, 22, 32–34], but added a third group that used side-by-side programming [21]. For side-by-side programming, the experiment found an overhead of 20% compared to solo programming, while for pair programming the overhead was 50%.

The other, by Dewan et al. [9], is an exploratory, qualitative one on *distributed* side-by-side programming (DSBS). The pairs are working in different rooms and are coupled by technical means only. Each programmer has a second screen display that always shows what the partner is seeing; in a part of the study their workspaces are tightly coupled in real time via desktop sharing. The authors provide several observations about how DSBS is used and about the problems it does and does not exhibit. Among these observations is a classification of work modes which we will discuss in Section 5.3 after we have presented our own results.

While there are no further qualitative studies of side-by-side programming yet, a few do exist on pair programming. We discuss those which appear relevant for our side-by-side programming research.

Chong and Hurlbutt report on field observations of 40 hours of pair programming sessions for at least four different pairs from two different companies [6]. The analysis is based on field notes and partial audio recordings (later transcribed). A coding scheme is mentioned, but neither its content nor its role is specified. The article reports, among others, a finding relevant to side-by-side programming: the programmer with greater task knowledge or code base familiarity dominates the process.

Sharp and Robinson performed an ethnographic study of eXtreme Programming (rather than pair programming alone) [27]. The work describes how pair programming is intertwined with the other practices of XP and, more importantly, how it contributes to and is influenced by the social culture of the development team. It concludes that pair programming is mostly about communication, in particular about achieving and maintaining a common understanding.

Cao and Xu study the influence of high/medium/low skill levels on the activity patterns exhibited by pairs [5]. They study a total of 6 pairs of student programmers with high-high, medium-medium, and high-low skill combinations, respectively, via protocol analysis of videotapes. Their conclusion, whose derivation is only partially spelled out in the paper, is that the high-high skill pairs exhibit the largest amount of "deep-level thinking" and high-low skill pairs the lowest.

Sallyann Bryant (now Sallyann Freudenberg) discusses a mixed quantitative-qualitative approach with which pair programming could be studied [3]. She presents a simple coding scheme with 11 codes for classifying pair members' behaviors and finds that the behavior type frequencies correlate with the pair programming experience mix in the pair (high-high, high-low, low-low).

Xu et al. compare intermediate programmers and experts with respect to the build-up of problem knowledge during pair programming [35]. The study, based on three long pair programming sessions, uses a coarse-grained protocol analysis where each "episode" (of about five minutes) comprises the discussion of a different programming or domain concept. The expert

pair was found for instance to discuss multiple concepts at once (rather than serially) and to be more likely to reconsider their previous design decisions than the two pairs of intermediates did.

Several studies address the conventional assumption that the "driver" (the person currently operating the keyboard) and the "navigator" (or "observer") have fundamentally different roles and work on quite different levels of abstraction: high and low for the navigator, middle for the driver. They find that this is not the case. Rather, the members of a pair normally move through different abstraction levels together [4, 6, 11].

1.2 Research questions

We will now specify question 1 more precisely. When two side-by-side programmers (who work separately by default) get together in order to work cooperatively towards some subgoal, we call this stretch of time a *cooperation episode*. The episode ends when either the subgoal has been reached, the programmers pick a new subgoal, or the programmers split up to work separately again.

Note that the subgoal can be implicit, even unconscious, for the programmers and is sometimes difficult to identify for the researcher even after the fact.

Given this definition, we ask these research questions:

- **P (purpose)**: When or why or for what purpose do side-by-side programmers start a cooperation episode?
- **T (termination)**: When or why do they cut off or terminate the cooperation episode?

Note that directly answering the *when* or *why* of question P will sometimes require understanding the programmers' internal thought processes, whereas the question *for what purpose* can (on some reasonable level of abstraction) usually be answered by looking at the content of the cooperation episode. It is thus likely that we will have to drop the ambition to answer the *when* or *why*.

Furthermore, the answer to question P is likely more interesting and relevant than the answer to question T.

1.3 Research approach

As mentioned above, for learning how two programmers *should* cooperate, the first step is finding out how they *do* cooperate and the second one is evaluating what about this behavior is helpful and what is problematic (and what might be useful to add to their behavior). The present study concerns the first step only.

Our research approach is based on detailed recordings of side-by-side programming sessions. In these recordings, we will identify the cooperation episodes and analyze them in detail.

The analysis involves preparing a conceptual description of the programmer/computer and the programmer/programmer interactions that abstracts from the details of the specific task in order to make visible the conceptual similarities and differences between individual cooperation episodes. We will then cluster these conceptual descriptions to derive recurring classes of cooperation episodes. These classes will form the basis for answering the research questions.

We will now describe the setup of our study (Section 2) and the data analysis procedure used (Section 3). We will then present the results of the analysis (Section 4) and discuss how reliable they are and what to do with them (Section 5).

2 Study setup

The subjects of our study are graduate students taking part in a voluntary, zero-credit university workshop that provided a crash course about modern Java web development frameworks: Hiber-

nate[1] (an object-relational mapper), Spring[2] (an inversion-of-control container), and Tapestry[3] (a framework for web frontend programming).

This workshop forms the context from which we have collected the data used in the present study, we will describe it in Section 2.1. The subjects themselves will be characterized in Section 2.3 and the data collection procedure in Section 2.2.

2.1 The data collection context

The workshop took place in a computer pool room during four consecutive days in the summer break 2007. Day 1 introduced Hibernate and Spring each by a short lecture, after which the participants spent the rest of the day learning the practical use of the frameworks by solving a given two-step exercise. Day 2 introduced Tapestry by a short lecture, followed by first a Tapestry-only exercise and then a second exercise that involved both Tapestry programming and integrating the results with the results of day 1. Day 3 was a pure programming day. The participants extended their code so as to form a simple book library management application including user management, book management, lending, and return.

For each of these exercises, we tried to make side-by-side programming likely without explicitly telling the participants what it was or that they should use it: the participants were told to work in pairs and split up the work; time was restricted and the amount of work was larger than one person could likely do alone; splitting up the work involved some coordindation and integration effort; the two computers of a pair were set up close to each other. The participants paired with the same partner each day.

By day 4, the participants were somewhat familiar with the three frameworks, very familiar with their own library application systems, and reasonably familiar with cooperating with their pair partner. On day 4, the pairs were asked to build several specific extensions into their respective existing systems. Fulfilling the given requirements involved building seven additional classes or pages, but these were not mentioned explicitly. The time for solving this task was limited to 2.5 hours. The respective programming sessions were recorded as described in Section 2.2 and form the raw data analyzed in the present study.

2.2 Data collection

We used Techsmith's *Camtasia Studio 4.0.2*[4] to losslessly record the 1280x1024 pixel desktop on each member's computer at 4 fps (frames per second). A webcam sitting on top of the monitor recorded the head and upper torso of the respective user. During a cooperation episode, the migrating partner was always fully or partially visible for his own and/or the partner's webcam. Each webcam's microphone recorded audio of the pair's conversations.

When a pair had finished its task, each member would individually fill in a two-part questionnaire.

- Part 1 concerned the task and the pair's cooperation and consisted of 9 open and 7 closed questions. It asked how complete the pair's solution was, how difficult the task had been and why, when/why/how often the pair had split up and joined, what had been good or not so good about the cooperation, and how harmonious the cooperation had felt.
- Part 2 concerned the workshop overall and the subject's background and consisted of 1 open and 23 closed questions. It asked about length of programming experience, a subjective estimate of their capabilities relative to their student peers ("among best x percent"), how much they had worked in pairs with this partner or with anybody before, how much they

[1] www.hibernate.org
[2] www.springframework.org
[3] tapestry.apache.org
[4] www.techsmith.com

knew about Tapestry or Hibernate before/after the workshop, how often they found the workshop too easy or too difficult.

As the final data collection step, we used Adobe Premiere CS3 in order to join (for each pair) the four pieces of video and two pieces of audio into a single 10 fps video 2560 pixels wide and 1024 pixels high. This process involved the following difficulties:

- Imperfections of the involved codecs. After a lot of experimentation we eventually rendered usable videos by Xvid 1.1.2 [5] via VirtualDub 1.7.7 [6].
- Camtasia Studio does not record webcam and desktop fully in sync. We synchronized the webcam and desktop videos manually by adjusting the desktop video playback speed separately for each 5 minute segment in Premiere.

2.3 The subjects

The workshop had 10 participants, so we had 5 pairs to start with. When analyzing the videos, we found that one pair had used full pair programming throughout, so the number of join and split evens is zero for them and we hence eliminated them from our analysis. For a second pair we lost one member's video due to a file naming conflict and decided that the other half alone was not usable for the analysis, so we eliminated this pair as well and ended up with 3 pairs. The subsequent information concerns these 3 pairs only and is based on their answers in the questionnaire.

	pair 1		pair 2		pair 3	
Gender (male/female)	m	m	m	m	f	m
Been a student since (no. of terms)	14	12	6	8	7	8
Java programming experience (years)	6	7	3	4	1	4
Java web development experience (years)	1	0	2	1	1	2
I am among the most capable x%	40%	5%	40%	40%	50%	40%
Quality of cooperation (1–5)[1]	4	3	5	4	5	4
Task difficulty (1–5)[2]	3	2	3	3	4	3

[1] 1: very bad, 3: OK, 5: very good

[2] 1: much too easy, 3: just right, 5: much too difficult

Table 1. Information on background and perceptions of the six pair members

As shown in Table 1, pair 1 had the most experienced programmers, but the largest difference in self-perceived capabilities. The lower capabilities of member 1 made this member somewhat anxious and self-conscious and the other somewhat unhappy. These two had never before worked together—in fact not even during the workshop, because the original member 2 did not appear for the experiment and so the 11th workshop participant, a Ph.D. student who had worked alone during the first three exercises, stood in as ersatz member 2. As a result of all this, this pair gave a not-so-harmonious impression.

Pair 2, though younger, had more previous web development experience, perceived their respective capabilities as medium, the task difficulty as appropriate, and their cooperation as very harmonious. They had worked together with each other several times previously.

Pair 3 reported the lowest capabilities and the highest task difficulty, but found their cooperation to be rather harmonious. This pair had never worked together before the workshop.

[5] www.xvid.org

[6] www.virtualdub.org

3 The data analysis process

From the data collection process described above, we ended up with three videos, each of approximately 2.5 hours length.

The data analysis proceeds in four phases: mark the relevant stretches in the videos, that is, identify the cooperation episodes (Section 3.1); conceptualize and encode previously expected phenomena in the episodes (Section 3.2); identify further concepts needed to classify the episodes (Section 3.3); cluster the so-encoded episodes in order to identify classes of episodes (Section 3.4).

3.1 Identify cooperation episodes

We viewed all videos at least once in full length and marked time stretches as "this is certainly cooperation" and "this is certainly none". Certain cooperation are situations of physical togetherness in front of just one computer and situations with spoken interaction. Certain non-cooperation are situations with extended lack of both of these features. We then iteratively reviewed the remaining "maybe" stretches in order to identify the precise beginning and end of each would-be cooperation episode (defined by making contact and breaking contact) and to make sure not to overlook cooperation episodes consisting of only spoken dialog, without body movement (non-physical cooperation).

This process resulted in 91 stretches. During the further analysis steps, 19 of these turned out to be actually two different episodes glued end-to-end. We broke these apart and ended up with 111 cooperation episodes of lengths between 5 seconds and 31 minutes as the net raw material for our study.

3.2 Apply the PP foundation layer concepts

The Pair Programming Foundation Layer (PP foundation layer) is a set of concepts for conceptualizing the semantics of a pair's interaction with each other, with their computer, and with their environment [25]. As the name implies, it was created for describing pair programming situations and so can be expected to fit the cooperation episodes of side-by-side programming well.

For the human/human interaction (HHI), the concepts are noun/verb combinations, where the noun is one of *activity, completion, design, finding, gap in knowledge, hypothesis, knowledge, off topic, rationale, requirement, source of information, standard of knowledge, state, step, strategy* and the verb is one of *amend, ask, agree, challenge, decide, disagree, explain, propose, remember, say, stop, think aloud*. Further concepts exist for human/computer interaction (HCI, i.e computer use) and for human/environment interaction (HEI, describing non-verbal activity that is not computer use).

Many of the HHI concepts are reasonably straightforward. For instance when the pair members talk about their problem solving process, *step* refers to a possible next single elementary action to be taken, *strategy* refers to a plan consisting of many steps, and *design* refers to a possible decision with respect to the structure of the work product (the program). *propose_step* means that one pair member suggests which next action to take (likewise *propose_strategy* and *propose_design*). The partner then typically reacts with either *agree_step/strategy/design* (i.e. states approval), *disagree_step/strategy/design* (i.e. rejects the suggestion), or *challenge_step/strategy/design* (i.e. rejects the suggestion by making an alternative one).

Other concepts, in contrast, are quite subtle. For instance *thinkaloud_activity* refers to the verbalization of an ongoing stream of actions the driver is performing on the computer. Parts of these utterances can often be interpreted as *propose_step* events, etc.; hence *thinkaloud_activity* acts as a container for other events.

Some of the possible combinations of the above nouns and verbs do not make sense and many of the remaining ones have never been observed. Since all of the PP foundation layer concepts

are required to be firmly grounded in actual observations (the PP foundation layer was derived via Grounded Theory methodology [29]), less than one third of the possible combinations are actually included as concepts. For instance, we have never yet seen an actual *disagree_strategy* event and so this concept is not in the PP foundation layer. However, a user of the foundation layer could immediately introduce the concept when it occurs, because the concept is obviously meaningful and consistent with the rest. Together, we call the combination of a noun (say, *strategy*) with any of its verbs a *concept class* and abbreviate it as *∗_strategy*.

In other words, the PP foundation layer ought not to be used as a fixed coding scheme, but rather as useful background knowledge and as a set of suggestions of potentially useful concepts. This point is important, because in our analysis we want to follow the Grounded Theory methodology which allows using prior expectations for sharpening one's attention, but clearly forbids constraining the conceptualization to only preconceived concepts [14].

We applied the PP foundation layer concepts for encoding each programmer's behavior during every cooperation episode. The PP foundation layer was sufficient for covering the material and the resulting conceptualization was sensible and consistent. However, it was not sufficient to discriminate all of the different types of cooperation episode we saw. We needed further concepts.

3.3 Introduce additional concepts

From the data, we discovered four new stand-alone concepts. The HEI concepts *move_over* and *move_back* describe temporarily joining the partner in front of his/her computer and discriminate physical from non-physical cooperation episodes. The HCI concepts *commit_changes* and *update_codebase* describe pushing work results to and getting them from a source code versioning system (such as CVS or SVN), which is the method by which our pairs join their distributed work results on one computer.

All other new concepts, which we recognized in the data while trying to understand the different cooperation episode types describe properties to be attached to existing concepts. Most of them are HHI:

- Instances of *∗_strategy* can often be described by an *explicitness* property with possible values *procedural* (enumerating the steps of the strategy explicitly, as for instance in *"First we need to create the* `Location` *thing and then we must add a* `location` *field to* `Copy`*"*.[7]) and *declarative* (describing the strategy as a whole, as for instance in *"I'll create the webpage and you make the Java stuff for it, OK?"*).
- Likewise, instances of *∗_design* and *∗_rationale* can have a property *granularity* with value *fine-grained* (e.g. *"In* `PublicationService` *we also need a* `removeLocation` *with* `Location` *as parameter"*) or *coarse-grained* (e.g. *"We also need an Entity with name* `Location`*"*).
- Instances of *∗_knowledge* and *∗_standard of knowledge* can have a property *specificity* with value *generic* (e.g. *"How can you separate multiple parameters? With blanks?"*) or *project-specific* (e.g. *"I have now spelled 'Gebäude' with* `ae`*"*).
- Instances of *∗_knowledge* can also have a property *type of knowledge* for which there are many possible values[8]. We only need the values *description of phenomenon* (e.g. *"What is it that's not working?"*) and *explanation for phenomenon* (e.g. *"Do you know how one registers a* `Location` *in the database? I am getting an Exception."*, which in this case is interpreted as asking for an explanation of why the exception occurs.).
- Instances of *propose_step* can have a property *type of step* of whose many possible values we only need *help me* (a query for help, e.g. *"Could you look? I'm getting yet another*

[7] This and all subsequent examples are actual utterances from our raw data, translated from German.

[8] In principle, *rationale* is just one such value. One main reason why we present *∗_rationale* as a concept class instead is that it requires a further property *granularity*, which would otherwise make the presentation overly confounded.

Exception"), *help you* (an offer to help), and *stop help me*/*stop help you* (a suggestion to terminate the cooperation episode).
– Instances of *verify_something* can have a property *outcome* with values *correct*, *incorrect*, and *don't know*.

3.4 Cluster phenomena to identify types

The actual search for different cooperation episode types started from the PP foundation layer concepts only, looking for dominance of individual concepts or fixed sets of concepts within individual episodes. We then gradually introduced the additional concepts and extended the search to cover also sequences (rather than just sets) of concepts.

Via observing, hypothesizing, cross-checking, formulating, and re-observing as it is induced by the Grounded Theory activity of open coding [29, Section II.7] (framed by the practices of theoretical coding [29, Section II.5] and constant comparison [29, Section II.1]), we finally arrived at the following seven major new concepts ("categories" in Grounded Theory lingo) that each represent one type of cooperation episode.

4 Results: The cooperation episode types

The following subsections each characterize one of the types we have identified and describe how to recognize an instance of this type based on the encodings previously applied to the raw data (see Section 3.2 and 3.3).

The name we have chosen for each type characterizes the purpose of the respective cooperation episode. Except for one type ("Make remark"), the types can be assigned to two different spheres according to their purpose: coordination issues and technical issues. This is shown in Figure 1.

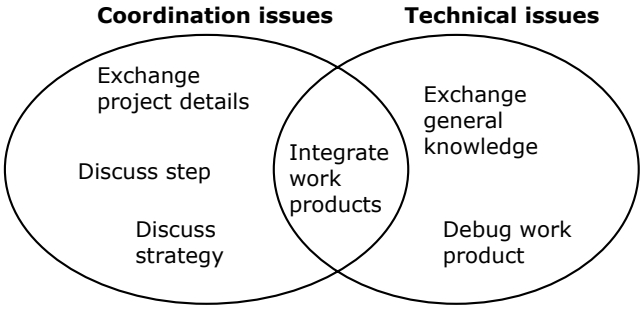


Fig. 1. Types of cooperation episode types

4.1 Type "Exchange project details"

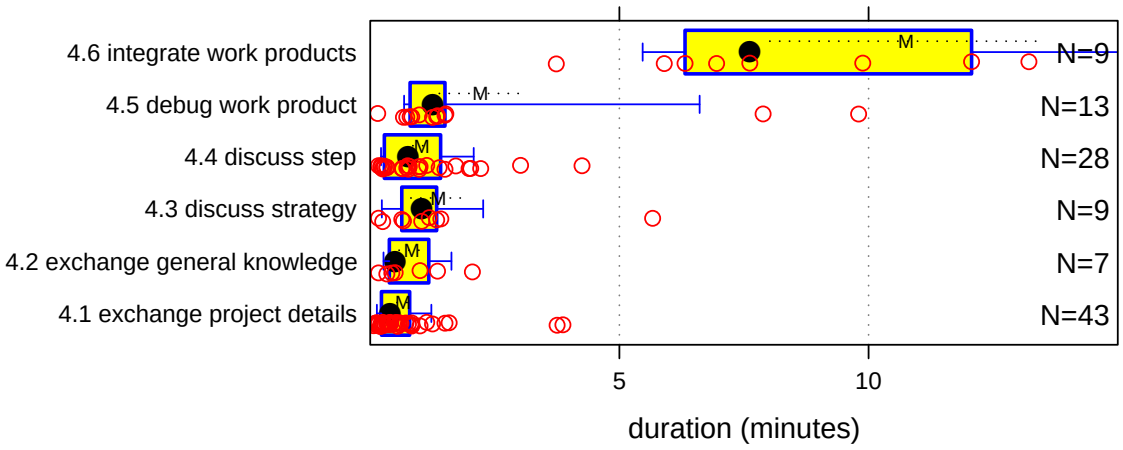
Instances of this type have a coordination purpose: The partners query and/or inform one another (or only one informs the other) about facts that are specific to the partners' current common task or project. Such facts may be status information (which would typically, but not exclusively, be encoded by concepts of classes $*$*_completion* and $*$*_state*), artifact-related details (typically concepts of class $*$*_knowledge(project-specific)*), background information ($*$*_rationale*) or similar.

The above description means the following:

– If the above concepts dominate the episode, the type of the episode is likely "Exchange project details".
– Other concepts may occur as well.

Further symptoms of "Exchange project details" are implicit queries for information by declaring where one's knowledge has a gap (*∗_standard of knowledge(project-specific)*).

Episodes of this type are very frequent (in our data this was the most common type by far), tend to be quite short (usually well under one minute), and are more often non-physical than physical. See Figure 2 for an overview of frequencies and durations of the different types of episode.



(Each mark represents one episode, the whiskers indicate the 10- and 90-percentiles, the fat dot is the median, the M and dotted line are the arithmetic mean plus/minus one standard error. One data point for 'integration' at 31 minutes is not visible.)

Fig. 2. Durations of cooperation episodes by type

4.2 Type "Exchange general knowledge"

Instances of this type have a support purpose: Either one partner receives technology-related or domain-related knowledge from the other (either after an explicit query or because the other has observed difficulties in this respect) or the partners collect and compile their fragmentary knowledge on a particular topic. In both cases, the episode is dominated by concepts of class *∗_knowledge(generic)*. Ideally, such an episode consists of just two events (*ask_knowledge(generic)*, *explain_knowledge(generic)*, but in practice additional interactions often occur.

In contrast to "Exchange project details", the information received in "Exchange general knowledge" episodes is meaningful also outside the scope of the current project.

Episodes of this type can be more or less frequent depending on the programmer pair, can be short or longer, and in our data were all physical.

4.3 Type "Discuss strategy"

Instances of this type have a coarse-grained task and work planning purpose: The partners roughly identify what needs to be done (*∗_strategy(declarative)*), agree on a rough high-level design for the artifacts to be produced (*∗_design(coarse-grained)*), and possibly discuss reasons for either of these (*∗_rationale(coarse-grained)*).

Episodes of this type typically occur once at the beginning of a side-by-side session and perhaps another few times during its later course and can be short or long. The initial episode of this type is usually non-physical, later ones are often attached to the end of an "Integrate work products" episode and are then usually physical.

4.4 Type "Discuss step"

Instances of this type have a fine-grained work planning purpose: Based on the current work status (∗_*completion*, ∗_*state*, but sometimes neither of those is mentioned explicitly), the partners discuss the next work step to be performed (∗_*step*) or possibly an explicit list of several such steps (∗_*strategy(procedural)*). They may also talk about a specific artifact element to be created (∗_*design(fine-grained)*), may postpone a step to be performed at some yet unspecified time in the future (∗_*todo*), or discuss the rationale for steps, e.g. in order to decide between alternatives (∗_*rationale(fine-grained)*).

"Discuss step" can initially look exactly like "Exchange project details", but the purpose is making decisions, and events in the later course of the episode will reflect this. "Discuss step" resembles "Discuss strategy", but concerns much less far-reaching decisions. Therefore, "Discuss step" episodes are much more frequent and can (but need not) be quite short. We have seen physical as well as non-physical ones.

4.5 Type "Debug work product"

Instances of this type have the purpose of turning a defective work product into one that works as intended.

There are two forms how such episodes start. One partner may explicitly ask for help (*propose_step(help me)*, e.g. "I have a problem with X, could you have a look?" or *ask_knowledge(explanation for phenomenon)*, e.g. "Do you have an idea why I keep getting X?") or the other may notice problems of the other (who mumbles about them) and jump in (also by *propose_step(help you)* or *ask_knowledge(description of phenomenon)*)

The further course of the episode is characterized by some (possibly long) sequence of events involving testing (*verify_something*) or code review (*verify_something*), discussing assumptions (∗_*hypothesis*) and insights (∗_*finding*) (or alternatively explaining what the culprit's oversight was (∗_*knowledge*)), and modifying the artifacts in question (*write_something*).

The end of the episode looks different depending on success. If the defect was found, the last few events must conceptually include an insight (*explain_finding*) what the defect is, the correction of the defect (*write_something*), and a final test to make sure the defect is repaired (*verify_something(correct)*). However, the last one or even the last two of these steps may occur outside the proper cooperation episode.

If, in contrast, the partners give up, one of the last few events will be *propose_step(stop help me)* or *propose_step(stop help you)*.

Episodes of this type tend to be long (although the simple task in our study reflects this only sometimes), their frequency depends on the pair, and except for very short ones they are physical.

4.6 Type "Integrate work products"

Instances of this type have the purpose of joining and consolidating the partial work products of the partners into a coherent common work product.

The instances of this type are composite: they often include sub-episodes that would qualify as one of the other types. However, these sub-episodes must be considered part of the integration episode, because otherwise the purpose of an integration episode would no longer be discernible; such definitions would misrepresent the intentions of the programmers.

Conceptually, the structure of an "Integrate work products" episode is as follows:

- Status: Make sure each partner's work is sufficiently complete for performing an integration. This will be a valid "Exchange project details" episode.
- Decision: Agree to actually perform an integration. This will be a valid "Discuss step" episode.

– Sync: Make all work products available on one computer: *commit_changes* by one partner followed by *update_codebase* by the other.
– Test: Prepare and perform a test of the new functionality (*verify_something*, often also *write_something*).
– Debug: Possibly correct any defects in the integrated artifacts. This would be a valid "Debug work product" episode.

The last two steps can be repeated multiple times, until all aspects that need to be checked for a successful integration have been tested and debugged. Underways, there may also be some further development of small missing bits and pieces.

Note that status, decision, and sync may be missing from the encoding because they have happened earlier (sync) or without explicit communication (status, decision).

Conceptually, the integration episode ends with the successful test of the last aspect that needed testing. Attentive readers will notice, however, that this ending criterion is not operational: based on the given concepts and encodings there is no way to decide which is the "last aspect that needed testing". An operational ending criterion can be defined as follows, though: Integration episodes are necessarily physical[9]. The latest possible end of the integration episode is thus the end of the physical cooperation (*move_back*). From that point, go backwards in time and cut off the last candidate sub-episode (or possibly two) if it qualifies as "Discuss step" or "Discuss strategy", which often occur at the end of an integration but should be considered separate episodes, because they usually concern a new topic.

Integration episodes are usually quite long and their frequency depends on the pair.

4.7 Type "Make remark"

This type is a rather special case; its instances have no fixed kind of purpose[10], are extremely short, and necessarily non-physical. They involve only one single utterance (no dialog or interaction at all[11]) and go as follows: One partner observes a situation faced by the other and comments on that situation. Period.

In the two (presumably quite typical) instances we have observed, the remark was once a *propose_step* and once an *explain_knowledge*. Both episodes occured with pair 2, the only one that had already worked together several times previously.

Alistair Cockburn considers "Make remark" events a Good Thing[12], because they reflect just the some-benefit-at-no-cost type of advantage for which he suggested the side-by-side programming practice.

5 Discussion

5.1 Threats to internal validity

Internal validity refers to the degree to which the results as presented were correctly derived from the raw observations underlying the study.

The only threat to validity for a Grounded Theory study is incomplete or incorrect grounding of the statements made. As for incompleteness, our grounding is complete except for one point: Our notion of "dominance of a set of concepts within a particular episode" is an intuitive concept we have not formally grounded in the observations (although we are sure that this can be done). Furthermore, due to lack of space in this article, the description of our grounding is limited to

[9] Similar event sequences via non-physical cooperation lose their composite nature and would be described by the respective sequence of sub-episodes only.
[10] That is why they are missing in Figure 1.
[11] That is why they are missing in Figure 2.
[12] Personal communication January 2008

brief examples (and sometimes not even that) rather than the detailed concept descriptions that are the hallmark of Grounded Theory.

As for correctness, our encodings will arguably contain a number of concept confusions (concept A has been chosen although concept B could be argued to be more appropriate). Such confusions can occur when multiple interpretations of an utterance are possible. For the given episode type classification, however, such confusions are unproblematic: Wherever they occur, the respective type description will accomodate A as well as B anyways, because many of the similar episodes will have clear As and many will have clear Bs at the corresponding point.

5.2 Threats to external validity

External validity refers to the degree to which the results as presented generalize to other settings than those observed in the study.

The setting of our study was not at all generally representative of realistic software development situations (laboratory environment, relatively inexperienced subjects, simple task context) and we have analyzed only three pairs. For most kinds of study this would lead to rather weak external validity. Not so in our case; we are convinced that the types we found are all valid, for the following reason: It is conceivable that industrial practice may exhibit additional types of cooperation episodes or (more likely) that it might be helpful to split up some of our types, but we have no reason to believe that one of our types is malformed or does not occur in other settings. Remember that most of our classification rests on the concepts of the PP foundation layer, which *does* reflect industrial practice.

5.3 Comparison to the Dewan et al. study

In their study of distributed side-by-side programming [9], Dewan et al. offer a list of what they call "work modes". These can be compared to our cooperation episode types:

- Concurrent uncoupled programming. This is the solo mode, without any immediate cooperation.
- Concurrent coupled programming. Both partners continue working on their own machine, but also talk to each other. This includes our "Make remark" episode type but did otherwise not occur in our setting. We can only speculate on the reasons: perhaps the larger distance of the partner's screen makes this mode unhelpful. Otherwise, our episode types "Exchange project details" and "Discuss step" would be good candidates for occurring in this mode.
- Pair programming. All our types can occur in this mode.
- Concurrent programming/browsing. This is like pair programming but the observer uses her own display for looking at different material than the driver. We have not seen such behavior, presumably for the reason discussed for concurrent coupled programming above.
- Concurrent browsing. Both partners browse code or documentation. This may or may not involve cooperation and is also not present in our data.

As we see, this study focussed on a more technical operational mode for describing what is going on whereas our own classification is oriented towards describing the actual content of the cooperation. We make two observations: First, the two resulting schemas are quite different, but compatible. Second, the results point out that different technical/organizational settings (such as the described distributed setup) may produce cooperation modes we have not seen in our study. Both of these observations corroborate the statements regarding validity we made above.

5.4 Usefulness of the cooperation episode types

What are the benefits from understanding the episode types? We can think of three. First, the names of the types of cooperation episode provide a standardized vocabulary that eases communication in much the same way as the names of design patterns do [23].

Second, knowing the episode types contributes to a clear mental model of the side-by-side process. In particular for programmers using metacognition [26], it provides a better tactical orientation by helping to answer the questions "What is the purpose of the current cooperation episode?" and "So what should I focus on?" .

Third, the results suggest an entry in a side-by-side programming etiquette that says "Do not shy away from asking your partner for project details, even if you could find them out yourself. Asking is often more efficient overall, at least if you keep the interaction short." Presumably, if this rule was wrong, we would not have seen such a large number of "Exchange project details" episodes.

6 Conclusion

Of the research questions formulated in Section 1.2, our results provide an answer for the question "For what purpose do side-by-side programmers start a cooperation episode?". We have found that they do it in order to either agree on coarse work strategies, agree on the next work steps, exchange knowledge (either project-specific or generic), perform debugging, or integrate their work results.

Explicitly knowing this set of cooperation episode types provides a valid and shared schema [28] of how sidy-by-side programming works which aids thinking about side-by-side programming more clearly. This can help practitioners to cooperate more effectively and efficiently [10].

It also helps to identify the next research questions regarding side-by-side programming: Subsequent work should now study what types of pairs excel (or limp) in which of the episode types and what kinds of behavior in each of the episode types makes an efficient and successful completion of the respective episode most likely.

Acknowledgments

We thank our subjects for participating in the study and reviewer 1 for rather concrete and keen remarks.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 1999.
2. Kent Beck. *Extreme Programming Explained: Embrace Change, Second Edition.* Addison-Wesley Professional, 2004.
3. Sallyann Bryant. Double trouble: Mixing qualitative and quantitative methods in the study of extreme programmers. In *Proceedings of the 2004 IEEE Symposium on Visual Languages – Human Centric Computing (VL/HCC 2004)*, pages 55–61, Washington, DC, USA, 2004. IEEE Computer Society.
4. Sallyann Bryant, Pablo Romero, and Benedict du Boulay. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, in press.
5. L. Cao and P. Xu. Activity patterns of pair programming. In *Proc. of the 38th Annual Hawaii International Conf. on System Sciences (HICSS 2005)*, page 88a, Washington, DC, USA, 2005. IEEE Computer Society.
6. Jan Chong and Tom Hurlbutt. The social dynamics of pair programming. In *ICSE07: Proceedings of the 29th Int'l Conf. on Software Engineering*, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.
7. Marcus Ciolkowski and Michael Schlemmer. Experiences with a case study on pair programming. In *Workshop on Empirical Studies in Software Engineering*, 2002.
8. Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley Longman, 2004.
9. Prasun Dewan, Puneet Agarwal, Gautam Shroff, and Rajesh Hegde. Distributed side-by-side programming. In *Proc. ICSE Workshop on Collaborative and Human Aspects on Software Engineering (CHASE).* IEEE CS Press, 2009.
10. B.D. Edwards, E.A. Day, W. Arthur, and S.T. Bell. Relationships among team ability composition, team mental models, and team performance. *J. of Applied Psychology*, 91:727–736, 2006.
11. Sallyann Freudenberg (née Bryant), Pablo Romero, and Benedict du Boulay. "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? In *AGILE 2007*, pages 84–91, Washington, DC, USA, 2007. IEEE Computer Society.

12. Brian Hanks, Charlie McDowell, David Draper, and Milovan Krnjajic. Program quality with pair programming in CS1. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 176–180, New York, NY, USA, 2004. ACM Press.
13. Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *IEEE Software*, 18(5):120–122, September 2001.
14. Udo Kelle. "emergence" vs. "forcing" of empirical data? a crucial problem of "Grounded Theory" reconsidered. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, 6(2), 2005.
15. Kim Man Lui and Keith C.C. Chan. When does a pair outperform two individuals? In *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 225–233. Springer, 2003.
16. Lech Madeyski. *Software Engineering: Evolution and Emerging Technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, chapter Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality, pages 113–123. IOS Press, 2005.
17. Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. The effects of pair programming on performance in an introductory programming course. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 38–42. ACM Press, 2002.
18. Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. The impact of pair programming on student performance, perception, and persistance. In *ICSE '03: Proc. 25th Int'l Conf. on Software Engineering*, pages 602–607. IEEE Computer Society, 2003.
19. Nachiappan Nagappan, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. Improving the CS1 experience with pair programming. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 359–362, New York, NY, USA, 2003. ACM Press.
20. Nachiappan Nagappan, Laurie A. Williams, Eric Wiebe, Carol Miller, Suzanne Balik, Miriam Ferzli, and Julie Petlick. Pair learning: With an eye toward future success. In *XP/Agile Universe*, volume 2753 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2003.
21. Jerzy R. Nawrocki, Michal Jasiñski, Lukasz Olek, and Barbara Lange. Pair programming vs. side-by-side programming. In *Software Process Improvement*, volume 3792 of *Lecture Notes in Computer Science*, pages 28–38. Springer, 2005.
22. John T. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, 1998.
23. Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern information during program maintenance. *IEEE Trans. on Software Engineering*, 28(6):595–606, June 2002.
24. Stephan Salinger, Laura Plonka, and Lutz Prechelt. A coding scheme development methodology using Grounded Theory for qualitative analysis of Pair Programming. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG '07)*, pages 144–157, Joensuu, Finland, July 2007. www.ppig.org, a polished version appeared in: Human Technology: An Interdisciplinary Journal on Humans in ICT Environments, 4(1):9-25, May 2008.
25. Stephan Salinger and Lutz Prechelt. What happens during pair programming? In *Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group (PPIG '08)*, Lancaster, England, September 2008. www.ppig.org, to appear.
26. Teresa Shaft. Helping programmers understand computer programs: the use of metacognition. *ACM SIGMIS Database*, 26(4):25–46, November 1995.
27. Helen Sharp and Hugh Robinson. An ethnographic study of XP practice. *Empirical Software Engineering*, 9(4):353–375, December 2004.
28. Eliot R. Smith and Sarah Queller. Mental representations. In Abraham Tesser and Norbert Schwarz, editors, *Blackwell Handbook of Social Psychology: Intraindividual Processes*, pages 111–133. Blackwell, London, 2001.
29. Anselm L. Strauss and Juliet M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE, 1990.
30. Laurie Williams. Integrating pair programming into a software development process. In *Proceedings of the 14th Conference on Software Engineering Education and Training (CSEET 2001)*, Washington, DC, USA, 2001. IEEE Computer Society.
31. Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.
32. Laurie Williams and Robert R. Kessler. Experimenting with industry's "pair-programming" model in the computer science classroom. *Journal of Software Engineering Education*, December 2000.
33. Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, 2000.
34. Laurie Williams and Richard L. Upchurch. In support of student pair-programming. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 327–331, New York, NY, USA, 2001. ACM Press.
35. S. Xu, V. Rajlich, and A. Marcus. An empirical study of programmer learning during incremental software development. In *Fourth IEEE Conf. on Cognitive Informatics (ICCI 2005)*, pages 340–349, Los Alamitos, CA, USA, 2005. IEEE Computer Society.