Chapter 14 Conclusion and Further Work

My goal was to understand how *knowledge transfer* in pair programming (PP) works and to formulate results that are meaningful to practitioners. I qualitatively analyzed 27 industrial PP sessions from ten companies and developed grounded theoretical concepts starting from individual *utterances* over knowledge transfer *episodes* and to whole *sessions*. I validated the high-level concepts with practitioners in four companies, none of which pointed to missing relevant elements. I therefore consider my overall theory of knowledge transfer in pair programming to be theoretically saturated.

I summarize my research contributions and advice for practitioners in Sections 14.1 and 14.2; I propose directions for further work in Section 14.3.

14.1 Research Contributions

- An extensive **review** of practitioner and scientific **literature** on PP effectiveness, influence of knowledge, task types, and pair constellations, showing that the often employed quantitative methods do not explain the observable effects and that *qualitative* approaches are needed to understand the underlying processes and mechanisms (Section 2.3).
- A refined **qualitative research process** for collecting and analyzing data to understand pair programming process phenomena based on video recordings (Sections 4.3 and 4.5).
- The first detailed description of **Focus Phases** of high productivity, which occur in some pair programming sessions and had been reported by other sources before, as well as their antagonist, the **Breakdown**, which had not been reported before (Section 6.3).
- The concept of **Togetherness** to describe what makes two software developers work together *as a pair*. Handling its five factors well can lead to Focus Phases—and to Breakdowns when not: Making sure to have a *shared understanding of the system* and of *software development in general*, as well as maintaining *one shared plan*, and possibly dealing with *workspace awareness* and a *language barrier* (Section 6.4). Prior PP research mostly did not consider the pairs' processes; differences in Togetherness might explain the effectiveness variations observed in experiments.
- A taxonomy of **Topics** that are actually addressed in industrial pair programming sessions: By far the *most* knowledge transfer pertains to system-specific S knowledge and *only some* to general software development knowledge, or G knowledge (Section 7.3.1).
- The notion of knowledge transfer during pair programming being structured in **Episodes**: During each, the pair pursues a Topic in one of six knowledge transfer Modes, i.e., Push, Pull, Parallel or Co-Production, Silent or Talking Pioneering (Chapter 9).
- A characterization of pairs that is not based on hard-to-agree-on global developer expertise (no 'expert/novice'), but on task-specific **Knowledge Needs** (Sections 11.2 and 11.3).
- A **Grounded Theory** of PP session dynamics of pairs acquiring and transferring knowledge: A relative difference in task-relevant S knowledge is addressed first; a relative difference in task-relevant G knowledge is hardly a problem, and may even pose an opportunity to transfer such knowledge *after* the pair acquired enough S knowledge to work on the task (Section 11.4). Prior PP research, in which subjects often worked with unknown systems or *none*, could not have observed this industrially relevant dynamic.

14.2 Practical Applications

These pieces of advice were condensed from my observations and address practitioners directly.

14.2.1 Maintain Togetherness

During pair programming, you and your partner want to work *as a pair* to possibly achieve the benefits of better design, fewer defects, faster progress, knowledge transfer, and more enjoyable work. This Togetherness, however, needs to be Maintained throughout a PP session. There are some **problematic signs** to look out for:

- You do not understand the intentions behind your partner's utterances and actions. Also: You feel like your partner does not understand your intentions, e.g., she may take longer than normal to react, her reaction may not match your actions, or she may not react at all.
- You cannot evaluate a proposal your partner made. Also: Your partner does not evaluate your proposal, or makes a proposal of her own without addressing yours.

These are all *conversational defects* and they are worth clearing up. Not all defects are necessarily problematic, but they may point to underlying problems such as:

- **Lack of Shared System Understanding**: You and your partner have no common mental model of the software system, no common way of referring to its parts and aspects.
- **Lack of Shared Understanding of Software Development**: You and your partner have no common toolkit (e.g., known libraries or tools) or way of approaching certain types of programming tasks and issues (e.g., writing tests before production code or using a debugger).
- **No Shared Plan**: You and your partner do not have a common conception of what steps to take to achieve which goal and where you are in the process.
- **Limited Workspace Awareness**: You and your partner cannot fully perceive each other's actions in the code editor, see and read the same things on the screen, or notice what each of you is looking at.
- **Language Barrier**: You and your partner may have difficulties expressing or understanding each other thoughts on a phonetic, lexical, or semantic level. Even within the same natural language, words and phrases do not mean the same to everyone.

All these problems reduce your Togetherness, but all can be mitigated by communicating explicitly. Not every PP session will have problems in each area, but too many unhandled problems might result in a Breakdown of the pair process, which then has none of the expected benefits and may be even worse than working alone. Addressing all the areas, however, might lead to a Focus Phase where you and your partner complete each other's thoughts and make fast progress.

Further Reading: Chapter 6 on *Process Fluency and Pair Togetherness*.

14.2.2 One Topic at a Time

Sometimes during a session, there are multiple things you want to understand or clarify at once, especially since there are two members in a pair who may want to pursue different topics. Experienced pairs manage to temporarily Limit their Scope such that only one topic is relevant at every given moment. Once they are done with it, they Return Explicitly to the original topic to make sure to not lose track. Unexperienced pairs, however, may start new lines of inquiry whenever something catches their attention, which leads to many expensive context switches and makes backtracking more difficult.

Further Reading: Chapter 10 on *Patterns of Episodes*.

14.2.3 Choose Mode of Knowledge Transfer

There are different styles for transferring existing knowledge and for acquiring new knowledge. Depending on the situation and your preferences, there are different Modes to choose from:

- **Pull vs. Push**: Knowledge which one partner already possess can be transferred either by a series of questions from the developer in need (Pull) or by explanations driven by the more knowledgeable pair member (Push). Pushing has the advantage that it may transfer knowledge whose lack the partner was not yet even aware of, which can also be confusing for her if the point of the push does not become clear soon.
  Some developers may also have difficulties giving pro-active explanations in push mode. Switching to an interview-style pull mode might help those pairs.
- **Co-Production vs. Pioneering Production**: Lacking knowledge can also be acquired through reading source code, using a debugger, or interacting with the application. Often, both partners are interested in the topic and engage in Co-Producing the knowledge. In case one partner is less interested, however, the other may Pioneer for a moment until she is satisfied and then continue to work together.
- **Silent Pioneer vs. Talking Pioneer**: Some developers prefer to read source code for themselves (Pioneer), even if their partner could explain it to them. If you decide to do this as a pair, the reader should be a Talking Pioneer, that is, to make clear what she is looking for and what she understood so far, so her partner can validate her findings and give useful pointers when the time is right. A Silent Pioneer, in contrast, makes it more difficult for the partner to follow along.

Further Reading: Chapter 9 on *Episodes of Knowledge Transfer*.

14.2.4 Embed Pair Programming Sessions in the Team Process

Before the Session Consider the technical task and the knowledge it requires about the software system as such and about software development in general (e.g., frameworks, technology stack, design patterns, testing strategies): What are your Knowledge Needs, i.e., what relevant knowledge does either of you *not* yet possess? Are there One-Sided Gaps where one of you knows more about some area, or Two-Sided Gaps where both of you lack knowledge?

A setting of Complementary Gaps can be mutually beneficial, because both partners can bring in some knowledge advantage. If your pair constellation is not yet complementary, maybe the task can be amended in a way that both partners' respective expertise can come into play, e.g., by keeping an eye open for code smells and possible refactorings.

A Two-Sided Gap regarding general software development knowledge, e.g., where both of you do not understand some technology, will probably not make for a good PP session if you also try to work on a technical task. Better choose a different task and/or pairing.

Discuss which Knowledge Needs you want to address in your session. Understanding the task-relevant parts of the software system is usually required for both of you, but sometimes only one needs to continue with the task and an unclosed One-Sided Gap may be tolerable.

After the Session Reflect on what either of you learned. Chances are that each of you remembers different episodes. Together, you get a fuller appreciation of which knowledge you transferred and acquired, and where newly discovered knowledge gaps are.

Further Reading: Chapter 11 on *Session Dynamics* and Section 13.2 on *Preparing and Reflecting on a PP Session*.

14.3 Further Work

My work is done, but there is more to do. The following two areas for further investigation occurred to me while considering the limitations inherent to my data (Section 4.3.4):

Ad-hoc Pairings I have no idea in which regards spontaneous pairings are different from the at least half-planned sessions that ended up in the recordings repository. Recording such sessions would probably require an always-on setup to get rid of session start-up times (similar to what Socha et al., 2015, 2016, did, see page 144).

Application Domain Knowledge Developers in consulting may encounter new domain concepts more often than my pairs, who work in their company's own product. Knowledge Needs of this type might influence pairs in different ways than S and G knowledge do.

Just as I used the *base layer* (Salinger & Prechelt, 2013), further pair programming research may build upon my methods and concepts. Based on interesting phenomena I have seen in my data but not analyzed, I deem the following two areas particularly relevant and insightful:

Fluency and Togetherness The Fluency of most analyzed sessions was normal, with both Breakdowns and Focus Phases being the exception. I do not think that splitting up the Fluency concept into more than these three levels is useful: Even though the appearance of normal PP process varies across sessions, these differences do not appear to matter much. Almost all analyzed pairs manage to achieve something useful in their sessions, with not many avoidable detours along the way. There are, however, three directions I consider useful:

- **Focus Phases** appear highly productive and enjoyable. But: Is there an actual difference between sessions with and without Focus Phases, or are they just an occasional side-effect of high Togetherness? If they are desirable, is there anything pair programmers can do (e.g., more strict Scope Limiting) to get more and longer Focus Phases?
- Similarly, further types of **Breakdowns** are worth investigating. I heard anecdotes about terrible pair sessions, but none of my recorded sessions came close to those stories. Assuming these stories are not over-dramatized, there appear to be practically relevant ways how PP can go 'wrong' which my concepts cannot describe.
- Finally, I only considered how pairs **Maintain Togetherness** regarding two factors: *Shared system understanding* and *shared understanding of software development*. I provide initial observations on how pairs deal with *language barrier*, *workspace awareness*, and *one shared plan* in Section 6.4.4. There might also be more factors influencing a pair's Togetherness besides the five I identified.

Decision Making The Episode concept can easily be transferred to other process aspects of pair programming, such as discussing design decisions. I also expect the Propellor concept to be applicable (i.e., one or possibly both developer(s) being active rather than reactive), and different Modes to exist, in particular something Push-like where one developer pitches an idea, and something Co-Production-like where both partner go back and forth on something. Similarly, the patterns of Branching Wildly, Scope Limiting, and Returning Explicitly strike me as not necessarily specific to knowledge transfer.

Years of qualitative research and building on the *base layer* led to the concept of Togetherness. Its five factors appear central to fully understanding how *pair programming* works: maintaining (1) a shared understanding of the system and of (2) software development, (3) one shared plan, (4) good workspace awareness, and (5) dealing with any language barrier. My Grounded Theory of *knowledge transfer* explains the mechanisms of the first two factors to a degree that is enabling and meaningful to practitioners; the foundation for the other three has been laid.