Explaining Pair Programming Session Dynamics from Knowledge Gaps

Franz Zieris
zieris@inf.fu-berlin.de
Freie Universtität Berlin
Berlin, Germany

Lutz Prechelt
prechelt@inf.fu-berlin.de
Freie Universtität Berlin
Berlin, Germany

ABSTRACT

Background: Despite a lot of research on the effectiveness of Pair Programming (PP), the question when it is useful or less useful remains unsettled.
Method: We analyze recordings of many industrial PP sessions with Grounded Theory Methodology and build on prior work that identified various phenomena related to within-session knowledge build-up and transfer. We validate our findings with practitioners.
Result: We identify two fundamentally different types of required knowledge and explain how different constellations of knowledge gaps in these two respects lead to different session dynamics. Gaps in project-specific systems knowledge are more hampering than gaps in general programming knowledge and are dealt with first and foremost in a PP session.
Conclusion: Partner constellations with complementary knowledge make PP a particularly effective practice. In PP sessions, differences in system understanding are more important than differences in general software development knowledge.

ACM Reference Format:
Franz Zieris and Lutz Prechelt. 2020. Explaining Pair Programming Session Dynamics from Knowledge Gaps. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380925

1 INTRODUCTION

Software development is knowledge-intense. In their daily work, software developers need to be knowledgeable about languages, technology stacks, approaches to design, coding, testing, and debugging, functional and non-functional requirements, and the system's current architecture and status. Rarely all of the relevant knowledge is readily available and so *"software development [...] is a knowledge-acquiring activity"* [4].

Pair programming (PP) is a practice of two developers working together closely on the same problem. Surveys show that *knowledge transfer* is an important expected benefit of pair programming [7, 24] and some PP sessions' main purpose is to transfer knowledge [20, 25]. In practitioners' *expectations*, knowledge transfer can mean to: (1) *Combine:* The partners possess different knowledge to begin with and these combine favorably for solving the session's task faster or better. (2) *Understand:* Two developers together acquire the lacking knowledge faster and more reliably and thus catch defects in the making and produce better solutions. (3) *Learn:* Beyond the current task, the two developers learn together and from another, improving their abilities to work on future tasks.

Research goals: Generally, we want to understand how PP actually works and want to provide actionable advice to practitioners for effective PP (behavioral patterns and anti-patterns). Specifically for this article, we want to understand the role of knowledge transfer for how a PP session develops. We do *not* quantify effects and do not compare to solo programming.

Research approach: We perform deep qualitative analyses of a broad variety of industrial pair programming sessions.

Research contributions: First, we go beyond a simple dichotomy of "expert" and "novice" developers based on years of work experience and instead characterize two types of knowledge relevant for software development: system-specific S and generic G knowledge. We use these types to characterize (a) the extent of the developer's knowledge needs for working on the current task and (b) how the developers exchange and acquire knowledge to meet these needs.

Second, we describe how six different *pair constellations* (in terms of the pair's initial knowledge needs) shape the dynamics of the whole PP session and how a single global structure emerges from these six that is common to all analyzed PP sessions.

Third, we formulate and validate ideas on how practitioners may use these insights to make more informed decisions about whom to work with on which task and how to organize the resulting pair programming session.

In the following Sections, we first summarize related work (Section 2). We then describe our data collection and analysis method (Section 3) and discuss the classification of *knowledge needs* that is central for the present work (Section 4). We explain the idea of pair constellations and our finding how they, in general, lead to the overall session dynamics (Section 5). We describe five prototypes of such session dynamics with examples (Section 6). We discuss the validity our results and describe our first attempts at putting them to practical use with industrial practitioners (Section 7) before we conclude (Section 8).

2 RELATED WORK

2.1 Pair Programming Effectiveness

Pair programming studies in education focus on learning outcomes more than economic aspects of improved code quality and effort. A meta-analysis has shown a positive effect of pair programming on assignment and exam scores [23]. In industry, the focus is on

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

effort spent and quality produced. Here, a meta-analysis [16] found mere tendencies and a lot of between-study variance. Arisholm et al.'s large (quasi-)experiment [3] could not determine consistent moderating effects of task complexity and individual developer expertise on pair performance.

However, experimental studies are strongly unrealistic: In industrial contexts, it takes years to learn the specifics of a project and become fully productive [27, 32]. PP appears to require some of that learning to have happened before but can then help with the rest [17, 30].

The pair members' traits (e.g., personality type) as such appear to have little impact on the effectiveness of PP performance [15, 31] and even individual performance does not predict PP performance well [12, 19], so it appears to be important how PP partners actually work together.

2.2 Peeking Into The Pair Process

A few qualitative and qualitative-quantitative studies have looked at the PP process itself. PP consists of a lot of discussions in which both partners verbally contribute to almost all topics [8] with an equal share on all levels of abstraction [9].

Chong & Hurlbutt [11] observed that a more experienced pair member dominates the session while a newly hired partner would ask many questions, time pressure permitting. Plonka et al. [20] identified experts' teaching tactics: *nudging* (making suggestions instead of telling), *preparing the environment* (e.g., opening a useful file), *pointing out problems* instead of telling the solution, *gradually adding information*, or *giving clear instructions*.

Jones & Fleming [18] and our own prior work [34], however, has shown that knowledge transfer in pair programming is not limited to such "expert-novice" constellations. There are episodes of explicit knowledge transfer essentially throughout all PP sessions, even "expert-expert" constellations. Jones & Fleming [18] identified four types of knowledge that pair members transfer in bug-fixing sessions: programming language details, development tools, code structure, and how to reproduce a bug. We identified four modes of individual knowledge transfer episodes [33, 34]: *Pushing* is explaining without prior request (akin to Plonka's teaching tactics), *pulling* is knowledge transfer driven by many questions of the knowledge recipient. *Co-producing* means both developers acquire and consolidate new knowledge together by e.g. code-reading and discussion. *Pioneering production* means one pair member does this alone (e.g., if the partner already knows or does not care). PP only works well as long as the partners frequently resynchronize their ever-changing session-specific knowledge [33, 34].

We are not aware of a characterization of PP sessions as a whole that explains what makes pair programming work (or not) and where the high variance seen in experiments likely comes from.

ID	Length	Pair	Session Content
Company A: Content Management System (Java, Objective-C, SQL)			
AA1	02:22	A1 A2	Fix five similar bugs touching both frontend & backend
Company B: Social Media (PHP, JavaScript, SQL, HTML, CSS)			
BA1	01:47	B1 B2	Read foreign code, implement cache, discuss specification
BB1	01:21	B1 B2	New feature from scratch (template); discuss requirements
BB2	01:51	B1 B2	↳ impl. model, controller, template; discuss requirements
BB3	01:32	B1 B2	↳ implement template, controller; discuss requirements
Company C: Graphical Geo Information System (Java)			
CA1	01:18	C1 C2	Implement new form in GUI (C1 already started)
CA2	01:14	C2 C5	Architecture discussion (C5 already started), refactoring
CA3	02:10	C6 C7	Implement context menu entry, incl. test case & refactoring
CA4	01:34	C4 C7	Implement selection feature w/ special key-binding
CA5	01:23	C3 C4	Implement feature to split graphical elements
Company D: Estate Customer Relationship Management (Java, XML)			
DA2	02:23	D3 D4	Planned feature impl., turned to widespread refactoring
Company E: Logistics and Routing (C++, XML)			
EA1	01:17	E1 E2	Step-by-step debugging of error in route display
Company J: Data Management for Public Radio Broadcast (Java)			
JA1	01:07	J1 J2	Walkthrough of J2's code, discuss possible refactorings
JA2	01:15	J1 J2	Review of J2's new API, define requirements
Company K: Real Estate Platform (Java, SQL, CoffeeScript)			
KA1	02:00	K1 K2	Dev. env. setup, discuss inter-system API design, 1st impl.
KB1	00:53	K2 K3	Add new class to model, write and debug database migration
KC1	00:59	K2 K3	Test env. setup, discuss test approaches for GUI feature
KC2	02:01	K2 K3	↳ trying diff. test approaches, struggling w/ debugger
Company O: Online Project Planning (CoffeeScript)			
OA1	01:24	O3 O4	Understand foreign component, try to read state for testing
OA2	01:32	O3 O4	↳ try to set up (parts of) component for testing
OA5	01:09	O1 O3	Bug fix: amend test cases, refactor prod. code, fix the bug
OA8	01:16	O3 O4	Failing test: Investigate prod. and test code, correct mocks
Company P: Online Car Part Resale (PHP, SQL)			
PA1	00:58	P1 P2	Walkthrough of DB migration (written by P1), discuss req.
PA2	01:30	P1 P2	↳ test of migration, debugging, refactor test cases
PA3	01:31	P1 P3	Implement new API endpoint w/ tests (P3 already started)
PA4	01:42	P1 P3	↳ implement DB access with OR-mapper

Sessions AA1 to KC2 were selected from the *PP-ind* repository [35]; OA1 to PA4 were recorded for this study (see Section 3.1). Some sessions continue an earlier one (↳). Sessions OA1 to OA8 are in English, all others are in German. Developers C4, C6, and O3 are female.

Table 1: Context & characterization of analyzed PP sessions

3 RESEARCH METHOD

3.1 Type and Origin of Data

We analyze sessions of professional software developers working in pairs on their every-day development tasks. Part of our data we draw from the *PP-ind* repository of industrial pair programming session recordings collected by Plonka, Prechelt, Salinger, Schenk, Schmeisky, and Zieris between 2007 and 2016 [35]. This repository contains 52 recordings of 32 pairs from 11 different companies featuring 49 developers working on many different types of tasks. These sessions have a typical length of 45 minutes to 2.5 hours, averaging 1.5h. Developers were not restricted in their choice of task and partner; participation was voluntary and based on informed consent. Each session has a unique identifier, such as CA2 which denotes the third company, first project, second session. Developers are identified by their company and an index, such as C4.

Each recording comprises a desktop screen-capture, a webcam video showing the upper bodies of the two developers, and an audio track with the developers' conversation. Most recordings are complemented with questionnaires with self-reported developer backgrounds and developers' characterization of the session's task.

For this study, we recorded 14 additional sessions in two companies (O and P) in similar manner, featuring 8 developers in 8 pair and "mob" constellations (groups of three or four). We do not discuss mob sessions here. See Table 1 for an overview of the sessions analyzed for this study. Additionally, we conducted 1-on-1 and group interviews with developers, scrum masters, and technical managers from companies O and P in order to gather more background information and to validate our findings (see Section 7.3).

All companies (A to P) develop their own software product in-house, so that the application domain is stable for the developers. This is different for consulting companies whose developers more often encounter a new application domain. For validation, we also conducted individual and group interviews with developers and technical managers in two consulting companies Q and R (but did not record PP sessions).

3.2 Analysis Process

We base our research method on the Grounded Theory Methodology (GTM) in its Straussian form [29]. In particular, we make use of the practices of open coding, axial coding, and selective coding [29, Chapters 5, 7, 8]; we applied theoretical sampling to focus more on the interesting phenomena once we identified them and to reach theoretical saturation [29, Ch. 11].

We selected sessions in the manner of *theoretical sampling* [29, Ch. 11] with different "pair constellations" who work on different types of tasks (e.g., implement new feature from scratch, amend existing feature, bug fixing, module integration, refactoring and other maintenance, testing)—see Table 1 for information on the analyzed sessions. We did not transcribe the sessions, but performed our analysis directly on the video material.

3.2.1 Open Coding. During *open coding* data is "broken down" and conceptualized as "what it is" [29, p. 63]. Existing grounded concepts strengthened our *theoretical sensitivity* [29, Ch. 3]. To help with reconstructing what the pair programmers are doing, we employed the base layer for PP research [21]. It conceptualizes certain *speech acts* [5]: A set of 68 concepts structures the PP process with utterance-level granularity into so-called "base activities" such as making and evaluating proposals [21, Ch. 4 & 6] or asking questions and offering explanations [21, Ch. 16]. We used the base concepts to identify session segments relevant for our general interest in knowledge transfer phenomena; we reuse our notions of *knowledge transfer episodes* and *modes* (*push*, *pull*, etc., see Section 2.2 and [33]) as vocabulary to talk about PP processes throughout this paper. We illustrate this with a concrete example in Section 3.2.2.

An early result of our open coding was our operationalization of *knowledge* and different knowledge types (see Section 4). We came back to open coding and constant comparison whenever new insights from axial and selective coding required amending or adding concepts. In this article, we typeset our new concepts in blue sans-serif font as such: Some Concept; for brevity, we do not report on intermediate concepts.

3.2.2 Analysis Process Example. What follows is an annotated transcript from the beginning of session EA1 (0:04:23–0:14:00) where developer E2 explains to his colleague E1 what he already knows about a display error of route segments on a map. The software is running in debug mode, execution is halted at a breakpoint. E2 switches between source code and GUI to explain the observable failure and related code segments. The transcript is translated from the original German dialog and slightly shortened ("[...]"), with marked <developer actions> and source code `identifiers`. We annotate *base concepts* [21] (flushed right, per utterance) and knowledge transfer *episodes* with their *modes* [33] (semi-graphically in the right margin).

[E2 push]

E2: "I'll show you what I did [...] The error is this [...] you see, the last segment has an extra point." *explain_knowledge*
E1: "Yep." *agree_knowledge*
E2: "The last point should have been this one, but it takes that one. <hovers two points in GUI>" *explain_knowledge*
E1: "Yes." *agree_knowledge*

E2 *explains* and E1 *agrees* for five minutes in this fashion.

E2: "<looking at source code> `result` is the point count [...] which gets increased here." *explain_knowledge*
"<opens inspector> And now it's 131." *explain_finding*
"Before it was 1, I guess that's the start point." *propose_hypothesis*
E1: "M-hm." *agree_hypothesis*
E2: "And then 130 were added." *explain_finding*
"And now they are in this `pPoints`, there <opens inspector> they are." *explain_knowledge*

[E1 pull]

E1: "The polygon points of the route, they are still in there?" *ask_knowledge*
E2: "Yes, exactly. This `TraceFerry()` has an output parameter where the points get copied to." *explain_knowledge*
E1: "M-hm." *agree_knowledge*
E2: "It's called multiple times. It's a big array, first for the stub, and later it says 'copy starting here'." *explain_knowledge*
E1: "M-hm." *agree_knowledge*
E2: "And in `TraceFerry()` they get copied." *explain_knowledge*
E1: "OK." *agree_knowledge*

[E2 push, continued]

E2: "Exactly. And, erm, now in this `pPoints` array, there are the points." *explain_knowledge*
"The last point should be correct now. Or it's not. We'll see about that." *propose_hypothesis*
"[...] <inspects values> Yes, it is the one, it's doubled. So, the error is somewhere in `TraceFerry()`. [...]" *explain_finding*

[E1 pull]

E1: "So you say, it's actually—we interrupt the route—I mean <takes mouse> it goes here, goes here, goes back again, and takes that as the endpoint, or what?" *ask_knowledge*
E2: "Exactly, it got all these points first, then this one, then that one instead of this." *explain_knowledge*
E1: "Yes, and goes back again. OK." *agree_knowledge*

[E2 push, continued]

E2: "[...] I first thought it goes wrong here, but it's not. Instead it's `TraceFerry()`, where it comes out wrong." *explain_knowledge*

With a long *push* and two shorter *pull* episodes it took the pair almost 10 minutes to get the point where E2 already was. After they reached the limits of E2's existing understanding, the pair then continues in *co-produce* mode, debugging the source code together.

Our research interest in knowledge transfer led us to two questions: (1) What is the effect of this behavior? (2) What role does it play in the session overall? We come back to these questions and the above excerpt in Sections 4.2.1 and 5.3.1 to illustrate how we arrived at our concepts.

3.2.3 Axial Coding. In *axial coding* one considers the conceptualized behavior: what phenomena it is directed at, its context, and its consequences [29, Ch. 7]. In our case, the behavior is pair programmers transferring knowledge in *episodes*, and the phenomena are perceived knowledge gaps pertaining to different knowledge types. As the relevant context for all their in-session behavior, we identified the developers' Knowledge Needs resulting from their individual pre-existing knowledge and the specific demands of their task (see Section 4). We did not consider higher levels of the *conditional matrix* [29, Ch. 10] such as team or company.

3.2.4 Selective Coding. *Selective coding* is the integration of concepts to a theory around a central narrative under systematic consideration of context properties [29, Ch. 8]. We identified *system-specific knowledge*, S, as the most important knowledge type, and *generic software development knowledge*, G, as another (see Section 4.1). Our pairs also talked about other types of knowledge, such as their application domain or aspects of the company culture, but these were rare and had little relevance in their sessions. We considered pair constellations based on the developers' knowledge needs regarding the two dimensions S and G which led to six recurring initial constellations (see Section 5). Across all these, we identified three common prototypes of session dynamics (see Section 6).

3.2.5 Validation and Theoretical Saturation. We validated our findings with our subjects in companies O and P, and—as another round of *theoretical sampling* [29, Ch. 11]—with developers and technical managers from consulting companies Q and R for whom knowledge of the application domain presumably works differently (Section 7). *Theoretical saturation* is reached when collecting fresh data no longer sparks insights with regard to new properties of the central concepts [10, p. 113]. Our discussions with companies O, P, Q, and R brought no new facets regarding our concepts to light, so we reached theoretical saturation in this sense.

4 KNOWLEDGE NEEDS

We do not a characterize developers as "experts" or "novices", but consider their concrete situation in a PP session. They work on some task with specific knowledge demands and bring some body of existing knowledge to the table. Their "task" is not necessarily well-defined and can be modified (explicitly or implicitly) as the session proceeds. Depending on the pair's design decisions, for instance, different areas of knowledge become more or less relevant. These decisions, in turn, may depend on what the developers know and do not know.

Considering all that happens in a PP session, each developer has an overall Knowledge Need, her gap in knowledge with regard to the current task, which we operationalize as follows:

(1) Direct Operationalization: Individual knowledge transfer *episodes* have a *topic* [33]. When developer A explains something to developer B about some topic X, and B acknowledges this explanation (as seen several times in Section 3.2.2), this indicates that B had a Knowledge Need regarding X (and A had not). Occasionally, pair programmers also express uncertainty or talk about their respective knowledge levels explicitly [21, Ch. 13–15], thus marking a Knowledge Need.

(2) Indirect Operationalization: Developers may not yet be aware of the extent of their Knowledge Need and thus do not formulate questions to trigger an explanation. But signs like correcting an obvious mistake or being puzzled by new discoveries allow the researcher to see it. Conversely, being able to formulate and evaluate proposals indicate a lower Knowledge Need.

For the sake of simplicity, we also speak of a developer's "knowledge level" to refer to all the things she demonstrates to know in a PP session. After a successful *knowledge transfer episode*, her overall Knowledge Need gets lower and her level gets higher.

Considering not only one developer but the whole pair, we call a knowledge gap either *one-sided* or *two-sided* depending on whether only one partner or both have an according Knowledge Need.

4.1 Types of Knowledge

Nearly all knowledge transferred in our PP sessions relates to solving the session's task. At first, we characterized pair members as either having a high or a low Knowledge Need regarding this "task knowledge", and later distinguished three degrees (details follow).

In our data, there were knowledge transfer episodes pertaining to many different types of knowledge, such as the company culture and structure or the application domain. But the vast majority of the topics across all sessions can be classified into two major types:

S ("specific") knowledge is about understanding the software system at hand: Its requirements, its overall architecture, and gazillions of small facts regarding its detailed design structure, test/build infrastructure and procedures, configuration state, defects, idiosyncrasies, implementation gaps, and so on.

G ("generic") knowledge is about general software development methods and technology: Programming language details, design patterns, development tools, and technology stacks, etc.

S knowledge is mostly narrow and factual. Large numbers of S items are typically transferred in any PP session. In contrast, G knowledge is more widely applicable, but much fewer G items are typically transferred in a PP session.

The following subsections provide (first for S, then for G) a characterization for different degrees of Knowledge Needs.

4.2 S need – Need for System-Specific Knowledge

Considering all in-session activity from a researcher perspective, a pair member has an overall need for S knowledge, the S need. We characterize three degrees of S need:

- **Low S need**: The developer provides explanations about the current state to her partner, she alludes to things not yet seen in the session, and she evaluates findings, explanations, and hypotheses proposed by her partner. She does not ask questions about S knowledge and is rarely puzzled by new discoveries.
- **Mid S need**: The developer has some knowledge about the system in general, but not enough about the particular area relevant for the task. For instance, she may be not up-to-date with recent changes in that area. The developer may acknowledge her lack of knowledge and proposes to "look into things" or formulates hypotheses. Alternatively, if she is not aware of her lack of S knowledge or does not act on it, she might make proposals that are misled and which her partner rejects thus pointing out the S need.
- **High S need**: The developer knows barely anything about the system's relevant parts. She acknowledges her lack of knowledge and asks her partner about the system. She does not refer to system parts or properties until the pair has looked at them. Proposals and hypotheses coming from the partner are not evaluated.

The degree of S need depends on prior involvement with the relevant parts of the system (e.g., authorship), on forgetting details, and many specifics of the current task.

4.2.1 Analysis Process Example (c'd): But how did we find the S need concept? As explained in Section 3.2.1, we use the pair programming literature to increase our theoretical sensitivity. Salinger et al. [22], for example, identified the role of a *task expert*, who provides her partner with task-relevant knowledge. In the example from Section 3.2.2, developer E2 would be the *task expert* who explains all he knows about the bug. However, our first question, *What is the effect of this?*, cannot be answered from this perspective. It occurred to us that what matters here is the *partner*, E1, who gains system understanding: The beginning of the session EA1 is about addressing *his* S need.

4.3 G need – Need for Generic Software Development Knowledge

Again, we distinguish three degrees of G need based on the developer's behavior:

- **Low G need**: The developer is able to explain the meaning of programming language idioms or how to use certain libraries or tools, if need be. She does not ask questions in this regard.
- **Mid G need**: The developer asks informed questions about the used technology or the development approach, and occasionally reads in the documentation.
- **High G need**: The developer asks fundamental questions concerning programming language, standard libraries, or basic tools, and/or uses documentation extensively. She might also express uncertainty and verbalize a lack of ideas on how to proceed.

By these terms, "experts" and "novices" would be developers with low and high G needs, respectively, for the majority of tasks in their job. The same developer will often have different degrees of S need (and can have different degrees of G need) for different tasks.

In practice, a developer's G and S needs are not independent. For an individual developer and a given task, the combination of perfect system understanding (low S need) and no applicable general development knowledge (high G need) is unlikely, since understanding a system without having a grasp of the used technology is difficult. Having a low G need and high S need, on the other hand, is plausible and may lead to quick acquisition of S knowledge.

5 PAIR CONSTELLATIONS AND SESSION DYNAMICS

5.1 Session Context and Goal: Initial and Target Constellation

With respect to a specific development task, each developer has a degree of S and G need, possibly changing over time. A PP situation can thus be characterized by each developer's momentary S and G needs.

For systematically solving a task, the pair needs to address its S need and attain complete understanding of the system's task-relevant aspects. Depending on the goal of the particular session, the pair has one or more options to break this down to the individual level: Meeting the S need is often desirable for both developers, e.g., if they are expected to be able to work on similar tasks alone or with a different partner in the future. In other cases, the pairs are content with only one developer meeting her S need, leaving a one-sided S gap between the partners.

In contrast to S, not all G needs have to be addressed in a session. More complete G knowledge facilitates important steps such as addressing an S need, designing a good solution, implementing and debugging that solution smoothly. Filling a G gap may be part of the session goal, e.g., for training purposes.

From a researcher's perspective, pair programmers begin a session with an **initial constellation** of each partner possessing some S and G knowledge, plus a more or less clear idea of the session task, i.e., what they want to achieve with their session. The task might be for example fixing a problem (for which S needs have to be met) or educating the partner (addressing an S and/or G need in some respect). The intended outcome of a session in terms of S and G needs to be met denote the session's **target constellation**.

5.2 Constellation Changes

Overall, knowledge gaps tend to shrink during a session. It is important for a pair to become aware of knowledge gaps, but in our terminology, their G and S need are not affected by such insights alone. In principle, pairs can take two approaches to deal with knowledge gaps they are aware of: (1) Limiting the scope of the current task, thereby making some of their S and/or G needs obsolete or (2) transferring existing or acquiring new knowledge to address their respective S or G needs.

As for (1): In our data, a pair's *initial* session scope discussion is often not recorded. But our developers also sometimes decide *during* the session that some subtask is not mandatory and stop pursuing it (e.g., in the beginning of session BB1, Section 6.1.1), or they may shift their focus mid-session, thus effectively changing what knowledge is relevant (e.g., in DA2 which should have been a feature implementation but pivoted to a large refactoring).

As for (2): The remainder of this paper is concerned with this approach only, that is, with *knowledge transfer*.

5.3 Overall Session Dynamics

In all analyzed sessions, the pairs first deal with a one-sided S gap if one exists, then with any two-sided S gap, both limited by their awareness of these gaps. We therefore call these the **primary gap** and the **secondary gap**. The target constellation acts as a moderator for both steps: In case not both developers need to reach high S knowledge, parts or all of the primary and secondary gap may remain unfilled. Once the S needs are met to the intended degree (and only then), the pair transfers (or does not) G knowledge if one partner has a G need: the **G opportunity**.

Different orders appear to be the exception: (1) If neither partner possesses required G knowledge (two-sided G gap), the pair will have to acquire it together and this can happen when a secondary gap is still open. (2) If two-sided S *and* G gaps are large enough, the pair may become overwhelmed by difficulty. The session then breaks down and no or nearly no progress happens.

5.3.1 Analysis process example (c'd): How did we find the concepts of primary and secondary gap? Remember Section 3.2.2, where we discussed E2's bug-related explanations to E1 and our second question: *What role does it play in the session overall?*

Considering the session EA1 as a whole, the long-running *push* episode in the beginning enabled the pair to work as peers to then solve the actual task together. In our data, this is common behavior in the beginning of a session, especially when one developer already worked on the task before. It occurred to us that these pairs start with an *asymmetric* situation regarding their S needs and they turn it into a *symmetric* one which allows them to make further progress closely together (*co-produce* episodes in the case of EA1). So achieving S need symmetry comes first (closing the primary gap), acquiring the remaining S knowledge together follows (closing the secondary gap).

5.4 Session Visualizations

Below, we will provide schematic representations of pairs' knowledge constellations and their trajectory throughout several example sessions (Fig. 2). Each developer is represented by a point on a two-dimensional coordinate system, with the degree of G need decreasing from left to right and the degree of S need from bottom to top. In a *qualitative* sense, the vertical distance between a pair's two points hence represents the primary gap, the distance from the top represents the secondary gap, and the partner's horizontal distance represents the G opportunity.

The pair's points are drawn at their initial constellation. The reduction of knowledge gaps is indicated by arrows originating at the developer whose understanding improves: upward for increasing S knowledge, to the right for improved G knowledge. Multiple arrows starting at the same height indicate multiple attempts made to address a Knowledge Need. The trajectories do not depict technical progress at all. Arrow length does not represent time at all, but only the (qualitative!) reduction of a knowledge gap. Arrow color indicates the *mode* in which the knowledge gap is narrowed (see Section 2.2 and [33] for details). The numbers in the trajectories correspond to numbers in the article text. For readability, a single arrow (e.g., ↑) might represent multiple knowledge transfer episodes pertaining to similar topics.

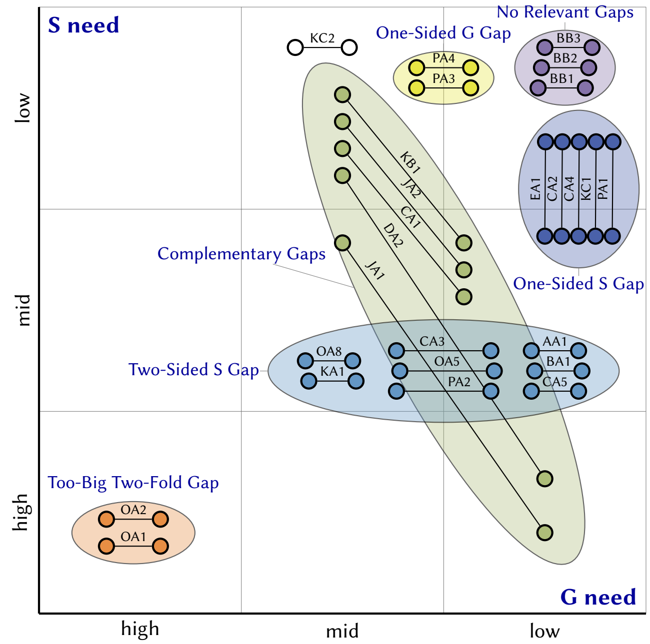
6 SESSION DYNAMICS PROTOTYPES

We will now describe how pairs actually deal with their Knowledge Needs: We have identified six different initial constellations in our data, each having a different combination of primary and secondary gaps and G opportunity, each leading to a characteristic session dynamic. These form a set of session dynamics *prototypes* that is very useful to understand how PP works and when (and for which goals) it is most useful. See Fig. 1 for the initial constellations of all analyzed sessions. Other initial constellations are conceivable, but we have not seen them.

6.1 No Knowledge Gaps, No Opportunity

In many pair programming sessions, there is a point at which both partners have all necessary system understanding as well as all needed general programming knowledge to work productively on the task. This is the No Relevant Gaps constellation (see Fig. 1). The only pair we have seen that had this as its initial constellation forms our first and simplest example.

6.1.1 Example 1: Greenfield Development. Developers B1 and B2 work on a new feature from scratch over the course of one afternoon in three sessions (BB1 to BB3) with short pauses in between. Both are proficient in the involved technologies and need to interact with existing code only through few and well-understood interfaces (neither S nor G need). The newly developed code stays small enough to be fully understood by both developers at all times. Apart from a short orientation phase in the beginning, when they decide on where to visually place the feature in the GUI, they are in construction-only mode throughout the session, i.e., defining requirements and discussing design proposals, with zero debugging. B1 and B2 develop no S need through all three sessions while the body of S knowledge increases along the way.



S and G need denote a developer's gap in task-relevant knowledge with regard to the *specific software system* and *generic software development*, respectively (see Section 4.1). Each pair of points represents one PP session (see Table 1); sessions are grouped in six recurring pair constellations.

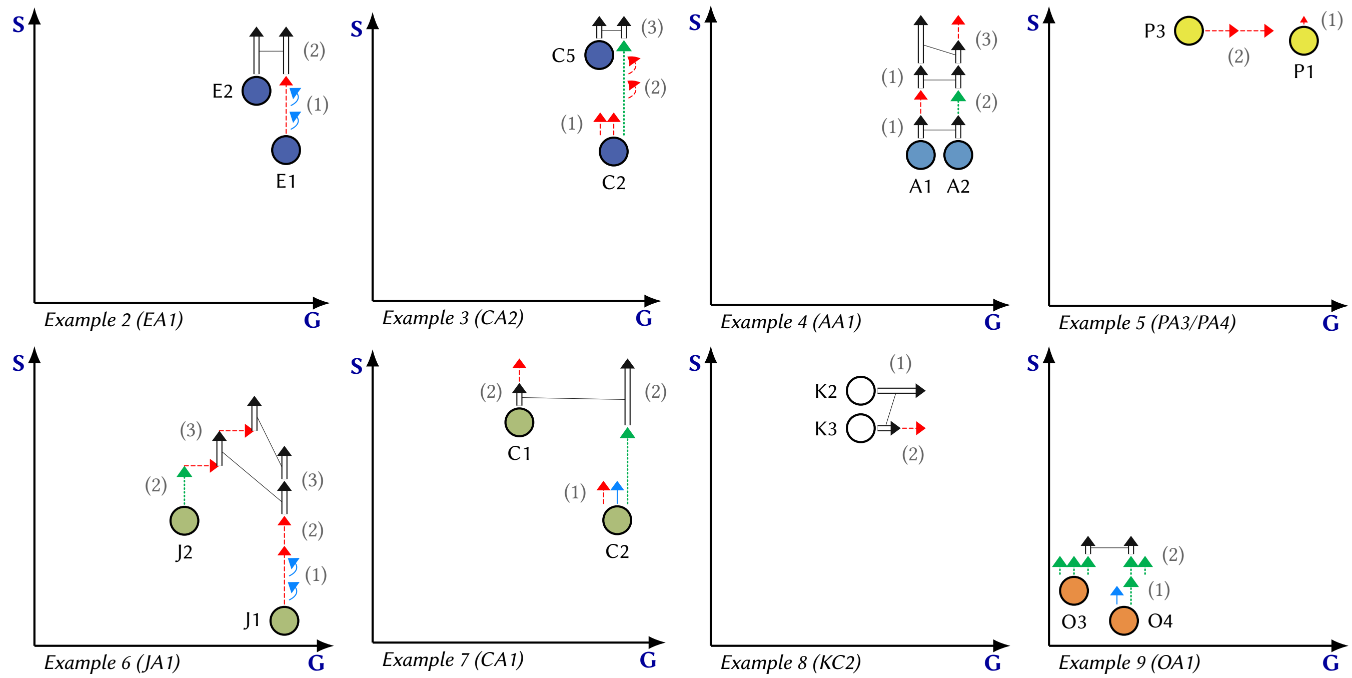
Figure 1: Initial pair constellations of the analyzed sessions

6.1.2 Discussion. This nice and easy initial constellation is likely only in development-from-scratch situations, which are not frequent. It is fragile and can be destroyed by any lengthy debugging episode or by between-session pauses long enough to let developers forget relevant details of what they built.

6.2 Dealing with a One-Sided S Gap

In some pair programming sessions, one developer has an S advantage, e.g., because she already started work on the task. Two constellations have this property: One-Sided S Gap and Complementary Gaps, each of which happened to be the initial constellation of five our analyzed sessions (see Fig. 1). Whenever a one-sided S gap—the primary gap—exists, the pair addresses it first.

In most cases, the developer with the larger S need is aware of the gap and the pair can address it proactively, as illustrated in Section 6.2.1. If, however, she is not aware of her S need, she



S and G denote a developer's task-relevant knowledge with regard to the *specific software system* and *generic software development*, respectively. Arrows indicate mode of knowledge transfer [33]: (solid line, blue): *Pull*; developer asks about topic of interest. (dashed line, red): *Push*; developer receives not-explicitly-requested explanations. (dotted line, green): *Pioneering*; developer acquires new knowledge through reading source code/inspecting artifacts. (double line, black): *Co-production*; both developers acquire new knowledge in a tightly coupled fashion (corresponding arrows are connected with a thin line). Numbers refer to descriptions in main text.

Figure 2: Trajectories of discussed examples (Example 1 (BB1 to BB3) has No Relevant Gaps)

might make poor or non-applicable proposals which need to be identified as such. This can take some time and be frustrating for the developers, see Section 6.2.2.

6.2.1 Example 2: Bringing Partner Into Ongoing Work. In session EA1, developers E1 and E2 want to fix a bug which E2 has already worked on before (more S knowledge than E1). So E2 steps through the code with a debugger, demonstrates the failure in the running application, and comments on the state of individual variables (*pushes*). His partner E1 asks for details (*pulls*). They quickly close their primary gap (phase 1) and continue debugging together (2). We already discussed phase (1) in the *Analysis Process Example* in Section 3.2.2; see also Fig. 2 for the numbers.

6.2.2 Example 3: Closing The Primary Gap Painfully. In session CA2, C2 and C5 want to implement a new feature for just one edition of their software. C5 has already started the implementation and is familiar with the system modularization (low S need). C2 does not know C5's recent changes; additionally, some aspects of the system's architecture have slipped his mind (mid S need). It takes the pair frustrating 11 minutes and multiple attempts to close their primary gap (refer to numbers in Fig. 2):

(1) C5 tries to explain his recent changes and alludes to the underlying architecture that motivated them. C2 does not engage in these *push* episodes: he does not listen to C5 at all and keeps hushing him.
(2) Instead, C2 starts reading the source code (*pioneering*), which leaves him puzzled several times, because he is not aware of the underlying rationale. C5 tries to follow C2's mostly silent reading process and intersperses architectural explanations (S *pushes*). C2, however, appears to misinterpret these as a discussion of general design principles, which would be G *pushes*, and ignores them. This continues until C2 eventually recognizes the underlying system structure and finally understands C5's changes from before the session.

With their primary gap closed, the pair continues in a Two-Sided S Gap constellation and works on their secondary gap (3).

6.2.3 Discussion. The most common way how pairs address their primary gap appears to be that the partner with more S knowledge starts a *push*, into which the partner hooks in *pulling* for details. In many sessions (e.g., CA4, EA1, JA1, and PA1) this is enough to close the primary gap.

If the pair member with more S knowledge does not provide good explanations in *push* mode, her partner with the S need may take the lead with a more interview-style *pull*-driven mode, e.g. in sessions CA1 and DA2.

If this is not enough to close the primary gap either, the partner with the S need may switch to reading up the necessary information herself (*pioneering*). In session DA2, such a switch was necessary when a partner could not explain well because of his lack of relevant G knowledge. Developer C2 above, however, appears to *generally* prefer to *pioneer* for closing a primary gap, even though his partner is willing and able to provide suitable information. Closing a primary gap was the only setting where we observed issues due to (presumed) personal preferences.

6.3 Dealing with a Two-Sided S Gap

Pairs that have no member with all relevant S knowledge need to find ways to acquire it to close their joint S gap. Such a secondary gap appears to be common: eight of our analyzed sessions had a Two-Sided S Gap as their initial constellation (see Fig. 1), many others reached this constellation after closing their primary gap.

6.3.1 Example 4: Pairing-Up Throughout. In session AA1, developers A1 and A2 want to fix five similar bugs and need to work with two different sub-systems, neither of which is fully understood by either partner. Since they both want to meet their S need, they keep their understanding in sync along the way. Their session illustrates a number of ways how pairs can deal with their secondary gap (see Fig. 2 for the numbers):

(1) *Co-produce*: Most of the time, A1 and A2 address their secondary gap collectively in *co-produce* episodes where they formulate hypotheses about the system, read source code, try out the application, and integrate their insights.
(2) + *Pioneer* plus *push*: The developers disagree on the relevance of some S topics. A2 occasionally pursues a *pioneering* episode and afterwards explains what he learned (*push*). A1 hardly opposes A2's initiatives, as some of them lead to task-relevant insights which the pair probably would have missed otherwise.
(3) + *Co-produce* plus *push*: During later *co-produce* episodes, sometimes one developer is faster at understanding something than the other (A1 is more proficient in one sub-system, A2 in the other).[1] In such cases, the faster developer would *push* explanations for his partner to catch up.

6.3.2 Discussion. For closing a secondary gap, *co-producing* is common behavior in many sessions (e.g., CA1, CA2, and JA1).

If one pair member understands faster, e.g., due to a G advantage as in session CA1 (discussed below) or a local S advantage in some area as in session AA1, a + *co-produce* plus *push* makes sure no partner falls behind.

If the developers have different goals or preferences, not all topics need to be understood fully by both and the more invested pair member may + *pioneer* plus *push*. Occasionally, this also happened in other sessions, such as JA1 (discussed below).

6.4 Opportunity: Reducing a One-Sided G Gap

A difference in G knowledge between the partners can be an opportunity to transfer valuable general software development knowledge. In our sessions, pairs never seized their G opportunity before any known primary gap and secondary gap were closed. Some pairs started from a One-Sided G Gap (e.g. in sessions PA3 and PA4), others from a Complementary Gaps (e.g. JA1 or DA2) or Two-Sided S Gap (e.g. OA5).

[1] One could split the S dimension for characterizing pair constellations that are complementary with regard to different parts of S, potentially leading to as many S dimensions as there are topics. To keep the discussion tractable, we do not do this.

6.4.1 Example 5: Initially-Misunderstood Teaching. In sessions PA3 and PA4 (see also Fig. 2), frontend developer P3 and backend developer P1 work in the backend of their system. Both know the relevant parts of the system well (no secondary gap), but P3 already started implementing a new API endpoint (small primary gap). Since they are on P1's technological home turf (no G need), he understands P3's explanations quickly and the primary gap is soon closed (1). From thereon, P1 explains the newest PHP language features and how to employ test-driven design whenever he sees an opportunity (G *pushes*, (2)).

In session PA3, P3 misinterprets these explanations as a lead-in for unnecessary S *pushes* and gets confused, but after talking to P1 about P1's intentions in a break he then acknowledges them as valuable lessons in session PA4.

6.4.2 Example 6: Embracing a Difference. Session JA1 is about improving the maintainability of a module that J2 wrote a year earlier and J1 never saw before. The module is basically a state automaton implemented with deeply nested if-statements. In addition to a seized G opportunity, this session also illustrates how one developer (J1) does neither need nor want to fully met his S need as he will only ever work on that module together with J2. Again, refer to Fig. 2 for the numbers:

(1) The pair deals with its primary gap via a long running *push* with hooked-in *pulls*: J2 explains and J1 asks for details.
(2) To address their secondary gap, J2 repeatedly reads through the complex low-level control structure and then explains the high-level states and transitions of the automaton (*pioneer* plus *push*). Both partners take care to keep the *pushes* from going into too much detail.
(3) After J1 got the big picture (only a mid S need left), the pair starts reading source code together (reducing the secondary gap further through *co-production*). In doing so, J2 looks for code smells to explain possible refactorings (G *pushes*) thus using the G opportunity.

6.4.3 Example 7: Missing the Opportunities. In session CA1, the pair wants to implement a new GUI feature similar to an existing feature. C1 already worked on it for an hour when C2 joins him. This gives C1 a modest S advantage, which needs to be addressed. C2 is more proficient with the object-oriented paradigm (a G advantage). They deal with their primary and secondary gap, but do not use their G opportunity (see corresponding numbers in Fig. 2):

(1) To close C2's S gap, C1 first tries to explain what he did (*push*). This is not effective and C2 starts to ask specific questions about existing classes (*pull*), which C1 begins to answer. But C2 quickly gives up on this in favor of trying out the new GUI elements and reading in the new code himself (*pioneering*), which eventually achieves the desired understanding.
(2) Later in the session, the pair is able to close newly detected two-sided S gaps in *co-production* episodes. In these cases, C2 is always the first to understand (presumably due to his better G knowledge) and often explains his findings to C1 (*push*).

During the session, there are multiple occasions at which C2 *could* have explained some G knowledge to C1 (i.e.,), e.g., how he got to his insights so swiftly or how some of C1's proposals violate good design. But he does not and merely alludes to the underlying knowledge.

6.4.4 Discussion. While session PA3 (see Section 6.4.1) was at times frustrating for P3 who only received explanations, but had no opportunity to provide some, Complementary Gaps constellations such as in JA1 (see Section 6.4.2) or DA2 and KB1 (see Fig. 1) were mutually satisfying sessions: One developer needs to understand the system (S need), and her colleague may help with this. Yet, the developer with the S need can use her advantage to teach G knowledge. Session CA1 shows, however, that not all pairs in such a constellation seize the G opportunity (see Section 6.4.3). A constructive pattern for the partner with the higher G need might be to → *pull* for G knowledge whenever the partner does something "magic" without ⇢ *pushing*.

6.5 Two-Sided G Gaps?

There are sometimes PP sessions where both pair members lack G knowledge needed for their task—a non-routine situation. We have seen one instance of a pair attempting to acquire it (Session KC2, Section 6.5.1). If both pair members have a high S need *and* a high G need, the pair lacks the technical background to build up the required S knowledge and faces a Too-Big Two-Fold Gap. We have seen this constellation twice with the same pair (Sessions OA1 and OA2, Section 6.5.2).

6.5.1 Example 8: It's not easy! In session KC2 (see Fig. 2), developers K2 and K3 want to write a test case for an auto-completion feature K2 implemented earlier. They already addressed their primary gap in session KC1 before lunch and now want to programmatically simulate keystrokes. Both have a mid G need: they know their tools and where to look for help, but cannot implement a test case right away. They attempt to read documentation together (⇒ G *co-production* (1)), which helps K2 somewhat, but not K3. Forty minutes later(!), they notice this one-sided G gap and close it (⇢ G *push* (2)). But they never complete their G knowledge acquisition and after two hours they give up. The next day, K3 said he found a simple solution alone.

6.5.2 Example 9: Disaster. In session OA1, the developers were tasked to write test cases for some new functionality they did not implement and which is built with a technology they are not familiar with. O3 and O4 have both a high S need and a high G need. O3 has a slight S advantage, as she already opened and skimmed the relevant source code. O4 has a slight G advantage since he knows a bit more about the programming language. (See numbers in Fig. 2.)

(1) To close the small primary gap, O4 has to ↑ *pioneer* since O3 does neither ↑ *push* nor react to O4's ↑ *pull* attempt.
(2) For most of the session, they address their secondary gap and try to acquire S knowledge by individually reading in the source code (↑ *pioneering*). At some point, and for lack of better ideas (high G need), they together resort to "printf" debugging (⇑ *co-production*), but do not gain much S knowledge in this way either.

The developers express confusion on fundamental issues (e.g., O3: *"Type? Function? I don't even know what this is."*), but never attempt to address their G need. The same pattern continues in session OA2 on the same day after lunch. The pair eventually decided to not continue with this task.

6.5.3 Discussion. A two-sided G gap is too rare in our data to make much of it, but it appears to be difficult to resolve. In session KC2, the pair had no S gap to deal with and presumably were simply too tired to put their newly gained G knowledge to proper use.

Our interpretation of session OA1 and OA2 is that the situation was so difficult overall that the pair failed to manage the combined complexity of task solving plus coordinating the PP process. We do not expect Too-Big Two-Fold Gap to be common as developers likely anticipate and avoid such a situation. In the OA1/OA2 case, for example, the pair only tackled this task because they were the only team members available and the task had high priority.

7 VALIDATION AND APPLICATION IN PRACTICE

Our analysis yielded two results: (1) A characterization of two types of task-relevant knowledge, generic software development knowledge, G, and system-specific knowledge, S; and (2) descriptions of six different PP session types (based on the degrees of the developers' G and S needs) for which we claim (based on our observations of the overall session dynamic, see Section 5.3) that some types make PP particularly favorable and that this fact can be useful in practice.

We will now discuss some limitations inherent in our analysis approach (Section 7.1), support the G/S concepts by relating them to similar concepts in existing literature (Section 7.2), and validate the session dynamics results and our usefulness claim by describing practitioner reactions to them and experience with practical industrial application of the ideas (Section 7.3).

7.1 Limitations

(1) We only get to see knowledge that can be verbalized (as opposed to tacit knowledge) and so mostly deal with declarative knowledge (as opposed to procedural knowledge) [2, p. 78]. In software engineering, the procedural knowledge relies on a thick foundation of declarative knowledge, so this is hardly a problem at all.

(2) We only get to see knowledge-in-transfer that is actually verbalized, but not the larger body of knowledge-in-use. The difference is smaller than it sounds because in practice there is almost always *some* difference in understanding between pair members that leads to a verbal exchange in the context of the session.

(3) One-sided knowledge gaps are easier to see than two-sided ones, but we usually detect the latter as well because they become visible as uncertainty on how to proceed.

(4) Overall, we found no indication that either of (1)–(3) is problematic but they make it impossible to be sure that developers *always* address primary and secondary gaps when they exist. But if they address them, they do it in that order.

(5) Furthermore, all our developers and work context were western. We have not seen all kinds of software engineering settings (e.g., no consulting contexts). However, our data comes from nine different application domains.

7.2 G and S in Existing Literature

The likelihood that the concepts G and S are contrived is lower if other people report on similar concepts, so we integrate them back into existing literature [28].

Jones & Fleming [18] also investigate knowledge transfer in PP. They identified *general development knowledge* (pertaining to tools and programming language) and *project-specific knowledge* (about code structure and bug reproduction) as different knowledge types which get transferred. In our terminology, which we developed independently, these would be parts of G and S knowledge, respectively. Furthermore, Jones & Fleming are only concerned with explicit "teaching", i.e., only *pushing*, but no *pulling*, *co-producing*, or *pioneering* of such knowledge.

Many studies postulate the importance of S knowledge. For example, the whole subfield of program comprehension revolves around acquiring S knowledge [1]. Sillito et al. [26] analyze the types of questions developers ask about their code base and characterize how well current tools support extracting information from it. In contrast, Fritz et al. [13] acknowledge the value of another *developer* as an information source and provide a tool that identifies S-knowledgeable colleagues in a task-specific manner.

Salinger et al. [22] identify the *task expert* role in some PP sessions, who may provide the S knowledge to close the primary gap.

7.3 Validation with Practitioners

Glaser & Strauss [14, pp. 239f & 245] argue (and Strauss & Corbin agree [29, p. 23]) that a grounded theory meant for practical application needs to be *understandable* to people working in the respective area and should allow the user some *control* over daily situations. We first validated our concepts by presenting them to practitioners from four companies, two of which are consulting firms (Q and R) we approached because our analysis was based on companies with in-house development only. Then, we also validate three ideas of how to put our findings to use in everyday development practice. We mark each validation result as positive (■), negative (□), or half-positive (⬓).

7.3.1 Validation of Concepts. We discussed the two dimensions of developers' task-specific G and S knowledge, five initial pair constellations,[2] and particular session trajectories in interviews with groups of various sizes: two Scrum Masters (SM) and Product Owner (PO) in company O (after recording the OA* sessions); six developers (D), SM, and PO in company P (before recording the PA* sessions); ten D, SM, and two technical managers (TM) in company R; and a 1-on-1 interview with a TM in company Q.

(1) The two dimensions were understood in all discussions. The O-SMs immediately started to characterize their developers as to their typical G and S levels; O-PO used the dimensions to characterize recent difficulties across all teams as a "collective G gap". To Q-TM, the classic "expert vs. novice" is *"too simplistic, too naive, offensive even"*, whereas S and G *"resonate better"*, they *"get better to the heart of the matter"*; he found thinking about "resolving an S-gap or a G-gap?" more compelling than "am I a novice?". ■

(2) The five constellations were quickly understood. O-SM and O-PO independently identified Complementary Gaps as interesting, as the *"most real and valuable"* pairing; O-PO, a P-D, and Q-TM recognized it from recent experience of working together with different roles (such as system administrator) or as a common consulting theme. O-PO found it useful to have names for the constellations. Q-TM had recent experience with four constellations but not with Too-Big Two-Fold Gap. Nevertheless, after just seeing the Too-Big Two-Fold Gap picture, he immediately comprehended the issue: *"Tricky, isn't it? The developers do not get very far."* ■

(3) In consulting companies Q and R, we wanted to assess the importance of application domain knowledge (a possible third type "D"), so we asked for types of relevant knowledge before presenting the G/S dimensions. We learned that D knowledge is seen to have little impact on PP session dynamics: Several R-Ds explained that there are only small D knowledge differences within their team, but sometimes gaps which only the (non-technical) client can fill. The far more serious issue is their overall lack of S knowledge, as due to the legacy system, a developer with a mid S need is already considered a rare "expert". They pair program to carefully build up and maintain S knowledge. Q-TM ranked the problems imposed by G knowledge lowest because G knowledge can be hired if needed or be built along the way through pair rotation. ■

(4) While the dimensions G/S were understood, the task-specificity of a developer's Knowledge Need was not. Instead, both developers and managers tended to think of programmers as having a relatively fixed, experience-based *level* with little changes over the course of a session. Unfortunately, without the task-specific understanding of S and G needs, the pair constellations can no longer be recognized as a valuable tool for forming effective pairs. □

7.3.2 Validation of Practical Ideas. With these four companies, we also discussed three practical ideas for putting our findings to use in everyday software development, all of which would involve a "G-S chart": (5) Consider *task-specific* knowledge when forming pairs [6, p. 59], (6) set goals and orient during a PP session, and (7) reflect on a PP session after the fact.

(5) *Forming Pairs:* We firmly expect our findings to be useful for forming effective pairs, but due to point (4), although many respondents *tended* to like the idea, none of them reacted enthusiastically in this respect. Some were mostly positive, but raised *practical* concerns: P-PO said their teams are too small to regularly offer more than one pairing to chose from. O-PO believed all their pairings would have the same constellation. ⬓

(6) *Setting Session Goals:* Before starting session PA4, we asked developers P1 and P3 to discuss and draw onto a blank G-S chart their initial and their target constellation. They quickly agreed on a One-Sided G Gap with no primary and secondary gap and the goal to address P3's G need regarding an OR mapper. After the session, both explained that filling out the chart did not affect their session, but *could* have, had there been discrepancies in their respective Knowledge need assessments to be resolved. ⬓

(7) *Reflecting on a Session:* We individually asked the developers after sessions PA1/PA2 (same pair, same day) and PA3 (different pair) to trace out their trajectory and to discuss the results without us intervening. In both cases, the pairs had actually seized their G opportunity during the session but only remembered the primary and secondary gaps. The G transfer resurfaced during the reflection, P3: *"Right, I totally forgot about that. That was really cool."* ■

[2] We had not yet seen One-Sided G Gap.

Concept	Description
S knowledge/G knowledge	*System-specific knowledge* (requirements, overall architecture, detailed design structure, test/build infrastructure, defects, idiosyncrasies, implementation gaps, etc.) and *Generic software development* knowledge (programming language details, design patterns, development tools, and technology stacks, etc.); see Section 4.1.
Knowledge Need	Degree of individual developer's knowledge gap resulting from her existing knowledge and the specific demands of the task. May pertain to either knowledge type (S and G need, see Sections 4.2 and 4.3).
One-Sided/Two-Sided Gap	Characterization of a pair: Either only one pair member or both partners have a Knowledge Need in some regard.
Initial/Target Constellation	Constellation of the developers, each with her individual S and G needs, which they assume at the *beginning* of their session and which they want to *achieve* with their session, respectively. We identified six recurring constellations (see Section 5.1 and Fig. 1):
No Relevant Gaps	Neither partner has an S or G need; rarely the initial constellation, the target constellation in most cases (Section 6.1).
One-Sided S Gap	One has an unmet S need; e.g. when joining a partner who already started, sometimes also target constellation (Sec. 6.2).
Two-Sided S Gap	Both lack system understanding; initial constellation e.g. for debugging tasks, not a target constellation (Section 6.3).
One-Sided G Gap	One partner has an unmet G need; opportunity to transfer G knowledge in absence of S need (Section 6.4).
Complementary Gaps	One partner has an S advantage, the other for G; satisfactory session possible due to mutual learning (Section 6.4).
Too-Large Two-Fold Gaps	Both have high S and G needs; difficult and undesirable constellation (Section 6.5).
Overall Session Dynamics:	Three prototypes of session dynamics describing pairs' trajectories from all initial constellation changes (Section 5.3).
1. Closing the Primary Gap	Narrow a one-sided S gap between the developers: either through pro-active explanations, an interview mode, or through solitary reading of the less knowledgeable partner (Section 6.2.3).
2. Closing the Secondary Gap	Narrow a two-sided S gap of both developers by building understanding together and staying in sync (Section 6.3.2).
3. Seizing the G Opportunity	Narrow a one-sided G gap between the developers after any S need has been addressed sufficiently (Section 6.4.4).

Table 2: Overview of our grounded concepts

8 CONCLUSIONS AND FURTHER WORK

Our qualitative analysis of 26 real-world industrial pair programming sessions from nine companies led to the following results:

(1) The expert/novice discrimination used in much existing PP research is not useful. One must consider task-specific knowledge and discriminate S and G knowledge (Section 4.1).
(2) The overall knowledge transfer dynamics of all PP sessions follows a single pattern: Synchronize S knowledge, acquire needed S knowledge together, possibly transfer G knowledge (Section 5.3).
(3) By far most knowledge transfer activity in a PP session concerns S knowledge. One-sided S gaps can require multiple attempts to resolve (Section 6.2). In contrast, resolving one-sided G gaps is usually optional (Section 6.4).
(4) Resolving two-sided S gaps are a bread-and-butter activity in PP and pairs do it routinely (Section 6.3), whereas two-sided G gaps appear to pose difficulties (Section 6.5).
(5) In summary, there is plenty of evidence that differences in S knowledge differences are more important those in G knowledge: S needs are addressed first, more knowledge transfer is concerned with S knowledge, resolving S need is more often mandatory, and S knowledge transfer is the type that pairs are more skilled with. General programming experience differences are overrated.

Our initial attempts at putting these results to practical use have shown that further work will need to create a better didactic concept to communicate the task-specificity of S and G needs. Only once this is solved, we will be able to demonstrate the insights' usefulness by "selling" PP to teams not currently using it at all. Presumably, those teams do not recognize which are the pair constellations or tasks when PP is going to be most useful.

ACKNOWLEDGEMENTS

We thank all participants in our study for talking to us, and allowing us to record and scrutinize their pair programming sessions. We also thank our reviewers for their constructive feedback to improve our manuscript.

REFERENCES

[1] 2019. *ICPC '19: Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada). IEEE Press.
[2] John R. Anderson. 1976. *Language, Memory, and Thought.* Psychology Press.
[3] Erik Arisholm, Hans Gallis, Tore Dybå, and Dag I.K. Sjøberg. 2007. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering* 33, 2 (2007), 65–86. https://doi.org/10.1109/TSE.2007.17
[4] Phillip G. Armour. 2000. The five orders of ignorance. *Commun. ACM* 43, 10 (2000), 17–20. https://doi.org/10.1145/352183.352194
[5] J. L. Austin. 1962. *How To Do Things With Words.* Clarendon Press.
[6] Kent Beck. 1999. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.
[7] Andrew Begel and Nachiappan Nagappan. 2008. Pair Programming: What's in It for Me?. In *Proc. 2nd Int'l. Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. ACM, New York, NY, USA, 120–128. https://doi.org/10.1145/1414004.1414026
[8] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. 2006. The Collaborative Nature of Pair Programming. In *Extreme Programming and Agile Processes in Software Engineering*, Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi (Eds.). Lecture Notes in Computer Science, Vol. 4044. Springer, 53–64. https://doi.org/10.1007/11774129_6
[9] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. 2008. Pair Programming and the Mysterious Role of the Navigator. *International Journal of Human-Computer Studies* 66, 7 (2008), 519–529. https://doi.org/10.1016/j.ijhcs.2007.03.005
[10] Kathy Charmaz. 2006. *Constructing grounded theory: A practical guide through qualitative analysis.* SAGE Publications.
[11] Jan Chong and Tom Hurlbutt. 2007. The Social Dynamics of Pair Programming. In *Proc. 29th Int'l. Conf. on Software Engineering (ICSE '07).* IEEE Computer Society, Washington, DC, USA, 354–363. https://doi.org/10.1109/ICSE.2007.87
[12] Madeline Ann Domino, Rosann Webb Collins, Alan R. Hevner, and Cynthia F. Cohen. 2003. Conflict in Collaborative Software Development. In *Proc. 2003 SIGMIS Conf. on Computer Personnel Research.* ACM, 44–51. https://doi.org/10.1145/761849.761856

[13] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. 2010. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proc. 32nd Int'l. Conf. on Software Engineering* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 385–394. https://doi.org/10.1145/1806799.1806856

[14] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* AdlineTransaction.

[15] Jo E. Hannay, Erik Arisholm, Harald Engvik, and Dag I.K. Sjøberg. 2010. Effects of Personality on Pair Programming. *IEEE Transactions on Software Engineering* 36, 1 (2010), 61–80. https://doi.org/10.1109/TSE.2009.41

[16] Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology* 51, 7 (2009), 1110–1122. https://doi.org/10.1016/j.infsof.2009.02.001

[17] Hanna Hulkko and Pekka Abrahamsson. 2005. A multiple case study on the impact of pair programming on product quality. In *Proc. of the 27th Int'l Conf. on Software Engineering* (St. Louis, MO, USA). ACM, New York, NY, USA, 495–504. https://doi.org/10.1145/1062455.1062545

[18] Danielle L. Jones and Scott D. Fleming. 2013. What Use Is a Backseat Driver? A Qualitative Investigation of Pair Programming. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 103–110. https://doi.org/10.1109/vlhcc.2013.6645252

[19] Matthias M. Müller and Frank Padberg. 2004. An Empirical Study about the Feelgood Factor in Pair Programming. In *Proc. 10th IEEE Int'l. Software Metrics Symposium (METRICS)*. 151–158. https://doi.org/10.1109/METRIC.2004.1357899

[20] Laura Plonka, Helen Sharp, Janet van der Linden, and Yvonne Dittrich. 2015. Knowledge transfer in pair programming: An in-depth analysis. *Int'l. J. of Human-Computer Studies* 73 (2015), 66–78. https://doi.org/10.1016/j.ijhcs.2014.09.001

[21] Stephan Salinger and Lutz Prechelt. 2013. *Understanding Pair Programming: The Base Layer.* BoD, Norderstedt, Germany.

[22] Stephan Salinger, Franz Zieris, and Lutz Prechelt. 2013. Liberating Pair Programming Research from the Oppressive Driver/Observer Regime. In *Proc. 2013 Int'l. Conf. on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1201–1204. https://doi.org/10.1109/ICSE.2013.6606678

[23] Norsaremah Salleh, Emilia Mendes, and John Grundy. 2011. Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 37, 4 (2011), 509–525. https://doi.org/10.1109/tse.2010.59

[24] Christian Schindler. 2008. Agile Software Development Methods and Practices in Austrian IT-Industry: Results of an Empirical Study. In *Proc. Int'l. Conf. on Computational Intelligence for Modelling, Control and Automation (CIMCA), Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC), Innovation in Software Engineering (ISE)*. 321–326. https://doi.org/10.1109/CIMCA.2008.100

[25] Todd Sedano, Paul Ralph, and Cécile Péraire. 2016. Sustainable Software Development through Overlapping Pair Rotation. In *Proc. 10th ACM/IEEE Int'l. Symposium on Empirical Software Engineering and Measurement*. ACM Press, 19:1–19:10. https://doi.org/10.1145/2961111.2962590

[26] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451. https://doi.org/10.1109/tse.2008.26

[27] Susan Elliott Sim and Richard C. Holt. 1998. The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. In *Proc. 20th Int'l. Conf. on Software Engineering* (Kyoto, Japan) *(ICSE '98)*. IEEE Computer Society, Washington, DC, USA, 361–370. https://doi.org/10.1109/ICSE.1998.671389

[28] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research. In *Proc. 38th Int'l. Conf. on Software Engineering (ICSE)*. ACM Press, 120–131. https://doi.org/10.1145/2884781.2884833

[29] Anselm L. Strauss and Juliet M. Corbin. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques.* SAGE Publications, Inc.

[30] Jari Vanhanen and Harri Korpi. 2007. Experiences of Using Pair Programming in an Agile Project. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, Washington, DC, USA, 274b. https://doi.org/10.1109/HICSS.2007.218

[31] Thorbjørn Walle and Jo E. Hannay. 2009. Personality and the Nature of Collaboration in Pair Programming. In *Proc. 3rd Int'l. Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 203–213. https://doi.org/10.1109/ESEM.2009.5315996

[32] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *Proc. 18th ACM SIGSOFT Int'l. Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/1882291.1882313

[33] Franz Zieris and Lutz Prechelt. 2014. On Knowledge Transfer Skill in Pair Programming. In *Proc. 8th ACM/IEEE Int'l. Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, 11:1–11:10. https://doi.org/10.1145/2652524.2652529

[34] Franz Zieris and Lutz Prechelt. 2016. Observations on Knowledge Transfer of Professional Software Developers During Pair Programming. In *Proc. 38th Int'l. Conf. on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 242–250. https://doi.org/10.1145/2889160.2889249

[35] Franz Zieris and Lutz Prechelt. 2020. PP-ind: A Repository of Industrial Pair Programming Session Recordings. arXiv:2002.03121 [cs.SE].