A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming

Stephan Salinger, Laura Plonka, and Lutz Prechelt

Freie Universität Berlin, Institut für Informatik,
Takustr. 9, 14195 Berlin, Germany
salinger,plonka,prechelt@inf.fu-berlin.de

Abstract. Since a number of quantitative studies of pair programming (the practice of two programmers working together using just one computer) have produced somewhat conflicting results, a number of researchers have started to study pair programming qualitatively. While most such studies use coding schemes that are fully or partially predefined, we have decided to go the long way and use Grounded Theory (GT) to ground each and every statement we make directly in observations.
The first intermediate goal, which we talk about here, was to produce a coding scheme that would allow the objective conceptual description of specific pair programming sessions independent of a particular research goal.
The present article explains how our initial attempts at using the method of Grounded Theory failed and which practices we developed to avoid these difficulties: predetermined perspective on the data, concept naming rules, analysis results meta-model, and pair coding. We expect these practices be helpful in all GT situations, in particular 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 also present a few preliminary hypotheses regarding pair programming that we have stumbled across.

1 Introduction

During the last few years, pair programming, as it is known from extreme programming [1], has been the subject of many empirical investigations. This research focussed 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 etc.

Unfortunately, the results of this research are often contradictory. For instance regarding total effort, Williams found that pair programming results in a 15% increase compared to solo programming [2], Lui and Chan found 21% [3], and Nawrocki et al. found 48% [4]. Most likely these differences are caused by differences in moderator variables such as programmer and pair experience, type of task etc., but neither do we 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 [5] and working from rich sets of data (full-length audio, programmer video, and screen video of pair programming sessions). The present paper 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 overcoming these obstacles;
- a sketch of a pair programming process coding scheme.

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

We will first give a short introduction to Grounded Theory (Section 2) and describe the nature and origin of our raw data (Section 3). The heart of the paper describes how and why plain GT does not work well under these constraints (Section 4) and which practices help to make it work better (Section 5). Section 6 presents the application of the modified GT process and a few of its initial results, namely excerpts of a coding scheme for describing the activities occuring during pair programming. We close by outlining related works (Section 7) and offering a summary and outlook (Section 8).

The paper focuses on research method, not on research results. The results mostly serve to illustrate the method.

2 The Grounded Theory methodology

As mentioned above, the initial analysis of pair programming has to be exploratory. In order to be as open as possible with respect to the nature and content of the results, we pick Grounded Theory as our analysis approach.

GT, first described in [6], is a data analysis approach that is largely data-driven (i.e. uses hardly any prior assumptions nor pre-defined terminology) 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 [5], 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 suggest a concrete set of relationships to check for (in particular: *causal conditions* lead to *phenomena* which exist in a *context* featuring *intervening conditions* and leading to *participant's strategies* which create certain *consequences*). These relationships (plus the slightly fuzzy notion of forming *categories*) they call *paradigmatic model*, a term we will use a few times further below.
3. *Selective coding* extracts a subset of the concepts and relationships thus found and formulates them into a coherent theory. Selective coding is not relevant for the development of a coding scheme and will not be discussed in the present article.

Strauss considered the following three aspects to be the core of the GT method, saying "When you do all of these, then it is Grounded Theory, if you do not, then it is something else" [7]:

- *Theoretical coding:* Codes are theoretical, not just descriptive; they reflect concepts which have potential explanatory value for the phenomena described.
- *Theoretical sampling:* The selection of the material to be analyzed is made incrementally in 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.

3 Data used for the analysis of pair programming

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

3.1 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 worktime (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 JMS, JNDI, and the JBoss application server for programming, configuring, and testing the actual extension. The task was non-trivial so that only three of the pairs were completely successful.

For the analysis described in the present article, we used the session of one of the successful pairs only; it is 2 hours and 58 minutes long.

3.2 Observation method: Data capturing procedure

Since we do not know in advance what will be important and what will not, we need to start from a rather rich data set. We use three different data sources:

– Audio recording captures verbal communication among the participants as well as other noises, vocal or other, that may help with the interpretation of the remaining data.
– Frontal-perspective video of the programmers (shot from above-behind the screen and reaching down to about waist level) captures aspects of facial expression, gestures, posture, direction of attention, and — most relevantly — who is currently operating mouse and keyboard.
– Full-resolution screen recording captures almost all computer activities of the programmers on a fairly fine-grained level.

All three recordings are made at once using Camtasia Studio [8] and unified into a single, fully synchronized video file in which the camera video is superimposed semi-transparently onto a corner of the screen video so that all information is visible at once (multi-dimensional video).

The session was recorded in an otherwise silent office. Combined with the high audio quality of the Logitech 5000 webcam, this provides good acoustical playback conditions.

4 Problems of a plain Grounded Theory data analysis approach

Attempting GT-style exploratory analysis of the rich data set described above[1], we quickly recognized that transcription was not practical. Too much relevant information is found in the screen recording for which it is not obvious how to transcribe it at all, not to speak of the effort for doing so: source code fragment input, using features of the development environment (such as browsing across different files or positions within files), pointing with the mouse during discussion with the partner, etc.

[1] Actually a precursor, but very similar in all respects.

This is why we decided to work on the raw video directly and chose the qualitative data analysis software ATLAS.ti [9] for doing so, which is one of the few products that allows creating direct annotations to video.

One of us, Stephan Salinger, started open coding in the manner suggested by Strauss and Corbin. The short-term goal was to characterize the activities occuring 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 no fewer than 194 different 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 (kinds of) observations to code and which to ignore (code verbal interaction? factial expressions? gestures? posture? directions of gaze? sub-verbal vocal noises? nervous tics? computer input? input methods? computer output? and so on) and consequently were overwhelmed by the data.
– No predefined granularity: We had no prior decision on the level of detail that would be worth coding. As a result, we produced codes on different levels of detail (say, coarse ones such as *handle problem* and finer ones such as *test defect fix*), which where 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 stick closely to observations that any observer would agree with to codes that interpret the observation to a degree that they must 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 (say, 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 one 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 various 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, and "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 started the whole analysis again from scratch (but very slowly and carefully, with a lot of backtracking) and concurrently redesigned the coding procedure. The result of this redesign were a number of heuristic practices described below that help using the GT analysis process.

5 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. Section 6 will present an application of the practices that also shows how they work together and mutually support one another.

5.1 Practice 1: Perspective on the data

Strauss and Corbin 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 a good idea 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 constancy 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 re-checking and adjusting the answer to this question helps deciding when to stop the analysis, when to modify (or throw overboard) 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 refering 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 judgement (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? Or 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.

5.2 Practice 2: Concept name syntax rules

Choosing the names of concepts is another area where we found that giving up some of the freedom postulated by plain GT is beneficial, because our 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 as described below. Within the confines we set 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.
- It facilitates handling and overlooking a large set of concepts.
- Some relationships between concepts are implicitly recorded as well, which much simplifies axial coding and the forming of categories.
- A concept name explicitly represents several aspects at once, which simplifies the basic GT practice of "constant comparison".
- It becomes easier to understand where difficulties in delineating one concept against another come from and correspondingly easier to obtain insights as to the weaknesses of the overall current conceptual description.

In our case, the concepts needed to describe individual activities by one or both of the pair members[2], so a concept name is structured like a complete sentence:

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

[2] For other domains of analysis, other code naming structures might be preferable.

for example "P1.ask_knowledge" and "P2.explain_knowledge". The criterion part can be used for additional specialization where needed. Given such codes, subsequent analysis can very easily abstract for instance the verb part (to compare contexts of objects) or the object part (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 in plain GT finding relationships involves axial coding, in our case recording at least some relationships became a fringe benefit of open coding.

5.3 Practice 3: Analysis results meta-model

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 relationships and relationships are not quite the same thing.

Second, even after the initial learning phase some of the differences were subtle enough that we misapplied them every once in a while and became confused when we tried 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 Section 6.)

Fourth, we extended the terminological framework by some additional ideas owing to the nature of our data, in particular the notion of *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 meta-model, that is, a model of the concepts that describe the structure of an analysis result. We formulated this meta-model as a UML class model [10], which is shown in Figure 1.

Here is a very short description of the most important elements of the model: `Quotation`s define fragments of the data (scenes in the video) that the analysis refers to. `Annotation`s connect `Quotation`s with `Concept`s. `Concept`s can be grouped into `ConceptClass`es; a single `Concept` can be a member of many `ConceptClass`es. `ConceptRelation`s are used to describe relationships between `Concept`s, for instance according to the paradigmatic model. In many cases, such a relationship is not valid for all pairs of `Annotation`s that use these `Concept`s; it can then be expressed individually by using `AnnotationRelation`.

The other elements of the meta-model are not relevant for the present article.

Besides describing the structure of analysis *results* (to avoid terminological confusion), the meta-model also acts as a repository of ideas for the analysis *process*. For instance, when unsure whether a certain `ConceptRelation` will always hold, the meta-model suggests to initially annotate the currently known

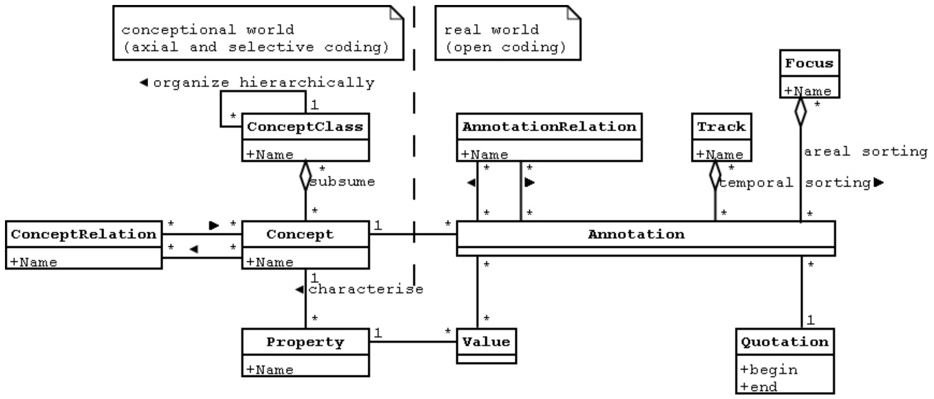

Fig. 1. Meta-model of analysis results

instances only (`AnnotationRelation`) and defer the creation of the more general `ConceptRelation` until sufficient evidence is available.

5.4 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 of a pair 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 right upon their introduction. This effect is further supported by the structured naming scheme (practice 2).
– The differentiation between similar concepts also becomes more precise, not just due 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 that hence much more often has 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 time leading to fewer incorrect concept assignments and/or less rework. If it is inherently impossible to fully resolve them (which is not uncommon at all), the reason for this will be understood much more thoroughly by the discussion, 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.

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 which will also show how the practices complement one another.

6 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 really help vitally.

We first introduce four concepts from our coding scheme and then present some episodes from the process in which we created them. As an add-on (and slightly off-topic for this article) we state a few hypotheses about pair programming that we have derived based on our coding scheme.

6.1 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` — most concepts being members of either two or three of them.

As an illustrative example, we present the four concepts of the *think aloud* `ConceptClass`. They are shown in Table 1; the descriptions are heavily abbreviated.

Table 1. The concepts of the *think aloud* 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 w.r.t. to the current strategy and goal
thinkaloud_completion	States that a simple work step has been completed

6.2 Use of the practices: a few examples

We will now explain how we arrived at these four concepts in order to show the practices in action and illustrate their interaction.

Soon during the coding process we recognized that the so-called *Driver* [11] frequently verbalized what he was doing on the computer. Based on this observation, we made two decisions:

First, we started developing two `ConceptClass`es (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 `ConceptClass`es 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` (so far having only one member) based on the verb "to think aloud". This effect leads to extended differentiation of concepts where needed but incurs only little additional complexity for the coding scheme.

As the second member of this class we introduced *thinkaloud_finding* when we found a phenomenon that was obviously thinking aloud, but that also obviously 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 is has priority where *thinkaloud_finding* might also be applicable.

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 meta-model (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), as both coders initially suggested encodings based on the existing codes and only the non-acceptance of these suggestions (and their supporting arguments) by the other lead to the discovery of the "is-precondition-of" relationship and the fourth code *thinkaloud_completion*.

6.3 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 clues that Driver and Observer do indeed work on different levels of abstraction, as claimed in the pair programming literature [11].
- We have observed what we call *pair phases*, characterized by a high density of communication acts refering 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 short (usually under three 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 apparrently tend to implicitly produce 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.

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 so that it would be helpful to understand when and why they occur.

7 Related work

7.1 Qualitative analysis of pair programming

We know of no other work analyzing the process of pair programming that uses a GT approach (they all work with at least partially pre-defined coding schemes) and also none that works directly with video data (multi-dimensional or other).

Wake [12] presents a list of typical pair programmer activities, but provides little information on how it was derived.

Bryant [13] studies the difference of 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 real study, she then recorded the sequence of events in real time according to this schema.

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

Cao and Xu [14] investigate the activity patterns of pair programming. Pair working sessions were videotaped and then transcribed. The analysis used a coding scheme that started out from a combination of the schemes from [15] and [16]. 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 us, it ignores all information contained in the computer interaction, but for the rest it still grounds on objectively observable communication acts only.

In contrast, Xu and Rajlich [17] use the dialog-based protocol[3] 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 involves classification heuristics derived from a theory on self-directed learning [18]. Xu and Rajlich proposed to do the coding assignment by two or more coders. In contrast to our approach, the coders work separately and compare the results afterwards. This approach is sensible only with a fixed coding scheme, because 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 three studies use rather more 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 presumed Driver/Observer role differences etc.

7.2 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 [19]. The video was recorded with four cameras from different angles. The analysis involved four analysts and three steps. (1) One analyst identified video segments with interesting verbal or non-verbal team interactions. (2) Two analysts created conceptual descriptions of the segments by consensus. (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+2) and axial coding (step 3).

[3] By the way, [17] suggests to use screen-capture and voice recording only rather than videotaping to avoid influences due to camera-consciousness — we have never observed this to be an issue at all.

8 Conclusion and further work

We have described why a straightforward application of the standard Grounded Theory method to multi-dimensional video data of pair programming sessions is not likely to be successful and have presented and illustrated a set of four analysis practices that provide a systematic way to hold the analysis problems at bay.

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 meta-model.
– Application of the coding scheme to produce actual grounded theories of several aspects of the pair programming process. This will require selective coding which we expect to exercise even those parts of the meta-model 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.

References

[1] Beck, K.: Extreme Programming Explained: Embrace Change, Second Edition. Addisson-Wesley Professional (2004)
[2] Williams, L.: Integrating pair programming into a software development process. In: CSEET '01: Proceedings of the 14th Conference on Software Engineering Education and Training, Washington, DC, USA, IEEE Computer Society (2001)
[3] Lui, K.M., Chan, K.C.: When does a pair outperform two individuals? In: Extreme Programming and Agile Processes in Software Engineering. Volume 2675 of Lecture Notes in Computer Science., Springer (2003) 225–233
[4] Nawrocki, J.R., Jasiñski, M., Olek, L., Lange, B.: Pair programming vs. side-by-side programming. In: EuroSPI. Volume 3792 of Lecture Notes in Computer Science., Springer (2005) 28–38
[5] Strauss, A., Corbin, J.: Basics of Qualitative Research: Grounded Theory Procedures and Techniques. Sage Publications, Inc. (1990)
[6] Glaser, B.G., Strauss, A.L.: The Discovery of Grounded Theory: Strategies for Qualitative Research. Aldine de Gruyter, New York (1967)
[7] Legewie, H., Schervier-Legewie, B.: Im Gespräch: Anselm Strauss. Journal für Psychologie **3** (1995) 64–75
[8] TechSmith Corporation: Camtasia studio 4.0.1. (http://www.techsmith.com)
[9] ATLAS.ti: User's Manual for ATLAS.ti 5.0. (http://www.atlasti.com)

[10] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley Professional (2005)
[11] Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Software **17** (2000) 19–25
[12] Wake, W.: Extreme Programming Explored. Addison Wesley Boston (2002)
[13] Bryant, S.: Double trouble: Mixing qualitative and quantitative methods in the study of extreme programmers. In: VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, Washington, DC, USA, IEEE Computer Society (2004) 55–61
[14] Cao, L., Xu, P.: Activity patterns of pair programming. In: HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences, Washington, DC, USA, IEEE Computer Society (2005)
[15] Lim, K., Ward, L., Benbasat, I.: An empirical study of computer system learning: Comparison of co-discovery and self-discovery methods. Information Systems Research **8** (1997) 254–272
[16] Okada, T., Simon, H.: Collaborative discovery in a scientific domain. Cognitive Science **21** (1997) 109–146
[17] Xu, S., Rajlich, V.: Dialog-based protocol: an empirical research method for cognitive activities in software engineering. In: International Symposium on Empirical Software Engineering. (2005) 383–392
[18] Xu, S., Rajlich, V., Marcus, A.: An empirical study of programmer learning during incremental software development. In: (ICCI 2005: Fourth IEEE Conference on Cognitive Informatics. (2005) 340–349
[19] Xiao, Y., Seagull, F., Mackenzie, C., Klein, K.: Adaptive leadership in trauma resuscitation teams: a grounded theory approach to video analysis. Cognition, Technology & Work **6** (2004) 158–164