HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 9–25

A CODING SCHEME DEVELOPMENT METHODOLOGY USING GROUNDED THEORY FOR QUALITATIVE ANALYSIS OF PAIR PROGRAMMING

Stephan Salinger
Institut für Informatik
Freie Universität Berlin
Germany

Laura Plonka
Institut für Informatik
Freie Universität Berlin
Germany

Lutz Prechelt
Institut für Informatik
Freie Universität Berlin
Germany

Abstract: *A number of quantitative studies of pair programming (the practice of two programmers working together using just one computer) have partially conflicting results. Qualitative studies are needed to explain what is really going on. We support such studies by taking a grounded theory (GT) approach for deriving a coding scheme for the objective conceptual description of specific pair programming sessions independent of a particular research goal. The present article explains why our initial attempts at using GT failed and describes how to avoid these difficulties by a predetermined perspective on the data, concept naming rules, an analysis results metamodel, and pair coding. These practices may be helpful in all GT situations, particularly those involving very rich data such as video data. We illustrate the operation and usefulness of these practices by real examples derived from our coding work and present a few preliminary hypotheses regarding pair programming that have surfaced.*

Keywords: *pair programming, grounded theory, coding scheme development, qualitative data analysis, video data.*

INTRODUCTION

During the last few years, pair programming, as it is known from extreme programming (Beck, 2004), has been the subject of many empirical investigations. This research focused mainly on the measurement of bottom-line pair programming effects, whereas the underlying process of pair programming has been regarded as a kind of black box, the output of which is analyzed quantitatively with respect to its performance, error rate, programmer satisfaction, and so forth.

URN:NBNfi:jyu-200804151350

Unfortunately, the results of this research are often contradictory. For instance, regarding total effort (measured in person-hours of developers' work time), Williams (2001) found that pair programming results in a 15% increase compared to solo programming, Lui and Chan (2003) found 21%, and Nawrocki, Jasiński, Olek, and Lange (2005) found 48%. Most likely these differences are caused by differences in moderator variables, such as programmer and pair experience, type of task, and so on, but we do not know the complete set of relevant moderator variables nor the nature and mechanism of their influence.

Our goal as software engineering researchers is to understand pair programming in such a way that we can advise practitioners how to use it most efficiently. We propose that the only way to obtain such understanding is to understand the mechanisms at work in the actual pair programming process. Obviously, this understanding must first be gained in qualitative form before we can start quantifying and, since we do not know much yet, the investigation has to start in an exploratory fashion.

We have started such an investigation based on the grounded theory (GT) methodology (Strauss & Corbin, 1990) and working from rich sets of data (full-length audio, programmer video, and screen video of pair programming sessions). The current article presents a number of important methodological insights gained during this research and a few initial results. Its contributions are the following:

- a description of stumbling blocks for a GT-based analysis in this area;
- a set of practices that extend the plain GT method and help overcome obstacles;
- a sketch of a pair programming process coding scheme.

In subsequent research, the coding scheme is intended to form the basis for more detailed conceptual descriptions of the pair programming process. It also should support the proposition of hypotheses and theory construction.

We will first give a short introduction to GT and describe the nature and origin of our raw data. The heart of the article describes how and why plain traditional GT does not work well under these constraints and which practices help it work better. Thereafter we will present the application of the modified GT process and a few of its initial results, namely excerpts of a coding scheme for describing the activities occurring during pair programming. We close by outlining related works and offering a summary and outlook. This article is an improved and slightly extended version of Salinger, Plonka and Prechelt (2007) and focuses on research method, not on research results. The results primarily serve to illustrate the method.

THE GROUNDED THEORY METHODOLOGY

Selecting Among Qualitative Research Methods

We have already argued why we believe that it is time to study pair programming in an exploratory manner. We want to avoid posing specific hypotheses and generally make as few assumptions as possible. Using predefined coding schemes (see Hughes & Parkes, 2003, for a list) implies making such assumptions and hence should be avoided. Considerations like these quite naturally lead to using GT as the research method, because GT is an approach that makes the fewest number of assumptions.

Alternative methods, such as protocol analysis (Ericsson & Simon, 1993) or verbal analysis (Chi, 1997), appear less suitable because they start from at least partially predefined coding schemes or theoretical models. They are also more specialized than appropriate: They were designed for investigating cognitive processes.

Verbal analysis aims at the ability to quantify qualitative data, which could be an advantage. Unfortunately, such quantification requires a well-defined granularity of segmentation, so making such decisions at the start of the analysis prematurely structures the exploration space and prevents a completely open exploratory approach.

The Basic Ideas of Grounded Theory

GT, first described in Glaser and Strauss (1967), is a data analysis approach that is largely data driven and aims at producing a theory that describes interesting relationships between things, situations, events, and activities (together called *phenomena*) reflected in the data by means of abstract *concepts*. The term *grounded* indicates that this theory will contain only statements derived from actual observations in a manner that can be traced back to these data: The theory is grounded in the data.

We use the variant of GT described by Strauss and Corbin (1990), who suggest three (partially parallel) activities for a GT-based data analysis:

1. *Open coding* describes the data by means of conceptual (rather than merely descriptive) codes, which are derived directly from the data.
2. *Axial coding* identifies relationships between the concepts described by these codes. Strauss and Corbin (1990) suggest a concrete set of relationships to check for (in particular: *causal conditions* leading to phenomena that exist in a *context* featuring *intervening conditions* and leading to *participant's strategies* that create certain *consequences*). These relationships (plus the slightly fuzzy notion of forming *categories*) they call a *paradigmatic model*, a term we will use further below.
3. *Selective coding* extracts a subset of the concepts and relationships found and formulates them into a coherent theory. Selective coding is not relevant for the development of a coding scheme and thus will not be discussed in the present article.

Strauss considered the following three aspects to be the core of the GT method, saying in an interview that only these are required in order to call something GT (Legewie & Schervier-Legewie, 1995):

- *Theoretical coding:* Codes are theoretical, not just descriptive. They reflect concepts that have potential explanatory value for the phenomena described.
- *Theoretical sampling:* The selection of the material to be analyzed is made incrementally during the course of the analysis, based on what is expected to be most relevant for the theory under development.
- *Constant comparison:* Observed phenomena (and their contexts) are compared many times in order to create codes that are precise and consistent.

Theoretical sampling is of less interest in the present article, but theoretical coding and constant comparison are of vital importance to understand the discussion.

DATA USED FOR THE ANALYSIS OF PAIR PROGRAMMING

In the following subsections, we describe our observation context (programmers and task). We also describe the data capturing method used.

Observation Context: The Origin of Our Data

We observed (in the manner described below) seven pairs of graduate students who all worked on the same task. Six of them had worked together as pairs previously. The average work time (which was not limited) was 3.8 hours. The students were all participants of a highly technical course on enterprise information systems and the Java2 Enterprise Edition (J2EE) architecture and technologies. The specific task called for an extension of an existing Web shop application. The task required broad passive J2EE knowledge for analyzing and understanding the existing system and specific operational knowledge about Java Message Service (JMS), Java Naming and Directory Interface (JNDI), and the JBoss application server[1] for programming, configuring, and testing the actual extension. The task was not easy; only three of the pairs were completely successful. The other four pairs terminated their work before it was completely finished. They did not believe it to be possible to solve the remaining problems in an acceptable time frame.

For the analysis described in the present article, we used the session of one of the successful pairs only. This session ran 2 hours and 58 minutes.

Observation Method: Data Capturing Procedure

Since we did not know in advance what would or would not be important, we needed to start from a rather rich data set. We used three different data sources:

- An audio recording captured verbal communication between the participants, as well as other noises, vocal or other, that may have helped with the interpretation of the data.
- A frontal-perspective video of the programmers (shot from above and behind the screen and reaching down to about waist level) captured aspects of facial expression, gestures, posture, direction of attention, and, most relevantly, who was operating mouse and keyboard at any given time.
- A full-resolution screen recording captured almost all computer activities of the programmers on a fairly fine-grained level.

All three recordings were made simultaneously using Camtasia Studio[2] and unified into a single, fully synchronized video file in which the camera video was superimposed semitransparently onto a corner of the screen video. In this way, all data was visible at once (multidimensional video).

The session was recorded in an otherwise silent office. Combined with the high audio quality of a high-end webcam[3], this arrangement provided good acoustical playback conditions.

PROBLEMS OF A PLAIN GROUNDED THEORY DATA ANALYSIS APPROACH

Attempting GT-style exploratory analysis of the rich data set described above (actually a precursor study, but very similar in all respects), we quickly recognized that transcription was not practical. Too much relevant information in the screen recording—source code fragment input, used features of the development environment (such as browsing across different files or positions within files), pointing with the mouse during discussion with the partner, and so on—proved unclear in how to go about, or impractical in the effort of, transcribing.

This is why we decided to work on the raw video directly. We chose the qualitative data analysis software ATLAS.ti[4] for achieving this task, which is one of the few products that allows direct annotation to video.

One of us, Stephan Salinger, started open coding in the manner suggested by Strauss and Corbin (1990). The short-term goal was to characterize the activities occurring during pair programming; the long-term goal was to identify recurring behavioral patterns and classify them as helpful, hampering, ambivalent, or neutral.

This approach generated as many as 194 distinct concepts and almost complete confusion and despair in the course of a few days of analysis due to the following problems:

- No predefined focus: We had no criteria for selecting which observations (verbal interaction, facial expressions, gestures, posture, directions of gaze, subverbal vocal noises, nervous tics, computer input, input methods, computer output, etc.) to code and which to ignore, and consequently were overwhelmed by the data.
- No predefined granularity: We made no prior decision regarding the level of detail worth coding. As a result, we produced codes on different levels of detail (e.g., coarse ones such as *handle problem* and finer ones such as *test defect fix*), which were difficult to delineate against one another subsequently.
- No predefined level of acceptable subjectivity: The nature of the codes chosen in GT can be anywhere on the spectrum, ranging from codes that reflect observations that any observer could agree with to codes that interpret the observation to a degree that could be called wishful thinking. GT as such does not provide a criterion for deciding where "grounded in data" ends and wishful thinking begins. As a consequence, we mixed objective–descriptive and subjective–evaluative attitudes for selecting codes. This led to codes of different nature (e.g., descriptive ones such as *uses documentation* and assumption-bearing ones such as *gains knowledge of detail*) existing side-by-side, which made it harder to decide which code to use in a particular case.
- Too many topics: The codes described too many different topics of interest, making it impossible to properly focus on anything. None of the resulting collections of information ever reached a useful degree of completeness.
- Lack of concept grouping: The diversity of topics also distracted from forming what GT calls categories: a few large groups of heavily interrelated concepts, say, human-human interaction (HHI) or human-computer interaction (HCI).
- Importance misjudgments: The high attention to a broad set of concepts overtaxed our ability to judge their importance so that, because of the large number of concepts we introduced, we completely overlooked a number of important ones.

After we had noticed and gradually understood a number of these problems, we stopped this mode of investigation completely. We restarted the complete analysis from scratch (but very slowly and carefully, and with considerable backtracking) and concurrently redesigned the coding procedure. The result of this redesign was a number of heuristic practices described below that help using the GT analysis process.

PRACTICES SUPPORTING THE ANALYSIS OF COMPLEX VIDEO DATA

The methodological heuristics presented here form the heart of the present article. These intertwined practices serve to reduce or solve the problems described in the previous section. After introducing them, we will present an application that shows how they work together and mutually support one another.

Practice 1: Perspective on the Data

Strauss and Corbin (1990) suggest that the start of selective coding (that is, after open coding and axial coding have been going on for quite some time) is the time when you should begin to decide what is important and what is less so. As described above, we found that this is not practical when working with rich video data. There are three reasons why a perspective used for the analysis should be defined before starting:

- to avoid drowning in detail;
- to provide consistency in the criteria used for creating and assigning concepts;
- to focus attention on the most relevant aspects.

This perspective can be defined by formulating answers to the following questions. These answers should be reviewed (and perhaps revised) several times in the course of the analysis:

1. In which respects do you expect the data to provide insight?
2. What kinds of phenomena do the researchers allow themselves to identify in the data?
3. What type of result do you want the analysis to bring forth?

Question 1 does not ask what you expect to find, only in what respects you expect to find *something.* The answer acts as a filter that tells you which phenomena should receive more attention than others. Furthermore, constantly rechecking and adjusting the answer to this question helps in deciding when to stop the analysis, when to modify (or even replace) your research question, and when to obtain further or different raw data. In our case, the expectation was that the data could help understand what activities dominate the pair programming process and how they relate.

Answer 2 provides the mechanism for systematically bounding the nature and amount of subjectivity to be found in the conceptualizations of the data. The strongest restriction would be to allow only concepts that express directly observable phenomena, resulting in a behaviorist (stimulus/response) research perspective. Weaker restrictions might also allow concepts referring to unobservable processes (such as attitudes or thinking processes of actors), concepts that involve predictions (such as “helpful for reaching goal X”), and/or

concepts expressing moral judgment (good, bad). We were convinced that, in our case, only the behaviorist perspective would enable us to trust our own results.

Finally, the result type is the standard used for deciding how much attention to invest in which kinds of phenomena when the analysis resources begin to get scarce (which very quickly they will). It helps to stay on track. Do we want to produce a full conceptual theory, just a conceptual structure (system of categories) for the data, or even just a coding scheme? In our case, the goal was just to produce a coding scheme, because we felt we knew so little about the internals of pair programming that we should not yet decide on an actual engineering research question.

Practice 2: Concept Name Syntax Rules

Choosing concept names is another area where we found that giving up some of the freedom postulated by plain GT is beneficial. We found that our initial freely chosen concept names turned out to be highly variable and hence difficult to understand, remember, and compare.

As a remedy, we developed a structured naming scheme. Within the confines we set for ourselves by Practice 1, that is, describing directly observable activities of the pair programmers, the scheme does not predetermine anything with respect to the meaning of a concept: It only prescribes the shape of its name. When working with this scheme, we observed the following benefits:

- A concept will be better understood right at introduction time.
- A naming scheme facilitates managing a large set of concepts consistently.
- Some relationships between concepts are implicitly recorded as well, which greatly simplifies axial coding and the forming of categories.
- A concept name explicitly represents several aspects at once, which simplifies the fundamental GT practice of constant comparison.
- It becomes easier to understand where difficulties in delineating one concept against another arise, and correspondingly easier to obtain insights into the weaknesses of the overall conceptual description in practice.

In our case, the concepts needed to describe individual activities by one or both of the pair members, although for other domains of analysis different code naming structures might be preferable. Our concept name was structured like a complete sentence:

code = <actor>.<description>
actor = P1 | P2 | P
description = <verb>_<object>[_<criterion>]

Examples for such concept names are P1.ask_knowledge and P2.explain_knowledge. The criterion element of the structure can be used for additional specialization where needed. Given such codes, subsequent analysis can very easily abstract, for instance, the verb element (to compare contexts of objects) or the object element (to compare the variants of action types). Without such complex codes, the same situation would probably be modeled by a tuple of codes with relationships. So while finding relationships in plain GT involves axial coding, in our case recording at least some relationships became a fringe benefit of open coding.

Practice 3: Analysis Results Metamodel

When we started practicing GT, we found some of the terminology and concepts confusing. First, where GT talks about phenomena, conceptualization, concepts, properties, categories, and relationships, our analysis software (ATLAS.ti) talks about quotations, annotation, concepts, concepts, families, and relationships, respectively—and even the term *relationships* denotes two different notions.

Second, even after the initial learning phase, some of the differences were subtle enough that we misapplied them every once in a while. As a result, we became confused when trying to reconstruct what we had meant to express.

Third, when decisions regarding the introduction or demarcation of codes became difficult (which they often did), we realized we needed guidance for systematically applying the ideas of GT to break out of the situation in an appropriate way. (An example of this will be given in the section presenting the practices' application.)

Fourth, we extended the terminological framework with additional ideas related to the nature of our data, in particular the notion of a Track for partitioning data in order to support data visualization for a better overview of nested and parallel activities.

Together, these issues prompted us to formulate an explicit analysis results metamodel, that is, a model of the concepts that describe the structure of an analysis result. We formulated this metamodel as a UML class model (Rumbaugh, Jacobson, & Booch, 2005), which is shown in Figure 1.

Here is a very short description of the model's elements: a *Quotation* defines a fragment of the data (a scene of the video) the analysis refers to. An *Annotation* connects Quotations with a *Concept*. Concepts can be grouped into a *ConceptClass*; a single Concept can be a member of many ConceptClasses.

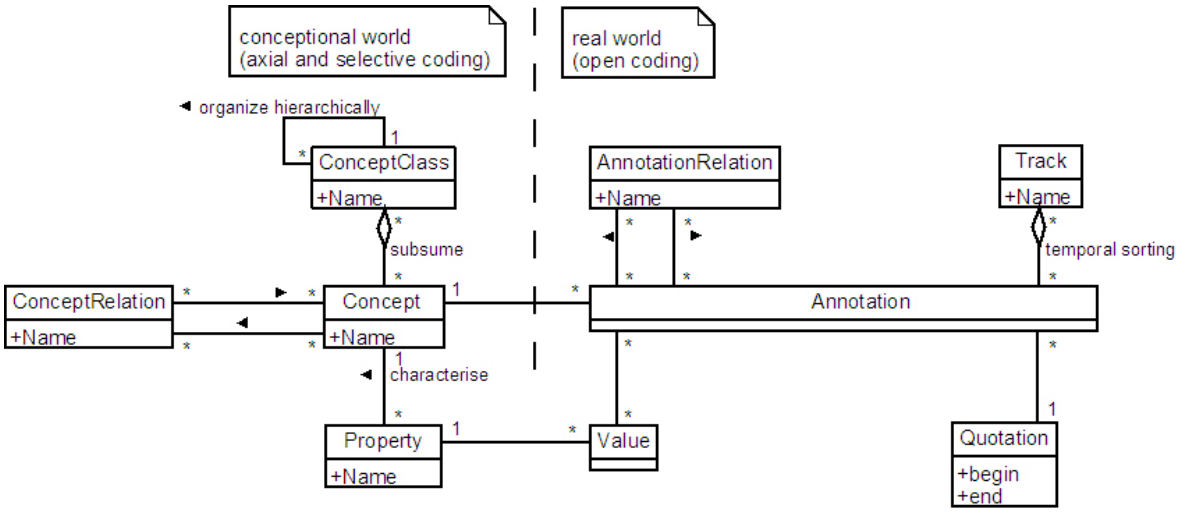

Figure 1. Complete metamodel of analysis results formulated as a UML class model. Boxes denote the various different kinds of elements occurring in our GT analysis results and the lines describe the relationships between them.

In order to further differentiate Concepts, they can be attributed with *Properties* that have *Values*. This allows developing concepts in a data-driven manner during axial coding and is helpful for identifying relationships between concepts (Strauss & Corbin, 1990).

A *ConceptRelation* is used to describe a relationship between Concepts, for instance according to the paradigmatic model. In many cases, such a relationship is not valid for all pairs of Annotations that use these Concepts; it can then be expressed individually by using *AnnotationRelation*. A *Track* allows for defining subsets of annotations that help identify various kinds of recurring relationships on the concept level, typically by means of appropriate visualization, as shown in Figure 2.

In addition to describing the structure of analysis *results* (to avoid terminological confusion), the metamodel also acts as a repository of ideas for the analysis *process*. For instance, when one is unsure whether a certain ConceptRelation will always hold, the metamodel suggests initial annotation of the currently known *instances* only (AnnotationRelation) and deferring the creation of the more general ConceptRelation until sufficient evidence is available.

Note that the metamodel is meant to be used throughout all phases of the GT research process. Some of its elements (e.g., Tracks) are used only rarely during the development of a coding scheme, as described in this article.

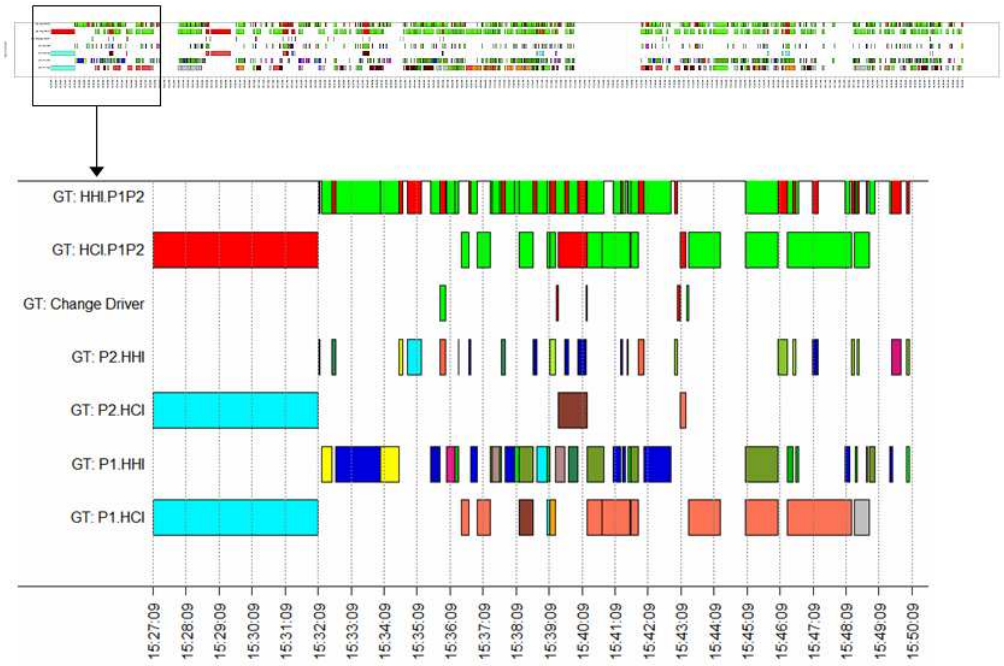

Figure 2. An example of a visualization of Tracks: The upper part shows a heavily scaled-down, automatically generated visualization of the GT annotations for a full pair programming session of 2 hours and 58 minutes. The lower part shows a magnified excerpt containing in particular the following four tracks: Track HHI.P1P2 represents the HHI activities of P1 (green) and P2 (red); HCI.P1P2 is the corresponding view of the HCI activities. Track P1.HHI represents each type of HHI activity performed by P1 in a different color; P1.HCI is the corresponding view of the HCI activities.

Practice 4: Pair Coding

The central and most important practice is pair coding. *Pair coding* means that all coding work is done by two people working together at one computer (much like pair programming, but that is just a coincidence). The key idea of pair coding is to require a consensus of two people for all important decisions: Which phenomena found in the data to single out for coding; where in time such a phenomenon starts and ends; which existing concept to use for coding this phenomenon; when to create a new concept; how to name that concept.

We found a number of benefits associated with pair coding as compared to a single researcher, some of them very important for successful GT work:

- Concept definitions become more exact, because they are scrutinized more closely upon their introduction. This effect is further supported by the structured naming scheme (Practice 2).
- The differentiation between similar concepts also becomes more precise, due not just to better definitions but also because a pair is less likely to let a concept slip in that is on a much different level of granularity than the others (and hence likely to have big overlaps with one or more existing concepts).
- Remaining concept differentiation problems will not be ignored but rather discussed. If they can be resolved, this will happen at an earlier point in time, leading to fewer incorrect concept assignments and therefore less rework. If it is impossible to fully resolve them (a not uncommon situation), the discussion will help understanding why, leading to a better understanding of the concepts involved.
- The perspective on the data (Practice 1) is maintained more consistently.
- The perspective on the data is refined more regularly and more thoroughly.
- A larger number of relevant phenomena are detected and encoded.

These results are in tune with psychological research suggesting that groups will often produce better decisions than isolated individuals (Shaw, 1981). Under adverse circumstances, *groupthink* (i.e., excessive concurrence seeking in groups) may make group decisions worse (t'Hart, 1988). But there is hardly any danger that this will happen in our setting: Groupthink is most likely in cohesive groups with a dominant leader, where the group is sharing common stereotypes and producing group pressures towards conformity (Janis, 1982). Since it is one of the routine tasks of any pair coder to challenge stereotypes used by the partner and to strive towards identifying possible different viewpoints, only a dominant person can pose any danger of groupthink in a pair-coding context. If the coders are equals, groupthink will be highly unlikely to happen.

Taken together, these four practices provided a quantum leap in the usefulness of our analysis results. The next section will illustrate this with a number of examples that will also show how the practices complement one another.

APPLICATION OF THE PRACTICES AND SOME RESULTS

This section will present a few fragments from the analysis process that used the practices described above and that led to our coding scheme for pair programming. We present these

examples to make the practices clearer, to explain how they interact, and to make it more credible that they help vitally.

We first introduce four concepts from our coding scheme and then present some episodes from the process in which we created them. Finally, we state a few hypotheses about pair programming that we have derived based on our coding scheme.

An Extract from the Coding Scheme

Our current version of the coding scheme (which ignores the subject part of the concept names) contains about 50 different concepts, clustered into about 20 overlapping ConceptClasses, with most concepts being members of either two or three of them. As an illustrative example we present the four concepts of the *ThinkAloud* ConceptClass. They are shown in Table 1; the descriptions are heavily summarized.

Use of the Practices: A Few Examples

Early during the coding process we recognized that the so-called driver (Williams, Kessler, Cunningham, & Jeffries, 2000) frequently verbalized what he was doing on the computer. Based on this observation, we made two decisions. First, we developed two ConceptClasses (see Practice 3) called HCI (human–computer interaction) and HHI (human–human interaction) for separating the computer-operating aspect from the verbalization aspect. These were ConceptClasses rather than individual concepts because the same separation would obviously be relevant in many other cases as well. Second, we postulated a new concept, ThinkAloud_Activity. By virtue of the concept naming syntax structure (Practice 2), this one concept immediately generated a whole ConceptClass (although having only one member at first) based on the verb *to think aloud*. This effect led to extended differentiation of concepts where needed but incurs only little additional complexity for the coding scheme.

We introduced ThinkAloud_Finding as the second member of this class, when we found a phenomenon that was obviously thinking aloud but did not explain computer activity. The demarcation appeared to be relatively clear. In the discussion of the pair coders (Practice 4), we agreed that ThinkAloud_Activity can be used only for the driver and that it has priority where ThinkAloud_Finding might also be applicable.

Table 1. The Concepts of the ThinkAloud ConceptClass.

Concept name	Description
ThinkAloud_Activity	Explains a current computer-operating activity
ThinkAloud_Finding	States a newly won insight (e.g., that some prior action was a mistake)
ThinkAloud_State	Reflects on the current state of work with respect to the current strategy and goal
ThinkAloud_Completion	States that a simple work step has been completed

Soon thereafter we encountered a programmer's explanation of the state of affairs and recognized it could be annotated as ThinkAloud_State, thus creating the third member of this set of concepts. But we soon found ThinkAloud_State to exhibit two problems. First, we had a case where it collided with ThinkAloud_Finding, because the finding concerned the state of work. Second, it designated statements on rather different levels of abstraction and granularity.

We solved both problems by using the metamodel (Practice 3), specifically by introducing the ConceptRelation "is-precondition-of" from the existing concepts *Propose_Step* (suggesting the next step) and *Propose_Strategy* (suggesting an approach for choosing many future steps). We postulated that ThinkAloud_State had to refer to a previous Propose_Strategy and introduced a new concept ThinkAloud_Completion that would refer to a previous Propose_Step. This solved both problems at once: We could now discriminate large and small granularity (strategic and tactical) and gained a criterion for when not to use ThinkAloud_Finding, which provided the demarcation to the other two.

This illustrates how open coding naturally leads into axial coding and how the combination of the paradigmatic model with the concept naming syntax (Practice 2) can show a way back into open coding, thus keeping the complexity of the resulting annotations down.

We are convinced that this route worked only because of the pair coding constellation (Practice 4), since both coders initially suggested encodings based on the existing codes and only the nonacceptance of these suggestions (and their supporting arguments) by the other led to the discovery of the "is-precondition-of" relationship and the fourth code ThinkAloud_Completion.

Some Hypotheses Based on the Coding Scheme

Although we have not yet started the analysis of the actual pair programming process as such, a number of phenomena recurred so consistently that we already call them hypotheses:

- We have found no evidence that the driver and the observer do indeed work on different levels of abstraction, as claimed in the pair programming literature (Williams et al., 2000). Similar results have been reported for pair programmer discussions by Bryant, Romero and du Boulay (in press), Freudenberg (née Bryant), Romero and du Boulay (2007; based on quantitative–qualitative work), and by Chong and Hurlbutt (2007).
- We have observed what we call *pair phases*, characterized by a high density of communication acts referring to just one narrow issue. They look a lot like what descriptions of pair programming suggest as the normal pair programming process, but we realized they are all of short duration (usually under 3 minutes).
- We believe that pair programming is not driven by strategic planning and monitoring. Rather, the plan is quite often only one step long: A single step is suggested, possibly discussed, decided (or revised), and immediately executed.
- Besides the unavoidable roles of driver and observer, pair programming sessions apparently tend toward implicitly producing a leader role as well. The leader is the person more skilled for the given task and influences speed and direction of the process much more strongly than the pair partner, no matter which role the leader is taking.

We expect that valuable insight about pair programming can be gained by investigating the reasons, consequences, and typical context conditions of the above trends. For instance,

we expect to find that pair phases are episodes of super-high productivity; it would be helpful to understand when and why they occur.

RELATED WORK

Qualitative Analysis of Pair Programming

We know of no other work analyzing the process of pair programming that uses a real GT approach: Most similar works use at least partially predefined coding schemes and most perform quantitative–qualitative analyses by means of protocol analysis or verbal analysis. We are also not aware of any work that is using video data directly in the analysis process.

Wake (2002) presented a list of typical pair programmer activities, but provided little information on how it was derived. Bryant (2004) studied the difference in interaction type and frequency in novice versus expert pair programmers. In a pilot study, she first refined Wake's list into a table of 11 behavior and interaction types. In the actual study, she then recorded the sequence of events in real time according to this schema and analyzed these data in a mostly quantitative way.

Such real-time categorization is obviously a good precondition for analyzing a large number of sessions, which is a positive approach. On the other hand, the simplicity of the categorization that is needed to make it possible also restricts the results to analyzing in terms of the rather simple concepts already presented in the predefined list. Neither subtle discriminations nor surprising new insights appear likely from this approach: It is applicable only in narrowly scoped investigations using predefined hypotheses.

Bryant et al. (in press) investigated behavior related to the driver and observer roles. They started from audio recordings, transcribed them, and annotated exactly each sentence with one out of the six predefined codes. The coding scheme is based on Pennington (1987) and characterizes the abstraction level. The analysis is mainly quantitative. This research aims at confirming or rejecting a conventional wisdom and is thus rather more hypothesis-driven than exploratory. A similar assessment applies to Freudenberg et al. (2007).

Cao and Xu (2005) investigated the activity patterns of pair programming. Pair working sessions were videotaped and then transcribed. The analysis used a coding scheme based on a combination of the schemes from Lim, Ward and Benbasat (1997) and Okada and Simon (1997). Then, during the analysis of the data, a new schema was developed in a manner not described. This work shares our behaviorist observation attitude; unlike our approach, however, it ignored all information contained in the computer interaction even though it was still grounded in only objectively observable communication acts.

In contrast, Xu and Rajlich (2005) used the dialog-based protocol in order to analyze the cognitive activities in pair programming, which involves a far greater amount of either subjectivity or generalized assumption. The coding scheme involved classification heuristics derived from a theory on self-directed learning (Xu, Rajlich, & Marcus, 2005). Xu and Rajlich proposed to do the coding assignment by two or more coders. In contrast to our approach, the coders worked separately and compared the results afterwards. This approach is sensible only with a fixed coding scheme; a GT-like generation of concepts would be very inefficient in this manner. Immediate discussion, as in pair coding (Practice 4), is much more efficient.

It is obvious that all five studies use rather predefined concepts during the analysis than concepts grounded only in the data. We fear that such approaches will be much more likely to fall prey to unwarranted assumptions according to conventional wisdom, such as the presumed driver/observer role differences, and so on.

Grounded Theory Work Using Rich Video Data

Even in the broader GT-related literature, examples of studies using video during the analysis (rather than transcripts of videos only) are rare. We found one such example in medicine that studied medical team leadership behavior (Xiao, Seagull, Mackenzie, & Klein, 2004). The video was recorded with four cameras from different angles. The analysis involved four analysts and three steps: (a) One analyst identified video segments with interesting verbal or nonverbal team interactions; (b) Two analysts created conceptual descriptions of the segments by consensus; and (3) Taxonomies for leadership actions from the conceptual descriptions were developed. This approach resembles our pair coding practice, at least in Step 2. If different people performed Steps 1, 2, 3 (the article is very unclear in this respect), we consider this a problematic procedure: It is almost antithetical to the GT philosophy, because it partially prohibits constant comparison and fully prohibits the intertwining of open coding (Steps 1 and 2) and axial coding (Step 3).

CONCLUSION AND FURTHER WORK

We have described why a straightforward application of the standard GT method on multidimensional video data of pair programming sessions is not likely to be successful. Furthermore, we presented and illustrated a set of four analysis practices that provide a systematic way to hold the analysis problems at bay:

- *Perspective on the data* helps avoid drowning in detail.
- *Concept name syntax rules* help create useful and consistent concept names.
- *An analysis results metamodel* helps keep the analysis process systematic and the results well structured.
- *Pair coding* mitigates the effects of limited or distorted perception.

We have used these practices to generate a general-purpose coding scheme of pair programming activities, of which we presented a small excerpt. In the future, we will proceed with the following steps:

- Validation of the coding scheme. We will encode sessions that have very different properties with respect to participants, task, and setting.
- Qualitative and quantitative evaluation of the coding process itself, based on its results, intermediate results, and process monitoring information (in particular timestamps) recorded by ATLAS.ti.
- Refinement of the coding scheme with respect to particular research applications, in particular by adding properties according to the metamodel.
- Application of the coding scheme to produce actual grounded theories of several aspects of the pair programming process. This will require selective coding through

which we expect to exercise even those parts of the metamodel not discussed in the present article.

Just like the four practices mutually support one another, these tasks will also exhibit synergy and so will be performed partially in parallel.

ENDNOTES

1. See http://labs.jboss.com/
2. A product of the TechSmith Corporation, http://www.techsmith.com
3. Logitech 5000 webcam
4. See http://www.atlasti.com/

REFERENCES

Beck, K. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Boston: Addison-Wesley Professional.

Bryant, S. (2004). Double trouble: Mixing qualitative and quantitative methods in the study of extreme programmers. In *Proceedings of the 2004 IEEE Symposium on Visual Languages: Human Centric Computing* (VL/HCC '04; pp. 55–61). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/VLHCC.2004.20

Bryant, S., Romero, P., & du Boulay, B. (in press). Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies.*

Cao, L., & Xu, P. (2005). Activity patterns of pair programming. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* (HICSS '05; p. 88a). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/HICSS.2005.66

Chi, M. T. H. (1997). Quantifying qualitative analyses of verbal data: A practical guide. *Journal of Learning Sciences, 6*, 271–315.

Chong, J., & Hurlbutt, T. (2007). The social dynamics of pair programming. In *Proceedings of the 29th International Conference on Software Engineering* (ICSE '07; pp. 354–363). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.87

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: Verbal reports as data*. Cambridge, MA, USA: MIT Press.

Freudenberg, S. (née Bryant), Romero, P., & du Boulay, B. (2007). "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? In *AGILE 2007* (pp. 84–91). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/AGILE.2007.1

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory: Strategies for qualitative research*. New York: Aldine de Gruyter.

Hughes, J., & Parkes, S. (2003). Trends in the use of verbal protocol analysis in software engineering research. *Behaviour and Information Technology, 22*, 127–140.

Janis, I. L. (1982). *Groupthink* (2nd ed.). Boston: Houghton Mifflin Company.

Legewie, H., & Schervier-Legewie, B. (1995). Im Gespräch: Anselm Strauss [An interview of Anselm Strauss]. *Journal für Psychologie, 3*, 64–75.

Lim, K., Ward, L., & Benbasat, I. (1997). An empirical study of computer system learning: Comparison of co-discovery and self-discovery methods. *Information Systems Research, 8*, 254–272.

Lui, K. M., & Chan, K. C. (2003). When does a pair outperform two individuals? In M. Marchesi & G. Succi (Eds.), *Extreme programming and agile processes in software engineering* (Lecture Notes in Computer Science 2675, pp. 225–233). Berlin, Germany: Springer.

Nawrocki, J. R., Jasiński, M., Olek, Ł., & Lange, B. (2005). Pair programming vs. side-by-side programming. In I. Richardson, P. Abrahamsson, & R. Messnarz (Eds.), *Software process improvement* (Lecture Notes in Computer Science 3792, pp. 28–38). Berlin, Germany: Springer.

Okada, T., & Simon, H. (1997). Collaborative discovery in a scientific domain. *Cognitive Science, 21*, 109–146.

Pennington, N. (1987). Comprehension strategies in programming. In G. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.

Rumbaugh, J., Jacobson, I., & Booch, G. (2005). *The unified modeling language reference manual* (2nd ed.). Boston: Addison-Wesley Professional.

Salinger, S., Plonka, L., & Prechelt, L. (2007). A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. In J. Sajaniemi, M. Tukiainen, R. Bednarik, & S. Nevalainen (Eds.), *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group* (pp. 144–157). Joensuu, Finland: Department of Computer Science and Statistics, University of Joensuu. Also available at http://www.ppig.org/papers/19th-Salinger.pdf

Shaw, M. E. (1981). *Group dynamics: The psychology of small group behavior*. New York: McGraw Hill.

Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. London: Sage Publications, Inc.

t'Hart, P. (1988, July). *Groupthink: Observations toward a theory*. Paper presented at the meeting of the International Society of Political Psychology, Meadowlands, NJ, USA.

Wake, W. (2002). *Extreme programming explored*. Boston: Addison-Wesley.

Williams, L. (2001). Integrating pair programming into a software development process. In *Proceedings of the 14th Conference on Software Engineering Education and Training* (CSEET '01; pp. 27–36*)*. Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/CSEE.2001.913816

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

Xiao, Y., Seagull, F., Mackenzie, C., & Klein, K. (2004). Adaptive leadership in trauma resuscitation teams: A grounded theory approach to video analysis. *Cognition, Technology & Work, 6*, 158–164.

Xu, S., & Rajlich, V. (2005). Dialog-based protocol: An empirical research method for cognitive activities in software engineering. In *International Symposium on Empirical Software Engineering* (ISESE 2005; pp. 383–392). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/ISESE.2005.1541848

Xu, S., Rajlich, V., & Marcus, A. (2005). An empirical study of programmer learning during incremental software development. In *Fourth IEEE Conference on Cognitive Informatics* (ICCI 2005; pp. 340–349). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.acm.org/10.1145/1145287.1145289

Author's Note

All correspondence should be addressed to:
Stephan Salinger
Institut für Informatik
Freie Universität Berlin
Takustr. 9
14195 Berlin
Germany
salinger@inf.fu-berlin.de

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi