Towards Understanding Communication Structure in Pair Programming

Kai Stapel, Eric Knauss, Kurt Schneider, and Matthias Becker

Software Engineering Group, Leibniz Universität Hannover,
Welfengarten 1, 30167 Hannover, Germany
{kai.stapel,eric.knauss,kurt.schneider}@inf.uni-hannover.de,
matthias.becker@stud.uni-hannover.de

Abstract. Pair Programming has often been reported to be beneficial in software projects. To better understand where these benefits come from we evaluate the aspect of *intra-pair communication*. Under the assumption that the benefits stem from the information being exchanged, it is important to analyze the types of information being communicated. Based on the Goal Question Metric method we derive a set of relevant metrics and apply them in an eXtreme Programming class room project. Data covering a total of 22.9 hours of intra-pair communication was collected. We found that only 7% of the conversations were off-topic (e.g. private), 11% about requirements, 14% about design, and 68% about implementation details (e.g. syntax). Accordingly, a great share of the information being exchanged in Pair Programming is on a low level of abstraction. These results represent a first data point on what kind of information is communicated to what extent during Pair Programming.

Keywords: Pair Programming, Communication, Empirical Study.

1 Introduction

When Kent Beck introduced eXtreme Programming (XP) it was considered to be the antithesis to document- and plan-driven software development [1]. Beck defined the XP practices in a provocant manner: Accordingly, all practices have to be *turned to ten*, i.e. best practices in software development should be practiced in an extreme manner. For example, Beck states that if reviews are good then all code should constantly be reviewed. Therefore, Pair Programming (PP) should be performed. Today, PP is one of the most prominent XP practices in use, even outside of XP. Many studies investigated the effectiveness of PP and its impact on code quality. PP is mostly considered to be superior to conventional single developer programming, because pairs produce better quality code, learning takes place, team building is speeded up, while development is about as effective as in the solo case [2,3,4,5,6]. These benefits are usually attributed to the intensive intra-pair communication that takes place when working in pairs. According to Lindvall et al. communication is one of the three most important success factors of PP [6]. Still, not many studies have invastigated the aspect

A. Sillitti et al. (Eds.): XP 2010, LNBIP 48, pp. 117–131, 2010.

communication in PP in detail. As a consequence, studying communication in Pair Programming and its effect on development success has been proposed by several research frameworks, e.g. by Gallis et al. [7] and Ally et al. [8].

In the work presented here we focus on analyzing intra-pair communication to better understand its connection to the benefits of PP. Two often mentioned benefits that are related to communication are learning and team building:

Learning: If one partner shares technical experience with his or her peer, learning takes place. The driver (the one who uses the keyboard) is even able to share tacit knowledge similar to the master-apprentice type of learning. Better educated developers are more likely to produce high quality code.

Team Building: If pairs closely collaborate and talk about off-topic issues, team building takes place. In addition, this kind of communication establishes a common context among partners which makes future communication simpler.

This paper is structured as follows. In Section 2 we discuss related work on measuring communication in software development and case studies investigating communication in PP. In Section 3 we define our research goals by applying the Goal Question Metric (GQM) method. This leads to the metrics we then apply in a case study with students in an XP class room project that is described in Section 4. Section 5 summarizes our results. In Section 6 we interpret our results and discuss their validity. Finally, we conclude in Section 7 and give an outlook on future work.

2 Related Work

Dutoit and Bruegge propose to use communication metrics to get project status information early on when no product related metrics are available yet [9]. They use digital communication media that is used for collaboration in class room software projects to derive communication metrics. For example, the number of messages per week in an electronic bulletin board are counted. These metrics are then applied in a series of software engineering project classes. The results are used to improve communication in the projects over time. Statistical analysis shows that bulletin board communication size and complexity has a significant positive correlation with project outcome. Here project outcome refers to source code as well as corresponding documents. In the work of Dutoit and Bruegge communication metrics are applied in non-agile projects and on non-verbal communication media. Still, counting and classifying communication events and using them as an early indicator for project outcome can also be applied in our agile setting on verbal intra-pair communication.

Sfetsos et al. investigated the impact of personality types on communication and collaboration-viability in Pair Programming [10]. They compared the effects of uniform versus mixed temperament type pairs on communication, collaboration-viability, and effectiveness using a controlled experiment with 42 pairs (84 subjects/students). Temperament types were classified according to

Keirsey Temperament Sorter (KTS). Communication was measured by counting communication events. Five different communication topics were considered: (1) Requirements gathering, (2) specification and design changes, (3) code, (4) unit tests, and (5) peer reviewing. The subjects (i.e. the navigator) had to self-report their communication activities as they were working on their tasks. Analysis showed that the number of communication transactions was significantly positive correlated with productivity (measured as points for correct solutions) and that mixed personality pairs communicated significantly more than uniform personality pairs. Although they have measured different types of communication similar to the classification we used (requirements, design, or code related, see table 3, M1.1-M1.3), they do not report their results concerning distribution of the topics and total amount of communication events. They also do not report about the durations of the communication events.

Bryant et al. analyzed 23 hours of pair programmers' dialogue and used the results as an indicator for collaboration of the pairs [11]. The dialogues stem from experienced professional pair programmers. They found that pairs had a high amount of verbal interactions. The pairs produced more than 250 verbal interactions per pair programming hour. Almost all tasks were contributed to by the driver and by the navigator in collaboration (more than 90% in 10 out of 12 task categories) with the driver contributing slightly more. They conclude that: "the benefits attributed to pair programming may well be due to the collaborative manner in which tasks are performed". As opposed to our study, off-topic dialogues were left out in the analysis. Therefore, they did not report the ratio of project relevant to off-topic conversations. Other differences to our study are that we observed students instead of professionals, that we had dedicated observers classifying the conversations on the fly as opposed to analyzing audio recordings afterwards, and that we had a different approach: we first chose how to classify communication and then measured it as opposed to first recording the communication and then classifying it.

3 Research Question

The main goal of the work presented here is to better understand the communication structure in Pair Programming. We chose the Goal Question Metrics paradigm [12] to refine this main goal to metrics that can be measured in a case study. In the first step we derived two GQM goals. For each GQM goal an abstraction sheet [12] was created. Based on these abstraction sheets the metrics for measuring communication in PP and finally a measurement plan were developed.

3.1 Goals

Our aim is to better understand the structure of Pair Programming communication. We decompose this into the following two GQM goals:

Goal 1. Understand the quality aspect *level of abstraction* of intra-pair communication in PP. In this context level of abstraction refers to the abstractness

of the information according to the software development phase. For example, requirements are on a high level of abstraction whereas program code issues are on a low level. Program design and architecture are somewhere in-between. So this goal tries to address the question: Are the developers talking about requirements, design patterns, syntax, or off-topic issues?

Goal 2. Understand the quality aspect *interactivity* of intra-pair communication in PP. Interactivity is the distribution of the drivers' and the navigators' share of communication. So this goal addresses the question: Who is communicating more, the driver or the navigator?

Abstraction sheets help to find the relevant factors that directly make up or indirectly influence the quality aspect of a GQM goal. Abstraction sheets allow to derive reasonable metrics for these factors. In table 1 and 2 the abstraction sheets for the two GQM goals are shown. Table 3 shows the derived metrics.

In an abstraction sheet the factors that directly make up the quality aspect of a GQM goal are called quality focus (QF). For the first goal we identified 4 quality factors (see table 1): the number of conversations about requirements (QF 1), design (QF 2), code (QF 3), and off-topic things (QF 4). We expect the greatest proportion of the communication to be about code (QF 3), because the students in our case study were rather inexperienced programmers. If the developers are more experienced in the usage of the programming language, they do not talk about syntax topics as much. We also expect a lot of conversations about requirements (QF 1), since the Story Cards that were used in the XP project are rather brief and do not contain many details. Usually, in XP the details of the requirements are directly exchanged between the On-site Customer and the developers. If a pair is talking to the On-site customer about a requirement each developer might interpret the details differently. This may lead to the need for discussion in the pair later. Design issues (QF 2) and off-topic (QF 4) conversations will probably not make up a big part of the communication, because the XP course required the participants to apply the practice Simple Design and because the developers have never worked together in a project before and therefore were unfamiliar with each other. Still, we believed once the developers get to know each other better, they will talk more about private things. Another aspect that characterizes the communication is the duration of the conversations about one topic. For example, we expect the conversations about design issues to be longer than the conversations about syntax issues.

Besides the quality focus an abstraction sheet allows for gathering variation factors (VF). Variation factors are factors that indirectly have an effect on the quality aspect. For our first goal we identified 7 variation factors, which include the developers' experience in requirements analysis, program design, and programming in general (i.e. writing source code). The variation hypotheses in abstraction sheet 1 describe how each variation factor influences the quality aspects.

The second goal is to understand the interactivity of the communication between driver and navigator. Here, we investigate whether driver or navigator contribute more to communication. Also, the frequency of the questions being

Table 1. Abstraction sheet for GQM goal "level of abstraction"

Goal 1: Analyze *intra-pair communication in Pair Programming* for the purpose of *understanding* with respect to the ***level of abstraction*** from the perspective of the *developers* in the context of a *class room XP project*.

Quality Focus	**Variation Factors**
QF 1. Conversations about requirements	**VF 1.** Quality of requirements
QF 2. Conversations about design (e.g. module structure)	**VF 2.** Complexity of software design
QF 3. Conversations about code (e.g. syntax)	**VF 3.** Developers' experience in requirements analysis
QF 4. Off-topic conversations (non software development related, private)	**VF 4.** Developers' experience in program design
	VF 5. Developers' experience with programming
	VF 6. Knowledge differences between developers
	VF 7. Familiarity with other developers

Baseline Hypotheses	**Variation Hypotheses**
1. Because of the brief nature of Story Cards and the developers being not very experienced in requirements analysis we expect many conversations about requirements ($> 30\%$)	1. More clear, thorough, and unambiguous, i.e. high quality, requirements lead to fewer communication about them (QF 1).
2. Because of the XP practice "Simple Design" we expect few ($< 10\%$) conversations about design topics. But if the developers talk about design the conversations will be rather long (> 3 min.)	2. The clearer and simpler a software design is, the less communication is needed about design issues (QF 2).
3. Because of the inexperienced programmers we expect many ($> 50\%$) but short (< 1 min.) conversations about code topics	3. Experienced requirements analysts talk less about requirements related issues (during programming), because their requirements are of good quality and have fewer room for interpretation (QF 1).
4. Because of the unfamiliarity with each other we expect a small proportion of off-topic or private communication ($< 10\%$)	4. Experienced designers talk less about design related issues (QF 2), because best practices in program design like design patterns are common knowledge.
	5. Experienced programmers talk less about syntax related issues (QF 3), because they do not make as many syntax mistakes as inexperienced programmers.
	6. Higher differences in software development knowledge between the developers lead to a lot communication about requirements (QF 1), design (QF 2), and code (QF 3)
	7. Good knowledge of the other person leads to a great proportion of private conversations (QF 4)

Table 2. Abstraction sheet for GQM goal "interactivity"

Goal 2: Analyze *intra-pair communication in Pair Programming* for the purpose of *understanding* with respect to the ***interactivity*** from the perspective of the *developers* in the context of a *class room XP project*.	
Quality Focus	**Variation Factors**
QF 1. Balance of communication proportions **QF 2.** Question frequency	**VF 1.** Number of navigators' clarification questions
Baseline Hypotheses	**Variation Hypotheses**
1. We expect the driver to talk slightly more than the navigator ($\approx 60:40$), because in a similar study the distribution between driver and navigator already was $60:40$ [11]. 2. According to Williams and Kessler [13] we expect a question at least every minute.	1. The number of the navigators' clarification questions is positively correlated with the question frequency (QF 2).

asked characterizes the interactivity. We expect the driver to talk slightly more than the navigator, because a similar study already found that the driver contributes slightly more to a given task [11]. Another assumption is that the pairs will discuss a question at least every minute [13]. A variation factor for the interactivity of the communication is the number of the navigators' clarification questions, which is positively correlated with the overall question frequency.

3.2 Metrics

The next step in the GQM approach is to create metrics based on the abstraction sheets. Metrics are needed to answer the questions associated with each goal. Ideally, at least one metric should be set up for each quality factor (QF) and each variation factor (VF). Table 3 summarizes the metrics we used in our case study. Each metric has an ID, a name, a scale type (e.g. ordinal, ratio), a range or value set, a "counting" rule that briefly explains how to measure it, and a reference to the corresponding abstraction sheet. The scale type is important for later analysis. For example, mean and standard deviation are not defined for ordinal scales. Hence, other characteristics like median and quartiles are needed for interpretation of results on an ordinal scale.

For the first abstraction sheet (see table 1) metrics for all but two factors were derived. We did not create a metric for the quality of requirements (VF 1), because we did not expect them to vary much. All pairs used Story Cards that the team created in collaboration with the customer at project start. Furthermore, measuring quality of requirements is a problem of its own [14]. We also did not explicitly measure the familiarity (VF 7) of the developers with each other. But, we know that none of them had worked in pairs with the others before. For the second abstraction sheet (see table 2) all metrics were created.

Table 3. Metrics

ID	Name	Scale type	Range	Counting rule	Abstract. sheet ref.
M1.1	Number of conversations about req.	ratio	$\mathbb{N}^+$	For each closed conversation about one requirement	Goal 1, QF 1
M1.2	Number of conversations about design	ratio	$\mathbb{N}^+$	For each closed conversation about one design topic, e.g. module structure	Goal 1, QF 2
M1.3	Number of conversations about code	ratio	$\mathbb{N}^+$	For each closed conversation about one program code topic, e.g. syntax	Goal 1, QF 3
M1.4	Number of off-topic conversations	ratio	$\mathbb{N}^+$	For each closed conversation about one not project relevant topic	Goal 1, QF 4
M1.t	Time of conversations	ordinal	short ($<$ 1 min.), med. (1-3 min.), long ($>$ 3 min.)	Map each closed conversation about one topic to a time interval	Goal 1, QF 1-4
M1.5	Complexity of design	ratio	$\mathbb{R}$, ≥ 1	Average cyclomatic complexity over all classes per iteration	Goal 1, VF 2
M1.6	Req. analysis experience	ordinal	1 (very exp.), 2, 3, 4, 5, 6 (none)	Self-evaluation of developers	Goal 1, VF 3
M1.7	Design experience	ordinal	1 (very exp.), 2, 3, 4, 5, 6 (none)	Self-evaluation of developers	Goal 1, VF 4
M1.8	Programming experience	ordinal	1 (very exp.), 2, 3, 4, 5, 6 (none)	Self-evaluation of developers	Goal 1, VF 5
M1.9	Deviation in developers knowledge	ratio	$\mathbb{R}$	Standard deviation of average self-assessment grades	Goal 1, VF 6
M2.1	Proportion of drivers' share in conversation	ratio	$0\% - 100\%$	Mark on scale (see figure 1)	Goal 2, QF 1
M2.2	Number of questions per hour	ratio	$\mathbb{R}$	For each question asked	Goal 2, QF 2
M2.3	Nr. of navigators' clarification questions per hour	ratio	$\mathbb{R}$	For each question asked	Goal 2, VF 1

4 Study Design

We chose the eXtreme Programming (XP) course at Leibniz Universität Hannover, Germany to apply our metrics. This course is a mature XP laboratory [15] where students first learn the XP practices in theory - including Pair Programming - and then apply their new XP knowledge in a short XP project in practice. The following tool set was used in the project: Eclipse, SVN, Ant, and JUnit4. In 2009 we combined the XP course with an empirical study course. Before the actual XP project started the students had to prepare an empirical study. We had 2 XP teams with 7 developers each. In each team 6 developers formed 3 pairs. The 7^{th} developer was the observer of the other team. The observers were switched every day. The development pairs had to switch regularly, at least every new day or after every finished Story Card. Intra-pair communication was measured just in one team. The project was a 5-day-project with 7 development hours a day. The first day was for a technical spike and requirements gathering. The students interviewed the customer and created the Story Cards. The remaining 4 days were 4 development iterations. An on-site customer with a real development task was available to the developers throughout the whole project.

4.1 Subjects

The observed team of this study consisted of 3 undergraduate (BSc level) and 4 graduate (MSc level) students. On average the students were in their 11.0^{th} (± 2.8) semester in computer science. Their curriculum covered programming lectures, topics like requirements engineering or design patterns, and a non-agile software project. According to their self-assessment they were least confident in their design abilities and most confident in their programming skills (see table 4, M1.6-M1.8). All but one student were new to Pair Programming. They have never worked in pairs with each other before. Since the class was also an empirical study class (and because of the present observer) the pairs knew they were being measured, but they did not know what about.

4.2 Data Collection

Before project start the developers had to fill out a self-assessment questionnaire. Results from this questionnaire were used for the metrics M1.6 - M1.9. For all other metrics a data collection sheet was prepared (see figure 1). During the project an observer monitored a pair, classified their conversations, and filled out the data sheet. One data collection sheet was filled in for a single pair for a period of up to two hours. After two hours or when the pair switched the observation was stopped and a new data collection sheet with a new pair was started. During an observation the durations and topics of the intra-pair conversations were noted. Each conversation duration was assigned to a group depending on whether it was shorter than one minute, longer than three minutes or in between. A conversation has been considered *closed* once the pair had stopped talking or the topic changed. Then a stroke was made in the corresponding column and

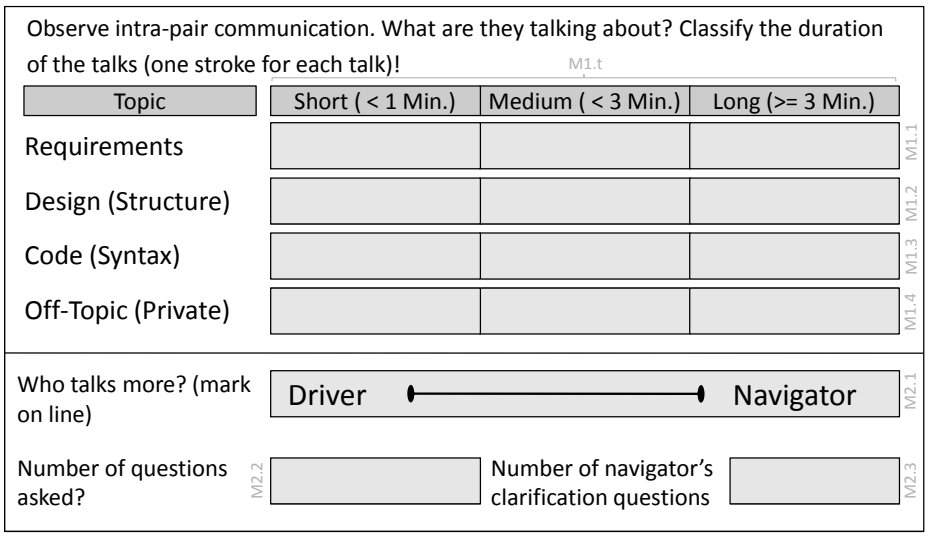


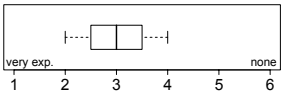
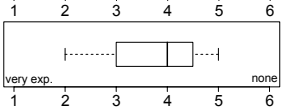
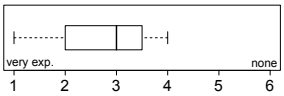
Fig. 1. Data collection sheet

in the row given by the topic. Corresponding to the metrics the available topics were requirements, design, code, and off-topic. Requirements covered conversations about functions or quality aspects of the software under development or preparations for a talk to the on-site customer. Design topics include dicussions about the program stucture, e.g. the module and package structure, or the usage of design patterns. Code topics include everything that is low-level source code related like syntax issues. Off-topic covers everything that is not software development related like private conversations. Parallel to this every question asked was noted in M2.2 and the number of clarification questions (the driver makes a decision and the navigator has to ask to understand it) was noted in M2.3. At the end of each observation the driver's share of the just monitored intra-pair communication was estimated by drawing a stroke on the line in M2.1, where the left end represents 100% and the right end 0%. According to the student's feedback the data collection sheet was easy to use. Putting an extra developer in the team for observation was a reasonable effort. There has been a short talk between the observers after they switched each morning to share experiences from data collection to ensure constant quality of the collected data.

5 Results

In total 22.9 hours of intra-pair communication were monitored. Out of the 21 possible pair combinations (7 developers forming pairs) data from 14 different pairs was collected and analyzed. Table 4 summarizes the results. In total, 216 conversations were observed. Almost half of the conversations were short (< 1 minute). The other half of the conversations was evenly split in medium ($1 - 3$

Table 4. Results

Metric	Name	Result	Range/Unit
M1.1	Number of conversations about requirements	23 of 216 (11%)	$\mathbb{N}^+$ of total (in percent)
M1.2	Number of conversations about design	30 of 216 (14%)	$\mathbb{N}^+$ of total (in percent)
M1.2	Number of conversations about code	148 of 216 (68%)	$\mathbb{N}^+$ of total (in percent)
M1.4	Number of off-topic conversations	15 of 216 (7%)	$\mathbb{N}^+$ of total in percent
M1.t	Time of conversations	short: 104 (48%), medium: 52 (24%), long: 60 (28%)	$\mathbb{N}^+$ for each time interval (in percent of total)
M1.5	Complexity of design	$m_{it1} = 1.00$, $m_{it2} = 1.04$, $m_{it3} = 1.16$, $m_{it4} = 1.12$	Average cyclomatic complexity per iteration
M1.6	Requirements analysis experience		Boxplot (min, lower quartile, median, upper quartile, max)
M1.7	Design experience		Boxplot (min, lower quartile, median, upper quartile, max)
M1.8	Programming experience		Boxplot (min, lower quartile, median, upper quartile, max)
M1.9	Deviation in developers knowledge	$s = 0.9$	Standard deviation of average self assessment grades
M2.1	Proportion of drivers' share in conversations	$\approx 40\%$ ($\pm \approx 20\%$)	$0\% - 100\%$ ($\pm$ std. deviation)
M2.2	Number of questions per hour	9.9 ($\pm$8.1)	$\frac{1}{\text{h}}$ ($\pm$ std. deviation $\frac{1}{\text{h}}$)
M2.3	Number of navigators' clarification questions per hour	2.1 ($\pm$1.6)	$\frac{1}{\text{h}}$ ($\pm$ std. deviation $\frac{1}{\text{h}}$)

min.) and long (> 3 min.) conversations. Two out of three conversations were about code issues, 14% about design, 11% about requirements, and 7% about off-topic things. Figure 2 summarizes the results for the metrics M1.t and M1.1 to M1.4 by showing the overall distribution of the conversation durations on the left and the conversation topics on the right. Code analysis showed that the average code complexity of the software did not exceed 1.16 throughout the project. The self-evaluation metrics showed that the developers estimated their requirements analysis experience with a median of 3, on a scale from 1 to 6 with 1 being very experienced and 6 having no experience at all, to be slightly more

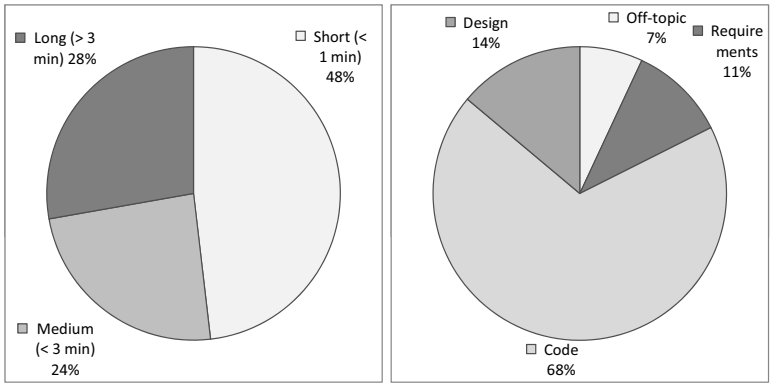


Fig. 2. Overall distribution of conversation durations (M1.t, left) and discussed topics (M1.1-M1.4, right)

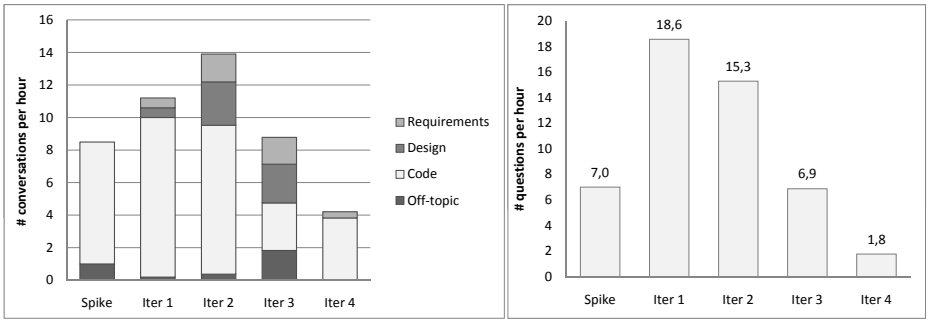


Fig. 3. Number of conversations per hour for each topic (left) and number of questions per hour (right) over the project time

experienced than the middle of the scale. With a median of 4 the developers were less confident in their design abilities. With a median of 3 and a lower quantile of down to 2 the developers were most confident in their programming skills. Still, they were far from classifying themselves to be very experienced programmers. The overall standard deviation of the combined measures of all software development related self-evaluation questions was 0.9. This means that about 68% of the developers ($\approx$ 5 of 7) self-evaluate their knowledge to be in a region that does not span more than 2 points, e.g. between 2 and 4. So, the majority of the students were on a similar level of knowledge.

On average the drivers' shares in conversations were approximately 40% with a standard deviation of approximately 20%. Over the observed 22.9 hours of inta-pair communication 227 questions were asked. Figure 3 (right) shows a decrease in the number of questions per hour over the course of the project. The navigators asked a total of 49 clarification questions.

6 Dicussion

6.1 Interpretation

Goal 1: Level of Abstraction of Communication. We expected the ratio of conversations about requirements to be more than 30%, because of the brief nature of the used Story Cards and the developers being not very experienced in requirements analysis (see abstraction sheet 1, baseline hypotheses). But the ratio of conversations about requirements was only 11%. So either the students were better than they thought or it is typical for PP that requirements are not being discussed that much. 14% of the conversations were about design issues, which was slighlty more than expected. This might be due to the fact that the students had not much experience in software design and therefore had to discuss these issues more. With a ratio of 68% by far the most conversations were about code. This is even more than the expected 50%. The students were not very experienced programmers, so this high value may be attributed to their need for learning low level programming. The decrease of conversations about code over the project time (figure 3, left) while the conversations about design remained constant and the decrease of questions being asked (figure 3, right) are also indicators for learning that took place. A proportion of 7% for off-topic conversations was even lower than the expected 10%. But the increase of the number of off-topic conversations in iteration 3 (see figure 3, left) indicates that once the developers get to know each other better they are more likely to have private conversations during PP. This is a sign for team building. We attribute the decrease of off-topic and design conversations in the last iteration to the stress at project end. Finally, the results support our assumption that conversations about design on average are longer than conversations about code. Figure 4 shows that there are more medium and long conversations about design than there are for conversations about code.

Goal 2: Interactivity of Communication. Along with the findings of Bryant et al. [11] we expected the driver to talk slighlty more than the navigator, about 60:40. But our results showed that it was the other way around. In our case it was approximately 40:60. So our assumption turned out to be wrong. The driver does not seem to constantly explain his or her actions. Maybe the navigator is guiding the programming by communicating the current and near future actions to take, while the driver is busy typing in the according source code.

Williams and Kessler state that driver and navigator communicate at least every 45 to 60 seconds [16], if only through utterances as in “Huh?”. We could not confirm this finding. In our study either driver or navigator asked a question every 6.1 minutes on average. This large difference might be due to several reasons: First, an utterance is much smaller than a real question. So we might have missed the small utterances during observation. Second, the students were new to PP and unfamiliar with each other. So they might not have been as communicative as the professional programmers Williams and Kessler refer to. Despite of the bias of our more coarse measurement we believe that novice pair programmers are much less communicative than expert pair programmers.

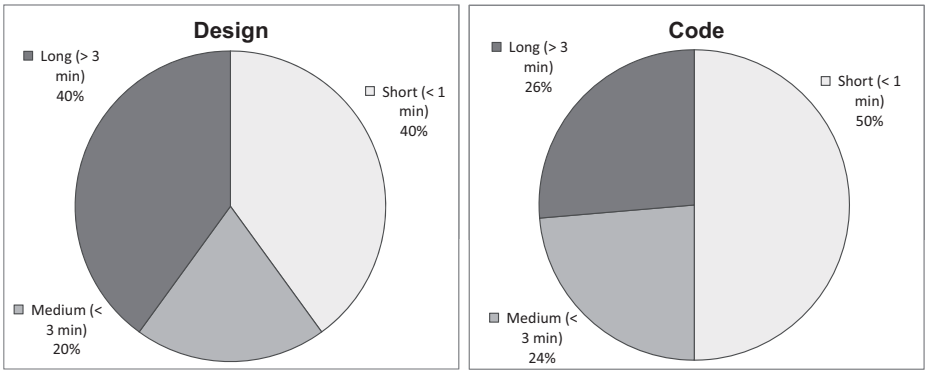


Fig. 4. Distribution of conversation durations for design topics and code topics

6.2 Threats to Validity

Conclusion Validity. The large standard deviations in some of our metrics indicate that we need a lot more data points, i.e. more pairs and longer observation periods, in order to increase conclusion validity. Another threat to conclusion validity is that the classification of the conversation topics was done subjectively by different observers. We tried to reduce that threat by setting some standards on how to classify the topics and had the observers discuss their experiences when switching. Despite the low number of data points and the subjectivity of our measures the results give a first indication of the structure and content of intra-pair communication.

Internal Validity. Internal validity is concerned with whether a treatment really causes the outcome of a study. The only treatment in our study was the fact that we observed pair programmers. The collected data and the observations of the supervisor show that the developers were actually working as pair programming suggests it [1]. Still, a threat to internal validity is that the distribution and frequency of communication also depends on other factors beyond PP. For example, personality types have already been shown to be significantly related to the frequency of communication in PP (see related work section) [10]. We did not measure personality types in our study. But, our data covers 14 out of 21 possible pair combinations, so the effects of personality types are partially evened out.

Construct Validity. The self assessment measures for the developer knowledge have a rather poor construct validity. They only reflect the developers' own opinion and not their actual knowledge. A second threat to construct validity is caused by the fact that the developers knew they were being observed. This might have had a distorting influence on the ratio of off-topic conversations, since observed developers tend to better follow the process (in this case PP) and do not talk as much about private things.

External Validity. There are two major external threats to validity: (1) We chose students that were new to PP and relatively inexperienced in programming at all. (2) The 5 day XP project might have been too short to show representative data for industrial pair programming settings. Usually, in industry developers know each other a little better and work together longer than 5 days. Despite of these threats our results can be used as baseline hypotheses for future studies in industrial contexts. Especially the difference between experienced pair programmers and deverlopers who are new to PP should be analyzed. Our assumption is that our results are most likely valid for developers who are learning PP.

7 Conclusion and Outlook

Our goal was to better understand the communication structure in Pair Programming. To reach this goal we applied the GQM method and derived relevant metrics to measure the intra-pair communication in PP. These metrics were then applied in a 5 day XP class room project with 7 students. 22.9 hours of communication were observed and analyzed. Two third of all conversations were about code related topics. This indicates that during PP neither design, requirements, or unrelated things are as important topics as communication about code. This holds at least for developers who are new to PP with programming experience at the level of a graduate student. Furthermore, the decrease of conversations about code over time, while conversations about design remained constant, and the decrease of the number of questions being asked indicates that learning took place. The developers increased their programming abilities and therefore did not have to talk as much about code issues anymore. Additionally, a small indicator for successful team building could be observed: Off-topic conversations slightly increased in the third iteration.

In future work we need to refine our measures further. Especially conversations about code need to be analyzed in more detail. Are syntax issues really dominating? If so, maybe Williams' and Kessler's "Myth 6: The navigator finds only syntax mistakes. How boring is that! Compilers can do that better than humans can anyway." [16] is almost not a myth after all. Maybe there is a chance to increase PP productivity even further by providing better low level code support and thereby allowing the pairs to concentrate their communication more on high level topics like design or even on requirements. Another interesting research question is whether some benefits of PP can be brought to solo programming by simulating a navigator, e.g. by using static code analysis data to provide more direct feedback about the code.

Finally, our findings could be useful for the design of tools that support *distributed* PP as described by Dajda and Dobrowolski [17]. For example, different support mechanisms for the different levels of abstraction can be created with an emphasis on low level code support.

Acknowledgements. We would like to thank all students who planned, executed, and participated in the study. This work was funded by the German Research Foundation (DFG project InfoFLOW, 2008-2011).

References

1. Beck, K.: Extreme Programming Explained. Addison-Wesley, Reading (2000)
2. Coman, I.D., Sillitti, A., Succi, G.: Investigating the Usefulness of Pair-Programming in a Mature Agile Team. In: XP, pp. 127–136 (2008)
3. Hulkko, H., Abrahamsson, P.: A Multiple Case Study on the Impact of Pair Programming on Product Quality. In: ICSE 2005, pp. 495–504. ACM, New York (2005)
4. Cockburn, A., Williams, L.: The Costs and Benefits of Pair Programming. In: Extreme Programming and Flexible Processes in Software Engineering XP 2000, pp. 223–247 (2000)
5. McDowell, C., Werner, L., Bullock, H., Fernald, J.: The Effects of Pair-Programming on Performance in an Introductory Programming Course. In: SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, pp. 38–42. ACM Press, New York (2002)
6. Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L., Zelkowitz, M.: Empirical findings in agile methods. In: Wells, D., Williams, L. (eds.) XP 2002. LNCS, vol. 2418, p. 197. Springer, Heidelberg (2002)
7. Gallis, H., Arisholm, E., Dybå, T.: An Initial Framework for Research on Pair Programming. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2003), pp. 132–142 (2003)
8. Ally, M., Darroch, F., Toleman, M.: A framework for understanding the factors influencing pair programming success. In: Baumeister, H., Marchesi, M., Holcombe, M. (eds.) XP 2005. LNCS, vol. 3556, pp. 82–91. Springer, Heidelberg (2005)
9. Dutoit, A.H., Bruegge, B.: Communication Metrics for Software Development. IEEE Transactions on Software Engineering 24(8), 615–628 (1998)
10. Sfetsos, P., Stamelos, I., Angelis, L., Deligiannis, I.: Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair Programming – An Empirical Study. In: Abrahamsson, P., Marchesi, M., Succi, G. (eds.) XP 2006. LNCS, vol. 4044, pp. 43–52. Springer, Heidelberg (2006)
11. Bryant, S., Romero, P., du Boulay, B.: The Collaborative Nature of Pair Programming. In: Abrahamsson, P., Marchesi, M., Succi, G. (eds.) XP 2006. LNCS, vol. 4044, pp. 53–64. Springer, Heidelberg (2006)
12. Briand, L.C., Differding, C.M., Rombach, H.D.: Practical Guidelines for Measurement-Based Process Improvement. Software Process: Improvement and Practice 2(4), 253–280 (1996)
13. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the Case for Pair Programming. IEEE Software 17, 19–25 (2000)
14. Knauss, E., Boustani, C.E., Flohr, T.: Investigating the Impact of Software Requirements Specification Quality on Project Success. In: Proceedings of 10th International Conference on Product Focused Software Process Improvement (PROFES 2009), Oulu, Finland. LNBIB, pp. 28–42. Springer, Heidelberg (2009)
15. Stapel, K., Lübke, D., Knauss, E.: Best Practices in eXtreme Programming Course Design. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008, pp. 769–776. ACM, New York (2008)
16. Williams, L., Kessler, R.: Pair Programming Illuminated. Addison-Wesley, Reading (2003)
17. Dajda, J., Dobrowolski, G.: How to Build Support for Distributed Pair Programming. In: Concas, G., Damiani, E., Scotto, M., Succi, G. (eds.) XP 2007. LNCS, vol. 4536, pp. 70–73. Springer, Heidelberg (2007)