# Observations on Knowledge Transfer of Professional Software Developers during Pair Programming

Franz Zieris
Freie Universität Berlin
Institut für Informatik
14195 Berlin, Germany
zieris@inf.fu-berlin.de

Lutz Prechelt
Freie Universität Berlin
Institut für Informatik
14195 Berlin, Germany
prechelt@inf.fu-berlin.de

## ABSTRACT

*Context:* Software development is knowledge-intense work, and so is pair programming. However, the importance of knowledge transfer in pair programming is usually only stressed for expert-novice constellations. *Goal:* Understand how knowledge transfer during pair programming works and eventually provide guidance for practitioners. *Method:* Detailed qualitative data analysis of full-length recordings of industrial pair programming sessions. *Results:* Expert software developers need to transfer knowledge, too, in order to conduct productive pair programming sessions. There is a diversity of beneficial and potentially problematic patterns, which even good pairs do not steadily apply or avoid, respectively. *Conclusions:* Pair programming is a versatile practice that even experts can profit from. Knowledge transfer skills do not automatically emerge from good software development skills, but can probably be learned.

## CCS Concepts

**•Software and its engineering → Agile software development; Pair programming; •General and reference** → *Empirical studies;* **•Human-centered computing** → *Ethnographic studies;*

## Keywords

Pair programming, knowledge transfer, grounded theory

## 1. INTRODUCTION

The general discussion on pair programming tends to distinguish two use cases. First, there is the *productive use* of pair programming in the XP sense [2]: Two developers work together to complete tasks with higher quality in less time. This view has been subject to a decade-long discussion on the overall efficiency (e.g. [5]). Second, there is the *knowledge transfer use*, a more recent perspective which regards pair programming as a mentoring technique to bring new team members or novices up to speed (e.g. [11]).



*ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA*
 ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889249

It seems to be common sense (at least tacitly) that these two use cases are distinct, i.e. developers are either good enough to perform a *productive* session (without any relevant knowledge transfer) or their skill levels are too far apart to be productive so they retreat to a *knowledge transfer* session. We will discuss this strict dichotomy and prove it wrong.

Pair programming is a useful practice because both developers can add value to the process in multiple ways, e.g. through complementary knowledge which enables pairs of developers to work on tasks that would be much more challenging for each of them alone. But the combination of the individuals' knowledge does not simply *occur*. Instead, knowledge transfer between the developers is part of the pair programming session itself – especially between expert developers.

In this article, we will discuss on the actual process of knowledge transfer in pair programming. We will put an entire pair programming session of over two hours length in the limelight. That session serves as the infrastructure for explaining and illustrating our main observations which are:

- **NOTEACHER**: There is usually no pair member who is more knowledgeable in all relevant areas.
- **INFERIOR**: Occasionally, it is the knowledge-wise inferior developer who has the bigger positive impact on the session's progress, either through solving a problem (insight) or through avoiding a mistake (diligence).
- **PUSH**: Existing knowledge is not just *pulled* from the knowledgeable developer through asking questions; it will also be *pushed*, even if her partner did not specifically ask for it.
- **TALK**: Producing new understanding by one single pair member that is accompanied by uttering immediate insights (rather than sinking into utter silence) allows the partner to both help and learn, too.
- **RESYNC**: If one developer pulled ahead, the pair needs to get close together again to fully use complementary knowledge. This requires some additional effort, which will pay off later on.
- **PARALLEL**: Even equally quick grasp does not guarantee common understanding. Producing new understanding as a pair requires additional synchronization. Otherwise the pair risks being on "parallel tracks" and taking avoidable detours in their process.

The remainder of this article is structured as follows. We will discuss related work in Section 2 and describe our methodology in Section 3. Section 4 illustrates our findings. We

discuss the limitations of our work (Section 5) before we conclude (Section 6).

# 2. RELATED WORK

## 2.1 Research on Work in Pairs

Pairs have been subject to cognitive research for decades. The interaction of two persons – often referred to as "dyads" – is usually studied under controlled conditions. Typically, individuals' and pairs' performances are compared, showing an *effect* of pair work, but providing no insight into the pair members' actual interaction (e.g. [1]). In other scenarios, which specifically focus on the collaborative process, pairs are tasked only with well-defined exercises with only few degrees of freedom, such as discovering a regulatory mechanism of genes through choosing between only two types of experiments [9], or understanding the mechanics of a sewing machine [7]. We are not aware of any research that considered non-programming pair work on tasks that exhibit the typical traits of software development with respect to task complexity, dealing with uncertainty and volatility, freedom of action and creativity, concreteness of outcome (source code).

From the viewpoint of organizational and social sciences (e.g. embodied in the discipline of *small group research*), pairs can be considered groups of two. However, most of the investigated phenomena (such as conflict and negotiation, decision making, coalition formation) are usually studied in groups of three and more (e.g. see [8, p. 531] for a rationale). Some other researchers argue that "dyads can be groups and often are" [15], but, ultimately, social science literature that could be used as a starting point for our research is scarce.

## 2.2 Research on Knowledge Transfer in Pair Programming

To our knowledge, there has not been any study yet that focused on the actual process of knowledge transfer between expert software developers.

Most studies are limited in so far as they only demonstrate the mere fact that knowledge is transferred. This is done either through questionnaires (e.g. [4, 10]), or through inference from product quality and – in case the subjects were students – test results at the end of the semester (e.g. [6]). Either way, the actual process of knowledge transfer cannot be understood by such approaches as they regard the pair programming sessions as black boxes.

Other studies "open" that box, e.g. Chong and Hurlbutt [3], who observed and partially recorded pair programming sessions. They state that large differences in the pair member's knowledge levels seem to hinder the exchange of ideas during the session. The work of Plonka et al. [11] focuses on expert-novice constellations and explicitly speaks of "teaching". However, they also state that "a certain degree of knowledge transfer would be expected within every PP constellation" as "developers never have identical knowledge".

Our work investigates knowledge transfer phenomena without filtering out particular sessions or constellations at an early stage. In [16], we reported our initial findings on what we called "Knowledge Transfer Skill": Good pairs manage to (1) not pursue multiple knowledge needs at once, (2) recognize complex Topics and split them into separate Topics, (3) recognize complicated Topics and deal with them in stages, (4) not lose sight of Topics. The present study adds many more concepts relevant for understanding knowledge transfer behavior.

# 3. METHOD

## 3.1 Data and data collection

As mentioned above, our research is concerned with the actual pair programming process. Over the last years, 48 professional developers from 11 different companies agreed to have 49 pair programming sessions from their everyday work recorded and analyzed. Such a recording consists of a full-resolution screen capture, a webcam video, and an audio recording of the developers' conversation. The typical session from our collection lasts between 1 and 2 hours. Where no session identifier is mentioned below, the data is from session AA1.

## 3.2 Data analysis

We employ the qualitative and highly systematic Grounded Theory Methodology (GTM), in particular, Strauss and Corbin's approach [14]. We used the practices of open, axial, and selective coding to identify and conceptualize relevant phenomena, and to develop their relationships.

In this article, we will not present the details of the analysis process, but only the resulting insights. Therefore, there exist many more concepts that were necessary along the research process, which do not appear in this article.

Since the recording of new sessions is tedious and the efforts not well predictable, we applied theoretical sampling only once, selecting session recordings from our repository, but not iteratively. Consequently, our research has not yet reached theoretical saturation.

## 3.3 Notation

When we refer to concepts resulting from the GT analysis, we will use small caps, as in Some Concept. Since we chose a storyline format for the main part of this article (see below), we do not define the concepts in an isolated section. Instead, we will inline the definitions, usually when a concept is used for the first time, and we will highlight that occurrence, as in ►Some Concept, to facilitate looking it up.

In verbatim quotes, we indicate program identifiers, pauses, comments, and replacements as follows: *"Huh? `RemoveTargets` (...) <*sighs*>. We could ask <**chief architect**>."*

## 3.4 Format

We chose a story-format for the main part of this article, which means we will follow the events of the session from the beginning in chronological order. We do so because iterating the concepts and findings one by one would be arduous to read. Furthermore, most of the examples are from the same session. This is not because example episodes would be scarce in our data – we analyzed over 400 – but because in order to be comprehensible each episode needs a lengthy preface explaining the context.

In Section 4.2, we will go through a pair programming session of professional software developers with varying granularity. The session parts relevant for this article will be presented with verbatim excerpts revealing the full depth of the actual knowledge transfer process; for the sake of brevity and continuity, the parts in between will by summarized but

not omitted. Each subsection adheres to the following structure:

- The heading provides an informal short name for the episode along with the list of the identifiers of the covered observations.
- **Situation**: A short narrative of what happened between the close-up scenes. This is for understanding the current task and not losing track of where the pair currently stands in its session. Here, we also introduce the domain and software concepts the pair is currently dealing with. This is to understand the developer's technical jargon.
- **What happens**: The concrete description of what the pair did in that episode, followed by the verbatim excerpt.
- **What it means**: The application of our concepts to the scene.
- **Discussion**: A discussion of the general observations that the example illustrates along with a rough assessment of how common similar situations are in our data.

# 4. RESULTS

## 4.1 Context

### 4.1.1 The Company

The pair programming session in focus was recorded in a German company that has been developing a web-based content management system (CMS) for many years. The system comprises two major components. First, the backend written in Objective-C, which deals with business rules and SQL database interaction. Second, the Java GUI which interacts with the backend through an XML API and renders HTML output to be displayed in a web browser.

### 4.1.2 The Developers

The pair members know their domain well, are experienced developers, and pair-program regularly. Developer $\mathcal{J}$ has very good structural knowledge of the Java frontend and its individual classes, as well as practical knowledge of the Eclipse IDE and the Java programming language. His colleague developer $\mathcal{C}$ is more familiar with the backend and the SQL database, the VIM editor and the UNIX shell, and the Objective-C programming language. However, these differences are small: Each of them would be able to find their way around in the other's system.

### 4.1.3 The Task

Their session, which lasts 2 hours and 19 minutes, can be characterized as corrective maintenance. The CMS supports multiple entity types (such as *task* or *unreachable link*) which can (among other things) be "active" or "inactive". The list displays of these entities should reflect their activeness through different icons and labels.

The starting point for this session were four lists that were reported as inconsistent as to how they render the entries' status: Sometimes the icons did not match, sometimes the labels.

They start their session in the frontend to find out:

1. how the icon selection and the label decoration is implemented;
2. that the four lists are implemented similarly but with important (and necessary) differences;
3. that "activeness" means different things, depending on the entity type;
4. that they need to amend the backend (and the communication interface) because one entity type is lacking the required status information; and
5. that there is another boolean property for all these entity types, called "`isMirror`", that is handled inconsistently across the types as well.

They end up with changes in both front- and backend, and with a consistent rendering behavior for all four types for the `isActive` property.

## 4.2 The Session

### 4.2.1 Understanding Status Quo (RESYNC PUSH TALK)

**Situation**

In the beginning, developers $\mathcal{J}$ and $\mathcal{C}$ read the bug tracker entry and reproduced the faulty behavior in the running application: Four list displays have four different ways to represent the inactiveness of their entries. Since the bug tracker entry stated no specific requirements, the pair discussed the intended graphical appearance of inactive entries. They tried to phone the product management, but could not reach anyone. Developer $\mathcal{J}$ was close to canceling the session in the fifth minute, but $\mathcal{C}$ came up with a software design that would allow to defer the actual graphic decision; $\mathcal{J}$ agreed. This is the first example of INFERIOR.

On the technical level, the first of the four lists is the `FinishTasksPage`, of which the entries have correctly rendered labels, but the same icon for both 'active' and 'inactive' states. Internally, these entries are `TaskNode` objects, a subclass of `Node`, which in turn defines two methods for rendering the icons: First, the parameterless `getIconName` which only depends on the `Node` object itself. Second, `getIconPrefix` which should encapsulate configurable behavior and is therefore provided with a `ViewConfig` parameter.

The `getIconName` method is already implemented in `TaskNode`; the prefix method is missing, so the empty default implementation is inherited. `TaskNode`'s `getIconName` method calls a static method of another `Node` subclass, called `ObjectNode`. This delegation can be considered a form of horizontal inheritance, which will cause the pair some trouble.

The pair is yet unaware of all of these technicalities.

**What happens**

The pair needs to find the source code responsible for creating the false output. After about two minutes of going through the source code in a top-down fashion, they arrive at a point where $\mathcal{J}$ has gathered enough understanding to make a concrete design proposal, which $\mathcal{C}$ does not yet understand. As $\mathcal{J}$ notices that his partner is still puzzled by the currently visible source code, he starts to explain it. $\mathcal{C}$ then understands the delegation and accepts the existence of the two-part icon name, but is perplexed about the rationale behind it, as he does not know about the `ViewConfig` yet. He starts to read the surrounding source code and keeps uttering his insights, allowing $\mathcal{J}$ to follow him and hook in with further explanations. A few moments later,

$\mathcal{C}$ understands both the current code and $\mathcal{J}$'s design proposal, and they can start with the implementation of `TaskNode.getIconPrefix()`.

In the developers' own words, it sounded like this:

> $\mathcal{J}$: *"Sure, we need to override `getIconPrefix` in `TaskNode` ... it's missing."* – $\mathcal{C}$: *"Uh? Is it reasonable to ..."* – $\mathcal{J}$: *"See, the icon name is plugged together ... and the `TaskNode` simply delegates this to the `ObjectNode`, that's why these are all static methods there, in order to not duplicate the code."* – $\mathcal{C}$: *"Ah! But why is it built so cumbersome?"* – $\mathcal{J}$: *"Who knows ..."* – $\mathcal{C}$: *"Well, I want to know. <*opens call-hierarchy of current method*> See, it's called from `Html.renderIcon()` <*opens that method*> and here it's done through the `ViewConfig`"* – $\mathcal{J}$: *"Sure, it's done the same way now, and it's right that way. We only need to overwrite that method."* – $\mathcal{C}$: *"Ah, it's because of the on/off, the showing and hiding and stuff."* – $\mathcal{J}$: *"Exactly."*

**What it means**

The excerpt presented above actually contains three instances of what we call a KNOWLEDGE TRANSFER EPISODE or just ►EPISODE: An interaction sequence that pertains to a single Topic that is pursued in a constant MODE (details follow).

First, neither of the two developers was in possession of the required information, the ►TARGET CONTENT (i.e. "The responsible method is `TaskNode.getIconPrefix()`"). But both of them understand the ►TOPIC (the focus of interest, i.e. "Which is the responsible method?"), so they could start to generate the TARGET CONTENT in a ►CO-PRODUCE EPISODE: Both developers are engaged in generating a shared understanding of the system's status quo.

Second, $\mathcal{J}$ perceived an ►EXTERNAL KNOWLEDGE NEED as $\mathcal{C}$ was lagging behind *("Uh")*, and then explained parts of the current design. He started a ►PUSH EPISODE, i.e. he explained without waiting for $\mathcal{C}$ to formulate an actual question.

Third, $\mathcal{C}$ still had an ►INTERNAL KNOWLEDGE NEED which he dealt with through the "solo variant" of CO-PRODUCE: He gained a better understanding as a ►TALKING PIONEER. Since $\mathcal{C}$ kept uttering his insights, $\mathcal{J}$ could provide him with relevant and timely information.

To summarize the concepts so far: EPISODES revolve around TOPICS which arise from perceived KNOWLEDGE NEEDS (internal or external) which can be satisfied by the TARGET CONTENT. Depending on whether any of the two developers already possesses the TARGET CONTENT, the EPISODE can run in the ►MODES PUSH and PULL (see below); otherwise it will be created in CO-PRODUCTION or PIONEERING PRODUCTION.

**Discussion**

Had $\mathcal{J}$ worked alone at this point, he could have started the implementation about one minute earlier – the time it took $\mathcal{C}$ to catch up. This is an example of RESYNC: The developers invested some time to be able to work as a unit, and we will see the benefits of their pair work in Sections 4.2.2, 4.2.8 and 4.2.9.

Furthermore, this is an example of PUSH. $\mathcal{C}$ did not need to formulate a question for $\mathcal{J}$ to start with explanations. It was $\mathcal{J}$ who deemed it useful to keep his partner close by.

Finally, this is an example of TALK. After $\mathcal{J}$'s explanations, $\mathcal{C}$ developed a new NEED that his partner would not clarify. But instead of silently reading source code – as some other developers in our session recordings do in such situations – he kept uttering his insights, thus making it easier for $\mathcal{J}$ to help him.

In comparison with our other sessions, this is a not too common beginning. Usually, one of the developers has already worked on the task, making him the "Task Expert" (see [12]). In such constellations, the Task Expert will at least try to start PUSHING the recent changes. Depending on the partner and her knowledge level, this might be sufficient to actually start the session – in other cases, the partner favors to engage in a PIONEERING EPISODE and take a look himself. ►SILENT PIONEERING, i.e. not letting the partner know anything about one's progress, is something that some developers do, but it can also lead to hard-to-detect misunderstandings.

### 4.2.2 *Effortless Pull and Push I (PUSH RESYNC)*

**Situation**

The pair has decided to overwrite the method `getIconPrefix` in the `TaskNode` class and has written down the method signature; the method body is still empty.

Technically, `TaskNodes` do not have a field for holding their 'active' state. Instead, there is only a so-called `ObjectHandle` to access the underlying business object called `CMMiniObject`. The `ObjectHandle` offers two `fetchMiniObject` methods to retrieve that business object: One without arguments, and another with a boolean argument `allowMicroObject`. If set to `true`, that method will return a fallback `MicroObject` in case the actual `MiniObject` is not available.[1]

**What happens**

$\mathcal{C}$ sees both 'fetch' methods offered by Eclipse's auto-completion, remembers a pending API change, and asks $\mathcal{J}$ about it. $\mathcal{J}$ answers promptly, understands why $\mathcal{C}$ is asking, and votes for the method with the boolean argument, which $\mathcal{C}$ then chooses.

> $\mathcal{C}$: *"Do the `MicroObjects` still exist?"* – $\mathcal{J}$: *"Yes, they do. They do here, on your machine. On the working branch, however ... `true`."*

They decide on storing the return value of `fetchMiniObject(true)` in a local variable, and $\mathcal{C}$ expects the type to be `MiniObject`, but such class cannot be found by the IDE. $\mathcal{J}$ sees this and corrects two things at once (the missing `CM`-prefix and the wrong class) just by saying:

> $\mathcal{J}$: *"`CM ...MicroObject`."*

**What it means**

$\mathcal{C}$'s question (and $\mathcal{J}$'s answer) form a very short ►PULL EPISODE, i.e. actively triggering explanations, e.g. by asking questions. $\mathcal{J}$ then inferred an EXTERNAL KNOWLEDGE NEED from $\mathcal{C}$'s actions and PUSHED information about the actual method return type (PUSH).

**Discussion**

Both EPISODES are about as short as they could be. Staying close together throughout the session is what enables this pair to achieve this (RESYNC).

Again, it may seem as if $\mathcal{J}$ would be better off alone: He knows the answers to all the questions that arise along the

[1] In our work, technical details like these frequently need to be understood (even without suitable discussion of the pair) in order to make sense of the pair's behavior.

way. But on the one hand, $\mathcal{C}$'s great moments are yet to come, and this is the necessary groundwork. One the other hand, without $\mathcal{C}$'s design proposal at the very beginning of the session, $\mathcal{J}$ would probably have deferred working on that task until the product management was available again.

In other sessions, neither PUSH nor PULL EPISODE are regularly that short. The first may require short check-backs, latter may require multiple attempts from different angles until the knowledgeable partner eventually understands the question. We already reported on much longer and more structured Pull Episodes, in particular the CLARIFICATION CASCADE [16].

### 4.2.3 *Push to clear-up Misconceptions (PUSH)*

**Situation**
The first version of `TaskNode.getIconPrefix()` was compilation error free. However, $\mathcal{J}$ noticed that the current implementation did not take its `ViewConfig` parameter into account. They decided to search for further reusable methods, found one, inspected its code, and decided to change their implementation to reuse that method.

Technically, the implementation of `TaskNode.getIconPrefix()` consists of two steps. First, the object's activeness is determined. Second, the activeness information is used to determine the icon prefix.

The first version implemented by the pair did the first step locally and delegated the second step to a static method of the sibling class `ObjectNode`. The newly found method is another static member of `ObjectNode` and is able to take over the first step, too, and can even consider the given `ViewConfig`. That method can handle both `MicroObjects` (the superclass, in case of an error) and `MiniObjects` (the subclass, the actual business object). It performs only superficial checks on the first, but in-depth checks on the latter.

**What happens**
After delegating the activeness-determination to the newly found method, $\mathcal{C}$ has recalled that it does only general checks on `MicroObjects`. $\mathcal{C}$ confuses super- and subclass, and starts to change the implementation so it uses the first `fetchMiniObject()` variant which returns actual `MiniObjects` instead. As this would bypass the error-handling of the second fetch method, $\mathcal{J}$ objects and constructs an easy-to-grasp example for the error handling. $\mathcal{C}$ understands, remembers the correct class hierarchy, and undoes his source code changes.

> $\mathcal{C}$: *<*changes the called method*> – $\mathcal{J}$: "No!" – $\mathcal{C}$: "Yes! We saw that it [`ObjectNode`] only asks `isActive()` and disregards the `editedContent` [part of the in-depth check]" – $\mathcal{J}$: "But what happens if the `Task` belongs a non-readable object?" – $\mathcal{C}$: "Uh, then I'll see an Exception." – $\mathcal{J}$: "Yup." – $\mathcal{C}$: "Ah, sure, it's fall-back <*undoes his changes*>, right, right."*

**What it means**
$\mathcal{J}$ did not hesitate to PUSH the rationale behind using the superclass as he saw that $\mathcal{C}$ was about to make a mistake from which he inferred $\mathcal{C}$'s KNOWLEDGE NEED.

**Discussion**
$\mathcal{J}$ helps $\mathcal{C}$ avoid an expensive mistake he would probably have made alone. This is an example of PUSH.

Across our analyzed sessions, there were very few instances of PUSH EPISODES which were not adequate, so behavior as that seen above is typical. The PUSHED information may not always be correct (as in Section 4.2.6) or crucial for the session. But we did not encounter PUSH EPISODES that caused actual harm.

### 4.2.4 *Doing the Right Thing (TALK INFERIOR)*

**Situation**
The pair already worked on two of the overall four lists. This first list (containing `TaskNodes`, see excerpts above) was simple because they could use an existing `ObjectHandle` to fetch a `MiniObject` for which the `ObjectNode` class already encapsulated the activeness-determining logic.

The second list displayed `TaskOverviewNodes` and like the first one, the labels were rendered correctly and the icons were not. But unlike the first node type, it used a low-level mechanism to call the XML-API of the backend for retrieving the data. After a CO-PRODUCE EPISODE the pair understood that they need to amend both the XML query placed by the Java frontend *and* the database interaction in the Objective-C backend to retrieve the objects' activeness information. They decided to skip the second list to be able to finish the Java frontend first, before switching to the Objective-C backend.

They just looked at the third list in the web browser: Here, the icons appeared to be rendered correctly (indeed, the pair will find an existing `getIconPrefix` implementation here). The entries' *labels*, however, did not yet reflect their activeness. The pair already found out which `Node` subclass the third list uses and opened its source code.

On a technical level, the third list uses `VersionEntryNodes`, which represent previous revisions of contents or "archived contents". The activeness of such entries only depends on the 'valid until' timestamp, which leads to a much simpler logic than `ObjectNode`'s activeness-determining logic.

**What happens here**
As they are about to start looking for the (faulty) label-rendering logic, $\mathcal{C}$ is puzzled as he discovers that the correctly-behaving icon-rendering uses a simple local method to determine the activeness, instead of delegating it. $\mathcal{J}$ wants to continue with the label-rendering, but $\mathcal{C}$ is unsure whether the underlying logic for both renderings is correct. After a short PIONEERING code-inspection, he comes to the conclusion that he does not trust the logic and that he would like to have that functionality in the backend instead. $\mathcal{J}$ disagrees and points out that there is nothing wrong with the logic as it is.

> $\mathcal{C}$: *"Uh, why does it have its own `isInactive()`?" – $\mathcal{J}$: "Don't know. And I don't want to ..." – $\mathcal{C}$: "No, I want to fix this. In the end, this one does it different than the other, and it's crappy again. <*inspects code, makes design proposal*> I mean, the backend knows whether the contents are active or inactive anyway. And I don't want to have a duplicated calculation here in the GUI that is probably broken anyway." – $\mathcal{J}$: "... Why should it be broken?" – $\mathcal{C}$: "Because it depends on more than just ..." – $\mathcal{J}$: "Nope, this <*points at current implementation*> is all that counts for archived contents" – $\mathcal{C}$: "<*reads again*> hm, yeah, guess you're right. Yes, you are right."*

**What it means**
$\mathcal{J}$ did not enter $\mathcal{C}$'s PULL EPISODE. As $\mathcal{C}$'s doubts about the local method are not resolved and the INTERNAL KNOWL-

EDGE NEED still exists, he starts a TALKING PIONEERING EPISODE to understand the code better. This way, $\mathcal{J}$ could follow $\mathcal{C}$'s thoughts and eventually comprehend his misunderstanding: $\mathcal{C}$ was not aware that the activeness of `VersionEntryNodes` is, in fact, easy to determine. Consequently, he expected a complex logic but found a simple one, which, to him, was suspicious.

**Discussion**
$\mathcal{C}$ was diligent here for a good reason: He noticed an anomaly in the code which the frontend expert next to him could not explain away. In the end, $\mathcal{J}$ was right about this method being unproblematic, but he certainly did not *know* this from the start in the sense of *justified true belief*. Arguably, $\mathcal{C}$'s diligence might have just as well led to the discovery of an actual problem, making this an example of INFERIOR.

And yet again, $\mathcal{C}$'s utterances while he PIONEERED made it easier for $\mathcal{J}$ to help him (TALK).

TALK INFERIOR behavior like this is common in our recordings.

### 4.2.5 *Lack of Synchronization (PARALLEL)*

**Situation**
After having understood that `VersionEntryNode` has a local implementation of `isInactive` for a good reason, the pair shortly discussed the pros and cons of moving such logic to the backend. They agreed on deferring this until the actual task is done. At this point $\mathcal{C}$ lost track of the latest open TOPIC – which was: *understanding the label-rendering logic of the third list* – and needed to be reminded by $\mathcal{J}$.

**What happens here**
They open the source code of the third list again, and start reading. They both come up with different ideas about where to look next, do not explicitly decide against one of them, but simply follow $\mathcal{C}$'s idea. After one minute, $\mathcal{C}$ even halfheartedly brings up $\mathcal{J}$'s idea again but it goes unheard. After three more minutes, they arrive at the same conclusion that $\mathcal{J}$ suspected from the beginning.

> <simultaneously> $\mathcal{J}$: *"Ah, the `Nodes` themselves should have such rendering stuff, right?"* || $\mathcal{C}$: *"The problem is, this is a `List`."*
> $\mathcal{J}$: *"Ah, this is a leftover old-style list page!"* – $\mathcal{C}$: *"Exactly [...] and those render themselves here <*navigates to a method of the superclass*>"* – <one minute of scrolling through classes> – $\mathcal{C}$: *"Maybe it's really the `Nodes` ... no."* – <three minutes of scrolling through classes> – $\mathcal{C}$: *"Ha, it says `node.getRenderingClass`. So it turns out the `Nodes` need to do this."*

**What it means**
This is no CO-PRODUCTION because the developers did not fully synchronize their insights. Instead, they performed a ►PARALLEL PRODUCTION.

**Discussion**
That mode is risky. It might save the synchronization effort and the pair might still end up with a similar understanding just like after a successful CO-PRODUCTION. But they might just as well overlook relevant aspects and waste time on non-optimal paths (PARALLEL).

In the example above the pair lost about four minutes. In another session – KA6, with two developers who are in the process of learning a new JavaScript framework – the pair did not notice it was not synchronized for a much longer period. Developers $\mathcal{S}_i$ and $\mathcal{S}_a$ were reading in the online documentation about how to programmatically trigger an event, which they needed for writing an integration test – but the only utterances they made were $\mathcal{S}_a$: *"Ah"* and $\mathcal{S}_i$: *"Ah-ha!"*. They closed the documentation, kept on working, and after 40 minutes, $\mathcal{S}_a$ is puzzled by a code statement $\mathcal{S}_i$ produced a few minutes ago. It was an idiom $\mathcal{S}_i$ memorized from the documentation while saying *"Ah-ha!"*. He understood it, used it, and $\mathcal{S}_a$ was probably not on the same page for 40 minutes.

PARALLEL behavior is common, but differs greatly in the amount of time that is lost.

### 4.2.6 *Complementary Knowledge (INFERIOR NOTEACHER)*

**Situation**
The pair fixed the third list, such that the labels of inactive entries are rendered correctly. The fourth list rendered the icons correctly, but not the labels. The pair found out the node type of the fourth list, and started to take a look at its icon-rendering implementation before planning to correct the label-rendering.

The nodes in question are `UnreachableLinkNodes` which represent internal or external links that are used somewhere without having a valid target. Similar to `VersionEntryNode`, the `UnreachableLinkNode` class implements the activeness-determination logic locally, which is rather simple: Dead internal links are considered to be "inactive".

**What happens here**
The pair does not understand the logic behind the implementation they are looking at. The frontend expert $\mathcal{J}$ notices this oddity in the code, but it is the backend developer $\mathcal{C}$ who – after understanding the pieces – can make the connection between something being 'resolved' and something being 'inactive'.

> $\mathcal{J}$: *"Ha! Ha! <*points at screen*> It uses `isResolvedInternal`? ... How does it relate to being inactive?"* – $\mathcal{C}$: *"Eh, what does `isResolvedInternal` even mean?"* – $\mathcal{J}$: *"Well, if it's an internal link and it's `resolved`."*[2] – $\mathcal{C}$: *"Well, it's `resolved`. But it's also not reachable, because it's on the 'Not reachable' page. Either because the external page is not available, OR because the internal link target is not active anymore. <*smirks at $\mathcal{J}$*> So they used this circumstance and say: 'Okay, if it `isResolvedInternal`, but it should be displayed here, then it's obviously inactive.'"* – $\mathcal{J}$: *"Ok, yeah."*

Although we cannot know how long $\mathcal{J}$ would have needed to come to the same conclusion, another episode one minute later indicates that $\mathcal{J}$'s understanding is not as solid as $\mathcal{C}$'s.

> $\mathcal{J}$: *"No-no-no, if it's `resolved` then everything's fine."* – $\mathcal{C}$: *"Nope, then it's inactive."* – $\mathcal{J}$: *"Huh? No way!"* – $\mathcal{C}$: *"`isResolvedInternal` means 'this link is a `resolved` internal link.'"* – $\mathcal{J}$: *"Sure <*facepalms*> and it's `unreachable`, we already know that. Yes, ok. T'was nonsense, you're right."*

[2] $\mathcal{J}$'s original German answer was not as trivial. Both developers spoke German throughout the session, *except* for when they referred to the English identifiers from the source code. In a sense, the developers could actually "hear" the monospaced font used in this article.

**What it means**
The first part is another Co-Production Episode in which both developers are actively engaged.

In the second part, *both* developers perceive an External Knowledge Need in their respective partner at the same time and *both* try to Push what they think should be the correct Target Content.

**Discussion**
This time, $\mathcal{C}$ was faster than $\mathcal{J}$, even though it was $\mathcal{J}$'s area of expertise (INFERIOR). But without $\mathcal{J}$'s reassuring explanation of what the called method actually does, it would have been much harder for $\mathcal{C}$, too (NOTEACHER).

The frequency of INFERIOR NOTEACHER behavior is induced by how often knowledge possessed by only one pair member is relevant for the task at hand. In some sessions you will never see such behavior, in others they occur all the time.

### 4.2.7 *Investing in Common Ground I (TALK RESYNC)*

**Situation**
The pair looked at all four lists and solved all problems they could tackle without leaving the frontend source code. The remaining changes for the session's task required to extend the backend API. They continued their work at the second list, which they skipped in the first pass (see Section 4.2.4).

The second list retrieves data from the backend through named commands which are wrapped in Java classes called `Accessors`. They do not contain an explicit list of fields to fetch; instead, the frontend sends the command to the backend, which creates an XML response with a certain structure, which the frontend simply *expects* to be well-formed when it creates Java objects from it.

**What happens**
$\mathcal{J}$ proposes to implement the frontend part first (pretending the backend would already send all necessary data), and then turn to the backend to actually include the new data in the response. $\mathcal{C}$, however, expects the frontend to ask for the data specifically – like a qualified SQL SELECT query – which is why he does not understand $\mathcal{J}$'s proposal.

Instead of following $\mathcal{J}$'s proposal, $\mathcal{C}$ wants to know how the backend extracts the fields-to-fetch from the command (not knowing that this is not how this list's command works). $\mathcal{J}$ eventually understands $\mathcal{C}$'s misconception and clarifies the issue.

> *$\mathcal{J}$: "We just need to add a new key to the map. Everything it gets from the backend is put into this map." – $\mathcal{C}$: "Everything?" – $\mathcal{J}$: "Yep. So, in that `Node`, we can pretend we already got something back." – $\mathcal{C}$: "And this `TaskAccessor` is only used for the `TaskOverviewPage`?" – $\mathcal{J}$: "Yep." – $\mathcal{C}$: "But who does it ask?" – $\mathcal{J}$: "It's a special thingy." – $\mathcal{C}$: "Ah, indeed ... `CMD_LIST_OVERVIEW` <*minimizes Eclipse and switches to the backend source code*>" – $\mathcal{J}$: "That's really a special ... huh, why do jump away now?" – $\mathcal{C}$: "I want to see how it's implemented." – $\mathcal{J}$: "But why? You simply define a new key, say you want that one too, and then we are done in the GUI – just that we only don't get the key yet. And in the end, we add it to the backend. <*waits and watches $\mathcal{C}$ scroll*>" – $\mathcal{C}$: "<*navigates through the backend*> See, it doesn't care ..." – $\mathcal{J}$: "Sure, you don't say what you want, but ..." – $\mathcal{C}$: "... here are no keys!" – $\mathcal{J}$: "No, of course not. It simply returns a set, but you don't ask for something specific. <*points at Java code*> These are the keys as they're returned." – $\mathcal{C}$: "Huh? But it needs to ask it somewhere?" – $\mathcal{J}$: "No, there is this command and it sends back a fixed set. In the frontend we only say how they're called so we can pull them out correctly." – $\mathcal{C}$: "Ah! Now I got it."*

**What it means**
Not only does $\mathcal{C}$ not understand $\mathcal{J}$'s initial design proposal. Moreover, he does not even *recognize* that he does not understand it: There is no Internal Knowledge Need.

Instead, $\mathcal{C}$ follows a different Need as a Talking Pioneer. $\mathcal{J}$ starts to perceive an External Knowledge Need as he begins to understand $\mathcal{C}$'s misconception, and eventually starts to Push.

**Discussion**
The Talking Pioneering Episode of $\mathcal{C}$ was necessary to clear up his misconception: $\mathcal{C}$ himself did not become aware of it until he saw the actual source code which did not fit his expectations; $\mathcal{J}$ needed to see $\mathcal{C}$ struggle – and he let him – before he could understand the misconception of his partner (TALK, RESYNC).

### 4.2.8 *Effortless Push II (RESYNC PUSH)*

**Situation**
After a few minutes with very little progress $\mathcal{C}$ realized that $\mathcal{J}$'s initial idea for how to retrieve additional data from the backend was not expedient. Together they found a better approach, and amended the `Node` class of the second list to include a new field `isActive`. They just started implementing the icon-rendering method.

Yet again, the pair fixed the icon-rendering by overwriting the `getIconPrefix` method with a delegation call to `ObjectNode`. That methods takes two boolean parameters: The already known `isActive` parameter and another parameter called `isMirror`, which indicates whether the underlying business object is mirrored on another server.

**What happens**
While implementing, $\mathcal{C}$ notices that he has no idea what to use for the `isMirror` parameter in the call he just wrote. He stops typing and $\mathcal{J}$ immediately provides him with the domain knowledge needed.

> *$\mathcal{C}$: "<*stops typing*> Ah ..." – $\mathcal{J}$: "There are no `tasks` on mirrors." – $\mathcal{C}$: "Really?" – $\mathcal{J}$: "Yup." – $\mathcal{C}$: "Right. <*types `false`*>"*

**What it means**
Even before $\mathcal{C}$ can formulate a question to articulate his Internal Need, $\mathcal{J}$ already has perceived an External Need and starts to Push.

**Discussion**
If the pair constantly invests in staying closely together throughout the session, such short Episodes are normal. The whole scene lasted about five seconds.

### 4.2.9 *Investing in Common Ground II (TALK RESYNC PUSH)*

**Situation**
The frontend part of the second list was already prepared to receive the entries' activeness information from the back-

end. Since that boolean value is not stored in the database as a distinct field, it needs to be computed by the backend logic. The pair already found the responsible Objective-C file which receives the frontend command, issues the database query, and prepares the result to be sent back. They also found a static method that performs the necessary time and date comparison.

The underlying database table offers the fields `validFrom` and `validUntil`, which are both already fetched and stored as local variables in the Objective-C code. The static method for date and time comparison is called `CMContent.isActive`: It takes the two parameters called `validUntil` and `from`, and returns `true` if the latter is chronologically before the first.

In addition to the static `isActive` method, there is also an instance method with the same name, which takes no parameters and internally calls the static method with `[self validUntil]` and the current timestamp "`now`", i.e. disregarding the instance's `validFrom` field. In other words: `CMContent` objects are active as long as the `validUntil` field is not in the past.

In the excerpt below, "`from`" refers to the static method's parameter name, and "`validFrom`" to the field fetched from the database.

**What happens**
As they read up on the methods implementations, $\mathcal{C}$ is faster than $\mathcal{J}$ and formulates a design proposal: He wants to reuse the static method and mimic the behavior of the instance method. $\mathcal{J}$ does not react. $\mathcal{C}$ therefore explains the logic behind it; $\mathcal{J}$ understands.

> $\mathcal{C}$: *"Ah, we need `from`, too. <*reads on*> But `from` is always `now`. So we don't need to fetch `validFrom`."* – $\mathcal{J}$: *"..."* – $\mathcal{C}$: *"You know? Because, if it's in the future, it's already valid ..."* – $\mathcal{J}$: *"That's stupid, but ..."* – $\mathcal{C}$: *"It's already `active`."* – $\mathcal{J}$: *"Really? Is it?"* – $\mathcal{C}$: *"It basically means that it's not deactivated."* – $\mathcal{J}$: *"Ah, ok."*

**What it means**
$\mathcal{C}$'s utterances did not suffice to fully engage $\mathcal{J}$ in the PRODUCTION of the TARGET CONTENT, qualifying this EPISODE as TALKING PIONEERING rather than CO-PRODUCTION. However, as soon as $\mathcal{C}$ noticed his partner was not as fast, he started to PUSH.

**Discussion**
The Objective-C backend is $\mathcal{C}$'s home ground, and this is only one example of him being more dexterous. But again, if the pair invests in staying closely together throughout the session, complicated issues can be discussed with minimum effort. That scene was a little more than 20 seconds.

In general, investing in common ground is a fundamentally important behavior in pair programming sessions and is hence frequent at least with skilled pairs.

### 4.3 Additional remarks on NOTEACHER & INFERIOR

A typical reaction of pair programming critics on the scenario of the above session – two sub-systems, one expert for each – would be to suggest the pair should clarify what needs to be changed in each sub-subsystem and split up as soon as possible to work separately. This reflects the aforementioned tacit assumptions that those experts (1) will not learn anything relevant and (2) the knowledge-wise inferior pair member will not be of much help anyway.

Regarding the second assumption, $\mathcal{C}$ may not have looked like much of a help while the pair was still working on the frontend, but this impression would be misled:

- He forced $\mathcal{J}$ to think about things that $\mathcal{J}$ would have ignored (Sections 4.2.1 and 4.2.4 where $\mathcal{J}$ just said *"Who knows ..."* and *"I don't want to [know now]"*), and which actually could have been problematic for their session.
- Many of the good design decisions along the way (which were not explicitly in the scope of this article) came from $\mathcal{C}$: It started with the design proposal without which $\mathcal{J}$ would not have worked on this task until the product management made a decision (see Section 4.2.1).

  When they were waiting for their frontend changes to be compiled and deployed, he made another frontend-related design proposal that $\mathcal{J}$ considered *"a great idea"* and jotted down immediately.

  Along the way, $\mathcal{C}$ also noticed and fixed several minor code smells, such as repeated Strings, which $\mathcal{J}$ probably already had a blinkered attitude to, but nevertheless found worthy to be dealt with.
- Finally, $\mathcal{C}$ solved the conundrum in Section 4.2.6.

And even though it is not explicitly shown in the excerpts above, $\mathcal{J}$ did do similar things while they were working on the backend.

Regarding the first assumption, we cannot quantify the effect of the transferred knowledge. It is difficult to pinpoint even the *in-session* effects of successful or bumpy knowledge transfer episodes. We cannot see how $\mathcal{C}$'s new frontend knowledge influenced his work and the work of his team *beyond* this session (in this sense, the "pair" is indeed a "problematic unit of analysis" [13]). But there certainly are effects, such as the developers becoming more versatile.

## 5. LIMITATIONS AND FURTHER WORK

We will discuss the limitations of our work along with its possible extensions.

- Our analysis so far involved only German software developers. The pair's language and culture certainly plays a role in what does and what does not happen in their programming sessions.

  A small idiosyncrasy was already remarked above: German software developers who use English identifiers in their source code can use both natural languages to easily distinguish (intentionally) blurry real-world concepts from sharp technological concepts (e.g. by saying something like *"Es ist aktiv, aber nicht `active`"* meaning *"It's active, but not `active`"*).

- We can make only very rough statements about the frequency of the described phenomena, because so far we are concerned with the existence of phenomena only. A qualitative-quantitative approach could later be useful to gain more objective characteristics of session types, pair constellations, or pair programming skill level. Also, our catalog of conceptualized phenomena is certainly incomplete.

- The goal of our research is to provide practical guidance to pair programmers. Some core elements of this were presented in this article, yet the didactics, i.e.

how to *teach* it to developers who do not behave this way yet, is unclear.

Due to the strict application of the Grounded Theory Methodology, especially through the practice of *constant comparison* (see Section 3), the concepts we report ought to be fully consistent and reasonably adequate.

# 6. CONCLUSIONS

We have shown that pair programming is *not* either productive or a knowledge transfer technique, but both at once. We qualitatively analyzed several in-vivo pair programming session recordings of professional software developers. We illustrated our findings using one full-length session of a productive expert-expert pair with many knowledge transfer episodes.

The main take-aways are:

- Pair programming is a practice that is useful in more cases than one might have thought. Even if you have a highly skilled developer who could tackle tasks on her own, collaborating closely with another expert can be beneficial.
- Knowledge transfer is an important element even of pair programming sessions with two expert participants.
- Do not focus on wall-clock time alone. Even though they are hard to pinpoint, arguably, the effects of successful knowledge transfer reach far beyond a single session.
- Even in expert-expert constellations, not every knowledge transfer episode goes frictionless. There is a clear potential for training software developers to become better pair programmers and we are working on a curriculum.

## Acknowledgments

This work was supported by a DFG grant and a PhD scholarship of the German Academic Exchange Service (DAAD). We thank our pairs for allowing us to record and scrutinize their sessions.

# 7. REFERENCES

[1] B. Bahrami, K. Olsen, P. E. Latham, A. Roepstorff, G. Rees, and C. D. Frith. Optimally interacting minds. *Science*, 329(5995):1081–1085, 2010.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[3] J. Chong and T. Hurlbutt. The social dynamics of pair programming. In *Proc. 29th Int'l. Conf. on Software Engineering*, ICSE '07, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *Extreme programming examined*, pages 223–243. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, 2009.

[6] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception, and persistence. In *Proc. 25th Int'l. Conf. on Software Engineering*, ICSE '03, pages 602–607, Washington, DC, USA, 2003. IEEE Computer Society.

[7] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10(2):151–177, 1986.

[8] R. L. Moreland, M. A. Hogg, and S. C. Hains. Back to the future: Social psychological research on groups. *J. of Experimental Social Psychology*, 30(6):527–555, 1994.

[9] T. Okada and H. A. Simon. Collaborative discovery in a scientific domain. *Cognitive Science*, 21(2):109–146, 1997.

[10] D. W. Palmieri. Knowledge management through pair programming. Master's thesis, North Carolina State University, 2002.

[11] L. Plonka, H. Sharp, J. van der Linden, and Y. Dittrich. Knowledge transfer in pair programming: An in-depth analysis. *Int'l. J. of Human-Computer Studies*, 73:66–78, 2015.

[12] S. Salinger, F. Zieris, and L. Prechelt. Liberating pair programming research from the oppressive driver/observer regime. In *Proc. 2013 Int'l. Conf. on Software Engineering*, ICSE '13, pages 1201–1204, Piscataway, NJ, USA, 2013. IEEE Press.

[13] D. Socha and K. Sutanto. The "pair" as a problematic unit of analysis for pair programming. In *Proc. Eighth Int'l. Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '15, pages 64–70, Piscataway, NJ, USA, 2015. IEEE Press.

[14] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE, 1990.

[15] K. D. Williams. Dyads can be groups (and often are). *Small Group Research*, 41(2):268–274, 2010.

[16] F. Zieris and L. Prechelt. On knowledge transfer skill in pair programming. In *Proc. 8th ACM/IEEE Int'l. Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 11:1–11:10, New York, NY, USA, 2014. ACM.