On Knowledge Transfer Skill in 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

Background: General knowledge transfer is often considered a valuable effect or side-effect of pair programming, but even more important is its role for the success of the pair programming session itself: The partners often need to explain an idea to carry the process forward. *Objective:* Understand the mechanisms at work when knowledge is transferred during a pair programming session; provide practical advice for constructive behavior. *Method:* Qualitative data analysis of recordings of actual industrial pair programming sessions. *Results:* Some pairs are much more efficient in their knowledge transfer than others. These pairs manage to (1) not attempt to explain multiple things at once, (2) not lose sight of a topic, (3) clarify difficult points in stages. *Conclusion:* Pair programming requires skill beyond software development skill. To be able to identify knowledge needs and then push such knowledge to or pull it from the partner successfully is one aspect of such skill. We characterize a number of its elements.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Human Factors

Keywords

agile software development, collaboration, pair programming

1. INTRODUCTION

Kent Beck defines pair programming (PP) as follows *"Write all production programs with two people sitting at one machine. [. . .] Pair programming is a dialog between two people simultaneously programming (and analyzing and designing and testing) and trying to program better."* [2, p.26]. He also notes that *"Pair programming [...] [is] a subtle skill"* [1, p.100]. We agree with this view [11, 12] and this article will analyze one key ingredient of that skill.

Pair programming can be useful to reduce elapsed time, to reduce defect density, to improve program design, to make sure more than one person is familiar with each part of the code, to increase the amount of knowledge available when solving a task, to increase focus and keep up discipline, to accelerate learning, and to build within-team trust, among other things, see for instance [6, 7, 15]. However, it requires two developers rather than just one and therefore unsurprisingly is a controversial technique.

A lot of small-scale empirical evaluation of pair programming, often in the form of controlled experiments, has produced remarkably unstable results [6]. We conjecture that one reason for the high results variance lies in ignoring pair programming skill (as opposed to technical software development skill) as a factor; we consequently pursue a research program for analyzing the actual pair programming process and the skill involved in it. Its long-term goal is to understand what constitutes that skill and to make it teachable and learnable through behavioral patterns (much like design becomes learnable by design patterns). Our research uses qualitative methods to conceptualize the course of a pair programming session and describe and evaluate the underlying mechanisms.

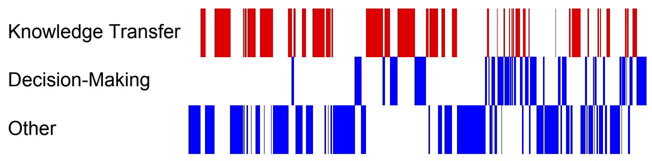

Figure 1: Pair activity type distribution plot for session KA1 of gross length 01:59:44; time runs from left to right. Knowledge transfer covers 35 percent of the time.

When reviewing pair programming sessions with a focus on what goes on in the verbal interaction of the pair members, two modes of work quickly become apparent as dominant: explaining (knowledge transfer) and decision-making. Figure 1 shows an example of how much these occur during one particular session. We picked knowledge transfer as the topic of the present research. Knowledge transfer serves three different roles: It can be the main purpose of a whole pair programming session, for instance to bring a new team member up to speed quickly. It can be a valuable side-effect of a session by spreading relevant knowledge about requirements, technology, existing code, etc. within a team without separate activity. It is always an unavoidable ingredient of a pair programming session, because the partners constantly explain their thoughts and ideas to each other. We ask the following research question:

> What mechanisms underlie knowledge transfer during pair programming and which of these work well or not so well?

We will now discuss some related work (Section 2), describe our method (Section 3), and then present the conceptualization of real-life examples of pair programming skill (Section 4) and derive practical advice from it (Section 5). We discuss the limitations of our work (Section 6) before we conclude (Section 7).

2. RELATED WORK

A survey by Begel and Nagappan confirms that pair programmers are aware of a strong knowledge transfer component inherent in pair programming; they rank it fourth on the list of pair programming benefits [3]. In contrast, the classical description of the driver and observer roles [16] suggests that knowledge transfer will usually only involve the observer telling the driver about her findings (because the partners each have their own and separate role responsibility) which would hardly be useful beyond the session itself.

Sillito et al. study pairs (and single programmers) with respect to the *questions* they seek answers for and found 44 types from four categories [13]. Their aim, however, is informing the design of programmer support tools and therefore the work does not analyze verbal knowledge transfer from one pair member to the other.

Plonka [9] studies industrial software developers and focuses on the use of pair programming as a means of transferring knowledge in expert-novice constellations. She describes six behaviors of experts to guide novices, such as making suggestions instead of telling what to do, or gradually adding information until the novice is able to solve the task. Her study focuses on a distinct type of pair constellation with a one-dimensional knowledge disbalance in sessions for which knowledge transfer is the main goal. Both of these limitations are lifted in our work.

More closely related to our work is that on distributed cognition of Flor and Hutchins [4, 5]. Like us, they are interested in the creation of common ground.[1] They observe two professionals performing a somewhat realistic program change task, focus on one knowledge transfer of about 20 minutes length which they report in great detail, and report as their main results four core knowledge representations used in the transfer process: of subtasks, of program structure, of program behavior, and of required program modifications. In comparison, our work uses more data and aims at findings that pertain to processes (not representations) and that allow for deriving practical engineering advice.

Knowledge and knowledge transfer are universally relevant topics; therefore the amount of *loosely* related work is almost limitless – one could go back as far as Platon's Allegory of the Cave [8]. In particular, various results from cognitive psychology, social psychology, and linguistics are somewhat relevant, e.g. from problem-solving research, conflict resolution research, or research on verbal interaction. The areas are too broad to cover them here.

[1]The definition of 'common ground' in that work includes only aspects that we consider 'knowledge'.

3. METHOD

Our analysis is based solely on recordings of pair programming sessions of professional software developers working on real tasks in their original office environment. A few sessions reflect distributed pair programming rather than local pair programming.

3.1 Data and data collection method

All pairs have volunteered to be recorded for research purposes in exchange for a reflection session we did with them the day after the recording after a quick analysis. Most of their companies used pairing only for difficult tasks. Most of the tasks involved extensions or modifications of a large existing code base. The sessions reflect normal daily practice. All pair members are experienced software engineers, speak German and work for German companies.

We recorded their screen with Camtasia Studio along with a webcam view from atop the monitor and audio. The small webcam video is laid over the desktop video in the bottom right corner.

In this manner, we have collected 43 sessions of 28 different pairs involving 42 persons from 9 different companies; the sessions have a typical length of one to three hours. From these, we selected for the analysis presented here 4 sessions of 4 pairs from 4 companies that appeared to reflect a particularly broad variety of knowledge transfer behaviors and levels of code knowledge (high-high, high-low, low-low pairs); see the discussion in Section 6.2. Session length ranges from 1:10 hours to 2:26 hours. The members of all but one of these pairs knew each other well. Two of the pairs had not paired before (which increases the breadth of behaviors observed). One pair performed distributed pair programming (DPP), all others local pair programming.

A session recording name such as CA2 means company C, development project A, second recording within this project. The four sessions we will use are:

Session CA2 (involves persons C2 and C5, length 1:23 hours): Two experienced (though not senior) developers from the geo-information system domain continue working on a task started by C5. The work consists of a prolonged design-discussion and a medium-sized refactoring in the first half of the session and of implementation and testing of a new feature in the second half.

Session DA2 (persons D3 and D4, 2:26 hours): One novice developer (D3) and one junior developer (D4) work on a large CRM system and try to implement a new toolbar. After two long discussions with two additional developers, they begin a refactoring task which lasts the whole session. Throughout the session, D4 provides D3 with information on programming styles, technologies, and so on, whereas D3 is more knowledgeable about the code base and the organizational background.

Session JA1 (persons J1 and J2, 1:10 hours): The session between a junior developer (J2) and an experienced external consultant (J1) was intended to perform a large refactoring, but turned into mostly an explanation of the legacy source code. During the session, the pair refactors small portions of the code and eventually decides to rewrite the whole module (which then they do in session JA2 to JA9). J2 has strong domain knowledge and intermediate programming skills; J1 has hardly any domain knowledge and very good programming skills. This distribution leads to frequent knowledge transfers in both directions.

Session KA1 (persons K1 and K2, 2:00 hours): Two junior developers from different teams start working on a new API for aggregating several data sources (K2's team's task) which eventually should be used by a mobile app (K1's team's project). The first half of the session is consumed by bringing their local Tomcat server and the existing applications into a runnable state; the second half contains requirements collection for the API and implementing the first executable version. The session involves reading a lot of somewhat-known source code and existing API specifications, and generates a lot of fresh common ground between the two.

In all four sessions, the used programming language was Java. In this article, we will present verbatim quotes from CA2, JA1, and KA1.

3.2 Data analysis method

Our data analysis uses Grounded Theory Methodology (GTM, [14]) and aims at a limited (and so far isolated) Grounded Theory (GT) of knowledge transfer in pair programming such that its main concepts directly inform useful behavioral patterns. We use open coding [14, Section II.5] (see also [10]), axial coding [14, II.7], but not yet selective coding [14, II.8]. We will present only the resulting conceptualization, not details from within the analysis process. The early phases of the analysis were accelerated by using (but always in fully grounded fashion) the pair programming base concepts identified in a previous stream of work [11], but the resulting concepts presented here are all new.

Our theoretical sensitivity [14, I.3] is oriented as described by the research question and primed by the pair programming base concepts. We apply theoretical sampling ([14, II.11]) once at the beginning of the research (based on our knowledge of the recorded sessions), but not iteratively. Therefore, our work does not provide complete theoretical saturation [14, II.11].

Note that the notion of skill in our articles title refers to actually exhibited skill, i.e., performance, not the underlying competence.

3.3 Notation

When we introduce the resulting concepts, we typeset their names in small caps and discriminate three levels of elaboratedness as follows. "V" (SOME CONCEPT^{V}) ("vague") represents informal concepts that appeal to intuition and for which there is hardly more description than their name. "S" (SOME CONCEPT^{S}) marks semi-complete concepts for which a concrete definition is available but where we expect that definition to be incomplete and/or unstable (from the point of view of more detailed further research on the topic). "C" (SOME CONCEPT^{C}) marks completely elaborated concepts that we consider stable. Where we subsequently use such a term, we set it in normal font (Some Concept) to avoid cluttering the text and also sometimes take the liberty to inflect it.

In the verbatim quotes, we indicate program identifiers, pauses, comments, and replacements as follows: *"Huh? `RemoveTargets`? (...). No idea. <*sighs*> We could ask <**chief architect**>."*. For pauses, each dot indicates about one second of silence, so the above was a three-second pause.

4. RESULTS

4.1 What is knowledge?

As mentioned above, two modes of dialog appear dominant in pair programming: explaining (knowledge transfer) and decision-making. For this work, we ignore decision-making, but want to understand knowledge transfer precisely. Hence, we need a definition of "knowledge".

Fortunately, we need not determine this from scratch: The pair programming base concepts [11] describe the basic activities that together form the pair programming process. They can serve as a foundation for deriving our definition; most of them conceptualize utterances. There are 60 such concepts, grouped into 12 concept classes. Any class consists of several concepts that each describe a different type of utterance such as *propose_step* (proposing what to do next), *agree_step* (saying yes to such a proposal), *challenge_step* (saying no by making a counter-proposal), and so on.

Of the 12 concept classes, 5 pertain to different types of knowledge: *finding* and *hypothesis* relate to fresh knowledge ("insight") that is deemed reliable or uncertain, respectively; *standard of knowledge* and *gap in knowledge* relate to meta-level discussion about what knowledge is available or not available, and finally *knowledge* itself relates to pre-existing knowledge brought in from before the session. These concepts together provide a suitable definition of "knowledge" and were a valuable starting point for our analysis.

4.2 Some terminology for knowledge transfer

To think about knowledge *transfer* in pair programming, we introduced three additional knowledge concepts: First, the need for knowledge. The need is difficult to observe directly and will hence not even be given a formal concept name. Second, what the knowledge transfer is concerned with. We will call this the TOPIC^{C}. Finally, the information that is able to fulfill the need. We will call this the TARGET CONTENT^{C}. The Topic needs to be recognized at least vaguely before the knowledge transfer can start. A Target Content can be an insight (and would then be talked about with *finding* utterances), an uncertain assumption (*hypothesis* utterances), or a piece of existing knowledge (*knowledge* utterances).

4.3 Modes of knowledge transfer

Any single knowledge transfer episode is almost always driven forward by only one pair member. This pair member has an idea of the Topic (appropriate or not) and pursues its clarification. We call this person the PROPELLOR^{S}. The Propellor can be the person in need of the Target Content, the CUSTOMER^{C}; we say such episodes run in PULL^{C} MODE^{C}. Or the Propellor is the person possessing the Target Content, the SUPPLIER^{C}; we say such episodes run in PUSH^{C} Mode.

Occasionally, nobody is in possession of the Target Content and the pair works to create it; we say such episodes run in PRODUCE^{S} Mode. There are two sub-Modes: Sometimes the pair collaborates closely and discusses the state of their knowledge, new hypotheses, observations, procedural suggestions, etc., intensely and continuously (CO-PRODUCTION^{S}). In other cases, e.g. if only one member fully understands the Topic, only this member performs the investigation, normally doing think-aloud to keep the partner informed, and the partner mostly signals understanding or lack thereof and only sometimes contributes a minor insight or thought (PIONEERING PRODUCTION^{S}). The latter sub-mode appears to be more frequent: The active member has an idea that is ahead of the partner, but lacks the knowledge for checking or elaborating it; rather than explaining this state of affairs, which would be difficult, she performs the evaluation alone so that the subsequent explanation becomes a lot easier.

4.4 The Episode structure

A KNOWLEDGE TRANSFER EPISODE^{C} (or EPISODE^{C} for short) is defined by a single Topic pursued in a constant Mode. If a second Topic appears and is followed, we consider this a separate Episode. If the Mode changes, we consider this a separate Episode as well. The latter is common when a Pull attempt fails and the Customer initiates Produce Mode, e.g. by starting to read source code after the partner was not able to answer certain questions.

This narrow definition of Episode is useful for understanding better the concepts we will present here and we will use it frequently. We will also sometimes talk about a larger piece of a session we will call SCENE^{V}. A Scene includes multiple Episodes or other phenomena of interest.

Episodes need not be contiguous, which is why more than one Episode may be incomplete at a certain time: An auxiliary or subordinated Episode may be sandwiched and, rarely, each pair member may pursue her own different Topic concurrently.

An Episode does not end just because of an utterance (or other event) that does not belong to it. An Episode ends only as soon as the Propellor no longer pursues its Topic: Obtaining or transferring the Target Content is no longer a goal – the researcher will usually have to analyze subsequent conversation to decide this. So far, we observed four possible reasons for no longer pursuing a Topic: The Propellor considers the Target Content to be transferred, the Propellor recognizes the transfer to be unnecessary, the Propellor gives up the attempt, or the Propellor loses sight of the goal.

4.5 Attributes of knowledge transfer utterances

To make sense of Knowledge Transfer Episodes, we first introduced a number of attributes with which to characterize individual utterances, mostly of the Propellor. Only one of them will be used in our subsequent analysis but we present several here to give the reader a more colorful idea of knowledge transfer phenomena and some additional insight into our research process.

INFORMATION TYPE^{S} has about a dozen different values such as design decision, characteristic of program artifact, technology fact, relationship to other tasks, and so on. The Information Type of an individual utterance is often closely related to the Topic of the overall Episode. If it is constant throughout several utterances, it typically characterizes the Topic. If it changes, this may reflect an attempt to explain or ask better (because the previous approach did not appear to work well), or it can be a sign of confusion or of an attempt to clarify several things at once.

SCOPE CHANGE^{S} describes whether the part of the world being discussed has been kept the same relative to the previous utterance or has been made smaller or larger. This concept led to the discovery of the Explanation Trigger Types; see below.

MEDIUM^{S} of a transfer can be pure verbalization, verbalization plus demonstration (such as a test run or a code walkthrough), or "by typing" (*"What I mean is this."*). The medium may change several times during an Episode.

ASSESSMENT^{S}s relate to a previous utterance and evaluate an explicit proposition or implicit assumption contained in it or implied by it, typically as part of a "question". Many such "questions" are not, in fact, questions – another observation that paved the way for the discovery of the Explanation Trigger Types.

IS TERMINATION ATTEMPT^{S} is one of a set of boolean attributes and characterizes whether the utterance aims at terminating the Episode. So far, we only observed the Propellor doing so because she felt the Episode was successful. However, other combinations (e.g. non-Propeller/giving up) are plausible as well.

IS HASTED REPLY^{S} means an explanation starts before the respective question was complete or the next question starts before the previous explanation was complete. It can speed up the process in case of needlessly detailed questions or explanations, but can also get the pair into trouble because the understanding of the hasty speaker may in fact be wrong.

REDUNDANTIZES^{S} means an utterance mostly or fully repeats information that was transferred before, but in a different formulation or with an additional question-and-answer pair in between. This attribute applies to questions as well as explanations. The former pointed to one particular Explanation Trigger Type: State Known Fact.

IS UNCERTAIN^{S} means the wording or intonation make it clear that the speaker is not fully sure of the correctness of her statements. This applies to explanations as well as to feedback on explanations (e.g. a hesitant *"Okaaaaay"*).

The final attribute we call EXPLANATION TRIGGER TYPE^{C}. It was the most productive of them all for our research and will be at the heart of the next subsection.

4.6 The Clarification Cascade

The following situation is common in pair programming knowledge Pull situations: A pair member recognizes a knowledge need (Topic) in herself that is too complicated to be fulfilled by only a single question. This TOPIC COMPLEXITY^{S} may either be recognized from the start or from an answer to a first, naive question where the answer is incomprehensible or reflects a misunderstanding.

What typically happens then is that the Customer devises a series of utterances-aiming-at-eliciting-explanations (EXPLANATION TRIGGERS^{C}) that serve to (1) check one-by-one the assumptions or hypotheses she has come up with so far and (2) guide the Supplier through a series of steps that serve to make the Supplier (a) understand the Topic and (b) generate answers in digestible bites. This series of utterances is an Episode. We call this type of Episode CLARIFICATION CASCADE^{C}.

4.6.1 Explanation Trigger Types

A Clarification Cascade is characterized by a typical sequence of utterance types; not each type occurs in every Clarification Cascade and the same type may occur multiple times in a row. What gives the Clarification Cascade its name is not the clarification of Target Content, it is the successive narrowing-down of a Topic. The need for further narrowing down may be on the Customer's side as well as the Supplier's side. These EXPLANATION TRIGGER TYPES^{C}

(or TYPES^C for short) are the following:

• FINDING^S: Form: Proto-questions, often in the form of thinking aloud, reading identifiers aloud, paraphrasing a snippet of code, or signaling and somewhat locating confusion.
Example 1: J1: *"So we go in there with the `currentTime`"*, which paraphrases a method signature and triggered J2 to elaborate on the parameter's meaning.
Example 2: J1: *"And that means what?"*, which locates confusion at "that".
Role: Locating a general area of interest when the speaker does not yet understand her own knowledge need well enough to *formulate* the Topic.

• DIRECT QUESTION^C: Form: Asking a question formulated as such, usually as an open question, only sometimes as a yes/no question.
Example 1: K1: *"What is that for, I mean, what does it give me?"*, where "it" is a member variable called `type`.
Example 2: D4: *"But can you attach, er, a context menu to it? You know, an SWT context menu or so?"*
Direct Questions are the prototypical entry point into a Clarification Cascade, but all Types can fill that role. In contrast to utterances of Type Finding (but like all other Types), isolated and multiple subsequent Direct Questions are both common.

• STATING KNOWN FACTS^C: Form: The speaker repeats something that was already stated earlier in the session in the same or similar form.
Context is required to give an example, so please see the long example of a complete Clarification Cascade below.
Role: This is a means for narrowing down the area of the partner's attention. It signals a Topic area of interest (that should be explained) and a subarea that is already understood (and thus needs no further explanation).

• SIMPLE STEP^C: Form and role: Like Stating Known Facts, Simple Steps refer to and use reliable common ground. The form is typically that of an assertion. The speaker of a Simple Step has a clear idea what she wants to know and the utterance is intended to lead the thinking of the partner towards a *particular* spot. To do this, it makes a statement that entices the partner to make a particular mental step – in the direction favored by the speaker.
We use paraphrased examples here because verbatim ones would require too much context.
Example 1: J1: *"A connection between these has to be created somewhere."* This speaker aims to find out why "these" are connected or how or what program part is responsible for it. The first part of the utterance is Stating a Known Fact. It is the last word "somewhere" that turns the utterance into a Simple Step.
Example 2: J1: *"In this block, the variable is always not null."* The word "always" is making the Step here. The speaker wants to investigate why there are statements for handling the case in which the variable *is* null.
Simple Steps may use irony.

• PROPOSITION^S: Form: The speaker states a proposition with the expectation that the partner will either accept or refuse it; a usually implicit and sometimes explicit yes/no question.
Example 1: J1: *"That means you* could *get it in each case simply by calling `getLastFile` again?"* This aims at validating an assumption about the semantics of the `getLastFile` method by posing a different manner of using it.
Example 2: J1: *"That which overwrites it (.) that is really, well, if you take any two files and compare them, then they are somehow always different."* The part up to the "well" is a failed first formulation attempt.
Role: Reducing the partner's possibilities for giving irrelevant information to zero.

4.6.2 The escalation of difficulty

Propositions are the Propellor's ultimate tool for obtaining relevant information (as opposed to just some information), but are difficult to devise; this is why they constitute the last level of the Clarification Cascade. The previous levels are easier: Finding requires only a vague association of the type "Hmm, this might be relevant somehow". Direct Question requires a first attempt at formulating this "this" and make palpable the speaker's interest in it. Stating Known Facts in addition requires to locate the area of this interest more narrowly. Simple Step is much more difficult because it requires at least a rough understanding of the difference between the speaker's and the partner's current line of thinking. Proposition is still more difficult in that it requires forming a specific hypothesis – a construction task.

4.6.3 The different effects of the Types

Each of the Types may trigger the partner to provide all of the explanation sought (the PERFECT ANSWER^S) but will usually result in much less (the ACTUAL ANSWER^S, which may also be the EXPECTED ANSWER^S), so that the cascade needs to continue. In this respect, Stating Known Facts and Simple Step are fascinating because the Expected Answer is always "yes" and has no information value. These two Explanation Trigger Types are clever intellectual tools, yet good pair programmers appear to use them intuitively.

4.6.4 An example Clarification Cascade

Few cascades ever comprise all of the Types at once. Here is a Scene containing a complete Clarification Cascade from session JA1 (time range 0:04:10–0:06:15) that has all Types except Finding:
J2: *"For now I could tell you what this plugin does overall."*
J1: *"Yep."*
J2 now starts a longer Push Episode that ends with *"It starts checking how the file its size still changes. That is, it looks until the file does not get bigger anymore, then it is apparently ready. And then it is fetched and handed over to processing."*
At his point, J1 has a question which J2 misunderstands and this starts a Clarification Cascade: J1: *"In what time window are you looking?"* (Direct Question)
J2: *"I start looking two minutes after the full hour, because then it's guaranteed that news files exist* if *any exist."*
J1: *"OK."*
J2: *"And monitor this file as long as needed until it's ready. That can take up to seven minutes, depending on the source."*
J1: *"Hm ya but mh the time window for the* change*?"* (another Direct Question)
J2: *"Yes, right, that is, er, time window for the change is variable, depends on how the news go. I can't know that. It is so they always start a new file. When the news are over again a file is created. This means, I never actually have more than the news."*
J1: *"Yes, no, I mean 'cause you said you look for so long, er, until the size stops changing, right?"* (Stating Known

Facts) *"Then you need to plan for a time window in which a change* could *happen."* (Simple Step)
J2: *"Yeah, well, until up to five before the hour. I really take my time."*
J1: *"<*laughs*> No I really mean the size now, the size of the time window, I mean (.) you wait for 10 seconds, then after 10 seconds you decide: In those 10 seconds nothing has changed, so the file appears to be ready."* (Proposition)
J2: *"Ahhh, that's what you mean. No, 30 seconds."*
J1: *"30 seconds, that's what I wanted."*
J2: *"That's 30 seconds long the time window. Now I got you."*
One can almost hear the relief that the difficult Clarification Episode was successful eventually. Note that in terms of our conceptualization, the main work result of the clarification process is *not* providing J1 with the value "30 seconds", it is providing J2 with J1's intended meaning of the word "time window".

4.6.5 Interpretation and context conditions

The clarification cascade is effectively an iterative question design technique (conscious or unconscious) for complicated situations. Its result is a kind of modularization-plus-unit-testing structure for the Pull-Propellor's knowledge acquisition process with the effect that the partner can catch mistakes or false assumptions in the speaker's reasoning, e.g. by rejecting a Simple Step.

Frequent occurrence of long cascades suggests that the pair's thinking may still lack common ground, because if it had enough, the partner should usually be able to understand a knowledge need more quickly. Frequent long cascades may thus indicate there is an improvement potential for the pair.

Long cascades may also happen frequently (and are not problematic then) if an experienced Customer lacks background knowledge (on the software product and/or its domain) but suspects a problem in the code: Such a suspicion often involves a number of assumptions that the asker wants to check one-by-one to avoid asking a question that is not understandable. Sometimes, the Customer is in fact a pseudo-Customer and works through a Clarification Cascade although she is fully sure there is a problem in the source code and what it is; in this case cascadic querying serves to save the partner's face.

We conjecture that being aware of the nature of Clarification Cascades and the individual Explanation Trigger Types will automatically improve engineers' knowledge transfer skills somewhat. Shortly reflecting (in mid-flight) on Clarification Cascades that were longer than ought to have been necessary (such as the example above) is also likely to help. We will formulate an explicit procedure for employing Clarification Cascade behavior in Section 5; likewise for behaviors arising from the remaining results subsections.

4.6.6 Clarification in Push Episodes?

Push Episodes exhibit a related phenomenon that also appears to result in a sort of modularization and involves alternating between Stating Known Facts and STATING NEW FACTS^{S}. However, we have not yet analyzed it sufficiently to fully present it here.

4.7 How to handle multiple Topics

For knowledge transfer to be successful, it appears to be necessary that the Propellor has a fairly clear understanding of the Topic, and, even if multiple things are to be clarified, only chooses one Topic at a time. If, in contrast, a Propellor tries to Push or Pull multiple things at once, confusion will usually result.

We provide one positive and one problematic example. In the positive case, a developer recognizes the difficulty of his Push plan (multiple Sub-Topics) and actively thinks about a helpful sequence.

4.7.1 Positive Example

From session JA1 (time range 0:57:30–0:59:40): During a code walkthrough by author J2, the following facts and circumstances are unclear for J1, and J2 needs to explain them: (1) The software loads files from multiple remote systems and then processes them. (2) Not each remote system will have the requested data each time it is queried. In that case, it asks a superordinate fallback system which will always provide a single set of ersatz data files which are then tagged as such and returned. (3) Therefore, the queries to several systems may return the same result in fact coming from the fallback. (4) Downloading these data is slow. (5) Therefore, files coming from the fallback system are cached to avoid downloading them multiple times. (6) Other files are deleted immediately, not cached. (7) When a file is deleted, a pointer to the file needs to be deleted in the program.

J1 already understands point (1), but needs to understand all seven, so J2 has to explain the whole situation. This explanation proceeds as follows (respective fact numbers as <*comment*>):
J1: *"Now we have `return response` here and `return response` there. That's..."*
J2: *"Yes, that is, problem is, that down there is, because this local file needs to be reset <*(7)*>"* (he points "down there" to `this.localFile = null;`) *"(..) that is (...) 'cause the thing is this: for <**machine1**> and <**machine2**> it is so that we want the local file deleted after processing <*(6)*>. Yes you see that here too when you look at the arguments of `processFile`. The second one is a `boolean` – ain't pretty but works – `deleteSourceFile`"*
J1: *"Unghhhh m-hm"* (a long guttural sound plus an affirmation)
J2: *"And now I just see... (..) No, that's correct so, 'cause we have <*sigh*> where should I start explaining this (...) er (.) yes. It's more complicated than you'd think."*
J1: *"Doubtless."*
J2: *"Because if <**machine2**> or <**machine1**> don't have their own data, if they failed somehow, then they use these <**fallback machine**> files. <*(2)*> Yes, those, and depending if it's their own ones... if that construct back here is `true`, then it is their own files."*
J1: *"Yesyes, I see."*
J2: *"Then it should delete them <*(6)*>. If it's <**fallback machine**> files, it should of course not delete them <*(5)*>, because there may be others who need them too <*(3)*> (...) Yes that, but that could all be done a bit different later."*
J1: *"A different kettle of fish! That is<*interrupted*>"*
J2: *"Yes that is a different issue, 'cause I did that only because the downloads took so long of those stupid files <*(4)*>. So I thought it need not take even longer so I'd use the same one for them all <*(5)*>. Of course if it runs on a local file system that's a whole lot faster and then it doesn't hurt*

to download twice, I guess, and we need not keep it."
J1 appears to understand the construct during the repetition of (6), but J2 goes on nevertheless. □

We do not claim our post-hoc explanation in points (1) to (7) to be the best serialization of this complex Target Content and the one actually created by the Propellor is even worse, but that is not the point. The point is: The Propellor became aware of the complexity of the Target Content and the difficulty of Pushing it; he took some time to think it through, informed his partner about the difficulty, finally found a serialization – and his partner even understood it sooner than expected.

4.7.2 Negative Example

From session CA2 (time range 0:09:30–0:19:40). In contrast to the competent developers above who successfully (if not exactly masterly) juggle 6 items needing clarification, the present example shows a developer struggling with merely two Topics.

Company C makes two variants (`Basis` and `Pro`) of its product. Both rely on a joint library. Outside the library, `Pro` may statically depend on `Basis` but `Basis` must not statically depend on `Pro`. Nevertheless there are framework parts in `Basis` that need to call `Pro` functionality.

C5 starts the session with explaining to C2 the previous work he has done. There are two Topics. First, to explain a key constraint: The goal is moving as little code as possible from `Pro` to `Basis` (C5 had consulted the chief developer with this) while absolutely avoiding static dependencies from `Basis` to `Pro`. Second, explain the new content (and state of this work) of new or modified classes.

But rather than explaining the constraint first, C5 pursues both Topics in parallel as follows (the technical background will be explained below):
C5: *"Then, let us, I'll first show you I guess what I've done?"*
C2: *"Ok."*
C5 starts explaining. C2 interrupts him and lets his dislike of C5's design shine through: C2: *"Okaaaay (...) do we really have a `ColumnAttribute` there? Is that so?"* He leans back with folded arms.
A bit later C2 interrupts C5 and explains a simpler, more direct design (which further complicates the Scene: there are two Propellors now): *"What data structures do we have for the GUI? Is a `ColumnAttribute` in them? If not, I would simply use them as they are."*
But C5 never isolatedly discusses the constraint topic, all respective statements are embedded in statements about his work results. For instance, he opens an interface that has only one method and says *"More, more than this isn't there yet because (...) because, er, it is (..) er, I, more (..) more, I have, we have, we need a `ColumnAttribute` to insert this in this `getAll` (.) when you fetch them all."*
The further explanations make clear (to the researcher who viewed the 10-minute scene a dozen times, took notes, and drew diagrams) that these "more"-statements are part of the work results topic: "I did not yet get farther than this, but we will need such an interface eventually." The "need a `ColumnAttribute`"-statements, however, are part of the constraint topic: `IColumnAttribute` is an interface from the joint library and is the result type of the `Basis` product's `getAllAttributeColumns` method. C5 wants this method to return a `ColumnAttribute` object that he needs for carrying functionality from the `Pro` product into the `Basis` product without introducing a static dependency.
But in the live situation, C2 never gets to the point of seeing this: C2: *"Is that so? Do we really need that in the attributes table for the visualization?"*
C5: *"We need for the visualization in the attributes table, if we want to make with `getAllAttributeColumns()`, an `IColumnAttribute`."*
C2: *"If we want to do it that way, OK."* C2 has now understood the existing design, but still not the rationale behind it.
C5: *"If we want to do it that way, that's right. But that's what I had taken to be our agreement. I haven't put in more yet."* Again, "agreement" is about the constraint, "more" is about work status.
As far as we can see, C2 over and over mis-takes the rationale explanations as justifications for the incompleteness of the results and continues to consider his own unsound "simpler" design a valid idea.
It takes all of ten minutes of Pushing from C5 (work content/state and constraint) alternating with Pushing from C2 (alternative "simpler" design) before C2 eventually asks the pivotal question: *"Is there any reason not to go the easy way?"*.
Only now do the two Topics finally come together in C5's answer: *"Um (..), well, right now there is no reason (.) I am not sure yet, whether the things, so, I did it this way (.) because (.) because we, because I wanted to move as little as possible to `Basis` of these things. And (.) the interface (.) um (.) that I already moved only knows things that are known in `Basis`."* Only now does C2 recognize that the functionality the pair is concerned with resides in the `Pro` product and must not be used directly, and that this fact is the reason for the seemingly circuitous design proposed by C5. □

The problem in this Scene is not with explaining two Target Contents, it is with recognizing there are two relevant Topics: C5 apparently is unaware that C2 is unaware of the imminent `Basis`-`Pro` coupling issue and so only alludes to it rather than stating it explicitly. The fact that C2 starts pursuing his own design idea further complicates the situation, but should in fact have been a trigger to make C5 aware of C2's lack of awareness.

4.7.3 Context conditions

Most Knowledge Transfer Episodes start and continue rapidly and without utterances indicating notably increased mental load (such as the *"where should I start explaining this?"* in the positive example above). It looks like usually only one uncertain aspect of the ongoing session stands out clear enough to be chosen as the Topic of an Episode.

So far, we have observed cases of multiple Topics only in Push Mode, presumably because a Pull-Propeller only needs to be aware of her own knowledge and its boundaries and much more easily manages to focus on one thing at a time.

We conjecture that problematic Scenes will typically arise only when a Propellor is not aware that she needs to explain a second Topic as well and mentions it only on the side, so being aware of this possibility is probably helpful for practitioners.

4.8 On focusing on Topics

Obviously, the Propellor's clear understanding of the Topic at the beginning of a Knowledge Transfer Episode is important. But *sticking* to that Topic is important as well.

4.8.1 Positive Example

From session JA1 (time range 0:06:15–0:06:45). This example continues a Scene which starts with a 50-second Push Episode of J2 regarding what the `NewsPlugin` does overall. This Push Episode is interrupted by a 75-second Pull of J1 regarding how often a certain file size was polled; a rather tiny detail (this one was the core of the long example from Section 4.6). Once that is finished, J1 imperturbably returns to the original Topic (although that had been introduced by J2!). The first three lines are repeated from the long example above.
J2: *"Ahhh, that's what you mean. No, 30 seconds."*
J1: *"30 seconds, that's what I wanted."*
J2: *"That's 30 seconds long the time window. Now I got you."* (end of repetition)
J1: *"And within this whole procedure the `NewsPlugin` does what? Exactly this monitoring and then the delegation to the individual channel plugins, huh?"* (Direct Question followed by Proposition).
J2: *"No, the (.) the `NewsPlugin` is only concerned with, it is called periodically, by the cron server (.), um, it gets triggered and then the `NewsPlugin` starts the respective `Processors`. And they take charge of the monitoring and the delegation then to the transcoding software."* □

Despite what could be considered a circuitous route, J1 never loses sight of J2's Push goal. At other times, however, a pair (even the same pair) may lose sight of the Topic while clarifying something else:

4.8.2 Negative Example

From session JA1 as well (time range 0:14:00-0:16:30). In this Scene, J2 explains a method. J1 asks about the meaning of the possible case `remoteFile == null`, never gets a semantic answer, but does not follow up. The code in question can be paraphrased like this:

```
this.remoteFile = this.getLastFile(this.remoteDir);
if (this.remoteFile != null) {
  this.remoteFileSize = this.remoteFile.getLength();
  return State.FILETRACKING;
}
log.error("No current file found");
return State.ERROR;
```

After J2's line-by-line explanation of this code, J1 wonders about the `null` case (which could mean for instance the file is missing, the directory is unreadable, the connection failed, there is no "last" file, or other things) and asks *"Is this an expected case? Can it happen?"* (Direct Question).
J2 does not hear this question, because this session is distributed pair programming and there is a short gap in the Skype connection. J1 asks it again as *"It should never slide into the `return ERROR` case, right?"* (Proposition)
J2 agrees, J1 is satisfied, but then J2 proceeds to explain *"Well, if it found a file, it does not get there. Because then it is in this `FILETRACKING` mode."*
J1 recognizes that J2 has interpreted his question on a control-flow level rather than a meaning level. He follows up several times, each time stating his question as a Proposition like above until eventually J2 agrees.
But the problem is still not gone: J2 continues by saying *"Of course I could as well throw an exception in `getLastFile`, a `FileNotFoundException`. That would even be nicer now that I think about it."*
J1 agrees, J2 writes a `TODO` into the code, the pair proceeds in the source code – and the original Pull Episode has simply disappeared without a proper ending. □

Should J1's issue be a serious problem, the pair has just lost an improvement opportunity.

4.8.3 Context conditions

Losing sight of a Topic does not appear to be a particularly frequent problem. Most cases we have seen were minor or even subtle. The frequency appears to rise, however, when there are two Propellors each pursuing their own Topic (see the next section), so the pair should make sure to develop well-oiled routines of "You go first"-behavior.

4.9 On determining the Propellor

Clarification becomes harder if each partner pursues a different Topic. On the other hand, having two Topics that are unclear, is quite normal in pair programming. Gelled pairs manage to determine a single Topic and a single Propellor quickly.

4.9.1 Positive Example

From session KA1 (time range 0:50:50-0:51:30). The pair has just finished a work item and committed the code. As for the next step, both pair members have done relevant work before the session and both now begin to speak at once:
K1: *"Shall we <*interrupts himself*>"* || K2: *"Well, you wanted to put the same into the mobile app – Mobile First, right?"* (Mobile First is company K's tactic of trialing new functionality in the mobile application first before publishing it in the web portal.)
As we see, K1 immediately stops himself and lets K2 go first; K2 promptly takes the initiative and starts Pulling what K1 has done before the session in Mobile First regard. K1 immediately and completely gives up his own idea for the next step and follows K2's:
K1: *"Yes, exactly. I even got quite far with it. I have over there built myself such an, a Mock-JSON (.) that would be cool if it roughly <*interrupted*>"*
K2: *"Have you checked it in?"*
K1: *"Uh, no, it's not checked in (.), I didn't want my previous web-app (..) to be trashed."*
K2: *"Ah OK, well I can show you what we have so far and then we can compare."* □

Although K2 interrupts K1 twice, K1 is not at all frustrated or angry. K2's decisive action led to a quick transfer of all relevant knowledge: (1) A data format for a new API that will be required has been thought out (Mock-JSON) by K1; (2) it may be complete but is not immediately ready for integration; (3) some work on the functionality behind the API-to-be has been done by K2; (4) that work is far from complete, and (5) K2 suggests the pair should continue there. The result is fluent session progress.

4.9.2 Negative Example

From session CA2 (time range 0:09:30-0:19:40). This is the same negative example as seen in Section 4.7 (regarding the `Basis-Pro` dependencies). The Scene is not only plagued by the non-recognizing of the second Topic, but also by a constant struggle for Propellorship: In principle, two Knowledge Transfer Episodes with different Propellors could interleave and so proceed at the same time; in practice, this is cognitively way too hard to be sensible.

In this Scene, the original Propellor C5 is polite and for a long time reacts on each of C2's proposals until eventually he attempts stopping C2 by saying: *"Let's carry on for now. It's not tied up yet, what I did (..) did so far."*
This is meant to mean *"Could we please defer your criticism until later?"*.
However, what C2 understands instead is *"OK, let us postpone my line of work and talk about yours now."*
Unfortunately, neither pair member appears to recognize this misunderstanding and the dual Propellorship continues as before.
When C2 finally stops Pushing his own design idea later, it is only to shut himself off of C5's explanations and start a Produce Episode of his own, reading code concentratedly. C2 works in Pioneering Production Mode without explaining what he does, thus shutting out C5 of his progress. C5 gives up his Propellor role and attempts to join C2 for a Co-Produce Episode: When C2's reading reaches an interesting line, C5 provides information about it, but this is apparently ignored by C2. This is hardly pair programming any more. □

5. ADVICE FOR PRACTITIONERS

This section formulates insights derived from the above observations in the form of explicit, procedural, practical advice; quotations refer back to the examples from Section 4.

5.1 For explaining an information need (Customer)

Naturally, the information need of a Customer is only perceived by herself unless she makes it palpable for the Supplier in form of a Topic. To do this, the Customer might use the following pattern:
(1) Signal the existence of an information need through a Finding.
(2) If this does not already make the Supplier deliver the Target Content, formulate a Direct Question that makes the Topic explicit.
(3) If the Supplier misinterprets the Topic, make use of common ground by Stating a Known Fact to provide the Supplier with context information.
(4) If this does not suffice, entice the Supplier to make a Simple Step that follows the line of thought towards the information need.
(5) If the Supplier still does not understand the Topic, formulate a Proposition that she can easily validate. Then, the Target Content is *constructed by* rather then *transferred to* the Customer.

5.2 For handling a complex Topic (Supplier)

When the Supplier becomes aware of the complexity of the Topic:
(1) The Supplier makes this difficulty explicit (like J2 did by uttering *"Where should I start explaining this?"*).
(2) The partner accepts her role as the Customer and allows the Supplier to be the Propellor, i.e. to take her time to finish her thoughts and to set the pace of her explanations.
(3) The Supplier claims the Propellor role, explains the different relevant aspects of the Topic separately, and ensures the Customer understands all of them, i.e. does not allow the Customer to finish the Episode prematurely.

5.3 For managing concurrent Episodes

When the Customer asks for a detail during a Push Episode which in turn starts a Pull Episode:
(1) The Customer ensures that her Pull Topic is actually clarified (e.g. by making it explicit, as J1 did in the positive example by saying *"30 seconds, that's what I wanted to know."*).
(2) Once finished, the Customer then hands over to the former Push-Propeller so she can continue her interrupted Episode.

If a second, somewhat unrelated Knowledge Transfer Episode appears while one is already running, the right thing to do is:
(1) postponing one of them immediately and
(2) resuming it once the other is finished.

In particular the second step will work much more safely if done explicitly. See the positive example from Section 4.8: J1 returns to the previous Topic by Pulling and explicitly mentioning the Topic again in his question.

6. LIMITATIONS AND FURTHER WORK

6.1 Validity

Results of GTM work explain phenomena that have specifically been observed to exist. Threats to the results' validity hence are restricted to inappropriate conceptualization. Many of the concepts we present show up directly in the examples shown (and so can partially even be validated by the reader). The validity of our results is therefore likely high.

6.2 Generalizability and breadth

Results of GTM work explain phenomena that exist; they neither claim to capture all such phenomena nor to quantify their frequency or distribution. Therefore, even the results derived from only a small amount of material will be valid and can be relevant.

Nevertheless, the amount of material we have analyzed is rather small and although the breadth of phenomena seen increases very quickly during the first few sessions analyzed, the set of phenomena we describe is likely incomplete; we have not yet reached theoretical saturation. Therefore, generalizing our results is not invalid (because (1) we do not claim completeness and (2) additional phenomena will not invalidate existing ones) but might be misleading. Further work will have to investigate more and different source material in order to achieve broad coverage of the pair programming knowledge transfer phenomena.

6.3 Depth

Our results so far consist of several islands of conceptualization that are only weakly connected to each other and to the pair programming process overall. The results do therefore not yet constitute an actual Grounded Theory of knowledge transfer in pair programming. Further work will have to perform additional analysis to achieve greater conceptual depth: to weave the partial conceptualizations together[2] (axial coding) and extract an overall narrative of pair programming knowledge transfer from it (selective coding).

[2]and into a conceptualization of the overall process that does not even partially exist yet.

6.4 Engineering implications

The goal of our research is to provide not only scientific understanding of pair programming but also practical advice to software engineers, e.g. in the form of pair programming process patterns. The advice derivable from the work presented here is mostly implicit (declarative) and only partially procedural. Further work will have to find out how to make the advice fully procedural and how to *teach* such behavior to pair programmers.

7. CONCLUSION

We have identified four elements of knowledge transfer skill in pair programming. When putting them together, they allow to formulate a rough sketch of the problem solving process for knowledge transfer challenges.

- Whenever both partners perceive a different (perhaps vague) knowledge need at the same time, they need to determine an order for satisfying those needs and must not allow two Propellors, that is, must not pursue both needs at once.
- The Propellor must make sure to recognize if the knowledge need contains multiple elements that together are too complicated to be explained at once and so need to be split up into multiple Topics for separate clarification.
- If a Direct Question is insufficient for successfully communicating the Topic, and it often is, the Customer needs to construct a sequence of Explanation Triggers of escalating difficulty along the (up to) five Explanation Trigger Types of the Clarification Cascade in order to lead first herself and then the partner towards a sufficiently precise understanding.
- The pair must not lose sight of a Topic until it is resolved or they find a good reason to give up. Losing sight can happen easily because additional Knowledge Transfer Episodes often intervene.

The above process sketch is incomplete; we have not analyzed all relevant sub-phenomena of pair programming knowledge transfer by far.

However, even these few behaviors are sufficiently difficult to make some pairs much more efficient in their knowledge transfer than others. We consider such pairs to possess the superior pair programming skill.

Note that the concepts we found are not bound to specific software engineering issues; they are completely generic. This is good, because it provides wide applicability. It also means that in principle, the same phenomena might also occur in dialogs in very different domains. However, we conjecture that this will not be common. Rather, software engineering dialog is special because of the enormous concreteness and precision required to create software. Insofar, the results presented here do not describe general dialog skill, they describe pair programming skill.

Acknowledgments

This work was supported by a DFG grant. We thank our pairs for allowing us to record and scrutinize their sessions.

8. REFERENCES

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

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley Professional, 2004.

[3] A. Begel and N. Nagappan. Pair programming: what's in it for me? In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 120–128, New York, NY, USA, 2008. ACM.

[4] N. V. Flor. Side-by-side collaboration: A case study. *International Journal of Human-Computer Studies*, 49(3):201–222, 1998.

[5] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *Empirical studies of programmers: Fourth workshop*, pages 36–64. Ablex Publishing Corp., 1991.

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

[7] J. T. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, 1998.

[8] Platon. *Politeia*.

[9] L. Plonka. *Unpacking collaboration in pair programming in industrial settings*. PhD thesis, Open University, 2012.

[10] S. Salinger, L. Plonka, and L. Prechelt. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):9–25, 2008.

[11] S. Salinger and L. Prechelt. *Understanding Pair Programming: The Base Layer*. BoD, Norderstedt, Germany, 2013. 978-3-7322-8193-0.

[12] S. Salinger, F. Zieris, and L. Prechelt. Liberating pair programming research from the oppressive driver/observer regime. In *Proc. 35th Intl. Conf. on Software Engineering (ICSE)*, pages 1201–1204. IEEE Press, 2013.

[13] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

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

[15] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.

[16] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, 2000.