Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Qualitative Untersuchung von Beeinflussungen der Aktivität des Partners bei der Paarprogrammierung

Franz Zieris
Matrikelnummer: 4138389
zieris@inf.fu-berlin.de

Berlin, 04. Februar 2012

Betreuer: Stephan Salinger
Gutachter: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Claudia Müller-Birn

Zusammenfassung

Paarprogrammierung (PP) zeichnet sich dadurch aus, dass zwei Personen gemeinsam Software entwickeln. Bevor man sich Gedanken darüber machen kann, wie man den PP-Prozess verbessern könnte, muss dieser erst einmal verstanden werden. Durch den Fakt, dass es Menschen sind, die miteinander interagieren, eröffnet sich ein breites Spektrum an möglichen Kommunikationsformen und -abläufen. Dazu gehören auch Meinungsverschiedenheiten über die konkrete Umsetzung oder generelle Arbeitsabläufe, bzw. das Infragestellen des Vorgehens des Anderen.

In dieser Arbeit wurde untersucht, auf welche Art und Weise und unter welchen Umständen während einer Paarprogrammierungssitzung ein Entwickler die gerade ablaufende Aktivität seines Partners beeinflusst. Dafür wurden detaillierte, qualitative Analysen von aufgezeichneten Sitzungen durchgeführt. Der Forschungsprozess dieser Arbeit richtete sich nach der *Grounded Theory Methodology* aus.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

04. Februar 2012

Franz Zieris

Inhaltsverzeichnis

0 Vorwort **1**
Paarprogrammierung? . . . 1
Motivation . . . 1
Vorgehensweise, Fokus und Zielsetzung dieser Arbeit . . . 2
Aufbau dieser Arbeit . . . 2

I Einführung **4**

1 Stand der Forschung **4**
1.1 Geschichte der Paarprogrammierungsforschung . . . 4
1.2 Anekdotische Berichte über Paarprogrammierung . . . 4
1.3 Quantitative Erforschung der Paarprogrammierung . . . 5
1.4 Qualitative Erforschung der Paarprogrammierung . . . 5

2 Forschungsmethode **6**
2.1 Datenbasis . . . 6
2.2 Auswahl der Forschungsmethode . . . 8
2.3 Forschungsprozess nach der Grounded Theory Methodology . . . 10
2.4 Basisschicht und Basiskonzeptmenge . . . 12
2.5 Forschungsstrategische Aspekte dieser Arbeit . . . 18

II Hauptteil **22**

3 Fokus der Analyse und Begriffsfindung **22**
3.1 Aktivitätsbegriff . . . 24
3.2 Dimensionen der betrachteten Phänomene . . . 24

4 Beeinflussungskonzepte **28**
4.1 Die Beeinflussungstypen . . . 28
4.2 Klassenübergreifende Funktionen von Beeinflussungen . . . 61

5 Phasen- und Rollen-Konzepte **62**
5.1 Phasen in Paarprogrammierungssitzungen . . . 62
5.2 Die Phasentypen . . . 65
5.3 Abschließende Bemerkungen zu den Phasen . . . 74

III Hinweise zur Herleitung der Ergebnisse **76**

6 Auszüge aus den Ergebnissen des offenen Kodierens **76**
6.1 Erweiterung der *activity*-Konzepte . . . 76
6.2 Erste Klassifizierung von Beeinflussungen . . . 78

7 Überlegungen aus der Phase des axialen Kodierens **79**
7.1 Einordnung von Interventionen 79
7.2 Vorgehensweise beim axialen Kodieren 80

IV Schluss **81**

8 Bewertung und Einordnung **81**
8.1 Was wurde in dieser Arbeit geleistet? 81
8.2 Verhältnis zur BS/BKM 81

9 Vergleiche mit bisherigen Erkenntnissen aus der Literatur **81**
9.1 Verschiedene Perspektiven von Driver und Observer 82
9.2 Abhängen des Partners und Rollenwechsel 82
9.3 Eigene Hinweise 83

10 Ausblick **83**
10.1 Inspiration aus der Kleingruppenforschung 83

Anhänge **86**

A Screenshots **86**

B Nebenbei „abgefallene“ Erkenntnisse **87**
B.1 Verhältnis zwischen *todo* und *design* 87
B.2 Wiederaufgreifen von *todo*: *remember_todo* 87
B.3 Selbstgespräche zu zweit 88

C Transkriptionshinweise **88**

D Acknowledgements **89**

Literaturverzeichnis **92**

Abbildungsverzeichnis

2.1 Metamodell der Analyseergebnisse 10
2.2 Darstellung des paradigmatischen Modells 11
2.3 Schematischer Aufbau der Basisschicht 14

3.1 Spektrum der Dimensionen „Negativity“ und „Miss of Expectations“ . . 26

4.1 Driver/Observer-Verteilung von Sitzungen D03 und D06 59

5.1 Funktionen von Einflussnahmen 65

10.1 Dyadische Interaktionssequenzen 84

A.1 Screenshot aus Sitzung D03 86
A.2 Screenshot aus Sitzung W02 86

Tabellenverzeichnis

2.1 Beispiele für die Klassen der produktorientierten Konzepte 14
2.2 Beispiele für die Klassen der prozessorientierten Konzepte 15
2.3 Beispiele für die Klassen der universellen Konzepte 16
2.4 Übersicht über die Fassadenkonzepte 17
2.5 Übersicht über die HCI/HEI-Konzepte 17

4.1 Übersicht über die Beeinflussungstypen 29
4.2 Beispiel-Episode für die Verschiebung des Aufmerksamkeitsfokus 35
4.3 Beispiel-Episode für eine Verantwortungsübernahme 42
4.4 Beispiel-Episode für *guidance* 47
4.5 Beispiel-Episode für *knowledge_request* 51
4.6 Beispiel-Episode für *finding_conveyance* 56
4.7 Beeinflussungstypen und ihre korrespondierenden Informationsbedürfnisse 61

5.1 Übersicht über die Phasentypen 64

6.1 BS-*activity*-Konzepte nach ihrer *Negativity* 76
6.2 Neue *activity*-Konzepte nach ihrer *Negativity* 77
6.3 Dimensionen der *Support*-Phänomene 78
6.4 Übergang der früheren *Support*-Typen in die jetzigen Beeinflussungstypen 78

C.1 Transkriptionsregeln 89

Verzeichnis der Exkurse

Exkurs 1: Arbeitsgedächtnis und Aufmerksamkeit 27
Exkurs 2: Fehler, Defekte und Versagen 32
Exkurs 3: Aufmerksamkeitsfokus und freie Ressourcen 36
Exkurs 4: Backchannel 74

Vorbemerkung

Wenn in wörtlichen Zitaten Referenzen auf Programmiersprachen oder Datei-/Ordnernamen enthalten sind, so werden diese in `nichtproportionaler Schrift` gesetzt. Konzeptnamen und Kodes werden *schräggestellt* gesetzt.

Sind im Text Personen in ihrer Rolle als Teilnehmer einer bestimmten Programmiersitzung gemeint, so werden sie mit *P1* und *P2* bezeichnet. Wird hingegen auf sie als reale Personen (also sitzungsübergreifend) Bezug genommen, dann werden sie mit $\mathcal{A}, \mathcal{B}, \mathcal{C} \ldots$ bezeichnet.

0 Vorwort

Paarprogrammierung?

Was ist Paarprogrammierung? Auch wenn der Bestandteil „*Programmierung*“ vielleicht Gegenteiliges vermuten lässt, muss währenddessen nicht zwangsläufig Programmcode entstehen; genauso kann man dabei auch gemeinsam konzeptuell über einen Algorithmus nachdenken oder einen Test durchführen:

> Als Paarprogrammierung (PP) bezeichnet man den Prozess, bei dem zwei Personen an einem Computer gemeinsam am selben Design, Algorithmus, Code oder Test arbeiten.
>
> nach Laurie Williams [41, S. iv]

Die Person, die dabei die Kontrolle über die Tastatur und die Maus hat, wird gemeinhin *Creator* [23] oder *Driver* [16, 41] genannt. Für die andere Person wurde eine Reihe von Rollennamen ersonnen, wie etwa *Non-Driver* [41], *Partner* [16] oder *Quality Assurer* [23]. Mittlerweile ist die Paarung *Driver – Observer* bzw. *Driver – Navigator* aber eine der gebräuchlichsten.
Entsprechend dieser klassischen Bezeichnungen werden auch die typischen Aufgaben der jeweiligen Rollen gesehen: der *Driver* führt die eigentliche Arbeit aus – der *Navigator/Observer* behält den Überblick, überdenkt strategische Konsequenzen und weist auf Schwachstellen hin.

Motivation

Mit wachsendem Interesse an *Extreme Programming* [6] – Paarprogrammierung ist hier eine der vorgeschriebenen Praktiken – häuften sich auch die Berichte über positive Erfahrungen mit Paarprogrammierung aus dem professionellen Umfeld. So sei der entstehende Code nahezu 100% defektfrei, die Aufgaben würden schneller erledigt und die Paare könnten länger arbeiten, ohne müde zu werden (z. B. [13][1] und [16, S. 107]).

Aber auch ohne diese „anekdotischen“ Belege hätten (und haben) im Laufe der Jahre sicherlich etliche Programmierer Erfahrungen der Arbeit zu zweit an einem Computer gesammelt. Die Grundidee ist schließlich denkbar einfach – und dass sich mit einem Partner an der Seite Arbeiten schneller bewältigen und Probleme eher lösen lassen, ist eine Erfahrung, die jeder individuell früher oder später macht (siehe auch [16, S. 107]). Allerdings kosten zwei Entwickler an einem Computer auch erst einmal doppelt so viel wie ein einzelner, sodass sich schnell die Frage nach der Wirtschaftlichkeit stellt: wiegen die beschriebenen Vorteile die doppelte Arbeitszeit auf?
Seit den 1990er Jahren wurden viele verschiedene Untersuchungen zur Klärung dieser Frage durchgeführt – mit gemischten Ergebnissen. Eindeutige Aussagen, wie sie für wirtschaftliche Entscheidungsträger wichtig sind, konnten so nicht erzielt werden.

[1] zitiert bei [42]

Um die Ursachen für die zum Teil widersprüchlichen Resultate zu ergründen, muss der *Prozess* der Paarprogrammierung besser verstanden werden; erst dann kann man sich wirklich konstruktive Gedanken darüber machen, wie der Prozess *verbessert* werden kann.

Vorgehensweise, Fokus und Zielsetzung dieser Arbeit

Durch den Fakt, dass bei der Paarprogrammierung Menschen miteinander interagieren, eröffnet sich ein breites Spektrum an möglichen Kommunikationsformen und -abläufen, die den Betrachungsrahmen der klassischen Informatik kräftig ausdehnen. Bevor solche Prozesse quantifiziert werden können, müssen sie zunächst einmal auf einer qualitativen Ebene verstanden werden. Da dies noch nicht umfassend der Fall ist, orientierte sich der Forschungsprozess dieser Arbeit an einer qualitativen Methode, genauer gesagt an der *Grounded Theory Methodology*, oder *GTM* (mehr zum konkreten Forschungsprozess in Kapitel 2).

Der Schwerpunkt dieser Arbeit wurde auf *Interventionen bei der Paarprogrammierung* oder allgemeiner: auf *Beeinflussungen der Aktitivät des Partners* gesetzt. Interventionen stellen hierbei nämlich nur *einen* Typus von Beeinflussungen dar. (Warum diese Verallgemeinerung sinnvoll ist, und welche Typen noch entdeckt wurden, wird in Kapitel 3 erläutert).
Dass bei der Untersuchung der Paarprogrammierung eine Fokussierung dringend nötig ist, hat sich nach den ersten, „kläglich gescheiterten“ Versuchen einer allumfassenden Sichtweise gezeigt (siehe [32]). Als eine Konsequenz daraus beschreibt Stephan Salinger [29, 31, 32] die *Basisaktivitäten* der Paarprogrammierung und schlägt vor, nachfolgende Analysen in Form aufeinander aufbauender Schichten zu gestalten. Hierfür wird die *Basisschicht* mit der *Basiskonzeptmenge* als Arbeitsgrundlage vorgelegt. Im Sinne dieser Metapher bestand die Zielsetzung dieser Arbeit in der Ausarbeitung einer (partiell) aufliegenden Schicht über der Basisschicht.

Aufbau dieser Arbeit

Diese Arbeit unterscheidet sich in ihrer Natur von „normalen“ Abschlussarbeiten der Informatik. Üblicherweise werden für Informatiker-Abschlüsse *ingenieursmäßige* Probleme gelöst, d. h. sie helfen, „ein praktisch relevantes Problem zu lösen“, wobei der Nutzen empirisch belegt wird [27]. Diese Arbeit hingegen hat einen *wissenschaftlichen* Fokus und ist auf die „Erweiterung des Wissens“ ausgelegt [27].
Die Tatsache, dass eine relativ „fachfremde“ wissenschaftliche Methode (die GTM) verwendet wurde und die Arbeit nicht über einen programmiertechnischen handwerklichen Anteil verfügt, führte zu einer Gliederung in die folgenden vier Teile:

I. Einführung: Im ersten Teil wird der Themenbereich dieser Arbeit, sowie das wissenschaftliche Umfeld, in das sie sich einbettet, vorgestellt. Dazu gehören ein Abriss über den aktuellen Stand der Forschung (Kapitel 1), eine Darstellung der konkreten Forschungsmethode und die Herkunft der verwendeten Daten (Kapitel 2).

II. Hauptteil: Im zweiten Teil werden die im Laufe des Forschungsprozesses gewonnenen Erkenntnisse in drei Kapiteln (3, 4 und 5) ausgebreitet.

III. Herleitung der Ergebnisse: Da die gewählte Forschungsmethode keinen perfekt geradlinigen Prozess garantiert, sondern zu jedem Zeitpunkt verschiedene Rich-

tungen ermöglicht, gab es zwangsläufig einige gedankliche Ausflüge, die nach einiger Zeit nicht weiter verfolgt wurden. Im dritten Teil werden die interessantesten der dabei entstandenen Ergebnisse präsentiert.

Unterteilt werden diese Resultate nach zwei „Evolutionsstufen“ der Forschungsarbeit in den Kapiteln 6 und 7.

IV. Schluss: Abschließend werden im vierten Teil die Resultate dieser Arbeit bewertet und eingeordnet (Kapitel 8). In Kapitel 9 werden Vergleiche bisherigen Erkenntnissen aus der Literatur geführt und schließlich rundet ein Ausblick in Kapitel 10 diese inhaltlichen Teil dieser Arbeit ab.

Ergänzend zum Inhalt gibt es noch folgende Anhänge:

- **Anhang A** beinhaltet exemplarische Screenshots aus den verwendeten Aufzeichnungen.
- **Anhang B** listet weitergehende Erkenntnisse und Forschungsideen auf, die während der Untersuchung entstanden sind, aber nur einen schwachen inhaltlichen Bezug zum Fokus dieser Arbeit hatten und deswegen nicht weiter verfolgt wurden.
- **Anhang C** fasst einige Hinweise zum verwendeten Transskriptionsschema zusammen.
- **Anhang D** enthält „Acknowledgements“, da in dieser Arbeit auch Material von außerhalb des Instituts verwendet wurde.

Teil I
Einführung

Im ersten Teil dieser Arbeit werden die für das Verständnis notwendigen Grundlagen gelegt.
Zunächst enthält Kapitel 1 einen Abriss über den aktuellen Forschungsstand. Im Anschluss beschreibt Kapitel 2 ausführlich den für diese Arbeit gewählten Forschungsprozess.

1 Stand der Forschung

Der Untersuchungsgegenstand dieser Arbeit waren Paarprogrammierungssitzungen. Die Ausrichtung und die Erkenntnisse der bisherigen Forschung auf diesem Gebiet (hauptsächlich von Informatikern) werden in diesem Kapitel zusammengetragen.

1.1 Geschichte der Paarprogrammierungsforschung

Bereits in den 1980er Jahren wurde verstärkt das gemeinsame Lernen (*„cooperative learning“*) von Kindern allgemein (z. B. von Brown & Palincsar [11]) und der „soziale Kontext des Programmieren-Lernens“ im Speziellen (z. B. Webb & Lewis [39]) untersucht. Dabei wurde festgestellt, dass das Zusammenarbeiten die Fähigkeiten zur Problemlösung zu steigern scheint.
Von diesem Effekt inspiriert, wurde die Grundidee für das akademische Umfeld aufgeriffen und Anfang der 1990er Jahre von Wilson et al. [46] experimentell überprüft. Dabei wurde die Qualität der erzielten Ergebnisse sowie die Zufriedenheit der paarweise arbeitenden Studenten mit denen von Einzelkämpfern verglichen und (wenn auch nicht uneingeschränkt) die vermuteten Vorteile für die gemeinsam Arbeitenden bestätigt. In den nachfolgenden Jahren wurden viele ähnliche Untersuchungen durchgeführt.
Während zu Beginn noch häufig von *„collaborative programming“* die Rede war, hat sich heute der Begriff *„pair programming“* oder eben *Paarprogrammierung* durchgesetzt.

1.2 Anekdotische Berichte über Paarprogrammierung

Die eingangs erwähnten Berichte über die Vorteil von Paarprogrammierung gegenüber der „Soloprogrammierung“ beziehen sich auf eine Reihe von Aspekten.

1. **Produktivität**: Paarprogrammierer produzieren mehr Code [16, Kap. 12]; sie arbeiten mehr als doppelt so schnell [42].

2. **Vielfalt**: Sie denken über mehr als doppelt so viele mögliche Lösungen nach wie zwei einzelne Entwickler [42][2].

3. **Qualität**: Sie produzieren besseren Code [16, Kap. 12]; es gibt nur ein geringes Risiko von hohen Defektraten [21, Kap. 9]

4. **Ausdauer**: Sie können länger arbeiten, ohne zu ermüden [16, Kap. 12].

[2] Betrifft dieses und das voherige Zitat aus [42]: Die Autoren Williams & Kessler beziehen sich hier auf Kent Beck. Allerdings konnte die genaue Quelle nicht ausfindig gemacht werden. Da es sich hier aber aller Wahrscheinlichkeit nach um anekdotische Ergebnisse und nicht die Resultate eines kontrollierten Experiments handelt, ist das an dieser Stelle nicht problematisch.

Einige dieser Aussagen werden begründet. So wirke z. B. Paarprogrammierung wie ein kontinuierliches Codereview und halte deswegen die Defektraten gering[3] [21, Kap. 9]. Außerdem sei es erstaunlich, wie viele offensichtliche, aber nicht bemerkte Defekte durch eine zweite Person entdeckt würden [42] (dieses Phänomen wird in Abschnitt 4.1.1 diskutiert).

1.3 Quantitative Erforschung der Paarprogrammierung

Ein Großteil der bisherigen Forschung zur Paarprogrammierung wurde quantitativ durchgeführt. Dabei wird der Paarprogrammierungsprozess als eine Einheit betrachtet wird, die in ihrer Gesamtheit durch verschiedene Maße wie etwa *Produktivität*, *Defektraten* oder *Zufriedenheit* charakterisiert werden soll.

Bei der Untersuchung des primären Kriteriums für die Entscheidung, ob Paarprogrammierung eingesetzt werden sollte, oder nicht – der Produktivität – liegen unterschiedliche Ergebnisse vor. Die Resultate reichen von „40% schneller“ [24], über „nur 15% mehr Zeit“ [45] bis hin zu „29% weniger Produktivität“ [37].
Bei der Messung von Defektraten (oder vielmehr dem Zählen von Programmversagen) haben Paare mal durchgehend *mehr* bestandene Testfälle [45], mal nach der Auslieferung dreimal so viele Defekte wie die Produkte der Solo-Vergleichgruppe [37].
Auch bei der Qualität der entwickelten Produkte erzielen Paare mal die gleiche Funktionalität mit weniger Programmzeilen [12] und mal gibt es keine Unterschiede in der Designqualität [37].
Worüber jedoch mehr Einigkeit herrscht, ist die Tatsache, dass Paarprogrammierung Spaß macht [45, 37] und dass die Programmierer mit ihren Lösungen zufriedener sind [45].

1.4 Qualitative Erforschung der Paarprogrammierung

Im Gegensatz dazu gibt es auch einige qualitative (oder qualitativ-quantitative) Forschungsansätze, die jeweils einzelne Aspekte der Paarprogrammierung untersuchen.
So beschreiben z. B. Williams & Kessler in [43] die Effekte „Pair Pressure“ und „Pair Learning“ in der Ausbildung von Software-Ingenieuren.

Die wesentlichen Grundlagen für diese Arbeit sind aber v. a. durch die Beiträge von Plonka [25] und von Salinger & Prechelt [31] zum 20. Workshop der *Psychology of Programming Interest Group*, sowie durch die Dissertation von Salinger [29] gelegt worden. Darin wird ein qualitativer Forschungsansatz vorgeschlagen, der den gesamten Paarprogrammierungsprozess erfasst.

[3] Im Original: *„[. . .] Pair Programming has a definite impact on the risk of a high-defect rate because pairing acts like a continuous code review.“*

2 Forschungsmethode

Im Vorwort dieser Arbeit wurde bereits erwähnt, dass für diese Arbeit eine qualitative Vorgehensweise gewählt wurde. Die Notwendigkeit dafür ergibt sich daraus, dass die Mechanismen, die dem Paarprogrammierungsprozess zu Grunde liegen, noch nicht gut genug verstanden sind, um die Unterschiede der Ergebnisse in den quantitiv ausgerichteten Untersuchungen erklären zu können.

Jeder Forschungsprozess greift ohne konkrete Datengrundlage ins Leere. Die Art und Herkunft des in dieser Arbeit verwendeten Materials wird in Abschnitt 2.1 beleuchtet. In Abschnitt 2.2 wird die Auswahl der qualitativen Methode diskutiert und in Abschnitt 2.3 dann die tatsächlich angewendete *Grounded Theory Methodology* erläutert. Die eingangs erwähnte *Basisschicht*, die das metaphorische Fundament für diese Arbeit bildet, wird in Abschnitt 2.4 in ihren Grundzügen dargelegt. Schließlich wird in Abschnitt 2.5 ergänzend zu den „theoretischen“ Vorgaben der GTM der tatsächliche Forschungsprozess inkl. zusätzlicher Praktiken zusammengefasst.

2.1 Datenbasis

Ein Großteil der Forschungsarbeit gründete sich auf der Analyse von umfangreichen Audio- und Videoaufzeichnungen. Für diese Arbeit wurden keine neuen Daten erhoben. Stattdessen wurde auf bereits existierende Bestände zurückgegriffen. In allen Fällen handelt es sich dabei um Aufnahmen von Sitzungen, in denen konkrete Probleme in Paararbeit gelöst werden mussten.

2.1.1 Material aus der Freien Universität

Art des Materials

Bei einem Großteil des Materials handelte es sich gewissermaßen um „Altbestände“ der *Arbeitsgruppe Software Engineering* am Fachbereich Informatik an der Freien Universität Berlin. Die selben Aufzeichnungen wurden (zum Teil) bereits in anderen Arbeiten untersucht (darunter [25, 26, 29, 30, 31, 32]).
Technisch betrachtet setzt sich das Material wie folgt zusammen:

- Hochauflösende Aufnahme des Bildschirms, auf dem die beiden Entwickler arbeiten. Dieser wurde mittels der Screencapture-Software *Camtasia* [36] mit einer Bildrate von 4 fps[4] aufgezeichnet. Zusätzlich wird am Bildrand ein Zeitstempel eingeblendet.
- Webcam-Video, das die beiden Entwickler entweder frontal oder von seitlich oberhalb zeigt. Die Auflösung dieser Bilder beträgt 320 x 240 Pixel (QVGA) bei einer Bildrate von 5 bis 15 fps. Das Webcam-Video ist in das Bildschirm-Video eingebettet, sodass beide Informationskanäle parallel untersucht werden können. Ein Standbild aus so einer Aufzeichnung ist im Anhang zu finden (Abbildung A.1).
- Tonaufnahmen der beiden Entwickler. Bei einigen Aufzeichnungen steht nur ein Mono-Kanal vom integrierten Webcam-Mikrofon zur Verfügung, bei anderen Aufnahmen kamen Ansteckmikrofone zum Einsatz, sodass die beiden Entwickler auf zwei Audiokanälen getrennt aufgenommen wurden.

[4] *Frames per Second*, Bilder pro Sekunde

Bei den vorhandenen Videodateien gab es an einigen Stellen einen leichten Versatz zwischen Bild und Ton, was für eine Analyse des Video-Materials nicht zwangsläufig ein Problem darstellen muss, solange sich die Asynchronität im Rahmen hält. Während der Untersuchung zeigte es sich jedoch, dass für das Verständnis der Abläufe kleine zeitliche Unterschiede von Bedeutung sein können – z. B. ob eine Äußerung *vor* oder *nach* dem Erscheinen einer Mitteilung auf dem Bildschirm begonnen wird. Aus diesem Grund wurden einzelne Videos über die hörbaren Tastenanschläge und Mausklicks neu synchronisiert[5].

Professionelle Entwickler

Ein Teil dieser Aufzeichnungen zeigt professionelle Entwickler, die in ihrem alltäglichen Umfeld gefilmt wurden (siehe [25] für nähere Informationen über den Aufzeichnungsprozess). Zusätzlich zu den Videoaufzeichnungen wurden von den Entwicklern vor und nach der Sitzung Fragebögen ausgefüllt. Außerdem wurden in einigen Fällen Interviews mit den Entwickler geführt, von denen die schriftlichen Notizen vorhanden sind.
Aus dieser Gruppe von Sitzungen wurden insgesamt sechs Sitzungen aus drei verschiedenen Unternehmen analysiert.

Studenten

Die Aufzeichnung der studentischen Sitzungen erfolgte im Rahmen einer J2EE-Lehrveranstaltung[6].
Im Gegensatz zu den professionellen Sitzungen war bei den studentischen Sitzungen die Aufgabe durch die Versuchleiter vorgegeben und sowohl die Problem- als auch die Lösungsdomäne bekannt. Für Analysen, die sich stark auf den Inhalt der Sitzung fokussieren, um z. B. die Qualität von Entscheidungsprozessen objektiv beurteilen zu können, ist das ein klarer Vorteil[7].
Aus diesem Umfeld wurde eine Sitzung analysiert.

2.1.2 Material aus dem Workshop

Während ich die Untersuchung des o. g. Materials bereits begonnen hatte, wurde ich über die *AG Software Engineering* auf eine E-Mail aufmerksam gemacht. Darin wurde für die Mitarbeit an einem 2012 erscheinenden Buch geworben, in dem der Designprozess bei der Softwareentwicklung aus vielen verschiedenen Blickwinkeln untersucht werden sollte. Als Datengrundlage wurden Videoaufzeichnungen und Transskriptionen von drei Sitzungen angeboten. Das Material wurde mir freundlicherweise auch ohne die feste Zusage eines Buchbeitrages zur Verfügung gestellt (siehe Acknowledgements im Anhang).

[5] Einige Videos hatten einen konstanten Versatz von ca. 1 Sekunde pro 90 Minuten; andere hatten sowohl positive als auch negative Verschiebungen. Für die Anpassungen wurde das Videoschnittprogramm *MAGIX Video Pro X3* [19] verwendet.

[6] *„Bau betrieblicher Informationssysteme mit Java2 Enterprise Edition“*, Freie Universität Berlin, Wintersemester 2005/06, https://www.mi.fu-berlin.de/kvv/course.htm?cid=590 (Letzter Aufruf: 01.02.2012)

[7] Im Rahmen der vorliegenden Arbeit kam dieser Vorteil allerdings nicht zum Tragen, da für Untersuchung wechselseitiger Beeinflussung der Aktivitäten kein vollständiges technisches Verständnis nötig ist. Es erleichtert zwar das Nachvollziehen der Gedankengänge der Beobachteten, aber letztlich reicht die Kenntnis der verwendeten Programmiersprachen und einiger grundlegender Paradigmen (wie z. B. Objektorientierung) aus, da ja in diesem Fall durch den Forscher keine Bewertung der Arbeitsergebnisse vorgenommen werden musste.

Im Gegensatz zu den bisherigen Sitzungen wurden hier keine Paar*programmierer*, sondern Paar*designer* beobachtet. Allen Teams wurde die gleiche Aufgabe gestellt, die in einem Zeitfenster von zwei Stunden bearbeitet werden sollte. Die Designerpaare hatten ein Whiteboard zur Verfügung, welches von den Versuchsleitern gefilmt wurde. Die gestellte Aufgabe verlangte einen Architekturentwurf für eine einfache Verkehrssimulation mit Straßenkreuzungen, Ampelschaltungen und variablen Verkehrsdichten. Bei den Softwaredesignern handelte es sich ausnahmlos um Profis, die in ihren jeweiligen Firmen ein hohes Ansehen genießen.
Aus den zur Verfügung stehenden Sitzungen wird hier Sitzung W02 untersucht. Ein Standbild dieser Sitzung ist im Anhang zu finden (Abbildung A.2).

Eigenschaften und Probleme des Datenmaterials

Im Gegensatz zu den Videoaufzeichnungen des FU-Materials lagen die Videos der Sitzungen in voller HD-Auflösung (also 1920x1080) vor, was zumindest theoretisch das Deuten von Mimik und Gestik erleichtern sollte. Gefilmt wurde aber nur das Whiteboard, sodass die beiden Designer nur sporadisch und oft nur von hinten zu sehen sind. Zusammen mit den Paardesignern war noch mindestens ein Interviewer anwesend, der auch manchmal zu hören, aber niemals zu sehen ist. Insgesamt ist die Ton-Qualität in Ordnung, sodass man die Äußerung meistens verstehen kann. Allerdings ist das Tonsignal oft mit Störgeräuschen wie konstant hohem Fiepen und Lüftergeräuschen überlagert.
Dass es überhaupt Transkriptionen zu den Sitzungen gibt, erschien zunächst fast unwirklich, weil das Transkribieren ein sehr zeitaufwändiger Prozess ist. Die tatsächlichen Transskriptionen können zwar nicht als Ersatz für die Videoaufzeichnungen dienen, stellen aber eine hilfreiche Ergänzung dar.

2.2 Auswahl der Forschungsmethode

2.2.1 Strukturalismus

Der Grundgedanke des Strukturalismus besteht darin, dass die Dinge nicht isoliert betrachtet werden dürfen, sondern stattdessen in einem größeren Kontext (der *Struktur*) gesehen werden müssen. Demnach ist es für die Bedeutung eines „Dings“ weniger entscheidend, was *in* ihm drinsteckt, sondern vielmehr, wie es sich nach *außen* hin, zwischen den anderen Dingen verhält.
Die Wurzeln des Strukturalismus liegen in der Denkweise des Linguisten Ferdinand de Saussure, für den *Wörter* für sich genommen keine Bedeutung hatten, da es keine inhärente Verbindung zwischen einem Wort und dem was es bedeutet gäbe[8]. Das führt dazu, dass Wörter nur über die Beziehung zu *anderen Wörtern* definiert werden können. Schließlich war Saussure der Ansicht, dass Sprache die Welt um uns herum erst erschafft. Durch die Art, wie wir die Dinge benennen, strukturieren wir sie und betten sie in einen höheren Kontext ein. Die Einteilung eines Kalenderjahres in vier Jahreszeiten oder die Benennung von Farben entlang des sichtbaren Spektrums sind Beispiele hierfür. [4, S. 38–43]

[8] Der Autor Peter Barry [4] merkt an dieser Stelle korrekterweise die Sonderrolle einiger Onomatopoetika an – also Wortmalereien wie „Kuckuck“. Aber selbst diese unterscheiden sich von Sprache zu Sprache (engl. „*cuckoo*“).

Aus forschungsstrategischer Sicht ist vor allem der letzte Punkt dieser Betrachtungsweise interessant: der *Benennung* eines Phänomens wohnt immer auch die *Festlegung* inne, was das Phänomen eigentlich ist. Einen „natürlichen“ Entitätenpool, aus dem Elemente entnommen und benannt werden können, gibt es also nicht. Bei der Beschreibung der *Grounded Theory Methodology* (genauer: in Abschnitt 2.3.1) werde ich darauf noch einmal zurück kommen.

2.2.2 Auswahl der Grounded Theory

Als Forschungsmethode wird die *Grounded Theory Methodology* (GTM) verwendet. Die GTM eignet sich für dieses Vorhaben, weil sie ohne Annahmen über die zu untersuchende Materie auskommt und durch den konkreten Bezug zu echten Daten belastbare Resultate liefert.
Konkret wird die Ausprägung nach Strauss & Corbin [34] verwendet, da sie im Gegensatz zur Fassung von Glaser & Strauss [14] einen „liberalen“ Umgang mit Literatur vorsieht (die nach Glaser & Strauss so weit wie möglich ignoriert werden sollte). Außerdem stellen Strauss & Corbin einen in Phasen unterteilten Forschungsprozess vor und bieten ein Kodierparadigma an (beides wird im weiteren Verlauf noch erläutert).

2.2.3 Ergänzungen zur Methode der qualitativen Analyse der Paarprogrammierung

Salinger, Plonka & Prechelt haben in [31] vier Praktiken für die qualitative Analyse des Paarprogrammierungsprozesses vorgestellt, die in Ergänzung zu den Praktiken der Grounded Theory eingesetzt werden können.

1. **Blickwinkel auf die Daten:** Die Beantwortung dreier Fragen soll dabei helfen, den Blick auf die Daten zu „schärfen“.

   (1) In welcher Hinsicht erwartet man von den Daten Einsicht?

   (2) Welche Art von Phänomenen erlauben sich die Forscher selbst in den Daten zu identifizieren?

   (3) Welche Art von Ergebnis sollte die Analyse erbringen?

2. **Syntaktische Regeln für Konzeptnamen:** Die Benennung der Konzepte soll einer Syntax folgen, weil die völlige Wahlfreiheit von Konzeptnamen zu verschiedenen Schemata führt, sodass die Namen schwieriger zu verstehen, zu erinnern und anzuwenden wären. Das Schema lautet wie folgt:

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

3. **Metamodell der Analysemethoden und -ergebnisse:** Die Motivation hinter einem Metamodell ist vereinfacht die Strukturierung des Prozesses und der Resulte. Das Metamodell für die Analyseergebnisse ist in Abbildung 2.1 als UML-Klassendiagramm dargestellt.

4. **Kodieren im Paar:** Die Idee hinter dieser Praktik ist, dass für alle wesentlichen Entscheidungen während der Analyse beide Personen übereinstimmen müssen. Dadurch wird z. B. die Perspektive auf die Daten konsistenter durch den Prozess hinweg eingehalten.

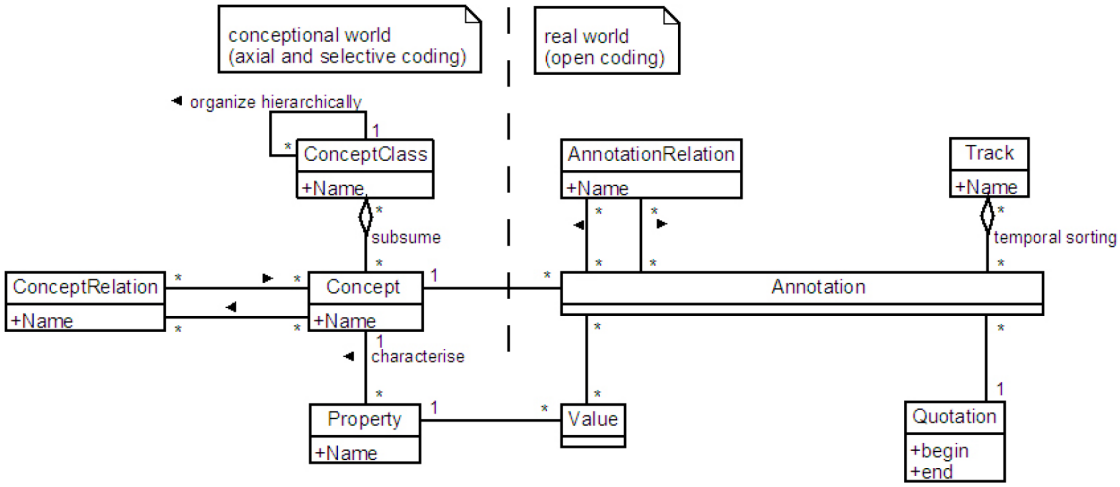

Abbildung 2.1: Metamodell der Analyseergebnisse (Praktik 2.), formuliert als UML-Klassendiagramm. Abbildung aus [31].

2.3 Forschungsprozess nach der Grounded Theory Methodology

Der Forschungsprozess nach der Grounded Theory Methodology umfasst im Wesentlichen die drei Phasen des *offenen*, des *axialen* und des *selektiven Kodierens*.

2.3.1 Offenes Kodieren

Während der ersten Phase nach der Datenerhebung wird das Material durchsucht und *Phänomene* benannt oder *konzeptualisiert* [34, S. 45]. Wie schon in Abschnitt 2.2.1 im Zusammenhang mit der strukturalistischen Denkweise von Saussure betont wurde, sind die *Benennung* eines Phänomens und die Festlegung seiner Grenzen eng miteinander verwoben. An dieser Stelle ist also der Blickwinkel auf die Daten ausschlagend für das Ergebnis der Konzeptualisierung.
Die Konzepte sind zu diesem Zeitpunkt erst einmal kaum mehr als Beschriftungen für die Phänomene. Der nächste Schritt nach Strauss & Corbin ist nun die Gruppierung gleichartiger oder ähnlich scheinender Konzepte zu *Kategorien*, die wiederum benannt werden müssen. Die so entstehenden Kategorien werden im weiteren Verlauf „entwickelt“, womit Strauss & Corbin zunächst die Identifizierung von Eigenschaften oder *Dimensionen* meinen [34, S. 50f]. Eine Kategorie kann mehrere Eigenschaften haben, und konkrete Phänomene haben entlang dieser *Dimensionen* eine bestimmte Ausprägung. Wenn sich innerhalb des so aufgespannten mehrdimensionalen Wertebereiches Phänomen-„Häufungen“ ergeben, können diese in spezielleren Kategorien zusammengefasst werden, die Strauss & Corbin als *Subkategorien* bezeichnen [34, S. 51].

2.3.2 Axiales Kodieren

In der Phase des axialen Kodierens werden die entwickelten Kategorien „auf neue Art“ zusammengesetzt. In dieser Phase wird der Schritt vom *Klassifizieren* hin zum *Interpretieren* gegangen [34, S. 75f].
Strauss & Corbin schlagen dafür vor, die einzelnen Phänomene im Rahmen eines kausalen Handlungsmodells – dem paradigmatischen Modell – zu betrachten, wobei *ursächliche Bedingungen* zu *Phänomenen* führen, die in einem bestimmten *Kontext* stattfinden. Darüber hinaus gibt es *intervenierende Bedingungen*, die die möglichen *Handlungs-*

strategien einschränken oder erweitern. Aus dem Handeln ergeben sich schlussendlich *Konsequenzen* (siehe [34, S. 78]).

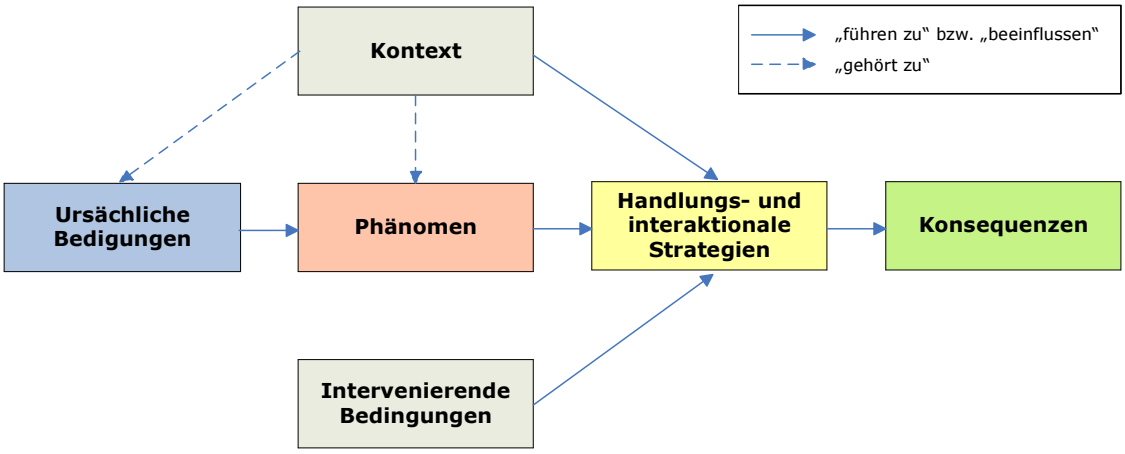

Abbildung 2.2: Darstellung des paradigmatischen Modells nach Strauss & Corbin [34]. Abbildung von Stephan Salinger [29, Abschn. 4.2.2]

Die Elemente, die an diese Positionen gesetzt werden, sind ebenfalls wieder Phänomene, sodass mit Hilfe dieses Modells Lücken in den bisher bekannten Kategorien entdeckt werden können. Da die Erkenntnisse aus dieser Betrachtungsweise für jedes einzelne Phänomen nicht isoliert bleiben, sondern mit anderen Phänomenen der gleichen Kategorie verglichen werden müssen, können so auch Zusammenhänge zwischen den jeweiligen dimensionalen Ausprägungen der Kategorien an den einzelnen Positionen festgestellt werden[9] [34, S. 77f].

2.3.3 Selektives Kodieren

In der Phase des selektiven Kodierens werden dann die kausalen Zusammenhänge einzelner Phänomene auf eine höhere Ebene gebracht und so die gesammelten Erkenntnisse zu einer Theorie integriert. Strauss & Corbin bezeichnen eine der wichtigsten Tätigkeiten in dieser Phase als das „Erzählen einer Geschichte" und das „Darlegen des Roten Fadens der Geschichte". Hierbei soll versucht werden, eine narrative „Darstellung über das zentrale Phänomen der Untersuchung" zu finden, um dann aus den entwickelten Kategorien die *Kernkategorie* zu *selektieren*. Alle anderen Phänomene müssen dann um diese Kernkategorie angeordnet und integriert werden. [34, S. 94f]

2.3.4 Praktiken bei der GTM

Theoretisches Sampling

Unter dem Begriff „Theoretisches Sampling" (engl. *theoretical sampling*) wird die Auswahl von Daten (z. B. durch weitere Datenerhebungen) unter der Beachtung der bisherigen Erkenntnisse verstanden. Das Ziel ist hierbei nicht, alle existierenden Fälle zu untersuchen (oder zumindest repräsentativ, sodass jeder Fall die gleichen „Chancen"

[9] Um diese abstrakten Aussagen mit einem (etwas) weniger abstrakten Beispiel aufzufrischen: Nachdem festgestellt wurde, dass die $\mathcal{P}$*-Phänomene* v. a. durch $\mathcal{U}$*-Ursachen* entstehen, kann man danach z. B. überprüfen, ob die $\mathcal{U}$s mit einer lauten Stimme eher zu $\mathcal{P}$s mit einer *roten* oder eher zu einer *blauen* Färbung führen.

hat, untersucht zu werden). Stattdessen erfolgt die Auswahl der Stichprobe aufgrund von Konzepten, die für „bedeutsam“ gehalten werden. [34, S. 148]
Die Beschreibung, wie theoretisches Sampling im Laufe dieser Arbeit angewendet wurde, folgt in Abschnitt 2.5.4.

Ständiges Vergleichen

„Ständiges Vergleichen“ (engl. *constant comparison*) ist eine Kernkomponente der Grounded Theory Methodology, sodass sie auch den Beinamen „Analysemethode der ständigen Vergleiche“ hat [34, S. 44]. Gemeint ist damit, dass während der Analyse die konzeptualisierten Phänomene, bzw. die Kategorien immer wieder miteinander verglichen werden müssen, um die Unterschiede und Gemeinsamkeiten zwischen ihnen auch entdecken zu können. Diese Praktik durchzieht den gesamten Forschungsprozess durch alle Phasen hinweg.

2.4 Basisschicht und Basiskonzeptmenge

Die Dissertation von Stephan Salinger [29] schlägt neben der Verwendung der GTM ein geschichtetes Modell vor, dessen Grundlage die Basisschicht mit der Basiskonzeptmenge ist, die alle elementaren Aktivitäten während des PP-Prozesses beschreiben. Über dieser sollen weitere Lagen (Layer) geschichtet werden, die jeweils andere Perspektiven auf den PP-Prozess einnehmen. Das Modell, das in dieser Arbeit entwickelt werden soll, wäre ein solcher Layer.

Hinweise: In dieser Arbeit kann die Basisschicht aus praktischen Gründen inhaltlich nur angerissen werden. Insbesondere bei der Darstellung der Basiskonzepte (größtenteils sogar nur die *Klassen* der Basiskonzepte) musste stark vereinfacht werden. Die Konzepte, die für das Verständnis der nachfolgenden Kapitel besonders wichtig sind, werden im Anschluss an die Übersichten detaillierter beschrieben. Als einzige echte Referenz kann nur die Dissertation von Stephan Salinger [29] gelten.

Weiterhin ist die einzige Literaturquelle für diesen Abschnitt (2.4) besagte Dissertation von Stephan Salinger [29]. Ausnahmsweise wird für diesen Abschnitt daher eine abweichende Kennzeichnung von Zitaten vereinbart, da praktisch *alle* Aussagen dieses Abschnitts inhaltlich aus einer einzelnen Quelle entnommen wurden. Ich lasse daher die Angabe von Abschnitt und Nummer in eckigen Klammern bei indirekten Zitaten weg; wörtliche Zitate werden nach wie vor durch Anführungszeichen gekennzeichnet.

2.4.1 Zielstellung und Grundprinzipien der Basisschicht

Die *Basisschicht* (BS) soll „einen Ausgangspunkt für das Verständnis der Abläufe des Paarprogrammierungsprozesses zur Verfügung [stellen]“. Die dafür entwickelte *Basiskonzeptmenge* (BKM) beschreibt dabei mit ihren *Basiskonzepten* die elementaren Aktivitäten dieses Prozesses.
Die Benennung der Basiskonzepte folgte dabei einer Syntax, sodass immer ein Verb und ein Objekt zusammengesetzt werden.

Unterscheidung zwischen Tätigkeitswelt und Diskurswelt

Die Objektbestandteile der HHI-Konzepte (siehe nächster Abschnitt) beziehen sich im Rahmen der Basisschicht immer auf die *Diskurswelt*. Damit sind „Formulierungen von Gedanken, Wissen oder Erkenntnissen über einen Sachverhalt“ gemeint. Dem gegenüber steht die *Tätigkeitswelt*, die die Sachverhalte selbst beinhaltet.
So gibt es z. B. das Konzept *propose_step*, mit dem der Sprecher einen zukünftigen Arbeitsschritt beschreibt bzw. vorschlägt. Als *step* wird hier nicht etwa die tatsächliche Arbeit bezeichnet, die diese Person oder sein Programmierpartner verrichten, sondern die Artikulation eines Arbeitsschrittes.
Analog dazu wird unter *source of information* „nicht eine Informationsquelle an und für sich, sondern das Wissen einer Person eben über diese“ verstanden.

Erfassen der primären Intention, Darstellung von Bezügen

Zwei der wesentlichen Grundprinzipien bei der Anwendung der Basiskonzeptmenge sind das *Erfassen der primären Intention* und das *Darstellen von Bezügen*.
Mit Ersterem ist gemeint, dass nach Möglichkeit durch eine Konzeptualisierung einer Äußerung das erfasst wird, „was die Person sagen möchte“. Salinger nennt in [29, Exkurs 3] das anschauliche Beispiel: *„Wollen wir einen Kaffee trinken?“*, was natürlich eine Frage ist, aber die primäre Intention dieser Frage ist der darin verpackte *Vorschlag*.
Mit dem *Darstellen von Bezügen* sollen in der Interaktion vorhandene Verweise nach Möglichkeit auch in der Konzeptualisierung erhalten bleiben. Dadurch sollen z. B. „Frage-Antwort-Paare oder Entscheidungsfindungsprozesse [. . .] sichtbar gemacht werden.“ In der Basisschicht erfolgt dies durch die Verwendung von entsprechenden Verbkombinationen wie *ask–propose* oder *propose–challenge–decide*.

2.4.2 Aufbau der Basisschicht

Schematisch ist der Aufbau der Basisschicht in Abbildung 2.3 dargestellt. Als gröbste Unterteilung kann die Differenzierung in die folgenden Klassen angesehen werden:

HHI-Konzepte[10] umfassen die Konzepte, die die Interaktionen zwischen beiden Paarmitgliedern beschreiben.

HCI/HEI-Konzepte[11] beschreiben alle Aktivitäten, die unter Einsatz von Tastatur und Maus erfolgen und sich auf den Computer beziehen, bzw. alle sonstigen Aktivitäten, die die Arbeitsumgebung betreffen. (Abschnitt 2.4.2.5)

Die mit Abstand umfangreichere Klasse bilden die HHI-Konzepte. Sie unterteilen sich wiederum in folgende „Hauptklassen“:

1. **Produktorientierte Konzepte** beziehen sich auf die Gestaltung des zu entwickelnden Programms. (Abschnitt 2.4.2.1)

2. **Prozessorientierte Konzepte** hingegen stehen im Zusammenhang mit der Gestaltung des Arbeitsprozesses (Abschnitt 2.4.2.2)

3. **Universelle Konzepte** fassen Äußerungen zu Wissenstransfer, Wissenslücken und Hypothesen zusammen. Sie können sich dabei sowohl auf das Produkt als auch auf den Arbeitsprozess beziehen. (Abschnitt 2.4.2.2)

[10] von engl. *human-human interaction*
[11] *human-computer interaction* und *human-environment interaction*

4. **Fassadenkonzepte** sind spezielle universelle Konzepte. Sie bilden eine Abstraktionsebene über den anderen Konzepten. (Abschnitt 2.4.2.4)

5. **Sonstige Konzepte** beschreiben unwichtige oder unverständliche Äußerungen. (werden hier nicht näher erläutert)

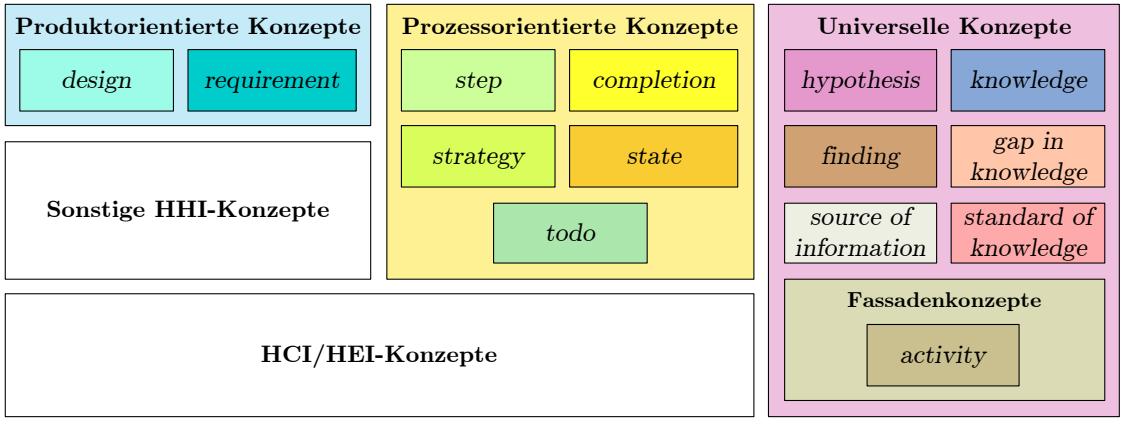

Abbildung 2.3: Die Basisschicht besteht aus den HCI/HEI-Konzepten (unten links) und den HHI-Konzepten (restliches Schema). Innerhalb der HHI-Konzepte gibt es drei große Gruppen (produktorientierte, prozessorientiert und universelle Konzepte) sowie zwei sonstige Konzepte *mumble_sth* und *say_off topic*.

Vereinfachtes und ergänztes Bild nach [29, Abb. 7.3]

2.4.2.1 Produktorientierte Konzepte

Die produktorientierten Konzepte umfassen die Unterklassen *design* und *requirement*.

Äußerungen, die mit Konzepten der Klasse *design* annotiert werden, befassen sich mit „Gestaltungsmöglichkeiten von Programmartefakten“. Hier wird ein sehr weit gefasstes Verständnis von „Design“ verwendet; eines, das Teile von einzelnen Codezeilen, aber auch die gesamte Architektur des Produkts umfasst.
Als *requirements* werden Anforderungen an das Programm definiert. Sie grenzen also die Gestaltungsmöglichkeiten ein, während für den *design*-Begriff gerade diese Freiheit essentiell ist.

Beispiele für je ein Konzept dieser beiden Klassen finden sich in Tabelle 2.1.

Konzept	Äußerung	Kommentar
propose_design	„Und wie nennen wir das Ding? `$id`?“	Der Sprecher schlägt die Benennung einer Variablen vor.
remember_-requirement	„Und die andere Sache, ich hatte es halt so verstanden, dass wir halt versuchen wollen, das über die Fassade nach außen für alle anderen Geschichten unsichtbar zu machen.“	Der Sprecher verweist auf eine Absprache.

Tabelle 2.1: Beispiele für die Klassen der produktorientierten Konzepte

2.4.2.2 Prozessorientierte Konzepte

Die prozessorientierten Konzepte „adressieren [...] Äußerungen, die sich auf die Gestaltung des Arbeitsprozesses des Paares beziehen“ und umfassen die Klassen *step* und *strategy*, sowie *completion* und *state*. Weiterhin gibt es die Klasse *todo*.

Ein *step* ist ein Arbeitsschritt (oder genauer: die Vorstellung davon). Für den Sprecher sind *step*s die atomaren Handlungseinheiten. Die *todo*s sind ähnlich ausgelegt wie *step*s, allerdings betont der Sprecher hierbei, dass der entsprechende Arbeitsschritt erst in der Zukunft und nicht sofort ausgeführt werden soll.
Eine *strategy* ist ein komplexeres Konstrukt und beschreibt „längerfristig ausgerichtetes, planvolles [...] Handeln“, das i. d. R. aus mehreren Teilschritten besteht.
Die Konzepte *completion* und *state* korrespondieren mit der Unterteilung in *step* und *strategy* und stellen Einschätzungen oder Bewertungen zum aktuellen Stand der Bearbeitung eben dieser dar.

In Tabelle 2.2 sind ein paar veranschaulichende Beispiele aufgeführt.

Konzept	**Äußerung**	**Kommentar**
agree_step	„Ich würd' mich da schon gerne rantrauen.“	Der Sprecher bezieht sich auf einen im Vorfeld gemachten Vorschlag für den nächsten Arbeitsschritt.
ask_strategy	„Wie fangen wir da an?“	Der Sprecher fragt seinen Partner vielleicht auch nach einem taktischen Vorschlag, weil dieser aber mit einem strategischen Vorschlag antwortet, wurde die Frage entsprechend kodiert.
explain_-completion	„Mir gefällt das alles immer noch nicht so richtig gut.“	Der Sprecher fasst die Einschätzung über den Erfolg beim aktuellen Arbeitsschritt zusammen.
explain_state	„Ja, jetzt sind wir streng genommen fertig, ne?“	Der Sprecher fasst die Einschätzung über den Erfolg beim Abarbeiten des gesteckten Sitzungsziels zusammen.
agree_todo	„Ja, das sollten wir auf jeden Fall auch noch machen.“	Der Partner hatte zuvor einen Vorschlag geäußert (*propose_todo*), dem der Sprecher hiermit zustimmt.

Tabelle 2.2: Beispiele für die Klassen der prozessorientierten Konzepte

2.4.2.3 Universelle Konzepte

Die universellen Konzepte können sowohl auf Äußerungen angewendet werden, die das Produkt, als auch den Prozess betreffen. Sie unterscheiden verschiedene „Formen“ von Wissen. Im Einzelnen gibt es die *source of information*, *gap in knowledge*, *standard of knowledge*, *knowledge*, *finding* und *hypothesis*.

Eine *hypothesis* ist eine Aussage, bei der der Sprecher deutlich macht (oder es deutlich wird), dass er sich nicht hundertprozentig sicher ist, ob sie auch tatsächlich wahr ist. Man könnte sie auch mit *Vermutung* umschreiben.

Unter dem *standard of knowledge* wird der Wissenstand einer Person verstanden. Wenn ein Programmierer z. B. sagt, dass er eine Idee hat, dann ist dies eine Äußerung zu seinem Wissensstand, da die eigentliche Idee ja nicht genannt wird.
Ein *gap in knowledge* ist eine Wissenslücke, die bei beiden Programmierern vorhanden ist, die das weitere Vorgehen behindert und derer sich die Entwickler auch bewusst sind.
Unter *findings* werden im Rahmen der Basisschicht neue Erkenntnisse verstanden – zumindest dann, wenn diese nicht in eine der bisher genannten Klassen zuzuordnen ist.
Einer der kompliziertesten Begriffe der Basisschicht ist *knowledge*. Im Wesentlichen bezeichnet er aber das Wissen, das im Rahmen einer Programmiersitzung mit der Illokution „Erklären“ transferiert wird, und nicht in eine der o. g. Klassen fällt[12]

In Tabelle 2.3 finden sich einige illustrierende Beispiele.

Konzept	**Äußerung**	**Kommentar**
remember_source of information	„Äh, muss ich in meinen E-Mails kurz nachschauen.“	Der Sprecher stellt fest, dass die vom Partner erfragte Information in seinen E-Mails steht.
explain_gap in knowledge	„Wir wissen nicht, ob das, was wir in der Datei ändern (..) bis zum <**Servername**> durchdringt“	Der Sprecher hält stellvertretend für das Paar fest, dass eine für das weitere Vorwärtskommen wichtige Information fehlt.
explain_standard of knowledge	„Keine Ahnung, was das Ding macht“	Der Sprecher stellt klar, dass er nicht weiß, wie eine bestimmte Methode zu benutzen ist.
ask_knowledge	„Aber <**Programm**> ist schon noch drauf?“	Der Sprecher fragt, ob der Partner ein bestimmtes Programm installiert hat.
explain_finding	„Das SQL-(~) brauchen wir nicht mehr. Wir haben ja nur 'ne 'ne Memcache-Abfrage.“	Der Sprecher äußert hier eine Erkenntnis und parallel noch ein *propose_design* (hier nicht kodiert).
propose_-hypothesis	„Also ich glaub nicht, dass wir am FeatureProxy was ändern müssen.“	Der Sprecher vermutet (ist sich nicht sicher), dass eine bestimmte Änderung nicht vorgenommen werden muss.

Tabelle 2.3: Beispiele für die Klassen der universellen Konzepte

2.4.2.4 Fassadenkonzepte

Die Fassadenkonzepte unterscheiden sich von den anderen Konzepten, da sie von vornherein auf Doppel- oder Mehrfachkodierungen ausgelegt sind. Die Basisschicht enthält nur eine einzelne Konzeptklasse: die *activity*-Konzepte.
Eine *activity* ist eine gerade ablaufende HCI- oder HEI-Aktivität und demzufolge beziehen sich die *activity*-Konzepte auch genau auf diese Aktivitäten. Da die activity-Konzepte im weiteren Verlauf der Arbeit in Unterkapitel 3.1 noch ausführlicher diskutiert werden, sei an dieser Stelle nur auf auf die Übersichtstabelle 2.4 verwiesen.

[12] Genau genommen wird dieses Wissen in der Basisschicht mit W_k bezeichnet. Die Konzepte der Klassen *knowledge* adressieren dieses Wissen nur.

Konzept	Beschreibung
think aloud_activity	Verbalisieren der gerade ablaufenden HCI/HEI-Aktivität (z. B. um den Partner „auf dem Laufenden“ zu halten)
disagree_activity	Widerspruch gegen eine HCI/HEI-Aktivität ohne Gegenvorschlag.
stop_activity	Eine HCI/HEI-Aktivität soll abgebrochen werden.
challenge_activity	Widerspruch gegen eine HCI/HEI-Aktivität und gleichzeitiger Gegenvorschlag.
agree_activity	Zustimmung zu einer HCI/HEI-Aktivität.
amend_activity	Zustimmung und Ergänzung einer HCI/HEI-Aktivität.

Tabelle 2.4: Übersicht über die Fassadenkonzepte. Die Basisschicht kennt nur eine einzige Klasse von Fassadenkonzepten – die *activity*-Konzepte.

2.4.2.5 HCI/HEI-Konzepte

Die HCI/HEI-Konzepte adressieren *Handlungsentitäten*. Eine Handlungsentität ist dabei „eine Folge von (atomaren) Tätigkeiten, [. . .] die durch ein kurzfristiges Handlungsziel bestimmt ist.“ Ein solches Handlungsziel kann z. B. sein, einen eben besprochenen Designvorschlag umzusetzen. Das Konzept *write_sth* bezeichnet dabei nicht nur das physische Tippen des Programmierers, „sondern die Schreibhandlung, die das Tippen beinhaltet.“

In Tabelle 2.5 sind die HCI/HEI-Konzepte der Basisschicht zusammengefasst und (soweit zum Zeitpunkt des Schreibens dieser Arbeit bekannt) kurz erläutert. Für tiefergehende Informationen verweise ich (wie generell in diesem Abschnitt) auf [29].

Kode	Beschreibung
write_sth	Schreibhandlungen, die direkt oder indirekt zum Programmcode gehörende Dokumente editieren.
search_sth	Suche nach wohldefinierten Objekten.
explore_sth	Erkundung von Artefakten, Dokumenten, etc. (keine Suche).
verify_sth	Überprüfung von Modifikationen an Artefakten, Dokumenten, etc.
read_sth	Vorlesen von Texten (Bildschirminhalte oder physische Texte).
show_sth	Zeigen auf Dinge (mit dem Finger oder der Maus).
do_sth	sonstiges

Tabelle 2.5: Übersicht über die HCI/HEI-Konzepte

2.4.3 Verwendung der Basisschicht

Die Basisschicht dient als sprichwörtliches Fundament für spezialisierte Analysen des Paarprogrammierungsprozesses. Hierfür sollen die Basiskonzepte ausdifferenziert, verallgemeinert, erweitert, beschnitten oder zur Verbesserung der *theoretischen Sensibili-*

tät[13] eingesetzt werden.

Im nächsten Abschnitt wird u. a. detaillierter beschrieben wie die Basisschicht und die Basiskonzeptmenge tatsächlich für diese Arbeite verwendet wurden.

2.5 Forschungsstrategische Aspekte dieser Arbeit

Zu Beginn dieses Abschnitts werden der geplante (2.5.1) und der tatsächliche Einstieg (2.5.2) in die Untersuchung einander gegenübergestellt und diskutiert. In Abschnitt 2.5.3 wird darüber hinaus die Verwendung der Basisschicht im Rahmen dieser Arbeit thematisiert.
Im Laufe der Untersuchung gewonnene Zwischenergebnisse, die im Endeffekt nicht weiter verfolgt wurden, werden detaillierter nach den eigentlichen Ergebnissen der Arbeit in Teil III betrachtet.

2.5.1 Geplanter Einstieg in die Untersuchung

Der geplante Einstieg in die Bearbeitung des Themas umfasste mehrere Schritte. Zunächst war eine methodische Einarbeitung vorgesehen, die sowohl das Erlernen der Grundlagen der *Grounded Theory Methodology*, als auch den Umgang mit der Software *Atlas.TI*[14] [22] umfasste. Im weiteren Verlauf sollte dann die *Basisschicht* verstanden werden, um eine Verständigung mit dem Betreuer „auf Augenhöhe“ zu ermöglichen, was natürlich das Studium seiner Dissertation [29] voraussetzte.
Vom zur Verfügung stehenden Videomaterial sollten vorerst nur die Sitzungen der professionellen Entwickler und erst im späteren Verlauf die in der Freien Universität Berlin aufgezeichneten studentischen Sitzungen untersucht werden. Innerhalb einer Sitzung sollten interessante Phänomene („Inseln“) ausgemacht werden und der zeitliche Kontext (links- und rechtsseitig) näher betrachtet werden.
Der weitere Verlauf sollte sich nach der GTM richten, war aber zu Beginn noch nicht näher spezifiziert.

2.5.2 Tatsächlicher Einstieg in die Untersuchung

Parallel zur GTM-Lektüre in [34] und [29, Kap. 4] wurde bereits die erste Sitzungsaufzeichnung (Sitzung D06) angesehen, um sich einen ersten Eindruck bilden zu können, wie ergiebig das Material tatsächlich ist. Dabei wurde allerdings kein bestimmter Blickwinkel eingenommen, da zu diesem Zeitpunkt noch gar kein Arbeitsthema ausgewählt war. Die Festlegung auf ein Thema manifestierte sich später in einem kurzen Exposé, das mit dem Betreuer diskutiert wurde.
Die nächste Phase bildete die Einarbeitung in die BS/BKM, zunächst durch das einführende Kapitel, in dem die Fragestellung, der gewählte Blickwinkel und die Herangehensweise bei der Entwicklung der BS/BKM erläutert wurden [29, Kap. 6] – später dann durch das nächste Kapitel, das sich den HHI-Konzepten widmet [29, Kap. 7].
In den darauffolgenden Wochen wurden sowohl die Grundideen, als auch die konkreten Konzepte der Basisschicht intensiv mit dem Betreuer diskutiert und so Unklarheiten beseitigt und Verständnisprobleme geklärt. Parallel dazu wurde versucht, die entdeckten

[13] Die *theoretische Sensibilität* bedeutet im Umfeld der GTM soviel wie: Fertigkeit, relevante Phänomene im zur Verfügung stehenden Material auszumachen.
[14] Atlas.TI ist ein Programm, das die qualitative Datenanalyse unterstützt – also eine sog. QDA-Software (*Qualitative Data Analysis*).

„Inseln“ aus den Sitzungen mit den jeweils schon zur Verfügung stehenden Basiskonzepten zu kodieren.

Im Gegensatz zum geplanten (idealisierten) Ablauf wurden die einzelnen Schritte also nicht sequenziell, sondern vielmehr parallel durchgeführt. Die sofortige praktische Anwendung der BKM lieferte dabei häufig den Anstoß für die Diskussionen und ermöglichte so ein viel tiefergehendes Verständnis, als es die „trockene“ Lektüre allein vermocht hätte.

Aus Material-Sicht hat sich im Laufe der Bearbeitung eine ungeplante Erweiterung ergeben, da sich zusätzlich zu den von vornherein bekannten Daten auch noch eine Möglichkeit ergeben hatte, an Audio- und Videoaufzeichnungen von mehreren Paardesignern zu gelangen, die eine von den Forschern gestellte Aufgabe jeweils zu zweit an einem Whiteboard in vorgegebener Zeit lösen mussten (siehe Abschnitt 2.1.2). Die Erkenntnisse aus der Untersuchung dieses zusätzlichen Materials führte letztendlich zu der gesonderten Betrachtung von kreativen Prozessen bzw. Phasen (siehe Abschnitt 5.2.8). Handwerklich gesehen handelt es sich bei der Verwendung dieses zusätzlich Materials übrigens um einen *weithergeholten Vergleich* [34, S. 69]. Strauss & Corbin empfehlen diese Praktik, um sich von Denkblockaden zu befreien – in dieser Arbeit eröffnete sie (auch ohne vorherige Blockade) einen neuen Blickwinkel auf die Daten.

Die Praktik „Pair Coding“ (Kodieren im Paar) aus [31] (siehe Abschnitt 2.2.3) wurde – nicht zuletzt weil es sich bei dieser Arbeit ja auch um einen Beleg selbstständiger Arbeitsweise handelt – nicht für die eigentliche Untersuchung angewendet. Dennoch kam sie in der Vorbereitungsphase zum Einsatz. So gab es einige Sitzungen, in denen ich mit meinem Betreuer bereits BKM-kodierte Episoden neu begutachtete, um die BKM „unter Aufsicht“ ausprobieren zu können.

2.5.3 Verwendung der BS/BKM

Anwendung der BS/BKM allgemein

Schon früh, während der ersten Kodierphasen, wurde festgestellt, dass es sowohl Äußerungen gibt, die sich nahezu prototypisch mit den Basiskonzepten annotieren lassen, als auch solche, die zumindest auf den ersten Blick *mehrere* potenzielle BS-Kodierungen oder auch *gar keine* erhalten könnten. Die genauere Betrachtung dieser „nicht-kanonischen“ Äußerungen führte häufig zu interessanten Gedankengängen, an deren Ende nicht unbedingt eine „formal korrekte“ Kodierung der Daten mit den Basiskonzepten standen, sondern vielmehr weiterführende Ideen und Einsichten. Zur Veranschaulichung folgt nun ein Beispiel:

> Der Observer hat seinen Vorschlag, kurz nachdem er diesen unterbreitet hat, widerrufen (*disagree_step* nach einem *propose_step*). Weil bei der entsprechenden Aufzeichnungen Bild und Ton nicht synchron waren und das *propose_step* das Neuladen einer lokalen Webseite beinhaltete, konnte nicht mit Sicherheit gesagt werden, ob das *disagree_step* noch vor, oder schon nach der Ausführung des vorgeschlagenen Schrittes geäußert wurde. Das Problem bei der Kodierung ergab sich nun, dadurch, dass sich *step*-Konzepte auf den nächsten, noch *durchzuführenden* Arbeitsschritt beziehen, dieser konkrete Arbeitsschritt (ein Tastendruck auf „F5“) schon abgeschlossen war.

Im Gespräch mit meinem Betreuer wurde dann aber festgestellt, dass eine derartig „formal korrekte“ Kodierung nicht das *Ziel* der Untersuchung sei. Wichtiger seien die dabei entstehenden Einsichten und weiterführenden Ideen.

Demnach haben sich einerseits die *Inhalte* der Basiskonzepte als hilfreich erwiesen, andererseits aber auch die *Grenzen* zwischen ihnen. Mit einer (hypothetischen) gröberen Gliederung der Basiskonzeptmenge wäre der im obigen Beispiel genannte Fall gar kein Grenzfall mehr gewesen und hätte demnach auch nicht zu weiterführenden Überlegungen angeregt.

Anwendung der *activity*-Konzepte der BS/BKM

Im Laufe der Untersuchungen wurden zwei Arten von Limitierungen durch die Basisschicht deutlich. Einerseits gab es Äußerungen, die nach dem Blickwinkel dieser Arbeit so verschieden waren, dass sie mit *unterschiedlichen* Konzepten annotiert werden mussten, während die selben Äußerungen im Rahmen der Basisschicht Vertreter des *gleichen* Konzepts wären. Anderseits gab es auch genau den umgekehrten Fall, bei dem die Basisschicht eine Differenzierung vornahm, die im Rahmen dieser Arbeit nicht vorhanden war.
Beide Beschränkungen waren nicht überraschend und entsprachen den Erwartungen. Bei Kodierungen mit den Basiskonzepten kann es vorkommen, dass Unterschiede nur durch weitere Konzepteigenschaften (*Dimensionen*) sichtbar werden, oder ganze Phänomene gar nicht kodiert werden können, weil die Basiskonzeptmenge eine andere (feinere) Granularität aufweist.
Auffälligkeiten beider Arten führten dazu, dass Überlegungen zu Ausdifferenzierungen bzw. Zusammenlegungen von Konzepten angestellt und überprüft werden mussten. In den ersten Wochen der Untersuchung führte das zu einem Satz von Heuristiken, die dafür entwickelt wurden, die – zu diesem Zeitpunkt in [29] noch nicht beschriebenen – *activity*-Konzepte konsistent anwenden zu können. Diese Heuristiken waren als Paraphrasierungen der Äußerungen der Entwickler konzipiert: Wenn sich die Äußerung sinngemäß mit dem entwickelten Satzschema umschreiben ließ, dann war das Kriterium für die Annotierung mit dem entsprechenden Konzept erfüllt.

Ein Beispiel für eine solche Heuristik wäre die folgende Paraphrasierung für das Basiskonzept *disagree_activity*:

> *„Ich bin mit dem, was du gerade tust, nicht einverstanden.“*

Oder für das Konzept *challenge_activity*:

> *„Ich bin mit dem, was du gerade tust, nicht einverstanden. Beende deinen aktuellen Schritt sofort, und mach stattdessen folgendes: . . . “*

Diese Paraphrasierungen halfen beim Umgang mit den Basiskonzepten, konnten aber nicht alle Probleme lösen. Eines dieser Probleme war die Vielfältigkeit der Zeitpunkte, zu denen der Sprecher seine Äußerung tätigen konnte: so kann die Aktivität, auf die sich der Sprecher bezieht, gerade ablaufen oder gerade eben beendet sein, oder lediglich vom Sprecher antizipiert sein. Weiterhin gibt es auch Äußerungen der Entwickler, die keinerlei Bewertung der Aktivität ihres Partners enthalten, sondern diese lediglich unterbrechen (mehr dazu im Hauptteil dieser Arbeit).

Kurzum: die *activity*-Konzepte der Basisschicht reichten nicht aus, um die vielen speziellen Aspekte der betrachteten Äußerungen zu erfassen, zeigten aber durch diese Limitierung deutlich auf, wo die Untersuchung im Rahmen dieser Arbeit ansetzen mussten.

Insgesamt war die Beschäftigung mit der Basisschicht an sich vielleicht eingehender, als es für diese Arbeit zwingend notwendig gewesen wäre, allerdings wurde so auch die allgemeine *theoretische Sensibilität* erhöht, was den weiteren Verlauf der Untersuchung sicherlich positiv beeinflusste.

2.5.4 Theoretisches Sampling

Was sich hinter *theoretischem Sampling* verbirgt und das diese als eine Praktik in der GTM vorgesehen ist, wurde bereits in Abschnitt 2.3.4 erläutert. Die „echte“ Form eines derartigen Samplings würde die gezielte Erhebung von weiteren Daten erfordern, was im Rahmen dieser Arbeit nicht praktikabel war.
Allerdings kann durch die Auswahl *welche* Ausschnitte aus den Daten näher untersucht werden, eine leichte Form des theoretischen Samplings durchgeführt werden.

Zu Beginn der Untersuchung hatte ich vermutet, dass ich vielleicht 12 oder 15 der verfügbaren Sitzungen untersuchen könnte. Das war allerdings zu einem Zeitpunkt, an dem die Ausrichtung der Arbeit noch nicht durchdacht war und sich demzufolge die Phänomendichte – die sich erst durch den Blickwinkel auf die Daten ergibt – nicht einmal geschätzt werden konnte. Letztendlich wurden acht Sitzungen untersucht: sechs Profi- und eine Studentensitzung, sowie eine weitere aus einem anderen Kontext (siehe Abschnitt 2.1.2).

Die Auswahlkriterien, nach denen diese Sitzungen schließlich gewählt wurden, sind nicht weltbewegend. Für einige der Sitzungen sind hier nun besondere Kennzeichen aufgelistet, die die Auswahl erleichterten.

- Sitzung D03 und Sitzung D06 zeigen die Arbeit von zwei Paaren, die eine Person gemeinsam haben.
- Sitzung D09 hat die Besonderheit, dass *beide* Entwickler eine Tastatur und eine Maus haben.
- Sitzung F02 zeigt zwei Entwickler mit einem großen Erfahrungsunterschied.
- Sitzung St10B zeigt keine Profis, sondern Studenten.
- Sitzung W02 stammt aus einem anderen Umfeld (Paardesigner).

Genauso gab es auch einige Sitzungen, deren Analyse zwar prinzipiell möglich gewesen wäre, die aber dennoch ausgeschlossen wurden. Gründe hierfür waren:

- Die Tonaufzeichnung hatte deutliche Lücken oder sehr starke Verschiebungen zum Bild (mehrere Minuten).
- Die Bildschirmaufzeichnung fehlte oder zeigte nur einen von zwei verwendeten Bildschirmen.
- Es wurde eine mir nicht geläufige Programmiersprache verwendet.

Teil II
Hauptteil

In diesem Teil werden die eigentlichen Resulte dieser Arbeit vorgestellt. Zunächst wird in Kapitel 3 näher erläutert, welcher Art von Phänomenen bei der Untersuchung die größte Aufmerksamkeit gewidmet wurde. Diese Phänomene werden in Kapitel 4 – den Erkenntnissen aus der Phase des *axialen Kodierens* folgend – in verschiedene Gruppen eingeteilt und klassifiziert. Da diese Arbeit aber nicht nur eine Taxonomie darstellen soll, wurde in Kapitel 5 großer Wert auf die Stellung der Phänomene im *Prozess* der Paarprogrammierung gelegt.

3 Fokus der Analyse und Begriffsfindung

Das zentrale Phänomen, um das sich die Analysen ausrichteten, sind die *Eingriffe* in die *Aktivität* des Partners. Um einen ersten Eindruck davon zu vermitteln, wie solche Phänomene konkret aussehen, werde ich nun vier verschiedene Szenarien vorstellen. Anhand dieser Beispiele wird anschließend der Betrachtungshorizont dieser Arbeit gezeichnet (Abschnitt 3.2).

Szenario 1
Die beiden Entwickler versuchen die Ursache für eine `NullPointerException`[15] zu ergründen. Der Driver *P2* scrollt abwechselnd im Quellcode und in der angezeigten Stacktrace[16], äußert sich aber nicht weiter dazu. Der Observer *P1* verhält sich ebenfalls ruhig und schaut auf den Bildschirm. Nach einiger Zeit sagt *P1*:

> *„Ah, wir sollten die – Stop, <**Name von P2**>! – wir sollten die, die Liste, die Liste oben einfach initialisieren.“*

Kurz bevor *P1* „Stop“ sagte, hatte *P2* wieder begonnen im Quellcode zu scrollen. Da sich der Vorschlag von *P1* aber auf die gerade noch angezeigte Codestelle bezog, forderte er *P2* auf, nicht weiter zu scrollen. Schließlich formuliert *P1* seinen Vorschlag zur Problemlösung aus.

Das Szenario stammt aus Sitzung D03.

Zusammengefasst hatte *P1* also einen Einfall, wie das akute Problem gelöst werden kann und wollte sich für die Erklärung die volle Aufmerksamkeit von *P2* sichern. Als Driver hat *P2* die Kontrolle über Maus und Tastatur und kann so seine eigenen Vorstellungen direkt umsetzen, während sein Partner dies vorher ankündigen muss.

Szenario 2
Nachdem ein Defekt im Programmcode erfolgreich beseitigt wurde, möchte der Driver *P2* – wie vorher besprochen – die Codekorrekturen in die Quellcodeverwaltung einchecken. Er öffnet dazu das entsprechende Menü und bewegt den Cursor zum Button *„Commit All*

[16] `java.lang.NullPointerException`. Ausnahmen dieses Typs werden z. B. geworfen, wenn im Programmcode vergessen wurde, ein Objekt zu erzeugen oder richtig zu übergeben. *http://download.oracle.com/javase/1.5.0/docs/api/java/lang/NullPointerException.html* (Abruf: 07.10.2011)

[16] Eine Stracktrace zeigt die Aufrufhierarchie bei einer geworfenen Ausnahme an. So kann nachvollzogen werden, durch welchen Methodenaufruf die Ausnahme ausgelöst wurde.

Outgoing Changes“, mit dem alle lokal vorgenommenen Änderungen hochgeladen würden. Bevor er ihn aber tatsächlich anklicken kann, wirft der Observer *P1* ein:

„*Nicht All!*“

Der Grund hierfür ist – wie *P1* im Anschluss seinem Partner erklärt – dass es neben den Änderungen, die das Paar in dieser Sitzung vorgenommen hat, auch noch weitere, ältere Änderungen gab, von denen *P1* nicht mit Sicherheit sagen könne, woher sie stammen, und ob es eine gute Idee sei, diese ebenfalls einzuchecken.

Das Szenario stammt aus Sitzung D06.

In diesem Fall hat der Observer *P1* einen potenziellen Fehler im Vorgehen von *P2* entdeckt, der sich aus den konkreten Umständen ergeben hat. *P1* hat somit seine Aufgaben als Observer „richtig“ erfüllt.

Szenario 3
Die Entwickler haben ein Shellskript geöffnet, in dem u. a. ein Aufruf von cURL[17] enthalten ist. Der Driver *P1* ist gerade dabei die aufzurufende Adresse zu ändern, als der Observer *P2* die Frage stellt:

„*cURL rufst du auf der Shell auf, ne?*“

P1 unterbricht seinen Schreibfluss, beantwortet die Frage mit „*Ja*“ und schreibt dann nach wenigen Augenblicken weiter.

Das Szenario stammt aus Sitzung S01.

P2 bezieht sich hierbei nicht direkt auf die Tätigkeit (oder *Aktivität*) von *P1*, sondern stellt lediglich eine Frage zum weiteren Vorgehen. Der *Auslöser* seines Gedankenganges kann dabei aber durchaus die Aktivität von *P1* gewesen sein. Im Rahmen dieser Arbeit ist es allerdings nicht (primär) von Bedeutung, ob sich *P2* inhaltlich auf die Aktivität seines Partners bezieht, sondern lediglich, *dass* er sie damit beeinflusst.

Szenario 4
Die Entwickler sind dabei einen Test zu schreiben. Der Driver *P1* denkt laut über die Bedeutung zweier Parameter in einem Funktionsaufruf nach, während der Observer *P2* vorschlägt den Test einfach zu starten. Noch bevor *P2* ausgesprochen hat, unterbricht ihn *P1* mit den Worten:

„*Warte mal eben.*“

Das Szenario stammt aus Sitzung D07.

Der Sprecher (*P1*) ist hier im Gegensatz zu den anderen drei Szenarien nicht in der Observer-, sondern in der Driverrolle. Genaugenommen umfasst dieses Szenario zwei Beeinflussungen: einen konstruktiven Vorschlag von *P2*, sowie dessen sofortige Zurückweisung durch *P1*. Weil die Reaktion sehr schnell kam, liegt die Vermutung nahe, dass *P1* nicht auf den Vorschlag von *P2* eingeht, sondern sich unabhängig vom Inhalt mehr Zeit einfordert.

[17] cURL („*Client for URLs*“, oder „*Client URL Request Library*“) ist ein Programm zur Datenübertragung, das u. a. die Protokolle HTTP und FTP unterstützt. http://curl.haxx.se/, (Abruf: 02.02.2012)

3.1 Aktivitätsbegriff

Im Rahmen der Basisschicht ist eine *activity* eine „gerade ablaufende HCI- oder HEI-Aktivität“ [29, Abschn. 7.1], und der Klasse *activity* sind die in Tabelle 2.4 aufgeführten Konzepte enthalten, die sich auf entsprechende Aktivitäten beziehen.
Aus Sicht der BS würde die Äußerung aus Szenario 1 mit einem *stop_activity* annotiert werden, weil der Observer den Driver auffordert, seine aktuelle HCI-Aktivität (Scrollen) zu unterbrechen. In Szenario 2 würde ein *disagree_activity* vergeben werden, weil der Observer der ablaufende HCI-Aktivität (Einchecken der Codeänderungen) widerspricht[18]. Die Szenarien 3 und 4 haben in der Basisschicht keine zugehörigen *activity*-Konzepte.

Im Rahmen dieser Arbeit wird ein anderer Aktivitätsbegriff verwendet. Ich erweitere den Begriff *Aktivität* um die „HHI-Aktivitäten“, beziehe also auch *Äußerungen* mit ein. Durch diese Erweiterung wird auch das Unterbreiten des Vorschlages durch *P2* aus dem 4. Szenario als Aktivität bezeichnet, die durch die BS-*activity*-Konzepte referenziert werden kann.

3.2 Dimensionen der betrachteten Phänomene

Um das bisherige grob vorhandene Verständnis der untersuchten Phänomene ein wenig zu formalisieren, werden an dieser Stelle zwei Dimensionen eingeführt. Diese helfen dabei, die Gemeinsamkeiten der Phänomene, die im Fokus dieser Arbeit liegen, in Worte zu fassen und greifbar zu machen. Letztlich wird durch sie festgelegt, welche Phänomene betrachtet werden und welche nicht.

3.2.1 *Miss of Expectations* (Verfehlen der Erwartung)

Es hat sich während der Analyse herausgestellt, dass die *Erwartungshaltung der Person* bezüglich des Verhaltens ihres Partners eine wichtige Rolle für das Verständnis von Interventionen und Aktivitätsbeeinflussungen spielt. Die Erwartungshaltung einer Person ergibt sich aus dem Kontext: zu jedem Zeitpunkt gibt es wahrscheinliche Verhaltensweisen (oder spezieller: Äußerungen), die der jeweilige Gegenüber auf Grund seiner Sozialisierung[19] und Erfahrung erwartet, und solche die ihn eher überraschen würden. In umgekehrter Betrachtungsweise lassen sich verschiedene Grade der Verfehlung eben dieser Erwartungen betrachten:

- Den niedrigsten Grad haben Äußerungen, die ziemlich genau der Erwartung des Partners entsprechen. Hat man eine Frage gestellt, erwartet man auch eine Antwort (bei rhetorischen Fragen verhält es sich natürlich anders).

  Gewissermaßen gibt es bei solchen geradlinigen Konversationen einen „Spielball“, der mit einer Frage an den Partner gegeben wird – und mit der Antwort wieder zurück. Das ist Teil der Sozialisierung.

- Einen leicht höheren Grad haben z. B. Äußerungen, deren *Auftreten* zwar erwartet war, bei denen allerdings der *Inhalt* nicht „erwartungskonform“ ist. Salinger

[18] Das Konzept *disagree_activity* kann auch verwendet werden, wenn sich der Widerspruch auf die Ausführung und nicht auf die Zielsetzung bezieht.

[19] Unter *Sozialisierung* eines Individuums versteht man die Verinnerlichung *(Internalisierung)* der typischen Verhaltensweisen, Meinungen und Wertevorstellungen innerhalb einer Gruppe. [15, S. 41]

bezeichnet derartige Antworten als „uneigentlich“, wenn sie z. B. betonen, dass es gar nicht nötig ist, die entsprechende Frage zu stellen [29, Abschn. 7.6.5].

Oder um es im Stile der eben genannten Metapher auszudrücken: Der Spielball wird anders zurückgespielt, als es normalerweise üblich ist.

▷ Auf einem mittleren Niveau bewegen sich Äußerungen, deren Auftreten nicht allzu wahrscheinlich, aber auch nicht völlig abwegig sind. Die Untersuchung der Sitzung W02 aus dem Paardesigner-Workshop (siehe Abschnitt 2.1.2) führte zu dieser Zwischenstufe: Hier wurden die meisten Äußerungen, auch wenn sie u. U. den Partner in dessen Sprechfluss störten, nicht wie Unterbrechungen aufgefasst.

Die Metapher lässt sich noch weiter ausbauen: Der Spielball hängt ständig in der Luft zwischen den beiden kreativen Köpfen. Sobald einer von beiden eine Idee hat, schlägt er auf den Ball, ohne dass dieser dadurch in den Besitz eines der Spieler kommen würde[20].

▷ Schließlich gibt es noch den höchsten Grad: dass *überhaupt* etwas gesagt wird, entspricht nicht der Erwartung des Angesprochenen. Dies ist z. B. dann der Fall, wenn der Driver in die Übersetzung seiner Gedanken in Programmcode vertieft ist, und der Observer ihn auf einen Defekt hinweist. Retrospektivisch kann es dem Driver natürlich klar sein, dass der Observer ihn jederzeit auf gefundene Defekte ansprechen könnte – es ist aber nicht sehr wahrscheinlich, dass sich der Driver dessen auch zu jedem Zeitpunkt während der Entwicklung bewusst ist. In diesem Sinne können solche Einwürfe als unerwartet angesehen werden.

Auch hier lässt sich die Metapher instrumentalisieren: Der Spieler passt den Ball zu seinem Partner, obwohl dieser im Moment mit anderen Dingen beschäftigt ist[21].

Angewendet auf die vier o. g. Szenarien ergeben sich folgende Werte für die Eigenschaft *Miss of Expectations*:

1. **Szenario:** *hoch*. Der Driver war in die Defektsuche vertieft.
2. **Szenario:** *hoch*. Der Driver wollte den vorher besprochenen Plan befolgen.
3. **Szenario:** *hoch*. Der Driver tippte eine längere URL aus dem Gedächtnis.
4. **Szenario:** *leicht bis mittel*. Der Observer hatte einen Vorschlag unterbreitet und bekam eine uneigentliche Antwort.

3.2.2 *Negativity* (Intensität der Ablehnung)

Die Äußerungen aus den Szenarien 1, 2 und 4 stehen der Aktivität des Partners *ablehnend* gegenüber; in Szenario 3 nimmt der Sprecher keinen inhaltlichen Bezug auf die Aktivität. Diese *Intensität der Ablehnung* bezeichne ich mit *Negativity*. Angewendet auf die vier Szenarien ergeben sich folgende Werte für die Eigenschaft *Negativity*:

[20] Tang et al. [35] beschreiben auf Grundlage des gleichen Materials u. a. die Koevolution von Problem und Lösung: immer wenn ein Stück mehr vom Problem verstanden wurde, wird die Lösung erweitert, wodurch sich wiederum neue Aspekte in der Problemdomäne ergeben usw.

[21] Allerdings enthält hier die Metapher eine negative Tendenz, weil ein beschäftigter Partner durch den Pass in jedem Fall gestört wird, während bei der Paarprogrammierung sowohl störende als auch hilfreiche Zuspiele vorkommen können.

1. **Szenario:** *mittel bis hoch*. Der Observer bittet den Driver aufzuhören im Code zu scrollen.
2. **Szenario:** *hoch*. Der Observer lehnt die Aktivität mit Nachdruck ab.
3. **Szenario:** *sehr gering*. Der Observer bewertet die Aktivität nicht negativ.
4. **Szenario:** *mittel bis hoch*. Der Driver unterbindet den Vorschlag des Observers.

3.2.3 Zusammenspiel der Dimensionen

Die Dimensionen *Miss of Expectations* und *Negativity* spannen einen Wertebereich auf, der sich in vier Quadranten zerlegen lässt und so vier verschiedene Arten von Äußerungen ergeben (siehe Abbildung 3.1).

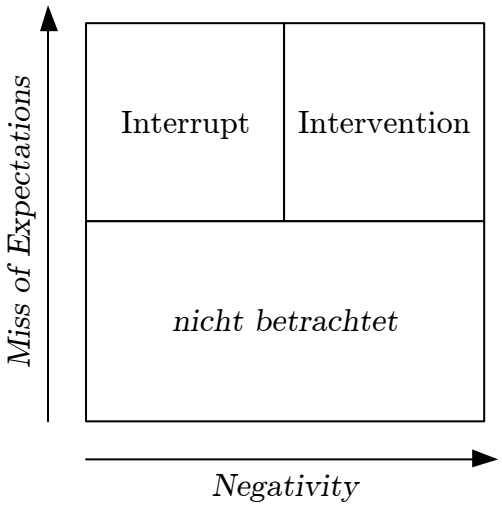

Abbildung 3.1: Spektrum der Dimensionen *Negativity* und *Miss of Expectations*.

Im Rahmen dieser Arbeit wurden nur Äußerungen betrachtet, die mindestens ein mittleres Niveau von *Miss of Expectations* hatten. Auf der *Negativity*-Skala wurde hingegen keine Vorabfilterung vorgenommen. Sie dient aber zur Definition zweier Begriffe: *Intervention* und *Interrupt*. Beide bezeichnen dabei Äußerungen mit einem großen *Miss of Expectations*-Wert.
Interrupts sind Unterbrechungen; sie haben wenig bis keine *Negativity*. *Interventionen* haben eine größere *Negativity* als *Interrupts*. Der Begriff *Interrupt* wurde nicht ganz zufällig gewählt. In Exkurs 1 wird die Parallele zum Interrupt in der Computertechnik näher erläutert.

Nach dieser Definition handelt es sich bei der Äußerung aus den Szenarien 1 und 2 um *Interventionen* und bei Szenario 3 um einen *Interrupt*. Szenario 4 hingegen hat einen geringeren *Miss of Expectations*-Wert und wird später in dieser Arbeit klassifziert.

Es sei an dieser Stelle noch einmal betont, dass es *keine* scharfe Grenze zwischen diesen beiden Begriffen gibt. Genau genommen hat eine *Intervention* auch immer die Eigenschaften eines *Interrupts*: sie unterbricht den Partner in seiner Aktivität. Gerade darin liegt ja die Bedeutung des *Miss of Expecations*.

Exkurs 1: Arbeitsgedächtnis und Aufmerksamkeit

Alan Baddeley [2] erforscht seit langem das Arbeitsgedächtnis. Bereits in einem frühen Modell hat er das Kurzzeitgedächtnis u. a. als ein Zusammenspiel einer „zentralen Exekutive“ und der „phonologische Schleife“ betrachtet. Die phonologische Schleife nimmt dabei ständig die (akustische) Umwelt war. Ihre Ausrichtung kann von der zentralen Exekutive beeinflusst werden und im Gegenzug gibt sie die empfangenen Signale weiter.

Die Interrupt-Behandlung in modernen Computern arbeitet ganz ähnlich. Durch einen Tastendruck (oder ein anderes Hardware-Gerät) wird der Signalpegel auf der Interrupt-Leitung erhöht und so der CPU die Möglichkeit zur Behandlung gegeben. Diese erfolgt aber nicht automatisch mit dem Heben des Pegels, sondern erst, wenn die CPU einen Zustand erreicht hat, den sie gefahrlos (also ohne Datenverlust) unterbrechen und nach der Interruptbehandlung wiederherstellen kann. [49]

Alles in allem ist der Vergleich zur Interruptbehandlung also durchaus angebracht.

4 Beeinflussungskonzepte

Nachdem im vorigen Kapitel mit Hilfe der Dimensionen *Miss of Expectations* und *Negativity* die wichtigsten Kennzeichen der untersuchten Phänomene formalisiert wurden, werden in diesem Kapitel die Ergebnisse des *axialen Kodierens* vorgestellt.
Während der Analyse stellte sich die Frage, an welcher Stelle des *paradigmatischen Modells* (siehe Abbildung 4.0) die Beeinflussungen der Aktivität des Partners einzusortieren wären. Einige Zeit lang versuchte ich es mit der Position des *Phänomens*, was allerdings mehr Probleme aufwarf, als dass es sie löste[22]. Als hilfreicher erwies sich dann die Position der *Handlungsstrategie*: in dem Moment, in dem eine Person in die Aktitivät des Partners eingreift, wurde die Entscheidung, genau das zu tun, ja bereits getroffen. Aus dem gegebenen *Kontext* heraus ergab sich durch die *intervenierenden Bedingungen* für die betreffende Person die Möglichkeit, so zu handeln. Damit lautet die Frage, die sich nach dieser Betrachtung aufdrängt: welche *Phänomene* werden eigentlich mit den Beeinflussungen behandelt? Diese Frage wird in diesem Kapitel beantwortet.

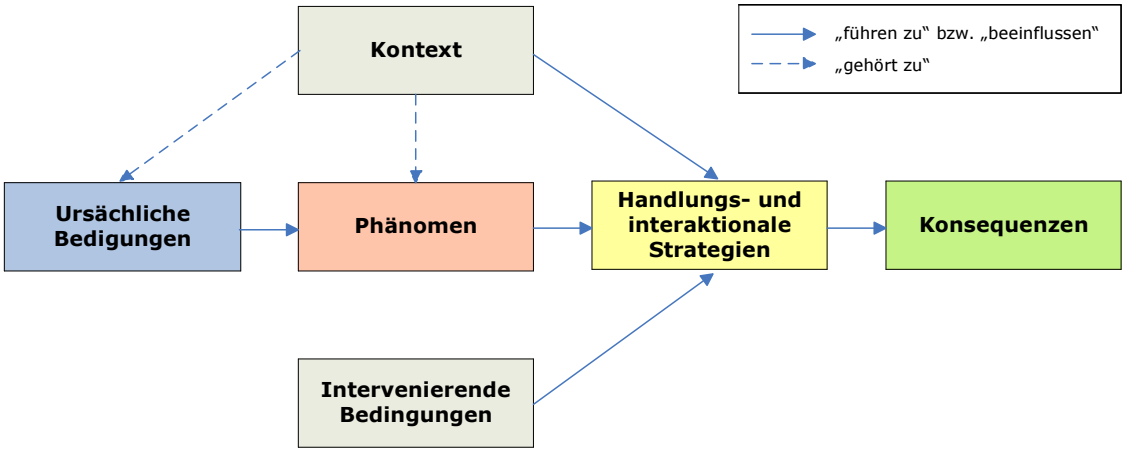

Abbildung 4.0: Darstellung des paradigmatischen Modells nach Strauss & Corbin [34]. (Wiederholung von Abbildung 2.2 von Seite 11.)

An dieser Stelle muss noch ein Unterschied zwischen den beiden Arten von Beeinflussungen hervorgehoben werden. Handelt es sich um eine *Intervention*, so nimmt die Beeinflussung die Position der *Handlungsstratgie* ein: Der Sprecher möchte an dieser Stelle schließlich die Aktivität des Partners beeinflussen. Ist es hingegen ein *Interrupt*, dann ist die Beeinflussung der Aktivität vielmehr ein Teil der *Konsequenz*: die primäre Illokution des Sprechers sieht gar keine Veränderung der Aktivität vor; sie ergibt sich lediglich daraus.

4.1 Die Beeinflussungstypen

Im Laufe der Untersuchung wurden mehrere hundert Episoden aus allen Sitzungen mit dem paradigmatischen Modell als Richtschnur analyisiert. Ausgehend von den beobachtbaren und teilweise auffälligen *Handlungsstrategien* wurde nach den *behandelten Phänomenen* geforscht und schließlich dafür die *Ursächlichen Bedingungen* gesucht. Schließlich wurden so acht verschiedene Klassen von *Handlungsstrategien* herausgearbeitet, die in Tabelle 4.1 überblicksartig zusammengefasst sind.

[22] Diese Problematik wird zusammen mit der Entscheidung für das letztendliche Vorgehen in Abschnitt 7.1 beschrieben.

Anmerkung: Die in diesem Kapitel beschriebenen Beeinflussungen haben naturgemäßg zwei beteiligte Rollen: den Beeinflusser und den Beeinflussten. Ersteren bezeichne ich im Folgenden mit *Sprecher*, letzteren mit *Partner*.

#	Klasse	Beschreibung
1	*defect_announcement*	Der Sprecher hat einen Defekt entdeckt. (Abschnitt 4.1.1)
2	*work_proposal*	Der Sprecher macht konkrete Vorschläge, *was* oder *wie* etwas zu tun ist. (Abschnitt 4.1.2)
3	*guidance*	Der Sprecher dirigiert seinen Partner. Entweder in Form einer Hilfestellung oder als aktive Steuerung. (Abschnitt 4.1.3)
4	*knowledge_request*	Der Sprecher hat eine Wissenslücke oder Verständnisschwierigkeiten. (Abschnitt 4.1.4)
5	*knowledge_conveyance*	Der Sprecher reicht Informationen oder Fachwissen nach. (Abschnitt 4.1.5)
6	*finding_conveyance*	Der Sprecher hat eine neue Erkenntnis (ausgenommen Defekte). (Abschnitt 4.1.6)
7	*proposal_rejection*	Der Sprecher weist einen Vorschlag seines Partners zurück, bevor dieser ihn komplett unterbreiten kann. (Abschnitt 4.1.7)
8	*action_taking*	Der Sprecher möchte selbst aktiv werden (z. B. etwas ausprobieren). (Abschnitt 4.1.8)

Tabelle 4.1: Übersicht über die Beeinflussungstypen

Auch wenn einige der Namen der Konzepte an die Basisschicht erinnern mögen, gibt es doch einen wesentlichen Unterschied zwischen beiden Konzeptarten: der Blickwinkel auf die Daten, oder anders: die Vorstellung davon, was ein Phänomen ist.
Die Basisschicht erfasst v. a. die *primäre Illokution* von Äußerungen und stellt Bezüge zwischen Äußerungen her. Die hier ausgearbeiteten Begriffe hingegen konzeptualisieren das Verhalten eines Entwicklers unter dem Gesichtspunkt der *Perlokution*, also der Wirkung die beim Empfänger erzielt wird. Außerdem werden mehrere Äußerungen des gleichen Konzepts pragmatisch zu einem Exemplar zusammengefasst (ähnlich wie die HCI-Konzepte in [29, Abschn. 8.1]).

In den folgenden Abschnitten werden diese nun der Reihe nach diskutiert. Da die Beschreibungen zum Teil recht umfangreich sind, folgen sie zur besseren Orientierung einem Schema, dass sich am paradigmatischen Modell orientiert:

1. **Allgemeine Beschreibung:** In dieser Sektion wird das beobachtbare Verhalten des Sprechers beschrieben (also ein Teil der *Handlungsstrategie*). Außerdem wird hier eine Zuordnung der Dimensionen *Miss of Expectations* und *Negativity* vorgenommen.

2. **Behandeltes Phänomen:** Es wird erläutert, welches *Phänomen* (oder welche Phänomene) mit dem beobachteten Verhalten behandelt werden.

3. **Ursächliche Bedingung:** Soweit es im Rahmen dieser Arbeit möglich war, werden in diesem Teil die ermittelten Ursachen für das Phänomen diskutiert.

4. **Rolle des Kontext und intervenierende Bedingungen:** Das bloße Auftreten eines Phänomens führt noch nicht automatisch zu den beobachteten Aktivitätsbeeinflussungen. Entscheidend für die tatsächliche Ausführung sind der engere *Kontext* des Phänomens, sowie die weiter gefassten *intervenierenden Bedingungen*, die der Person entweder neue Möglichkeiten eröffnen oder verschließen, also das Spektrum der möglichen Handlungsstrategien erweitern oder einengen.

5. **Beispiele:** Um die bis hierher gemachten Aussagen greifbarer darzustellen, habe ich versucht, möglichst anschauliche und vielsagende Beispiele zu finden. Es handelt sich hierbei selbstverständlich ausnahmslos um *tatsächliche* Episoden aus den untersuchten Sitzungen.

6. **Weitere Bemerkungen:** Im letzten Abschnitt jeder Klassen-Beschreibung wird den jeweiligen Besonderheiten und Eigenarten Rechnung getragen. Im Bedarfsfall werden auch Verweise auf die bisherige Forschung (auch aus anderen Disziplinen) gesetzt. Eine umfassende Einbettung der Gesamtergebnisse dieser Arbeit gibt es darüber hinaus im Schlussteil.

   Dieser Punkt wird im Einzelfall nicht ausgeführt, wenn sich für manche Konzepte alle wesentliche Punkte anhand des paradigmatischen Modells und den Beispiele verdeutlichen ließen.

4.1.1 *defect_announcement* (Bekanntgabe eines Defektfundes)

Allgemeine Beschreibung

Wenn der Observer während einer Sitzung einen Code-Defekt erkennt, teilt er dies seinem Partner mit. Dieses Verhalten ist ein Grundbaustein der Paarprogrammierung und wird praktisch in allen Publikationen zu diesem Thema erwähnt (z. B. von Williams et al. [45]). Der Driver kann natürlich ebenso einen Defekt entdecken – da er allerdings die Möglichkeit hat, seine Gedanken direkt selbst umzusetzen, ist eine Verbalisierung nicht zwingend nötig (was nicht heißt, dass er es nicht trotzdem tut).

Die Entscheidung, ob es sich bei einer entsprechenden Äußerung um eine *Intervention* oder einen *Interrupt* handelt, ergibt sich aus dem Kontext, in dem der Defekt gefunden wird. Beobachtet und erkennt der Observer, wie der Driver gerade im Moment einen Defekt produziert, wird er ihn i. d. R. sofort darauf hinweisen und ihn somit auffordern, den Defekt möglichst schnell zu korrigieren. In diesem Fall kommt es also zu einer *Intervention*.
Fällt dem Observer hingegen ein Defekt erst im Nachhinein auf (z. B. in Programmteilen, die schon vor Beginn der Sitzung existierten), dann kommt es häufig zu einem *Interrupt* – je nach dem, womit der Driver in diesem Moment beschäftigt war[23]. Auch hier gibt es keine natürlich vorhandene, scharfe Grenze zwischen den beiden Typen.

Behandeltes Phänomen

Das *Phänomen*, dass der „Entdecker“ behandelt, ist die Situation, dass der Programmcode eine strukturelle Schwäche aufweist, die im späteren Verlauf zu einem Programmversagen oder anderen Problemen führen kann. Defekte sollten in aller Regel behoben werden und dafür muss der Partner darüber informiert werden.
Defekte lassen sich entlang verschiedener Dimensionen betrachten (nach [28][24]):

- **Abstraktionslevel:** Es gibt einfache Syntaxfehler (wie etwa vergessene Semikola oder Klammern) und solche, die höhere grammatische Strukturen der verwendeten Programmiersprache betreffen (z. B. den schreibenden Zugriff auf eine als `final` deklarierte Variable). Solche Defekte werden für gewöhnlich von der Entwicklungsumgebung selbst angezeigt (spätenstens beim Kompiliervorgang). Wie sich durch die Beispiele weiter unten noch zeigen wird, sind diese „einfachen“ Defekte trotzdem eine nähere Betrachtung wert.

  Auf einem höheren Niveau befinden sich semantische Defekte. Sie werden vom Compiler nicht beanstandet, können aber ein Versagen während der Programmlaufzeit verursachen.

- **Konsequenzen:** Defekte lassen sich außerdem danach betrachten, welcher Natur ihre Auswirkungen sind. Zunächst stellt sich die Frage, welche Arten von Versagen der Defekt hervorrufen kann. Defekte, die zwar nicht beim Kompilieren re-

[23] Genau genommen hängt es davon ab, ob und wie der Observer die Tätigkeit seines Partners wahrnimmt. Ausschlaggebend wäre hier also nicht die objektive Realität, sondern das subjektive Bild, dass der Observer hat. Diese lässt sich aber leider nicht direkt beobachten (siehe Abschnitt 7.1 für eine Diskussion dieses Problems.)

[24] Die hier genannten Eigenschaften stellen nur eine Auswahl dar. In [28] werden zusätzlich noch *Ort* und *Typ* (zusammen ≈ Abstraktionslevel), sowie die *Herkunft* aufgeführt. Allein für die Bestimmung des Defekttyps existieren eine Vielzahl von Klassifikationssystemen.

gistriert werden, aber sofort danach spürbar die Ausführbarkeit des Programms beeinträchtigen, haben eine offensichtliche Auswirkung.

Allerdings führt nicht jeder Defekt sofort zu einem Versagen: manche Defekte haben erst bei potenziellen Änderungen am Programm und/oder dem Einsatzgebiet des Programms einen negativen Effekt. So verschlechtert es z. B. die Wartbarkeit des Programms (und erhöht damit die Gefahr, neue Defekte zu generieren), wenn Variablen, Methoden oder Klassen entgegen bestehender Konventionen oder einfach irreführend benannt sind.

Ursächliche Bedingung

Die Frage nach der *ursächlichen Bedingung* führt sehr schnell aus dem Fokus dieser Arbeit heraus. Ausformuliert würde sie in etwa lauten: *„Was ist die ursächliche Bedingung dafür, dass es Defekte in Software gibt?“*
Allgemein ließe sich sagen, dass Defekte durch Fehler von Menschen entstehen. Die gefundenen Defekte, die im Rahmen dieser Arbeit analysiert wurden, entstanden dabei sehr häufig durch kleine Unaufmerksamkeiten, wie in Beispiel 1, bei dem der Driver ganz offensichtlich den „Wald vor lauter Bäumen“ nicht gesehen hat.
In Exkurs 2 werden die drei Begrifflichkeiten *Fehler*, *Defekt* und *Versagen* noch einmal näher beleuchtet.

Exkurs 2: Fehler, Defekte und Versagen

Wenn das beobachtbare Verhalten eines Programms nicht korrekt ist, liegt ein *Versagen* (engl. *failure*) vor. Versagen entstehen, weil die Software *Defekte* (engl. *defect/fault*) aufweist – wobei allerdings nicht jeder Defekt zwangsläufig zu einem Versagen führen muss. Defekte entstehen, weil Menschen *Fehler* (engl. *error*) machen. Fehler können dabei entweder in einem *Falschtun* (engl. *commission*) oder in einem *Versäumnis* (engl. *ommission*) bestehen. Fehler passieren nicht nur beim Programmieren, sondern können in jeder Phase der Softwareentwicklung auftreten.

Fehler haben demnach einen Prozesscharakter, Defekte sind Struktureigenschaften und Versagen sind bestimmte Formen des Softwareverhaltens.

Quellen: [5, 28]

Für die Ursachen, dass die Defekte *entdeckt* werden, lassen sich aus Forschersicht zumindest Indizien finden. Als entscheidende Kriterien hierfür wurden im Laufe der Untersuchungen die *Sichtbarkeit* der Defekte und der *Aufmerksamkeitsfokus* der Person herausgearbeitet. Letzterer wird in Exkurs 3 ausführlich erläutert; zum Aspekt der Sichtbarkeit reicht es aus, festzuhalten, dass es *direkt sichtbare* und *indirekt sichtbare* Defekte gibt. Ein direkt sichtbarer Defekt steht „im Original“ auf dem Bildschirm; indirekt werden Defekte durch Annotationen (z. B. Symbole und farbliche Hervorhebungen) an mehreren Stellen in modernen Entwicklungsumgebungen angezeigt.
Kurz gefasst gilt folgendes: Um entdeckt werden zu können, muss ein Defekt *sichtbar* sein und er muss im *Aufmerksamkeitsfokus* der betreffenden Person liegen[25].

[25] Genaugenommen muss der Defekt mindestens einmal sichtbar *gewesen* sein. Je nach Gedächtnisleistung der Entwickler kann eine bestimmte Codestelle auch memoriert worden sein – was zumindest für einige Minuten im Bereich des Möglichen liegt – und der Defekt aus der Erinnerung heraus entdeckt werden. Neuropsychologisch gesprochen (siehe Exkurs 1) ist auch das Arbeitsgedächtnis nichts weiter als der gerade aktive Teil des Langzeitgedächtnisses.

In der Terminologie des paradigmatischen Modells handelt es sich dabei aber nicht um die Ursachen des Phänomens, sondern um Teile des *Kontextes* und der *intervenierenden Bedingungen*. Näheres wird im nächsten Abschnitt geklärt.

Rolle des Kontexts und der intervenierenden Bedingungen

Der Kontext bildet das nähere Umfeld für das Phänomen. Damit ist die *Sichtbarkeit* der Defekte klar dem Kontext zuzuordnen.

Per Definition sind die *intervenierenden Bedingungen* unabhängig vom Phänomen. Demnach nimmt der *Aufmerksamkeitsfokus* des Entdeckers diese Stellung im paradigmatischen Modell ein. Erst wenn der Defekt in den Fokus des Programmiers fällt, ergibt sich für ihn auch die Möglichkeit einer Behandlung. Der Aufmerksamkeitsfokus als intervenierende Bedingung kann das Spektrum der „wählbaren" Handlungsstrategien also *erweitern* (wenn der Fokus korrekt ausgerichtet ist) oder *einengen* (andernfalls).
Es ist einsichtig, dass farblich hervorgehobene syntaktische Defekte auch von einem „falsch" fokussierten Programmierer entdeckt werden können, während semantische Fehler erstens ein höheres Maß an Codeverständnis und zweitens auch einen genaueren, engeren Fokus beim Betrachter voraussetzen[26].

Beispiele

In Anbetracht der möglichen Abstraktionslevel von Defekten könnte man geneigt sein, die einfachen syntaktischen Defekte als uninteressant abzutun, da sie ja für gewöhnlich von der Entwicklungsumgebung deutlich aufgezeigt werden. Die Beispiele 2 und 3 zeigen, dass es sich dennoch lohnt, über diese „trivialen" Fälle nachzudenken, weil sie den Mechanismus illustrieren, über den die *Defektsichtbarkeit* und der *Aufmerksamkeitsfokus* zusammenhängen.

1. In Sitzung F02 soll eine Klasse ausgetauscht werden (Ersetzung von `CalenderTestView` durch `CalenderView`) und der Driver *P1* ersetzt zwei der insgesamt drei Vorkommen in der geöffneten Klasse manuell und übersieht somit ein Vorkommen (obwohl die IDE zu Beginn der Episode alle drei Vorkommen hervorgehoben hatte und die Klasse nur 10 Zeilen Code beeinhaltet). Die IDE kreidet kurz darauf die vergessene Stelle als syntaktischen Defekt an, weil die der neue Typ `CalenderView` nicht vom bisherigen Typ `CalenderTestView` erbt. Der Observer *P2* bemerkt diesen Fehler zunächst auch nicht, entdeckt dann aber die Verwendung des alten Namens und sagt:

   P2: *„Du hast hier <* zeigt mit dem Finger auf die Stelle *> das musst du noch machen."*

   Der Driver *P1* entgegnet ein lang gezogenes: *„M-hm"*, scheint aber nicht zu verstehen, was *P2* meint. Daraufhin ergänzt *P2*:

   P2: *„<* zeigt nochmals *> Hier‿hier ist doch immer noch 'Test'."*

   Dann fällt es auch *P2* und der Defekt wird behoben.

[26] Die Hervorhebung von Defekten in Entwicklungsumgebungen nutzt den Mechanismus der *Bottom-Up*-Aufmerksamkeit, während semantische Defekte *top-down* gesucht werden müssen (siehe Exkurs 3).

2. In Sitzung St10B hatten die beiden Programmierer das geöffnete Projekt nicht vollständig konfiguriert, sodass die Entwicklungsumgebung sämtliche Klassen aus einem bestimmten Paket als syntaktische Fehler ankreidete. Die Beiden entschieden sich zu Beginn der Sitzung, das Problem nicht zu beseitigen und stattdessen die Fehlermeldungen einfach zu ignorieren. Weil die Übersetzung des Programmcodes nicht von der IDE übernommen wurde, stellte das für die Entwicklung kein unmittelbares Hindernis dar. Allerdings waren in dieser Sitzung so permanent eine Vielzahl von „falschen“ Defekten sichtbar (direkt und indirekt).

3. In Sitzung D03 wurde ein Defekt, der schon seit ca. drei Minuten sichtbar war (zunächst indirekt, später auch direkt), erst durch die in Exkurs 3 erwähnte Weitung des Aufmerksamkeitsfokus nach Abschluss eines Arbeitsschrittes vom Observer entdeckt und konnte behoben werden. Die entsprechende Episode ist in Tabelle 4.2 ausführlich beschrieben.

4. Ebenfalls in Sitzung D03 wollten die Entwickler ein GUI-Element in der Anwendung rechtsbündig ausrichten. Der Driver hatte den Ausrichtungswert von `RIGHT` auf `FILL_HORIZONTAL` gesetzt und „hangelte“ sich danach durch die Menüpunkte der automatischen Codevervollständigung. Der Observer unterbrach diese Versuche mit der Äußerung:

   P1: *„Ich glaub das mit dem `RIGHT` können wir schon so lassen, weil wir müssen (!...!)“*

   P1 stockte danach und führte die Erklärung nicht zu Ende, sodass man nicht genau erfährt, wie *P1* zu seiner Überzeugung gelangt ist. Vorstellbar wäre, dass es ihm widersinnig erschien, die Ausrichtung `RIGHT` durch etwas Anderes zu ersetzen, wenn die Komponente doch gerade rechtsbündig angeordnet werden soll. Weiterhin äußerte *P1* im späteren Sitzungsverlauf, dass er sich mit dieser Technologie nicht auskennt, was diese Hypothese einer „intuitiven“ Ablehnung unterstützt.

Weitere Bemerkungen zu dieser Klasse

Die für diese Klasse herausgearbeiteten Begriffe von *Defektsichtbarkeit* und *Aufmerksamkeitsfokus* gelten selbstverständlich unabhängig davon, ob man im Paar oder alleine programmiert. Allerdings ist es entscheidend, *worauf* die Aufmerksamkeit gerichtet ist. In der Literatur wird (u. a. von Williams et al. [45]) die Tätigkeit des Observers häufig mit Aktivitäten wie „kontinuierlicher Code-Review“ und „Überdenken der strategischen Konsequenzen der Implementierung“ beschrieben; der Observer beobachte „aktiv“ (Arisholm et al. [1]). Beim Lesen dieser Formulierungen kann man den Eindruck gewinnen, dass der Observer das alles gleichzeitig machen würde. Das in Tabelle 4.2 angeführte Beispiel zeigt, dass das eben nicht der Fall ist: der Aufmerksamkeitsfokus erlaubt dem Observer nur *ein* Ziel im Auge zu behalten – und in diesem Fall war es das gleiche wie das des Drivers, wodurch der Defekt *trotz* seiner Sichtbarkeit unentdeckt blieb.

Es werden zwar im aktuellen Kapitel acht verschiedene Klassen vorgestellt, aber die Übergänge zwischen ihnen sind deswegen nicht automatisch trennscharf. Im Falle von *defect_announcement* gibt es folgende „benachbarte“ Klassen:

- Es kommt häufiger vor, dass der Observer ein Programmversagen zuerst, also vor dem Driver, verbalisiert. In der Regel sind die entdeckten Programmversagen aber durch Fehlermeldungen o. ä. auch ohne ausgeprägtes Domänenwissen leicht zu erkennen, sodass man nicht per se davon ausgehen sollte, dass der Driver das Versagen nicht als solches begriffen hätte. Daher sind diese Äußerungen seitens des Observers als „Think Aloud"-Phänomene kategorisiert (siehe Abschnitt 4.1.6), da sie dem Driver anzeigen, dass man dem Entwicklungsprozess weiterhin gedanklich folgt.

- Das Erkennen von bzw. das Hinweisen auf Code-Defekte kann auch unmittelbar mit einem Lösungsvorschlag einhergehen. Mehr dazu im nächsten Abschnitt 4.1.2.

HHI-Kode	**Äußerung**	**Kontext/Kommentar**
P2.write_sth (P2.propose_design)	<* tippt `addScaleRange` *>	P2 hat einen Dialog geöffnet, um eine Methode zu extrahieren. Er hat angefangen der Methode einen Namen zu geben und wartet nun.
P1.amend_design	„`Panel`?"	Nach drei Sekunden Pause ergänzt der Observer P1 den Methodennamen zu `addScaleRangePanel`.
P2.propose_step P2.write_sth + P2.challenge_design	„(∼) wart' mal (..) (##LabelScaleRange##) so?"	Unmittelbar nachdem P1 seinen Vorschlag beginnt, unterbricht ihn P2 und kurz darauf spricht er seinen eigenen Gedanken aus. Vor dem „so?" ändert er den Namen in `addLabelScaleRange`.
P1.agree_design	„Ja‿oder (.) ja."	P1 zögert mit seiner Zustimmung und ist offensichtlich nicht restlos überzeugt.
P2 schließt das Extrahieren der Methode ab, wobei ein Defekt entsteht, der am rechten Rand neben dem Code als neue rötliche Box *indirekt sichtbar* wird, was aber keinem der beiden auffällt. P2 stellt P1 eine Verständnisfrage zum Programmcode; anschließend diskutieren sie verschiedene Implementierungsmöglichkeiten. Zum Ende der Diskussion wird der Defekt durch Scrollen auch *direkt sichtbar*, bleibt aber zunächst unbemerkt.		
zweieinhalb Minuten später		
P2.write_sth	<* ändert eine If-Bedingung *>	P2 setzt eine Änderung um, die in den letzten Minuten diskutiert wurde.
P1.explain_finding	„OK"	P1 befindet die Umsetzung der besprochenen Änderung für richtig.
P1.explain_finding	„Uah. Jetzt kennt er das da unten nicht?"	Unmittelbar danach fällt P1 der Defekt auf, der zuvor beim Extrahieren der Methode entstanden ist.

Tabelle 4.2: Beispiel für die Verschiebung des Aufmerksamkeitsfokus. Das *P1.amend_design* ist außerdem ein Beispiel für eine Verantwortungsübernahme. Die Episode stammt aus Sitzung D03.

Exkurs 3: Aufmerksamkeitsfokus und freie Ressourcen

Die Tatsache, dass ein Defekt sichtbar war oder ist, ist noch nicht hinreichend für das tatsächliche Entdecken eines Defekts. Während des axialen Kodierens wurden weiterhin die Begriffe *Aufmerksamkeitsfokus* und *freie Ressourcen* entwickelt.

Unter *Aufmerksamkeit* verstehen Psychologen „das Entdecken und Lokalisieren von Sinnesreizen und Objekte, das Integrieren ihrer Merkmale [und] das Beachten ihrer Veränderung" [33, S. 124].
Mit dem Begriff *Aufmerksamkeitsfokus* einer Person sollen in dieser Arbeit die Aspekte der Programmiersitzung subsummiert werden, die sich im Arbeitsgedächtnis (siehe [8, 2]) dieser Person befinden – dazu zählen insbesondere auch Teile des optischen Sichtfelds der Person, sowie das „worüber sie gerade nachdenkt". Dieser Fokus kann *eng* oder *weit* gefasst sein, wobei *eng* bedeutet, dass sich die Person nur auf einen kleinen Anteil ihrer kognitiven Wirklichkeit[a] konzentriert, während ein *weiter* Fokus größere Teile weniger intensiv betrachtet.

Die *freien Ressourcen* sollen den Anteil des Arbeitsgedächtnisses bezeichnen, der nicht mit dem Inhalt des Aufmerksamkeitsfokus „belegt" ist.
Man kann davon ausgehen, dass der Bedienung von Maus und Tastatur durch den Driver kognitive Prozesse zu Grunde liegen, die sich im gewissen Sinne als „Mehrbelastung" gegenüber dem Observer verstehen lassen. Oder andersherum ausgedrückt: der Observer verfügt über mehr freie Ressourcen, die er zum Lesen weiter entfernten[b] Programmcodes oder dem Nachdenken über andere Aspekte der Sitzung nutzen kann: *„while one partner is busy typing [...] the other partner can think more strategically"* [45]. Ob er dies *tatsächlich* tut, und so strategischer denkt als der Driver, spielt hier allerdings erst einmmal keine Rolle.

Im Verlauf einer Sitzung lässt sich manchmal die *Verschiebung des Aufmerksamkeitsfokus* beobachten. So wird z. B. nach Beenden eines Arbeitsschrittes wieder Ressourcen freigegeben und der Fokus geweitet. Einige Defekte werden genau dann entdeckt (siehe Beispiel in Tabelle 4.2).
Die Ausrichtung der Aufmerksamkeit wird von zwei Seiten gesteuert. Psychologen sprechen in diesem Zusammenhang von *Top-Down*- und *Bottom-Up*-Aufmerksamkeit. Im ersten Fall wird die Aufmerksamkeit bewusst fokussiert (z. B. das konzentrierte Betrachten einer roten Ampel, um das Grün nicht zu verpassen). Andersherum können sich Reize auch „von unten" in das Bewusstsein drängen (wenn der Kraftfahrer hinter einem rhythmisch zu hupen beginnt) und den Fokus verschieben. [9, S. 497]

Es sei noch einmal angemerkt, dass weder der Aufmerksamkeitsfokus, noch das Ausmaß an freien Ressourcen direkt beobachtet werden können. Es handelt sich hierbei um theoretische Begriffe, die im Laufe der Untersuchungen entwickelt wurden und helfen, die beobachteten Phänomene nachvollziehbar zu erklären.

[a] Damit ist die Gesamtheit der äußeren Sinnesreize und der „inneren" Gedankengänge gemeint.
[b] Hier ist „Entfernung" i. S. v. „Abstand zum Cursor" zu verstehen.

4.1.2 *work_proposal* (Design- und Arbeitsschrittvorschläge)

Allgemeine Beschreibung

In diese Klasse fallen alle Aktivitätsbeeinflussungen bei denen der Sprecher:

- Vorschläge oder Anweisungen zu Designaspekten gibt (*Design* meint hier – wie in der Basissischt – jede Gestaltungsmöglichkeit des Produkts)
- Vorschläge oder Anweisungen zum weiteren Handeln des Partners gibt.

Ausschlaggebend für die Annotation mit dem Konzept *work_proposal* ist wiederum ein Mindestmaß an *Miss of Expectations*, da ansonsten jedes *propose_design* bzw. *propose_step* in diese Klasse rutschen würde.

Das untere Ende des betrachteten *Miss of Expectations*-Bereiches bilden hier Beeinflussungen, bei denen ein im Vorfeld gemachter Vorschlag derart aufgegriffen wird, dass die Zielstellung erhalten bleibt, aber der Weg ein völlig anderer ist. Abstrakt gesprochen nimmt der Sprecher hier eine *Intervention* vor und verlangt, die aktuelle Aktivität (die den bisherigen Plan verfolgt) durch eine andere zu ersetzen (die dem neuen Plan folgt). Beispiel 3 veranschaulicht den Typ *„Aufgreifen eines Vorschlags“*.
Wurde hingegen noch gar nicht mit der Umsetzung begonnen, dann kann sich der Sprecher auch nicht negativ darauf beziehen und tätigt so im „schlimmsten“ Fall einen *Interrupt*. Üblicherweise ist aber der Grund, warum mit der Umsetzung noch nicht begonnen wurde, der, dass die Diskussion noch gar nicht abgeschlossen ist, somit etwaige Korrekturen an den Vorschlägen keinen hohen *Miss of Expecations* haben und derartige Äußerungen nicht mal als *Interrupt* gelten können.
Ebenso wie bei gleichbleibender (oder sehr ähnlicher Zielsetzung) der *Weg* „ausgetauscht“ werden kann, können dem Sprecher auch neue Zweifel an der Richtigkeit der *Zielsetzung* kommen und er dementsprechend in die begonnene Umsetzung *intervenierend* eingreifen. Beispiel 4 zeigt einen solchen Vorgang *„Widerruf einer getroffenen Entscheidung“*.

Die Elemente dieser Klasse stellen mit Abstand den größten Teil der während einer Sitzung getätigten Beeinflussungen dar.

Behandeltes Phänomen

Bei der Untersuchung der o. g. Verhaltensweisen konnten zwei potentielle Phänomene ausgemacht werden, die der Sprecher mit Hilfe der Beeinflussung behandelt.

1. **Einfall des Sprechers.** Dem Sprecher ist etwas aufgefallen, von dem es unsinnig wäre, es dem Partner vorzuenthalten.

   Das eingangs beschriebene *Aufgreifen eines Vorschlags*, sowie der *Widerruf einer getroffenen Entscheidung* sind Behandlungen eben dieses Phänomens. Ersteres beinhaltet aus Sicht des Sprechers eine *Verbesserung*, durch die Arbeitszeit gespart werden kann, die Lösung in der Software „eleganter“ gemacht, etc. Mit der zweiten Verhaltensweise sollen Fehler vermieden werden. Der Sprecher ist von der getroffenen Entscheidung doch nicht überzeugt und äußert diese Bedenken entsprechend.

2. **Verantwortsbewusstsein.** Der Sprecher fühlt sich *verantwortlich* für das weitere Vorgehen und wird dadurch motiviert, aktiv etwas zum Prozess beizusteuern.

Ursächliche Bedingung

So wie zwei verschiedene Phänomene durch den Sprecher behandelt werden, gibt es auch verschiedene Ursachen für ihr Auftreten.

1. **Einfall des Sprechers.** Warum eine Person einen Einfall hat und die andere nicht, lässt sich nicht zuverlässig ermitteln. Einen Anhaltspunkt liefert wiederum der *Aufmerksamkeitsfokus*, dessen konkrete Ausrichtung schon für die Erklärung, warum Defekte entdeckt werden, hilfreich war.

   Der Sprecher (in der Rolle des Observers) sieht manchmal einfach eher, welche Codezeile angepasst werden muss, um den vorher besprochenen Plan umzusetzen (siehe Beispiel 6).

   Im Vorfeld der *Widerrufe getroffener Entscheidungen* ließen sich deutliche Unsicherheiten bei der ursprünglichen Entscheidungsfindung ausmachen. Beiden Entwicklern kann es an Fachwissen und/oder Erfahrung mangeln, wodurch es ihnen (oder zumindest dem späteren Sprecher) im Anschluss an die geführte Diskussion an Vertrauen in die getroffene Entscheidung fehlt. Wenn dem Sprecher dann rationale Argumente einfallen, kann er auch eine konstruktive Frage stellen (siehe z. B. Beispiel 4).

2. **Verantwortsbewusstsein.** Für das Phänomen des *Verantwortlich fühlen* konnten mehrere Ursachen rekonstruiert werden:

   (a) Nach einem Versagen der Software vermutet der Observer, dass der oder die Defekte irgendwo in den Codebereichen liegen, die er selbst implementiert hat. Somit fühlt er sich für das resultierende Versagen *verantwortlich* und unterbreitet ungefragt Korrekturvorschläge (siehe Beispiel 5).

   (b) Wenn der Driver einige Zeit lang scheinbar „nichts“ tut, dann kommt es vor, dass der Observer die Inititiative ergreift und so die *Verantwortung* übernimmt. (siehe Beispiele 1 und 6)

Rolle des Kontexts und der intervenierenden Bedingungen

Die zwingende Bedingung für das Unterbreiten von Vorschlägen ist eine entsprechende Idee. Allerdings muss hier „Idee“ nicht „umfassende Lösung“ bedeuten.
Im Falle von *Handlungsanweisungen* können diese auch relativ „einfallslos“ sein, wie Beispiel 1 zeigt.
Bei *Designvorschlägen* ließ sich eine interessante Eigenschaft herausarbeiten: Sie können sich nicht nur unmittelbar an ein *defect_announcement* anschließen, sondern sogar *stattdessen* erfolgen. Im Einzelnen ließ sich die folgende Abstufung nach dem Abstraktionslevel der getätigten Äußerung feststellen:

Problemdomäne (*defect_announcement*)

1. Der Sprecher weist darauf hin, *dass* ein Defekt existiert.
2. Der Sprecher sagt, wo sich seiner Ansicht nach der Defekt befindet.

Lösungsdomäne (*work_proposal*)

3. Der Sprecher sagt, was seiner Ansicht nach zur Behebung des Defekts zu tun ist.
4. Der Sprecher sagt, wie seiner Ansicht nach die Lösung durchgeführt werden muss.

Jede dieser Stufen kann nochmals unterteilt werden, indem man solche Äußerungen gesondert betrachtet, die eine *Idee* beinhalten, um die jeweilige Stufe zu erreichen. Außerdem kann der Sprecher natürlich auch lediglich die Tatsache äußern, *dass* er über das jeweilige Wissen verfügt, ohne es selbst zu äußern[27]. Eine solch feine Granularität trägt hier aber nicht zum Verständnis bei, sodass darauf verzichtet wurde.
Die höheren Stufen enthalten die niedrigen Stufen implizit: Wann immer der Sprecher einen Lösungsvorschlag unterbreitet (Stufe 3), sind darin automatisch die Aussagen enthalten, dass der Sprecher weiß, dass es einen Defekt gibt (Stufe 1) und wo sich dieser befindet bzw. worin dieser besteht (Stufe 2).

– Dazu ein Beispiel aus Sitzung D03: Der Driver *P2* hat eine Codeänderung vorgenommen und die Klassendatei gespeichert. Der Quellcode ist offensichtlich syntaktisch korrekt; dennoch zeigt die IDE einen Syntaxfehler an. Der Observer *P1* äußert darauf hin:

  P1: *„Mach' mal (.) ach er hat's schon gefunden."*

  Gerade als *P1* seinen Vorschlag zur Behebung dieses Fehlverhaltens äußern wollte, verschwindet die Markierung in der IDE. Davon abgesehen hätte es sich hierbei (womöglich) um eine Äußerung der Stufe 3 (oder 4) gehandelt, die implizit die Aussagen auf Stufe 1 und 2 enthalten hätte, die ausdrücken, dass es überhaupt einen Grund gibt, etwas zu tun.

Je nach Kontext ist es auch möglich, dass der Sprecher lediglich eine Äußerung einer niedrigen Stufe macht (z. B. sich kurz künstlich räuspert - Stufe 1) und beide Programmierer sofort wissen, dass der Driver z. B. zum wiederholten Male einen Buchstabendreher getippt hat (Stufe 2) und diese zwei Buchstaben nun austauschen soll (Stufe 3).

Eine *intervenierende Bedingung*, die die Handlungsmöglichkeiten *einschränkt*, ist die Priorisierung oder der *Vorrang* der Aktivität des Partners. Durch diese werden Äußerungen (vorübergehend) zurückgestellt, weil der Partner z. B. gerade etwas Bedeutsameres macht (siehe Beispiel 2).
Eine *begünstigende intervenierende Bedingung* ist die gefühlte *Verantwortung*, v. a. dann, wenn sie schon einmal übernommen wurde und sich kurz darauf wieder die Gelegenheit ergibt. Der Sprecher wartet dann eine wesentlich kürzere Zeitspanne ab, bis er wieder eingreift (siehe Tabelle 4.3 für eine entsprechende Episode).

Beispiele

Die folgenden Beispiele stammen zwar überwiegend aus einer einzigen Sitzung, was aber keinen Rückschluss darauf erlaubt, dass in den anderen Sitzungen derartige Phänomene nicht beobachtet werden konnten. Die Auswahl der abgedruckten Beispiele erfolgte hauptsächlich nach ihrem Vermögen, mit wenig Vorwissen beim Leser die o. g. Punkte bestmöglich zu illustrieren.

1. In Sitzung D03 haben beide Programmierer festgestellt, dass die Demo-Anwendung wegen einer `NullPointerException`[28] nicht ausgeführt werden kann. Der

[27] In der Basisschicht entspricht das der Unterscheidung zwischen einem *explain_standard of knowledge* und einem *explain_finding* [29, Abschn. 7.6.3].
[28] `java.lang.NullPointerException`, siehe auch Fußnote 16 auf Seite 22

Driver schließt das Demo-Menü, weist eine Hypothese des Observers bezüglich der Ursache zurück und verharrt dann mit der Maus am unteren Bildschirmrand. Nach sechs Sekunden schlägt der Observer vor:

P1: *„Guck doch mal in die Konsole was los ist.“*

Der Driver stimmt dem Vorschlag sofort zu. Da der Driver seinen Blick nach unten richten muss, um die Mauszeiger zu sehen, kann man davon ausgehen, dass er diesen vorher gar nicht beachtet hat, sondern auf den darüberstehenden Programmcode geschaut und dort vielleicht nach einem Defekt gesucht hat. Weiterhin lässt sich auch nicht mit Bestimmtheit sagen, dass *P1* annimmt, sein Partner sei untätig – wahrscheinlicher ist, dass er die Defektsuche mit Einbeziehung der Stacktrace einfach für aussichtsreicher hält.

2. Zu Beginn von Sitzung F02 schlägt der Observer *P2* vor, die Remote-Desktop-Verbindung im Vollbildmodus zu öffnen:

   P1: *„Können wir's groß machen?“*

   Zu Beginn war *P2* der Driver und hatte auch die Remote-Verbindung selbst geöffnet, hat dann aber die Tastatur zu *P1* geschoben. In diesem Moment fällt *P2* wohl die zu geringe Fenstergröße auf, und zeigt mit dem Finger auf den Bildschirm. Da im gleichen Moment aber *P1* zu sprechen beginnt, hält sich *P1* zurück und sagt nichts. Erst eine Sekunde nachdem *P1* ausgeredet hat, äußert *P2* seinen Vorschlag – er gewährt seinem Partner also den Vorrang und wartet ab.

3. In Sitzung D03 konnte beobachtet werden, dass der Driver einen Design-Vorschlag machte und ihn sofort umzusetzen begann. Der Observer hat den Vorschlag aufgegriffen und dann in seiner Art völlig verändert, sodass nur noch die Grundidee hinter dem Vorschlag erhalten blieb. Konkret hatte der Driver *P2* vorgeschlagen, einige Codezeilen zu verschieben (um den Code insgesamt besser zu strukturieren), worauf der Observer *P1* entgegnete:

   P1: *„Mach doch ne Methode 'raus, und (!...!)“*

4. In Sitzung D03 wurde von beiden Entwickler diskutiert und beschlossen, dass nicht mehr eine als „deprecated“[29] gekennzeichnete, statische Methode verwendet werden soll, sondern stattdessen eine dynamische Methode, für deren Aufruf ein Objekt benötigt wird. Der Driver wendet sich gerade der Tastatur zu, als der Observer einwirft:

   P1: *„Aber reicht das nicht als statischer Zugriff?“*

   Im Vorfeld hatte der Observer geäußert, dass er nicht wisse, wie der empfohlene Weg zu implementieren sei. Der Driver scheint etwas bessere Kenntnis darüber zu haben, bricht aber seinen ersten Ansatz mit einer zweifelnden Bemerkung ab

[29] Methoden, die mit `@Deprecated` annotiert sind, sollen in neuem Code nicht mehr verwendet werden, z. B. weil es mittlerweile bessere Alternativen gibt. *http://download.oracle.com/javase/1,5,0/docs/api/java/lang/Deprecated.html* (Abruf: 06.10.2011)

und wendet sich (vermutlich unbewusst) einer anderen Thematik zu. Der Observer weist ihn auf diese Abweichung hin und unterbricht ihn dann beim zweiten zögerlichen Anlauf mit der o. g. Frage.

Trotz der besseren Sachkenntnis des Drivers scheint dieser keine konkreten Vorstellungen zu haben, wie die Implementierung korrekterweise auszusehen hat. Es ist denkbar, dass diese Zögerlichkeit den Observer dazu bringt, einen einfacheren Weg zu suchen und so die Empfehlung zu umgehen.

5. Ebenfalls in Sitzung Sitzung D03 haben die Entwickler ein Versagen der Software festgestellt und die Stacktrace wird angezeigt. Der Observer *P1* beginnt parallel mit dem Driver *P2* zu lesen:

   P1: *„(#UpdateView#) (..) (#BasicLabel#) (!!...!!) Oh das hängt jetzt (!...!) warte mal. Das hängt jetzt schon mit dem Model zusammen, das ich erweitert hab.“*

   Der Driver *P2* entgegnet dem nur *„OK (.) dann, wart' mal kurz.“*[30], woraufhin der Observer *P1* unbeirrt den Vorschlag unterbreitet:

   P1: *„Entweder (.) kommentieren wir das aus (!...!)“*

   Dass sich *P1* für das Programmversagen verantwortlich fühlt, könnte erklären, warum er sich trotz der abweisenden Haltung seines Partners so aktiv in den Prozess einbringt.

6. An anderer Stelle in Sitzung D03 haben die Programmierer gerade einige Codezeilen in eine Methode extrahiert. Der Driver *P2* führt eine Tastenkombination aus, die aber keinen Effekt hat, was ihn sichtlich irritiert (er blickt zwischen Tastatur und Bildschirm hin und her). Er wiederholt die Kombination und sagt dazu halblaut *„Wird's bald?“*. Fast zeitgleich gibt der Observer *P1* die Anweisung:

   P1: *„Jetzt noch da unten verwenden dann.“*

   *P2*s Tastenkombination funktioniert nun offensichtlich und er stimmt dem Vorschlag von *P1* zu: *„Genau.“*.

   Im Prinzip haben hier beide Entwickler die gleiche Aktivität übernommen: sie suchen nach den passenden Codestellen bzw. denken über geeignete Codestrukturen nach. Der Driver *P2* war hier aber noch durch andere Dinge abgelenkt (weil sich der Cursor nicht wie erwartet bewegte), was dem Observer *P1* einen gewissen „Vorteil“ verschaffte[31]. Aus Sicht des Observers *P1* hat der Driver *P1* zu diesem Zeitpunkt den Prozess nicht vorangetrieben, sodass er sich *verantwortlich* fühlte und eine Anweisung gab.

Weitere Bemerkungen zu dieser Klasse

Keine weiteren Bemerkungen nötig.

[30] Das ist eine Phrase, die diese Person sitzungsübergreifend verwendet, um seinen Partner zu sagen: „Sei bitte ruhig, ich möchte nachdenken.“

[31] Es handelt sich hierbei um einen Teil der auf Seite 36 beschriebenen „Mehrbelastung“, die dem Driver *freie Ressourcen* raubt.

Kode	Äußerung	Kontext/Kommentar
P1.explain_finding	„Eine Demo, ein Test und drei andere Stellen.“	Der Driver *P2* hatte zuvor eine Suche nach allen Verwendungen einer Klasse gestartet und in den ersten vier Sekunden nach Erscheinen der Trefferliste keine Anstalten gemacht, etwas zu äußern oder eine HCI-Aktivität zu beginnen. Nach diesen vier Sekunden äußerte *P1* die nebenstehende Bewertung.
P2.click_sth	<* klickt auf eine Fundstelle *>	*P2* sagt weiterhin nichts, klickt aber unmittelbar nach P1s Bewertung, wodurch sich das entsprechende Element in der Baumansicht öffnet.
P1.explain_finding	„Das ist unsere Stelle.“	*P1* äußert seine neue Einschätzung weniger als 2 Sekunden nachdem *P2* dieses Element geöffnet hatte, also vermutlich unmittelbar nachdem *P1* erkannt hat, um welche Klasse es sich dabei handelt.

Tabelle 4.3: Beispiel für eine Verantwortungsübernahme durch den Observer. Die Zeitspanne zwischen Ereignis und Verbalisierung der Erkenntnis ist beim zweiten Mal deutlich kürzer, was für eine zuerst übernommene und dann „behaltene“ Verantwortung spricht. Der Driver ist die gesamte Zeit über still und kommentiert erst die letzte Äußerung des Observers (nicht abgedruckt). Die Episode stammt aus Sitzung D03.

4.1.3 *guidance*[32] (Dirigieren)

Allgemeine Beschreibung

Während einer Sitzung kommt es hin und wieder vor, dass der Observer dem Driver Anweisungen gibt, wohin dieser navigieren soll. Es kann sich dabei um Stellen im Code handeln (bestimmte Klassen/Methoden) oder um GUI-Elemente verschiedener Anwendungen (Menüpunkte, Buttons, etc.).
Diese Anweisungen stellen einen Spezialfall von *work_proposal* (siehe voriger Abschnitt) dar, sind aber konkreter und kleinteiliger formuliert und haben einen „räumlichen" Bezug, da entweder ein benanntes Ziel oder eine Richtung (hoch, runter, „da hin", etc.) vorgegeben wird.

Behandeltes Phänomen

Für diese Klasse von Beeinflussungen konnten zwei verschiedene Phänomene identifiziert werden, die auf die gleiche Art und Weise mit einem *guidance* behandelt werden. Dadurch ergibt sich in diesem und in den nächsten Punkten eine inhaltliche Zweiteilung. Bei den beiden Phänomenen handelt es sich um:

1. **Unsicherheit beim Partner.** Der Driver hat offensichtlich nur sporadische Kenntnisse im Umgang mit der gerade benutzten Anwendung oder es fehlt ihm die Orientierung im Programmcode.

   Geht es um die Bedienung einer Anwendung wartet der Observer gewöhnlich einige Augenblicke ab, um zu sehen, ob der Driver den nächsten Schritt selbst vollziehen kann, und schreitet dann im Bedarfsfall ein.

   Im Falle von Orientierungslosigkeit im Quellcode greift der Observer dagegen schneller ein.

2. **Verwehrtes Engagement.** Der Observer wollte eigentlich selbst aktiv werden (siehe Abschnitt 4.1.8), was ihm aber vom Driver verwehrt wurde. In der Konsequenz übernimmt der Observer die „Leader"-Rolle[33] und diktiert dem Driver die Schritte, die er ansonsten selbst ausgeführt hätte.

Der erste Fall ist also ein Dirigieren in Form einer Hilfestellung für den Partner; der zweite entspricht eher einem Anleiten oder Anführen.

Ursächliche Bedingungen

Für die beiden o. g. möglichen Phänomene konnten auch verschiedene Ursachen identifiziert werden:

1. **Unsicherheit beim Partner.** Hier gibt es einerseits die Ursache für die Unsicherheit an sich, und andererseits die Ursache dafür, dass der Sprecher den entsprechenden Eindruck gewonnen hat.

[32] Rein syntaktisch unterscheidet sich der Name dieses Konzepts von den anderen sieben. Der Grund hierfür ist aber keine inhaltliche Sonderstellung, sondern lediglich die Begrenzung meiner Sprachkenntnisse und die fehlende Bereitschaft, Sprache zu verbiegen. Dass die Wahl hier nicht auf ein zweiteiliges Wort viel, scheint vielleicht nicht optimal, allerdings war die Schaffung eines völlig reibungsfreien Vokabulars nicht das primäre Ziel dieser Arbeit. Falls doch dieser Anspruch erhoben werden sollte, kann ich noch *guide_partner* oder *partner_guidance* anbieten.

[33] Diese Rolle wurde bereits in [31] erwähnt.

Der Eindruck von Unsicherheit kann auf vielfältige Weise entstehen; konkret beobachtet wurden die folgenden Indizien (in absteigender Ausdrücklichkeit):

- Der Partner hat sich selbst zu seinem niedrigen Wissenstands geäußert (durch ein *explain_standard of knowledge*).
- Der Partner hat im Vorfeld Fragen formuliert, die auf weitere Defizite schließen lassen (z. B. durch bestimmte *ask_step*, *ask_knowledge*[34]), widersprüchliche Aussagen getätigt oder „unsinnige“ Vorschläge unterbreitet.
- Der Partner äußert ein isoliertes *„Äh“*, was wie ein unspezifisches *„Und was jetzt?“* interpretiert werden kann.
- Der Partner tippt nur zögerlich oder bewegt die Maus stockend, so als wisse er nicht, wohin er klicken soll.

Die Gründe für die Unsicherheit an sich können vielfältig sein. Denkbare Ursachen umfassen fehlendes Fachwissen oder das Gefühl, dem Partner unterlegen zu sein. Hierbei ist zu beachten, dass sich Fachwissen nicht automatisch mit Berufserfahrung gleichsetzen lässt. In Sitzung F02 haben bspw. ein Auszubildener und ein Praktikant zusammen gearbeitet. Der Praktikant war erst seit einer Woche bei der Firma, trotzdem machte der Auszubildene wesentlich häufiger einen unsicheren Eindruck. Letztlich reichen die Ursachen für die Unsicherheit aber aus dem Fokus dieser Arbeit und wurden nicht weiter untersucht.

2. **Verwehrtes Engagement.** In den beobachteten Fällen wurde die Intention, die Driver-Rolle zu übernehmen, nicht oder nicht ausdrücklich verbalisiert. Aus Forschersicht konnten immer physische Anzeichen dafür beobachtet werden (wie z. B. aufrechtes Hinsetzen, Bewegen der Hände in Richtung von Maus oder Tastatur, siehe Beispiele weiter unten), die allerdings nicht im Sichtfeld der Partner lagen, weil diese ihren Blick auf den Bildschirm gerichtet hatten.

Rolle des Kontexts und der intervenierenden Bedingungen

Auch hier unterscheide ich wieder zwischen den beiden Fällen von Hilfestellung und Anführung.

1. **Unsicherheit beim Partner.** In erster Linie muss der Sprecher die Unsicherheit seines Partners natürlich wahrnehmen. Bei der Analyse konnte vor jeder *guidance*-Episode eines oder mehrere der o. g. Indizen beobachtet werden, sodass es zumindest recht wahrscheinlich ist, dass der jeweilige Sprecher sich von eben diesen Anzeichen hat leiten lassen. Nichtsdestotrotz könnte es auch noch weitere Indizien geben, was sich z. B. (zum Teil) mit einer kurzen Befragung der Probanden direkt im Anschluss erörtern ließe.

   Um dann auch tatsächlich helfen zu können, muss der Sprecher über entsprechendes Wissen oder zumindest ein gewisses Grundverständnis verfügen. Oft reichen im letzteren Fall kleine Hinweisreize (z. B. die Anzeige aller möglichen Menüpunkte in einer Auswahlliste) aus, damit sich der Sprecher erinnert und kann so seinen Partner schon dirigieren, bevor dieser überhaupt die Zeit hatte, alle Punkte durchzulesen (siehe Beispiel weiter unten).

[34] Nicht jedes *ask_step/knowledge* lässt derartige Rückschlüsse zu. Vielmehr ist das beim Fragenstellen mitübertragene Wissen entscheidend.

2. **Verwehrtes Engagement.** Die Voraussetzung für dieses Verhalten sind einerseits die Kompetenz, die nötig ist, um das weitere Vorgehen zu planen, sowie die Motivation, dies auch zu tun. Andererseits muss der erste Versuch der Umsetzung gescheitert sein und die Motivation darf nicht so groß sein, dass der Sprecher nicht einfach ein zweites Mal fragt.

Beispiele

1. **Unsicherheit beim Partner.**

   – In Sitzung F02 soll eine neue Funktionalität implementiert werden, die es so ähnlich schon in der Anwendung gibt. Der Observer *P2* schlägt vor, die Implementierung dieser Referenzstelle aufzusuchen und sich daran zu orientieren. Der Driver *P1* navigiert selbstständig durch die Liste der verfügbaren Projekte und Pakete, kommt dann aber nicht weiter und liest nacheinander die Namen der angezeigten Klassendateien vor. Als dies *P2* bemerkt, äußert er den Vorschlag:

     P2: *„Ja (.) guck mal in den ähm in der plugin.xml.“*

     P1 hört sofort auf, die Klassennamen vorzulesen, als er die Hilfestellung von *P2* bemerkt. Danach navigiert er sofort zur genannten Datei.

   – In Sitzung D07 versuchen die Programmierer einen Testfall zu schreiben, der nicht fehlschlägt. Aktuell wird jedoch eine Exception geworfen und eine Stacktrace angezeigt. Der Driver *P1* schaut die Liste angestrengt an und klickt nach einiger Zeit auf einen Eintrag, stellt dann aber fest, dass die geöffnete Codestelle nicht die richtige sein kann. Der Observer *P2* bringt sich an dieser Stelle ein und sagt, während *P1* mit dem Mauscursor über die Listeneinträge fährt:

     P2: *„(∼) das FeatureNew (!...!) achso richtig, genau, da (.) ja.“*

     P1 klickt auf einen Eintrag; unmittelbar danach äußert *P2* sein *„ja“*.

   – In Sitzung D06 haben die Entwickler beschlossen, die Anwendung vor dem Einchecken der Codeänderungen zu testen. Der Driver *P2* äußert, während er mit der Maus über die Menüleiste der IDE fährt:

     P2: *„Oh Gott (∼) (.) wie läuft'n das noch (.) hier muss ich machen?“*

     P1 bejaht das sofort, *P2* öffnet ein Menü mit knapp 40 Auswahlmöglichkeiten, woraufhin *P1* dirigiert:

     P1: *„Und dann '<**Programm**> Examples Windows'.“*

     P2 klickt auf den genannten Menüpunkt und die Anwendung wird übersetzt und ausgeführt. Also diese nach 90 Sekunden gestartet ist, bewegt *P2* den Cursor zögerlich über die GUI und ein langgezogenes *„Äh“*, woraufhin der Partner weiter dirigiert:

     P1: *„Einfach 'Attributtabelle'.“*

     Dieses Beispiel illustriert gleich zwei der o. g. Indizien für die Unsicherheit des Partners.

2. **Verwehrtes Engagement.**

   – In Sitzung D06 haben die beiden Entwickler beim Testen des Programms ein Versagen festgestellt. Aus dem Kontext heraus macht *P1* einen kompetenteren Eindruck (er hat den Testfall, der zum Versagen führte vorgeschlagen und hatte sofort eine Vorstellung, in welcher Datei nach einem Defekt zu suchen ist). Sein Versuch, die Driver-Rolle zu übernehmen, scheiterte, sodass er stattdessen seinen fachlich weniger kompetenten Partner anleitet. Die Episode ist in Tabelle 4.4 abgedruckt.

   – In Sitzung D03 hat der Observer *P2* vorschlagen, die letzte Viertelstunde für den Beginn einer neuen Aufgabe zu verwenden, wozu der Driver *P1* auch zustimmte. *P2* sagte dann:

     P2: *„Und zwar (.) zumindest mal anschauen.“*

     P2 setzte sich aufrecht hin, schob sein Glas zur Seite und bewegte beide Hände in Richtung Tastatur. *P1* stimmte mit *„Ja“* zu, behielt aber die Tastatur bei sich. *P2* lehnte sich zurück, wartete kurz während *P1* unzusammenhängende Sätze äußerte und forderte dann:

     P2: *„Zeig doch einfach mal ganz kurz das Model.“*

Weitere Bemerkungen zu dieser Klasse

Beruht das Geben eines solches Hinweises allerdings deutlich auf neuen Erkenntnissen, dann handelt es sich um ein *finding_conveyance* (siehe Abschnitt 4.1.6). Hier war der Sprecher lediglich schneller als sein Partner, bzw. hat dies als Erster verbalisiert. Die Grenze zwischen einem *guidance* durch Hinweisreize und einem *finding_conveyance* ist allerdings – wie auch die Trennung zwischen *knowledge* und *finding* in der Basisschicht – nicht scharf.

HHI-Kode	Äußerung	Kontext/Kommentar
P2.explain_finding	„Tut.“	P2 ist nach einem ausgeführten Testfall der Ansicht, dass das Programm funktioniert und die vorgenommenen Änderungen damit erfolgreich waren.
P1.propose_step (*P1.challenge_finding*)	„Mach mal, (..) änder' mal die Werte (.) und geh' wieder rein“	P1 schlägt die Durchführung eines weiteren Testfalls vor und wiederspricht damit der vorherigen Erkenntnis von P2.
P2.agree_step	„Hm (.) ja“	P2 stimmt dem Vorschlag zu und beginnt mit der Ausführung. Der Testfall schlägt fehl.
P1.explain_finding	„OK.“	*P1* erkennt als Erster, dass das Programm nicht wie gewollt funktioniert und sieht damit seine Erwartung bestätigt.
P2.explain_finding	„OK, es tut nich.“	P2 erkennt das Versagen ebenfalls.
P1.propose_step	„Zeig mal.“	P1 richtet sich dabei in seinem Stuhl auf und legt eine Hand auf den Tisch in die Nähe der Maus.
P2.propose_step	„Dann sollten wir schauen wieso.“	P2 geht auf den Vorschlag von P1 nicht ein und reagiert auch nicht auf dessen angedeuteten Wunsch, die Rollen zu tauschen. P1 zieht daraufhin seine Hand zurück.
P1.amend_step *P1.guidance*	„Die Action.“	P1 schlägt vor, eine andere Datei zu öffnen, ohne diese exakt zu benennen. Er führt damit seinen Vorschlag näher aus (*amend_step*).

Tabelle 4.4: Beispiel-Episode für *guidance*. Die beiden Entwickler stellen beim Testen des Programms ein Versagen fest. *P1* wird die Driver-Rolle verwehrt, sodass er stattdessen seinen Partner anleitet (letzte Zeile). Die Episode stammt aus Sitzung D06.

4.1.4 *knowledge_request* (Behandlung von Wissenslücken oder Verständnisschwierigkeiten)

Allgemeine Beschreibung

Der Sprecher fragt seinen Partner nach deklarativem Wissen, was in der Basisschicht am ehesten mit *ask_knowledge* annotiert werden würde. Der Begriff „Wissen“ schließt hier sowohl fachliches Wissen mit ein (also über bestimmte Technologien, oder den Aufbau und die Funktionsweise von Quellcode betreffend), als auch „prozedurales“ Wissen (Informationen über die Zielsetzung und die Art des weiteren Vorgehens). Es handelt sich also in jedem Fall um eine Aufforderung zum Wissenstransfer.

Behandeltes Phänomen

Ähnlich wie das Verhalten der Programmierer beim Entdecken eines Defektes (*defect_-announcement*), bei dem es „unsinnig“ wäre, diese Information dem Partner auf Dauer vorzuenthalten, lässt es sich bei Wissenslücken und Verständnisschwierigkeiten rational begründen, dass es eine gute Idee ist, sie anzusprechen. In dem Moment, in dem einer der Entwickler sich nicht mehr bemüht, der Sitzung geistig zu folgen – und dazu gehört auch ein relativ umfassendes Verständnis – verliert sich die Fokussierung der Aufmerksamkeit und Probleme können nicht mehr von beiden Entwicklern angegangen werden. Die Entwickler arbeiten nicht mehr im Paar und es werden tatsächlich die doppelten Arbeitsstunden ohne deutlichen Mehrwert bezahlt.

Nur weil es allerdings eine gute Idee ist, solche Defizite anzusprechen, heißt das nicht, dass sich auch jeder Zeitpunkt dafür eignet. Wenn der Partner gerade in Gedanken vertieft (*top-down*-aufmerksam) ist und ihn der Sprecher etwas fragt, kann der Partner seine Fokussierung und damit u. U. auch einen wichtigen Gedanken verlieren (*bottom-up*-aufmerksam).

Ursächliche Bedingung

Die Ursachen für fehlendes Fachwissen und mangelndes Verständnis sind verschiedener Natur. Das Nachvollziehen der Gedankengänge des Drivers (z. B. beim Mitlesen seiner Codeänderungen) ist ein häufiger Auslöser dafür, dass der Observer bemerkt, dass er etwas nicht komplett verstanden hat.

- Dass der Observer dem Prozess nicht mehr folgen kann, ist nicht zwangsläufig seine eigene Schuld. Der Driver kann z. B. einen spontanen Einfall haben, den er sofort in die Tat umzusetzen beginnt und dabei seine Aktivitäten im Eifer des Gefechts nur spärlich kommentiert. Der Observer kann dabei zwar versuchen, ihm zu folgen, es gelingt ihm aber nicht immer. Dann bittet er den Driver, ausführlicher zu erläutern, was er eigentlich vorhat (siehe Beispiel 1).

  Fälle, in denen der Driver den Observer sichtlich „abgehängt“ hat, dieser also gar nicht mehr *versucht* dem Driver zu folgen, konnten nicht beobachtet werden.

- In anderen Fällen hat der Observer beim Nachvollziehen gemerkt, dass er bestimmte Technologien nicht kennt, den Sinn eines Aufrufes nicht selbst erschließen kann oder sich schlicht in der Umgebung nicht orientieren kann. Hier ist es der Mangel an Fachwissen, der zum *Interrupt* des Partners führt (siehe Beispiel 2).

– Der in der Literatur häufig erwähnte Aspekt des „strategischen Denkens“ des Observers (z. B. [45]) konnte ebenfalls beobachtet werden. Der Observer stellt dabei z. B. fest, dass für das weitere Vorgehen noch nicht alle Fragen geklärt sind und versichert sich dann beim Driver, ob dies wirklich der Fall ist. Allerdings bezieht sich der Observer hier oft nur auf den nächsten anstehenden Arbeitsschritt und nicht auf größere Zusammenhänge (siehe Beispiele 3 und 4).

Wiederum könnte man die Fragestellung erweitern und die Ursachen dafür ergründen, *warum* einem Entwickler fachliches Wissen fehlt, oder warum der Driver seine Handlungen nicht ausreichend kommentiert. Im Rahmen dieser Arbeit wurde dieser Schritt allerdings nicht gegangen.

Rolle des Kontexts und der intervenierenden Bedingungen

– Die Grundvorraussetzung für dieses Verhalten ist die Erkenntnis, dass man selbst nicht über die benötigten Informationen verfügt. Dass das u. U. eine Weile dauern kann, zeigt Beispiel 1.

– Weiterhin muss der Sprecher einen begründeten Verdacht haben, dass der Partner auch über die gesuchte Information verfügt, z. B. weil dieser bereits Teile seines Wissens über eine bestimmte Technologie oder einen Teil des Quellcodes geäußert hat, die dem Observer bis dato unbekannt waren.

  Theoretisch wäre es denkbar, dass der Sprecher auch dann eine solche Frage stellt, wenn es keine Indizien dafür gibt, dass der Angesprochene diese beantworten kann. Die genauen Umstände solcher Fragen müssen in weiteren Untersuchungen geklärt werden.

– Auch wenn diese beiden Bedingungen erfüllt sind, muss der Sprecher nicht zwangsläufig sofort seine Frage stellen. Der Sprecher kann merken, dass es gerade nicht der günstigste Zeitpunkt für eine Zwischenfrage ist: Der Sprecher kann eine Priorisierung vornehmen. Im Falle von *knowledge_request* konnten solche Phänomene nicht beobachtet werden. Generell ist es schwierig, ein solches Phänomen zu beobachten, weil man aus Forschersicht nicht zuverlässig erkennen kann, wie lange eine Person eine Äußerung schon tätigen möchte.

  Dass es dennoch so etwas wie einen *Vorrang* gibt, hat sich bei anderen Äußerungen gezeigt (siehe Beispiel 2 von *work_proposal*). Im Laufe einer längeren Kommunikation kommt es irgendwann vor, das beide Personen zeitgleich reden wollen. Als Konsequenz nimmt sich meistens einer von beiden zurück. Als theoretische Grundlage hierfür wurde der Vorrang herausgearbeitet, der z. B. durch die Persönlichkeit und die relative Bewertung der eigenen und der fremden Aussage bestimmt wird[35]. Eine vertiefende Untersuchung in diese Richtung wurde jedoch nicht vorgenommen.

Beispiele

1. In Sitzung D06 hat der Driver *P1*, nachdem er eben schon einige Klassen verschoben hatte, angekündigt noch eine weitere Klasse umzulagern (*propose_design*). Während er damit beginnt, äußert *P2*:

[35] Diese Bewertungen erfolgen i. d. R. sehr schnell, sodass eigentlich gar keine Zeit bleibt, den *Inhalt* der Äußerung des Partners zu erahnen, oder sogar zu erfassen. Stattdessen werden vermutlich die bisherigen Äußerungen extrapoliert.

P2: *„Wo bist du? In was für ner Klasse? Oder in was für nem Modul?“*

Als Reaktion darauf erklärt *P1* noch einmal ausführlich, was er in den letzten Minuten getan hatte. Offensichtlich hat *P2* erst jetzt bemerkt, dass er gar nicht mehr nachvollziehen kann, was der Driver tut.

2. In Sitzung St10B hat der Driver *P2* vorgeschlagen eine bestimmte Datei zu öffnen, um sich von dort aus besser orientieren zu können. Er öffnet darauf hin mehrere Ordner im „Package Explorer“. Der Observer *P1* beobachtet das für einige Sekunden, lehnt sich dann zurück und äußert:

   P1: *„Hm, wir sind jetzt in (∼) (!...!)“*

   P2 beginnt sofort darauf zu erklären, was er an der Ordnerstruktur geändert hat. *P1* kam zwar nicht dazu eine echte Frage zu stellen, aber aus seiner Intonation hat *P2* trotzdem erkannt, dass *P1* nicht versteht, was gerade passiert und eine Erklärung benötigt.

3. Dieses Beispiel ist bereits aus Kapitel 3 als Szenario 3 bekannt. Zur Erinnerung: Während der Driver eine längere URL aus dem Gedächtnis abtippt, stellt ihm der Observer eine Frage zum weiteren Vorgehen (*„cURL rufst du auf der Shell auf, ne?“*), wodurch der Driver zumindest kurz ins Stocken gerät.

4. In Sitzung D03 haben die beiden Entwickler die Notwendigkeit einer Anpassung in der GUI festgestellt und der Driver hat sich dem Code zugewendet, um die nötigen Änderungen vorzunehmen. Der Observer hat (wie er später in der Sitzung auch selbst sagt) wenig Erfahrung mit den verwendeten Klassen und stellt dem Driver wiederholt Zwischenfragen, während dieser „eigentlich“ programmiert. Die entsprechende Episode ist in Tabelle 4.5 aufgeführt.

Weitere Bemerkungen zu dieser Klasse

Bei ihrer Untersuchung haben Williams et al. [45] festgestellt, dass die individuell arbeitende Kontrollgruppe zwei bis drei Mal so viele technische Fragen an die Versuchsleitung gestellt hat, wie die paarweise arbeitende Experimentalgruppe. Die Autoren begründen dieses Phänomen mit dem Wissensaustausch der Paarprogrammierer, die einen Großteil der Wissenlücken gegenseitig schließen oder umgehen konnten.

Die genannten und oben beschrieben Ursachen von Wissenstransfer können natürlich nur bei Paaren auftreten. Einzelprogrammierer kommen vielleicht gar nicht auf die Idee, dass ihnen Fachwissen fehlt. Umgekehrt wäre es durchaus möglich, dass Entwickler erst durch den Partner auf mögliche andere Lösungswege aufmerksam werden und so auch mal komplizierter als nötig denken. In diesem Fall wäre die Tatsache, dass ein Fachwissentransfer stattfindet, mit Hinblick auf die zu lösenden Aufgabe sogar ein negativer Aspekt.
Die genauen Mechanismen, wie Wissen bei der Paarprogrammierung (oder auch in einem Team mit wechselnden Paaren) transferiert wird, und welche Arten von Wissen das sind, müssen noch genauer untersucht werden. Dies wird aber anderen Arbeiten überlassen.

Kodes	Äußerung	Kontext/Kommentar
P1.propose_design + P1.knowledge_request	„Macht es denn Sinn das (.) den ToggleComponentEnabler jetzt dahingehend zu erweitern?“	*P1* macht einen Designvorschlag, hinter dem auch eine Frage nach Fachwissen steht.
P2.challenge_design	„Nein, den brauchen wir nicht erweitern. Wir brauchen nur das (.) labelScaleRangePanel zu erweitern.“	*P2* beantwortet die Frage und macht einen Gegenvorschlag.
P1.agree_design	„Achso, OK.“	*P1* stimmt dem Vorschlag und scheint seine Frage beantwortet zu sehen.
Es folgt eine Sprechause, in der *P2* eine Codezeile veändert. *P2* klickt eine Fehlermeldung an, wodurch sich eine Liste von vorgeschlagenen automatischen Korrekturen öffnet.		
P2.think aloud_activity	„(#IEnableable#)“	*P2* liest einen Teil einer der Einträge aus der Liste vor.
P1.propose_design + P1.knowledge_request	„Muss das Panel dann dieses Interface implementieren, oder ...?“	*P1* macht wieder einen Designvorschlag. Dieses mal ist die Frage nach Fachwissen deutlich zu erkennen.
P2.think aloud_activity	„IEn (!...!) IEnableableComponentContainer.“	Fortsetzung des vorigen *think aloud*s. Er unterbricht sich selbst, als er merkt, dass *P1* noch spricht.
P2.agree_design	„Genau, das hier“	Direkt nach seinem *think aloud* stimmt er dem Vorschlag von *P1* zu und beantwortet damit dessen Frage.

Tabelle 4.5: Episode, in der der Observer dem Driver zweimal eine Wissensfrage stellt, während dieser beschäftigt ist. Beim zweiten *knowledge_request* ist der Einfluss auf *P2* daran zu erkennen, dass er drei Anläufe braucht, um das Wort „IEnableableComponentContainer“ auszusprechen. Die Episode stammt aus Sitzung D03.

Wiederum steht diese Klasse nicht vollkommen isoliert zwischen den restlichen Beeinflussungkonzepten, sondern hat gewisse Schnittstellen:

– Die Äußerung aus Beispiel 3 könnte man auch so interpretieren, dass der Sprecher gar nicht wirklich *wissen* wollte, wie das weitere Vorgehen vom Driver angedacht ist, sondern bereits selbst darauf gekommen ist, und die Frage nur als Stilmittel benutzt. In diesem Fall würde er seinem Partner *anzeigen*, dass er dem Prozess weiterhin folgt und damit würde dieses Verhalten in die Klasse *finding_conveyance* fallen.

4.1.5 *knowledge_conveyance*[36] (Wissen „nachliefern“)

Allgemeine Beschreibung

Analog zur einer Wissenslücke bei sich selbst kann der Sprecher auch beim Partner ein *Wissensdefizit* vermuten und dessen Aktivitäten mit einem unaufgeforderten Wissenstransfer unterbrechen. Dabei ist mit dem Begriff *Wissensdefizit* gemeint, dass es sich um Wissen handelt, über dass der Partner (aus Sicht des Sprechers) idealerweise Verfügen sollte, von dem der Partner aber noch nicht einmal weiß, dass es überhaupt existiert. Mit den nachfolgenden Beispielen wird es noch klarer, wie dieses Konzept gemeint ist.

Wenn der Sprecher mit seiner Äußerung die aktuelle Aktivität des Partner stoppen und ersetzen möchte (*replace_activity*), weil ihm z. B. ein besserer Weg eingefallen ist, dann handelt es sich wegen der hohen *Negativity* um eine *Intervention*. Möchte er hingegen nur ein Missverständnis aufklären, das ihm erst im Nachhinein aufgefallen ist, dann unterbricht er die Aktivität nur und es handelt sich um einen *Interrupt*.

Behandeltes Phänomen

Dem Sprecher fällt auf, dass er über Wissen verfügt, dass dem Partner weiterhelfen könnte. Wie schon bei den Konzepten *defect_announcement* und *knowledge_request* wäre es nicht von Vorteil, dem Partner derartig prozessrelevante Informationen vorzuenthalten.

Ursächliche Bedingungen

Hier müssen zwei Fälle unterschieden werden:

1. Der Partner kann nicht über das Wissen verfügen. Die Ursachenanalyse würde hier der gleichen Frage folgen wie bei *knowledge_request*: Warum verfügt der Partner nicht über bestimmtes Wissen?

   Genau wie bei *knowledge_gap* führten die Betrachtungen dieser Fragestellung zu weit vom Fokus dieser Arbeit weg.

2. Der Partner sollte eigentlich über das Wissen verfügen, wendet es aber nicht korrekt an. Auch hier führt die Untersuchung der Ursachen weit vom Kern der Arbeit weg. Vereinfacht könnte man zusammenfassen, dass Menschen (1.) nicht zuverlässig auf alle Gedächtnisinhalte zugreifen können und (2.) Dinge auch komplett vergessen können.

Rolle des Kontexts und der intervenierenden Bedingungen

Kurzgefasst gibt es folgende zwingenden Bedingungen für das Auftreten dieses Typs:

- Der Sprecher muss über das nötige Wissen selbst verfügen.
- Der Sprecher muss erkennen, welches Wissen in der aktuellen Situation hilfreich ist.
- Und es muss ihm auffallen, dass der Partner über das Wissen nicht verfügt bzw. falls er darüber verfügen müsste, dass er es nicht angewendet hat (siehe Beispiel 2).

[36] engl. *conveyance* = Vermittlung, Übertragung

Beispiele

1. In Sitzung D09 suchen die Entwickler nach der „Einsprungstelle“, also der Klasse oder Methode, in der sie ihre ersten Änderungen vornehmen wollen. Der Observer *P1* hatte bereits angemerkt, dass er die entsprechende Stelle kennt und hat eine entsprechende Wegbeschreibung begonnen. Der Driver *P2* beginnt daraufhin den Code selbstständig nach vielversprechenden Verweisen zu durchsuchen, als *P1* wieder die tatsächliche Einsprungstelle einfällt:

   P1: *Äh, ne, wart mal (.) des gibt Factories dafür (.) und die Factories erzeugen dann die einzelnen Menüpunkte. Also du hast nur eine Factory (.) und die erzeugt alle (.) Menüpunkte."*

   P2 kann diese Informationen sofort umsetzen und hat schon nach der ersten Erwähnung von „Factories“ die betreffende Klasse markiert und kurz darauf geöffnet. Als *P1* seine Äußerung beendet und wieder auf den Bildschirm blickt, ist die Klasse schon seit drei Sekunden geöffnet. Somit hat die *Intervention* von *P1* auf den Arbeitsfluss von *P2* gar keinen störenden Effekt, was aber wohl eher an der Fähigkeit von *P2* liegt, gleichzeitig zuhören und im Code navigieren zu können[37].

   Der Partner verfügte in diesem Fall nicht über das nötige Wissen.

2. In Sitzung D03 haben die Entwickler die anzupassende Anwendung im Debug-Modus gestartet, wodurch eine Vielzahl von Änderungen sofort sichtbar wird, ohne dass dafür die Anwendung geschlossen, die Quellcodeübersetzung manuell neu gestartet und die Anwendung wieder geöffnet werden muss. Der Driver *P2* hat die Anwendung dennoch geschlossen, worauf ihn der Observer *P1* hinweist, als dieser sieht, welche Änderung *P2* vorgenommen hat:

   P1: *„Aber dann (.) hättste nicht schließen müssen. Das war ja im Debug-Modus.“*

   Worauf *P2* entgegnet: *„Achso! (.) Naja, denk ich nicht dran.“*.

   In diesem Beispiel verfügte der Partner zwar prinzipiell über das Wissen, hatte es aber nicht bemerkt, dass er es für den aktuellen Anwendungsfall nutzen kann.

Weitere Bemerkungen zu dieser Klasse

Über die Klassen *knowledge_request* und *knowledge_conveyance* lässt sich bei Weitem nicht der gesamte Wissenstransfer abbilden, da sie ja nur Fragen und Antworten „außer der Reihe“ abdecken. Jede Antwort auf eine *knowledge_gap*-Frage ist natürlich vom (ursprünglichen) Sprecher erwartet, ebenso wie *ask_knowledge*s während einer Diskussion.

[37] *P2* ist übrigens weiblich.

4.1.6 *finding_conveyance* (Erkenntnisse liefern)

Allgemeine Beschreibung

Während im vorigen Abschnitt der Typ *knowledge_conveyance* beschrieben wurde, der ganz bewusst eine namentliche Verwandtschaft zu den *knowledge*-Konzepten der Basisschicht hat und das Hauptaugenmerk auf Bestandswissen legt, folgt in diesem Abschnitt mit dem Konzept *finding_conveyance* das entsprechende Pendant, das auf Erkenntnissen beruht.

Aus Sicht der Basisschicht handelt es sich bei den Äußerungen, die mit diesem Konzept annotiert werden häufig um *explain_finding* und um *think aloud_activity*.

Phänomen

Im Laufe der Analyse haben sich zwei Phänomene herauskristallisiert, die mit ähnlichen beobachtbaren Verhaltensweisen behandelt werden:

1. **Erkenntnis, die der Partner noch nicht hatte.** Wie bei anderen in diesem Kapitel beschrieben Klassen auch, spielen neue Erkenntisse eine wichtige Rolle. Der Sprecher bemerkt eine Veränderung seiner Umwelt, bewertet die Situation neu und informiert seinen Partner im Bedarfsall. In diese Klasse fallen entsprechende Beeinflussungen aber nur dann, wenn sie nicht schon durch eine der anderen Klassen abgedeckt sind.

   Die Veränderung in der Umwelt muss sich dabei nicht zwangsläufig objektiv manifestieren. In Beispiel 2 fällt dem Sprecher ein ihm unbekanntes Paket ins Auge – vorhanden und auch sichtbar war es aber schon längere Zeit. Die Veränderung ergab sich nur subjektiv für ihn – vermutlich durch eine Veränderung seines Aufmerksamkeitsfokus.

2. **Aufmerksamkeit anzeigen.** Es kommt vor, dass der Observer zu der Aktivität des Drivers laut mitdenkt und so z. B. „Typ P"-Erkenntnisse[38] zeitlich vor dem Driver verbalisiert, obwohl er selbst gar keine HCI-Aktivitäten ausführt (außer vielleicht *read_sth*). Vermutlich zeigt er damit dem Partner (u. a.) an, dass er ihm weiterhin gedanklich folgt.

   War der Observer vorübergehend verunsichert und konnte dem Prozess nicht hundertprozentig folgen, findet dann aber wieder den Anschluss, dann kann das resultierende Verhalten wie ein *guidance* aussehen, weil der Sprecher Schritt-für-Schritt-Anweisungen gibt. Es ist aber kein *guidance*, weil es sich dabei nicht um wirkliche Anweisungen handelt, sondern nur um ein Nachvollziehen (vielleicht auch Antizipieren) der Schritte handelt.

Ursächliche Bedingungen

Für die „echten" Erkenntnisse, die auch für den Partner neu sind, bedarf es vom Sprecher manchmal nur einer „zündenden Idee". Am Beispiel in Tabelle 4.3 zeigt sich in der ersten Zeile keine revolutionäre Erkenntnis, sondern eher eine Einsicht: *P1* fasst die angezeigten Suchergebniss nur laut zusammen. Vielleicht erhofft er sich, dass das laute Aussprechen bei ihm oder seinem Partner eine Erkenntnis provoziert (siehe Anhang B.3

[38] *explain_findings* vom Typ *P* verbalisieren „die Wahrnehmung [...] von sich zeitnah in der Umgebung der Person abgespielten Ereignissen". [29, Abschn. 7.6.1])

für weitere Gedanken zu diesem Punkt). Eine andere, nicht unbedingt disjunkte Erklärung dafür könnte auch sein, dass der Sprecher den Prozess bzw. Sprechfluss nicht ganz versiegen lassen wollte, als der Driver länger keine Äußerungen mehr getätigt hatte.

Rolle des Kontexts und der intervenierenden Bedingungen

Zu diesen Aspekten sind keine weiteren Aussagen zu treffen.

Beispiele

1. In Sitzung St10B hat der Observer *P1* sich beinahe so verhalten, als würde er seinen Partner dirigieren. Dennoch handelte es sich dabei nicht um ein *guidance*, sondern um ein *finding_conveyance*. Sein Partner *P2* zeigte zwar zu Beginn der Episode Anzeichen von Unsicherheit, hat sich dann aber wieder erinnert und danach zielsicher die gesuchte Datei gefunden. *P1*s Kommentare sind daher am ehesten so zu verstehen, dass er *P2* anzeigen möchte, dass er sich wieder zurecht findet. Die vollständige Episode findet sich in Tabelle 4.6.

2. In Sitzung D09 stellt der Driver *P2* fest, dass die eben vorgenommenen Codeänderungen eingecheckt werden können und öffnet den entsprechenden Dialog. Observer *P1* sagt plötzlich:

   P1: *„Da gibt's ne Demo, was macht denn die?“*

   Auf *P2* wirkt diese Äußerung als ein *Interrupt*. *P2* entgegnet:

   P2: *„Keine Ahnung (.) können wir gucken.“*

   P1 äußert seine Erkenntnis unmittelbar nachdem er das entsprechende Paket bemerkt hat. Zu sehen war das Paket schon länger, dass es *P1* jetzt erst auffällt, spricht dafür, dass sich nach Beendigung seiner „Beobachtungsspflicht“ – das Einchecken von Codeänderung ist für dieses Paar ein Routinevorgang – sein Aufmerksamkeitsfokus weitet und für neue Eindrücke empfänglich wird.

Weitere Bemerkungen zu dieser Klasse

Keine weiteren Bemerkungen.

Kodes	Äußerung	Kontext/Kommentar
P2.search_sth *P2.explain_finding*	<* *öffnet Ordner* `java/xpetstore/domain` *> Nee, ne? <* *schließt die Ordner wieder** >	Das Paar hat entschieden, eine Klasse zu suchen, weil diese ein guter Einsteigspunkt sei. Die ersten Versuche schlagen aber fehl.
P1.mumble_sth	„Hm (.) wir, wir sind jetzt in (!...!)“	*P1* lehnt sich dabei zurück. Ihm ist die Ordnerstruktur offenbar nicht geläufig, stellt aber keine vollständige Frage.
P2.explain_knowledge	„Das ist das Projekt. Wir hatten früher immer nur (..) einen Teil daraus. (.) Das musste ich aber mal umstellen, weil sonst gings gar nicht mehr zu exportieren.“	*P2* scheint die Unsicherheit von *P1* sofort nachvollziehen zu können und erklärt, was sich am Verzeichnisbaum geändert hat.
P2.search_sth	<* *öffnet Ordner* `java` *>	Parallel zu den letzten Augenblicken der Äußerung von *P2*.
P1.propose_step	„Achso, dann *(#java#)* dann ähm (!...!)“	*P1* kann mit der Erklärung von *P2* offensichtlich etwas anfangen und beginnt sein *finding_conveyance*.
P2.search_sth *P2.explain_finding*	<* *öffnet Ordner* `util` *> „Ach doch, da ist es doch alles.“ <* *zeigt auf Ordner* `services` *>	*P2* findet sich nun doch zurecht.
P1.amend_step	*„(#services#)“*	*P1* liest einen Ordnernamen vor.
P2.search_sth	<* *öffnet Ordner* `services` *>	*P2* öffnet den Ordner, auf der er schon zuvor mit dem Cursor gezeigt hatte.
P1.amend_step	*„(#mail#)“*	*P1* liest einen Ordnernamen vor.
P2.search_sth	<* *öffnet Ordner* `mail` *>	*P2* öffnet den Ordner zeitgleich mit *P1*s Äußerung.
P1.amend_step	*„(#ejb#)“*	*P1* liest einen Ordnernamen vor.
P2.search_sth	<* *öffnet Ordner* `ejb` *>	*P2* öffnet den Ordner zeitgleich mit *P1*s Äußerung.
P1.explain_finding	„Da war se.“	*P1* hat die gesuchte Datei entdeckt. *P2* äußert sich dazu nicht.

Tabelle 4.6: Beispiel-Episode für *finding_conveyance*. Die Episode stammt aus Sitzung St10B.

4.1.7 *proposal_rejection* (Zurückweisen eines Vorschlags)

Allgemeine Beschreibung

Wie schon erläutert, können Beeinflussungen verschiedene *Negativity*-Werte haben: wenn sie der zu beeinflussenden Aktivität ablehnend gegenüberstehen (hoher Wert) handelt es sich um *Interventionen*, wird hingegen gar keine Wertung vorgenommen (niedriger Wert), handelt es sich um *Interrupts*.
Eine besondere Form von *Interventionen* richtet sich gegen die *Tatsache*, dass gerade eine Aktivität abläuft und nicht gegen den Inhalt. Die *Negativity* ist hier ebenfalls hoch, aber die Art der Beziehung zur Aktivität ist eine andere.

Behandeltes Phänomen

Diese Art der Beeinflussung wurde hauptsächlich (aber nicht nur) von einer einzelnen Person ausgeübt. Die simpelste Erklärung für ein entsprechendes Verhalten ist, dass der Sprecher in Ruhe nachdenken möchte und deswegen z. B. eine Erklärung seines Partners unterbindet.

Ursächliche Bedingungen

Die Ursachen für dieses Verhalten sind evtl. in der Persönlichkeit der Sprecher zu finden (siehe dazu auch das Konzept *action_taking* im nächsten Abschnitt).

Rolle des Kontexts und der intervenierenden Bedingungen

Zum Umfeld, in dem diese *Interventionen* gemacht werden, lässt sich v. a. festhalten, dass der jeweilige Sprecher sich konzentrieren muss, um ein Problem zu durchdenken, oder ein Programmartefakt zu verstehen.

Beispiele

Fast alle Vorkommen dieser Art, die des o. g. „hauptsächlichen“ Sprechers lauteten:

„Wart’ mal kurz!“

Zu den unterbrochenen Äußerungen der Partner gehören z. B.

- ▻ Partner: *„Dass man vielleicht einfach ähm (!!...!!)“* (Sitzung D03)

  Hier ist der Sprecher offenbar stark in seine Überlegungen vertieft, da er auch nicht auf die vorher gestellte fachliche Frage seines Partners reagiert hat.

- ▻ Partner: *„Ah, wir sollten die (!!...!!) Stop <**Entwicklername**>! Wir sollten die, die Liste, die Liste oben einfach initialisieren“* (Sitzung D03, direkt im Anschluss)

  Der Sprecher unterbricht seinen Partner wiederum, allerdings gibt dieser dieses mal nicht auf und beendet seinen Vorschlag.

- ▻ Partner: *„(#getVirtualAttributes#), markier’ die mal (!!...!!)“* (Sitzung D06)

  Das Paar sucht gerade nach einem Defekt, der Partner hat einen Vorschlag, den der Sprecher aber unterdrückt.

Weitere Bemerkungen zu dieser Klasse

Es gibt einen fließenden Übergang zum Typ *action_taking* (siehe nächster Abschnitt), wenn der Sprecher den Partner zum Aufhören auffordert, um selbst ranzukommen.

4.1.8 *action_taking* (Selbst aktiv werden)

Allgemeine Beschreibung

Eine einzelne Person hat in den Sitzungen wiederholt die Hoheit über Maus und Tastatur eingefordert, um danach durch Teile des Programmcodes zu scrollen oder die ausgeführte Anwendung auszuprobieren. Dabei wurden jeweils bereits laufende Erläuterungsversuche des Partners unterbrochen.

Auf der *Negativity*-Skala ist dieses Verhalten nicht allzu hoch anzusetzen. Der Schwerpunkt liegt für den Sprecher darauf, dass er die Aktivität *selbst* ausführen kann, und nicht zwangsläufig darauf, dass die bisherige Aktivität falsch sei. Es handelt sich also um *Interrupts*.

Behandeltes Phänomen

Eine Begründung für dieses Verhalten könnte ein starkes Informationsbedürfnis sein, dass der Sprecher unbedingt stillen möchte. Da die Erklärungen des Partners nicht zu Ende angehört wurden, schien die betreffende Person keine hinreichende Informierung mehr zu erwarten, bzw. sah sich selbst nicht im Stande die benötigten Informationen in Kürze zu erfragen oder die nötigen Anweisungen zum Ausprobieren der Anwendung zu erteilen.

Ursächliche Bedingungen

Das Bedürfnis, selbst die Maus und die Tastatur zu bedienen, wird auch von Williams et al. [45] erwähnt. In einem mehrstufigen Experiment mit Studenten haben alle Teilnehmer[39] angegeben, dass sie ihren ständigen Drang, dem Partner Tastatur oder Maus abzunehmen, erst nach der zweiten Übung überwunden hatten.

Die betreffende Person hatte allerdings in einem Fragebogen angegeben, bereits 6 Jahre Erfahrung mit Paarprogrammierung zu haben, sodass es sich hier wohl nicht um eine „Kinderkrankheit“ handelt.

Bei dem im Anschluss durchgeführten Interview hat er ebenfalls eingeräumt, dass er die Tastatur „ein wenig zu viel greift“. Er ist sich dieses Verhaltens also durchaus bewusst.

In Abbildung 4.1 sind die Rollenverteilungen der Sitzungen, in denen dieser Programmierer (beide Male als *P2*) teilgenommen hat, abgebildet[40]. Während in Sitzung D03 das Verhältnis der Driverzeiten deutlich zu Gunsten von *P2* ausfällt, sind in Sitzung D06 die Rollen „gerechter“ verteilt.

[39] Während des Experiments wurden die Studenten in zwei Gruppen eingeteilt: in einer arbeiteten die Studenten alleine, in der anderen Gruppe paarweise. Gemeint sind hier natürlich alle Teilnehmer der *Paarprogrammierung*sgruppe.

[40] Dieser Entwickler ist übrigens der einzige, der in zweien der untersuchten Sitzungen auftaucht. Durch diesen Umstand ließen sich sitzungsübergreifende Vergleiche anstellen, die bei den anderen Personen so nicht möglich waren.

Es scheint also, als ob die Ursache nicht unbedingt eine Eigenschaft der Persönlichkeit von *P2* ist, sondern eher in der *Kombination* der beiden Persönlichkeiten von *P1'* und *P2*[41].

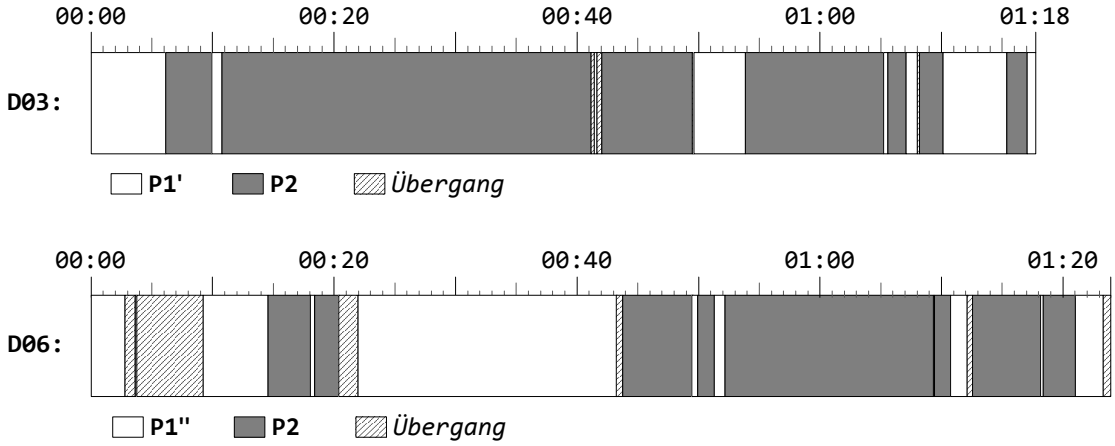

Abbildung 4.1: Driver/Observer-Verteilung von Sitzung D03 (oben) und Sitzung D06 (unten). Abgebildet sind die Phasen, in denen die jeweilige Person die Tastatur oder die Maus in der Hand hatte. Phasen, in denen weder Maus noch Tastatur berüht werden, sind – sofern währenddessen ein Rollenwechsel stattfand – als *Übergang* gekennzeichnet; andernfalls sind sie der Übersichtlichkeit halber nicht gesondert markiert worden. *P2* bezeichnet in beiden Diagrammen die selbe Person.

Noch eine Anmerkung zum oben verwendeten Begriff einer „gerechten" Rollenverteilung: Damit soll nicht impliziert werden, dass einer der beiden Fälle *wegen seiner Verteilung* besser sei! In dieser Arbeit wurden keine Untersuchungen angestellt, die die Driver/Observer-Verteilung ins Verhältnis mit einem Produktivitäts- oder Qualitätsmaß gesetzt haben.

In den Fragebögen nach den Sitzungen konnten die Programmierer Schulnoten für die Sitzung vergeben: Nach Sitzung D03 haben sowohl *P1* als auch *P2* die Note „2" vergeben, während nach Sitzung D06 *P2* mit einer Note „1" sehr zufrieden und *P1* mit einer „3" nur mäßig glücklich war. Eine vereinfache Formel der Art *„Je öfter eine Person der Driver sein durfte, umso angenehmer empfindet sie die Sitzung"* funktioniert also nicht.

Rolle des Kontexts und der intervenierenden Bedingungen

Wie in der Beschreibung von *guidance* erwähnt, ist einerseits eine gewisse Kompetenz nötig, um das weitere Vorgehen planen zu können und andererseits muss die Motivation vorhanden sein, dies auch zu tun. Wenn der Partner diese Ambition nicht wahrnimmt bzw. nicht zulässt, dann kommt es zu einem *guidance* durch den Sprecher; gelingt es aber, den Partner zu überzeugen, dann liegt ein erfolgreiches *action_taking* vor. Der Unterschied liegt hier oft im Nachdruck, mit dem der Sprecher seinen Wunsch äußert.

[41] Diesen Grundgedanken haben die Forscher Walle & Hannay [38] mit der Analyse von Audioaufzeichnungen von 44 professionellen Paaren verfolgt. Eines der Ergebnisse ist, dass die Persönlichkeitskombination der beiden Programmierer die Art der Zusammenarbeit beeinflusst. Die Autoren treffen leider keine Aussagen über Phänomene, die Ähnlichkeit mit *action_taking* haben.

Beispiele

1. In Sitzung D06 erklärt der Driver *P1* den Code, den er bereits programmiert hat. *P2* sagt währenddessen mehrfach leise *„Zeig' mal kurz"*, setzt sich aufrecht hin, rutscht mit dem Stuhl näher an den Schreibtisch und zieht die Tastatur langsam zu sich heran. *P1* erklärt weiter als *P2* mit dem Lesen des Codes beginnt und *P1*s Erklärungen schließlich unterbricht:

   P2: *„Moment, wo sind wir jetzt hier? (.) (#EditColumnAttribute#) (.) wart' mal kurz."*

   P2 lässt sich also nicht auf die Erklärungen von *P1* ein und fragt auch nicht nach, sondern „erschleicht" sich die Tastatur[42].

2. In Sitzung D03 hat der Driver *P2* nach einer Codeänderung die Anwendung erneut gestartet, was aber mit einer Fehlermeldung abgebrochen wird. Da er sich gerade seine Wasserflasche gegriffen hat, übernimmt der Observer *P1* die Tastatur und beginnt, nach dem Defekt zu suchen. Dann hat *P2* offenbar einen Einfall, bewegt die Hände in Richtung der Tastatur und sagt:

   P2: *„Ah (.) stopp' mal."*

   Der Partner *P1* fragt nach, ob er vorhat, eine bestimmte Änderung durchzuführen, worauf *P2* nur antwortet.

   P2: *„Nein (.) das ist, nein◡nein◡nein◡nein"*

   P2 behält weiterhin die Hände über der Tastatur, bis *P1* sie schließlich freigibt.

Weitere Bemerkungen zu dieser Klasse

Wie schon im vorigen Abschnitt angesprochen wurde, besteht vermutlich eine Verwandtschaft mit den Phänomenen der Klasse *proposal_rejetion*.

[42] *P2* benutzt die Phrase *„Wart' mal kurz"* immer wieder während seiner Sitzungen – hauptsächlich, um damit zu erwirken, dass der Partner mit Reden aufhört.

4.2 Klassenübergreifende Funktionen von Beeinflussungen

Eine zentrale Auslöserrolle spielt – abstrakt betrachtet – immer eine bestimmte Art Informationsbedürfnis. Entweder eines, das der Sprecher bei sich selbst wahrnimmt – oder eben eines, das er bei seinem Partner vermutet. Durch die gemeinsame Zielsetzung der beiden Entwickler, die u. a. durch die Sitzung gegeben ist (also z. B. „das Programm zum Laufen kriegen“, „ein neues Feature implementieren“), erweist es sich als Vorteil, wenn sich beide Personen auf einem ähnlichen Informationsstand befinden – zumindest was die aufgabenrelevanten Bereiche angeht. Um dieses Gleichgewicht herzustellen und zu halten, ist der permanente Ausgleich von Informationsdefiziten nötig, da die Programmierer (1.) verschiedene Ausgangspositionen haben und (2.) während der Sitzung verschiedene Erfahrungen sammeln und Einsichten erlangen.
Um diesen abstrakten Gedanken ein wenig greifbarer zu gestalten, werden hier die eben beschriebenen Beeinflussungstypen mit ihren korrespondieren Informationsbedürfnissen aufgelistet:

Klasse	**Art**	**Beschreibung des Informationsbedürfnisses**
defect_-announcement	⊕	Sprecher kennt einen Defekt, der Partner noch nicht.
work_proposal	⊕	Sprecher hat eine Idee/einen Vorschlag für das weitere Vorgehen, die der Partner noch nicht kennt.
guidance	⊕	Der Partner findet sich ohne Unterstützung nicht (gut genug) zurecht.
knowledge_-request	⊖	Der Partner weiß (vielleicht) mehr darüber.
knowledge_-conveyance	⊕	Dem Sprecher fällt auf, dass dem Partner noch Informationen fehlen.
finding_-conveyance	⊕	Sprecher hat eine neue Erkenntnis/Einsicht, die der Partner u. U. noch nicht hatte.
proposal_rejection	⊖/⊚	Teilweise, wenn der Partner zur Ruhe gebeten wird, um besser nachdenken bzw. Informationen aufnehmen zu können.
action_taking	⊖/⊚	Teilweise, wenn die Driver-Rolle eingenommen wird, um etwas selbst auszuprobieren oder nachzuschauen, was sonst zu umständlich zu erklären wäre.

Tabelle 4.7: Beeinflussungstypen und ihre korrespondierenden Informationsbedürfnisse. (**Legende:** ⊕ Überschuss, ⊖ Mangel, ⊚ indifferent)

Zusammenfassend ließe sich also festhalten, dass die untersuchten Phänomene viele verschiedene Funktionen erfüllen, eine gemeinsame Funktion aber die Befriedigung verschieden gearteter Informationsbedürfnisse ist.

5 Phasen- und Rollen-Konzepte

Im vorigen Kapitel wurden die beobachteten Beeinflussungen der Aktivität des Partners anhand des paradigmatischen Modells untersucht und acht verschiedenen Beeinflussungskonzepten zugewiesen.
Jede einzelne der so beschriebenen Beeinflussungen stellt aber nur einen kleinen Ausschnitt aus dem Gesamtprozess einer Paarprogrammierungssitzung dar. Der nächste Schritt in Richtung einer kompletten Beschreibung des Paarprogrammierungsprozesses wird in diesem Kapitel unternommen.

Die Phase des selektiven Kodierens, in der die Erkenntnisse des axialen Kodierens zu einer Theorie integriert werden, umfasst zunächst das Erzählen einer „Geschichte“ und die „Offenlegung des Roten Fadens“ dieser Geschichte [34, S. 94–95]. Die ersten Versuche, genau dies mit meinen bisherigen Erkenntnissen zu tun, sind daran gescheitert, dass sich partout keine Geschichte finden ließ.
Die Lösung dieses Problems bestand in der Betrachtung verschiedener *Kontexte*, in denen die Phänomene auftreten. Als geeignetes Unterscheidungskriterium haben sich bei der Untersuchung die *Phasen* der Sitzungen herausgestellt.
In Unterkapitel 5.1 wird das in dieser Arbeit verwendete Phasenkonzept erläutert. Die entwickelten Phasentypen selbst folgen in Unterkapitel 5.2. Schließlich werden dann noch abschließende Bemerkungen geäußert (5.3).

5.1 Phasen in Paarprogrammierungssitzungen

Zuerst stellte sich während der Untersuchung die Frage, welche Phasen während einer Paarprogrammierungssitzung überhaupt vorkommen. Daher werde ich in Abschnitt 5.1.1 zunächst die in der Literatur verwendeten Phasenbegriffe diskutieren, sowie im Anschluss das Verständnis der Entwickler selbst beleuchten (Abschnitt 5.1.2. Schließlich werde ich meinen eigenen Phasenbegriff entwickeln (Abschnitt 5.1.3).

5.1.1 Phasenbegriffe in der Literatur

Aus dem Bereich der Programmierung

In der (Paar-)Programmierungsliteratur werden zwar häufiger „Phasen“ erwähnt, der Begriff wird hierbei aber nicht genau definiert oder gar einheitlich verwendet wird. So gibt es z. B. die folgenden „Phasentypologien“:

- *Design*, *Coding*, *Testing*, *Acceptance*, *Maintenance*
  (Basili & Perricone [5], im Rahmen einer empirischen Untersuchung zu Fehlern bei der Softwareentwicklung)

- *Planning*, *Design*, *Design Review*, *Code*, *Code Review*, *Compile*, *Test*
  (Baheti et al. [3], im Rahmen eines Vergleichs von normaler und verteilter Paarprogrammierung)

- *Exploration*, *Planning*, *Iterations to First Release*, *Productionizing*, *Maintenance*, *Death*
  (Beck [6], als Beschreibung des idealen XP-Projekt-Lebenszyklus)

Diese Phasenunterteilungen beziehen sich jedoch auf das Softwareprojekt als Ganzes – beschreiben also (zum Teil) den Softwarelebenszyklus – oder implizieren eine sequen-

zielle Abarbeitung.

Eine alternative Aufteilung findet sich bei Williams & Kessler [44]. Hier wird in *New Development Time* und *Rework Time* unterschieden – je nachdem ob das betreffende Artefakt gerade erstellt wird, oder angepasst wird.
Schließlich wird durch Jekutsch [17] eine sehr feingranulare Aufteilung von „Aktivitäten“ vorgenommen: u. a. in *Changing Code*, *Browsing*, *Reading*, *Thinking*, *Pausing*, wobei jede dieser „Kernaktivitäten“ nochmals in speziellere Untertypen aufgeteilt wird.

Alle diese Phasenbegriffe sind entweder zu grob (weil sie sich auf den gesamten Softwareprozess beziehen) oder zu fein bzw. zu eng gefasst (weil sie für einen spezielleren Zweck – die detaillierte Analyse von Fehlern und Defekten – entwickelt wurden).

5.1.2 Phasenbegriff der beobachteten Entwickler

Die Unterteilung von PP-Sitzungen in sequentielle Phasen wird i. d. R. von den Entwicklern während des Sitzungsverlaufs nicht expliziert; in entsprechenden Fragebögen, die von den Entwicklern nach der Sitzung ausgefüllt wurden, wurden aber einige Phasenbezeichnungen genannt (wie etwa *„Orientierung für weitere Arbeit“*, *„Code-Sichtung“*, *„Design-Diskussion“ „Ausimplementierung“*, *„Testen“*).
Dabei ließ sich feststellen, dass die Entwickler untereinander inhomogenes Begriffsverständis hatten: die Abschnitte wurden häufig über ihre Zielstellung benannt (z. B. *„Ausrichten der Komponente“*), die aus Forschersicht tatsächlich aber aus *mehreren* Teilschritten bestanden (z. B. mehrere Try-and-Error-Zyklen, Diskussionen, etc.). Ganze Sitzungen wurden so von den Befragten in lediglich drei Phasen eingeteilt, was für das Verständnis von Aktivitätsbeeinflussungen viel zu grobkörnig ist. Ein Entwickler wählte ein gänzlich anderes Kriterium zur Bestimmung der Phasengrenzen[43]: er nannte in seiner Aufzählung u. a. die beiden Phasen *„<**Name des Partners**> Driver, [. . .] “* und *„Ich Driver, [. . .] “*.

Für die Festlegung, was eine Phase auszeichnet, wurden die selbstgewählten Bezeichnungen der Entwickler letztlich nicht verwendet. Es hat sich herausgestellt, dass es für die Bereitstellung eines Kontextes für die Aktivitätsbeeinflussung nicht entscheidend ist, wie die Entwickler ihre Sitzung selbst reflektieren, sondern was objektiv während bestimmter Abschnitte passiert[44].

5.1.3 Entwicklung der Phasentypen

Der Vorgehensweise der GTM folgend, wurden keine Phasentypen postuliert, sondern aus den Daten abgeleitet. Die beobachteten Sitzungen verliefen in Phasen wie z. B. *Code verstehen*, *Diskussion und Planung* und *Umsetzung*. In jeder dieser Phasen gibt es eine Reihe typischer Aktivitäten, der die Entwickler währenddessen nachgehen können. Abhängig von der aktuellen Phase gibt es auch nur begrenzte Möglichkeiten der Rollenverteilung – das Postulat eines „designierten Drivers“[45] [41] ist z. B. während einer

[43] Diese Unterschiede in der Betrachtungsweise illustrieren sehr schön die in Abschnitt 2.2.1 disktuierte Sichtweise, dass Benennung eines Phänomen es eigentlich erst erschafft.

[44] Man beachte an dieser Stelle den Unterschied zur BS/BKM: Die sog. *prozessorientierten Konzepte* adressieren explizit die *Äußerungen* des Paares [29, Abschn. 7.5], also Elemente aus der *Diskurswelt*. Der Phasenbegriff in dieser Arbeit stammt demnach aus der *Tätigkeitswelt* [29, Abschn. 6.5]

[45] Im Original: *„One [engineer] is the ‘designated driver’.“*

Code verstehen-Phase nicht adäquat, da u. U. beide Entwickler ohne Kontakt zu Maus und Tastatur den Quellcode studieren und wechselseitig neue Erkenntnisse oder Hypothesen äußern. Die Abfolge dieser Phasen ist nicht a priori festgelegt, sondern ergibt sich letztlich erst während der Sitzung[46].
Dem oben umrissenen Phasenbegriff folgend konnten acht Phasentypen herausgearbeitet werden. Jede tatsächlich auftretende Phase während einer Sitzung lässt sich durch einen der in Tabelle 5.1 aufgeführten Phasentypen konzeptualisieren.

#	Phasentyp	Beschreibung
1	*Organisatorisches*	Es werden Updates/Commits aus der/in die Versionsverwaltung getätigt und in der Ticketverwaltung gelesen und geschrieben. Ziel dieser Aktivitäten ist es, eine konsistente Arbeitsumgebung zu erhalten und den Prozess zu dokumentieren. (Abschnitt 5.2.1)
2	*Wissenstransfer*	Es wird erklärt, was bisher schon entwickelt wurde, wie bestimmte Technologien oder Abläufe funktionieren, oder welche Vorstellung und Pläne man für das weitere Vorgehen hat. (5.2.2)
3	*Testen*	Existierende Programmteile werden manuell oder automatisch (z. B. durch Unit-Tests) auf ihre Funktionstüchtigkeit überprüft oder schlichtweg ausprobiert. (5.2.3)
4	*Code verstehen*	Es wird versucht, existierenden Programmcode zu verstehen. Das kann durch einfaches Lesen oder Navigieren, oder im Falle einer Defektsuche auch durch Debugging geschehen. (5.2.4)
5	*Informationen einholen*	Es werden weiterführende Dokumente gelesen (Webseiten, Bücher, etc.) oder außenstehende Personen befragt. (5.2.5).
6	*Diskussion/ Planung*	Es wird das weitere Vorgehen besprochen, evtl. Alternativen diskutiert, Argumente vorgebracht, etc. (5.2.6)
7	*Umsetzung*	Die eigentliche Paar*programmierung*, bei der Programmcode neugeschrieben, angepasst oder gelöscht wird. (5.2.7)
8	*Kreative Tätigkeiten*	Das Benennen von Methoden, Variablen, Commit Messages etc. (5.2.8)

Tabelle 5.1: Übersicht über die Phasentypen

5.1.4 Eigenschaften der entwickelten Phasentypen

Ein interessanter Punkt, der später noch vertieft wird, ist, dass Einflussnahmen im Wechselspiel mit den Phasen stehen. Konkret können sie drei verschiedene Funktionen erfüllen: *eröffnend*, *evolvierend* oder *schließend*.

- ▷ Einflussnahmen sind *eröffnend*, wenn sie eine neue Phase beginnen (z. B. wenn einer der Entwickler durch ein *knowledge_request* eine Diskussion einleitet). Die alte Phase muss deswegen nicht zwangsläufig beendet werden, sondern kann einen umrahmenden Kontext bilden und nach Beendigung der Diskussion wieder aufgenommen werden.

[46] Natürlich gibt es wiederkehrende Muster, wenn die beiden Entwickler z. B. testgetrieben entwickeln (TDD, siehe [7]). Aber auch in diesen Fällen kann es erkenntnisgetriebene Abweichungen vom „Algorithmus“ des Prozessmodells geben.

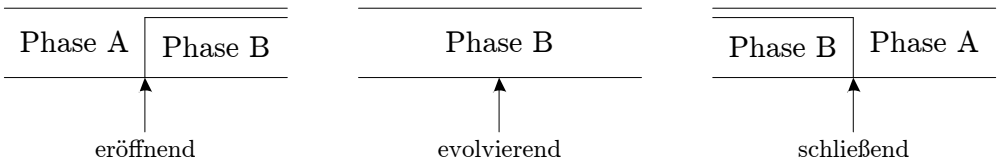

Abbildung 5.1: Funktionen von Einflussnahmen

- Einflussnahmen sind *evolvierend*, wenn sie den Prozess weiter voranbringen, ohne die aktuelle Phase zu verlassen, z. B. wenn einer der Entwickler seinen Partner durch eine Anwendung lotst (*guidance*).

- Einflussnahmen sind *schließend*, wenn die aktuelle Phase beendet wird und wieder zum höhergeordneten Kontext übergegangen wird, z. B. durch eine Äußerung wie *„Wir können ja erst mal weitergehen.“* (*work_proposal*)

Die so entstehenden Kontexte lassen sich wie öffnende und schließende Klammern verstehen, die sich ineinander schachteln lassen. Zum Erreichen des Projekt- oder Sitzungszieles müssen mehrere Zwischen- und Unterziele erreicht werden, deren Schachtelung nicht von Beginn an klar sein muss. Da z. B. während einer Diskussion festgestellt werden kann, dass die vorherige – eigentlich nur vorübergehend unterbrochene – Phase nicht zielführend war, oder es einfach vergessen wurde „wo wir gerade waren“, müssen einmal geöffnete Klammern auch nicht zwangsläufig „syntaktisch korrekt“ geschlossen werden.
Da die Festlegung, welcher zeitliche Abschnitt einer Phase entspricht, nicht von den Entwicklern selbst, sondern vom Forscher getroffen wird, spielt es auch keine Rolle, ob sich die Entwickler während (oder nach) der Sitzung über die Schachtelung der Teilaufgaben im Klaren waren.

5.1.5 Anmerkungen zur Entwicklung der Phasentypen

Ursprünglich war es geplant, die Phasentypen detaillierter auszuarbeiten und jeden Typ einzeln als Kontext für die Aktivitätsbeeinflussungen zu betrachten. Für jede Phase sollte so eine „beschreibende Geschichte“ entwickelt und eine selektive Kodierung begonnen werden, wobei die Rollen der darin vorkommenen Beeinflussungen untersucht werden sollten. Die kombinatorische Vielfalt[47] auf der einen und die zeitliche Begrenzung dieser Arbeit auf der anderen Seite haben es verhindert, dass dieser analytische Schritt noch gegangen werden konnte.
Die bereits entwickelten Eigenschaften der Phasentypen sollen deswegen aber nicht unter Verschluss gehalten werden. Die nachfolgenden Ausführungen sollten aber als frühes Entwicklungsstadium betrachtet werden.

5.2 Die Phasentypen

Wie schon die Vorstellung der Beeinflussungstypen, folgt auch die Darstellung der Phasentypen einem Schema, dass die Orientierung im Text erleichtern soll.

[47] 8 Beeinflussungstypen × 8 Phasentypen × 3 Funktionen von Beeinflussungen

1. Zunächst wird ein allgemeiner Überblick darüber gegeben, was innerhalb einer Phase dieses Typs passiert.
2. Anschließend werden die typischen Aktivitäten aufgelistet und bei Bedarf erläutert, die für diesen Phasentyp charakteristisch sind.
3. Dann werden (falls bereits untersucht) die für diesen Phasentyp identifizierten Rollenverteilungen beschrieben.
4. Im vierten Punkt werden (falls bereits untersucht) Aussagen über das Auftreten der definierten Beeinflussungstypen getroffen.
5. Außerdem werden zusätzliche Merkmale dieses Typs in einem optionalen letzten Punkt aufgeführt.

5.2.1 Phasentyp: Organisatorisch

Allgemeine Beschreibung

Die untersuchten Sitzungen begannen typischerweise damit, dass das Paar in der Ticketverwaltung einen oder mehrere Tasks öffnete, ihre aktuelle Aufgabe verortete und bei Bedarf nachtrug. Ebenfalls eine häufige initiale Tätigkeit ist das Aktualisieren der *Working Copy*[48] durch ein SVN-Update.
Aber auch im weiteren Verlauf einer Sitzung kommt es zu diesem Phasentyp, z. B. wenn die durchgeführten Codeänderungen in die Versionsverwaltung eingecheckt werden.

Typische Aktivitäten

Die typischen Aktivitäten während dieser Phase umfassen:

- das Öffnen der Entwicklungsumgebung
- das Öffnen/Ändern von Einstellungen in der Entwicklungsumgebung
- die Ausführung von SVN/CVS-Update des Quellcodes
- die Ausführung SVN/CVS-Commit der Änderungen im Quellcode

Typische Rollenverteilung

Hier findet sich die klassiche **Driver/Observer**-Verteilung. Der Driver führt die oben beschriebenen typischen Aktivitäten aus und benutzt dafür Maus und Tastatur. Der Observer verfolgt die Tätigkeit des Drivers und weist auf evtl. vergessene Punkte hin (*work_proposal*).

Beobachtete Einflussnahmen

Im Umfeld einer organisatorischen Phase konnten Beeinflussungen aller drei Funktionen beobachtet werden. Die drei folgenden Beispiele stammen alle aus Sitzung D06 und ereigneten sich in der abgedruckten Reihenfolge.

- **eröffnend:** Organisatorische Phasen traten v. a. zu Beginn der Sitzungen auf. Sie können dabei auch durch eine Einflussnahme eingeleitet werden, indem der Sprecher seinen Partner einen organisatorischen Schritt vorschlägt.

[48] Die *Working Copy* ist eine lokale Kopie eines Teils des SVN-Repositorys, auf der Änderungen vorgenommen werden können. http://svnbook.red-bean.com/nightly/en/svn.basic.version-control-basics.html#svn.basic.working-copy (Abruf: 28.11.2011)

P2: *„So, erst mal vielleicht 'n Update machen?“*

P2.work_proposal

Hier findet ein Bruch mit dem Schema der Rollenverteilung statt, da diese Einflussnahme kein typisches Observer-Verhalten ist: Der Sprecher hätte selbst seit einiger Zeit die Möglichkeit gehabt, Maus und Tastatur zu greifen (da sein Partner noch den Fragebogen ausfüllte). Stattdessen wartet er ab, bis sein Partner die Driverrolle übernimmt und noch bevor dieser selbst etwas tun kann, schlägt ihm *P2* das Update vor.

– **evolvierend:** Der Sprecher erinnert seinen Partner an ein weiteres Versionsverwaltungssystem.

P2: *„CVS auch?“*

P2.work_proposal

Diese Beeinflussung entspricht dem typischen Observer-Verhalten: er erinnert den Driver an eine (evtl.) wichtige Aktivität.

– **schließend:** Der Sprecher befindet die organisatorische Phase für beendet, weil alle Updates durchgelaufen sind und kein Handlungsbedarf mehr besteht. Der Sprecher äußert hier ein *explain_finding* und teilt dem Partner somit sinngemäß mit *„OK, wir können jetzt anfangen.“*

<* Entwickler warten auf Rückmeldung der IDE *>
P2: „OK“

P2.finding_conveyance

Dieses Verhalten ist weder eindeutig schemakonform noch schemabrechend.

5.2.2 Phasentyp: Wissens- oder Verständnistransfer

Allgemeine Beschreibung

Während einer Paarprogrammierungssitzung gibt es immer wieder Phasen, in denen Wissen über das Produkt, eine Technologie oder die Umgebung ausgetauscht wird, oder erst kürzlich entstandene Ideen erklärt werden.

Typische Aktivitäten

Üblicherweise können während dieser Phasen folgende Aktivitäten beobachtet werden:

- Äußerungen, die im Rahmen der Basisschicht mit *ask_knowledge* und *explain_-knowledge* kodiert würden.
- Das Aufsuchen von Codestellen (*search_sth*), um dort dem Partner etwas zu zeigen.

Typische Rollenverteilungen

Es konnten zwei verschiedene Rollenverteilungen beobachtet werden: Explainer/Listener sowie Teacher/Student.

Explainer/Listener Der *Explainer* ist die Person, die eine Idee oder eine (mitunter vage) Vorstellung hat und dies dem *Listener* näher bringen möchte. Der Explainer kann dies rein verbal, mit Stift und Papier oder an konkreten Codestellen tun, während der Listener zuhört und gelegentlich nickt oder anderweitig zustimmt, um seine Aufmerksamkeit zu signalisieren. Es ist auch nicht unüblich, dass der Listener an einigen Stellen nachhakt. Wenn der Listener selbst eine Vorstellung von der Thematik hat, dann ergibt sich ein Übergang zu einer Diskussionsphase, in der die beiden Entwickler gewissermaßen auf „Augenhöhe“ diskutieren können.

Teacher/Student Der *Teacher* ist die Person, die über das Wissen verfügt – allerdings hat sie im Ggs. zum Explainer ein tiefergehendes und gesetztes Verständnis von der Materie und zumindest ein rudimentäres „didaktisches“ Modell, mit dem er dem *Student* den Stoff zu erklären versucht. Auch hier kann es einen Übergang zu einer Diskussionsphase geben, wenn der Student Anstoß am „vermittelten Stoff“ findet.

Die Unterscheidung zwischen *Explainer* und *Teacher* basiert also letztlich auf der Differenzierung zwischen Bestandswissen, über das der Sprecher bereits vor der Sitzung verfügte, und neuen Vorstellungen, die erst im Laufe der Sitzung gereift sind. Die Trennung zwischen *Listener* und *Student* ist hingegen eine künstliche und nur dem besseren Klang der jeweiligen Verteilungsbezeichnungen geschuldet.

Als illustrierendes Beispiel sei der Beginn von Sitzung D06 genannt:

> Die Sitzung beginnt mit einer organisatorischen Phase und geht dann über in eine Wissenstransfer-Phase: *P1* erklärt seinem Partner, was er bisher gemacht hat. Die Erklärungen von *P1* scheinen *P2* aber nicht zu genügen, sodass er *P1* bittet, eine bestimmte Klassendatei zu öffnen, was dieser auch tut. Unmittelbar danach, als P1 mit seinen Erklärungen fortfahren möchte, stellt *P2* eine kritische, konkrete Frage, aus der sich eine Diskussion entspinnt.

P1 ist offensichtlich der Teacher, der zwar einige Schwierigkeiten hat, seine Arbeit in Worte zu fassen, aber dennoch theoretisch über das Wissen verfügt. *P2* ist der Student, der den Erklärungen des Teachers folgt, letztlich aber nicht vom Inhalt der Arbeit überzeugt ist, und eine losgelöste Diskussion beginnt.

Beobachtete Einflussnahmen

Eröffnet werden können diese Phasen z. B. durch *knowledge_request*s und *knowledge_-conveyance*s.
Wissenstransferphasen können auch abgebrochen werden, bevor sie richtig begonnen haben: *P1* wollte in Sitzung D06 als *Teacher* seinem Partner eine Klasse erklären, die dieser noch nicht verstanden hatte (siehe auch Typ „Code verstehen“, Abschnitt 5.2.4). *P2* blockt diesen Versuch allerdings mit einem *proposal_rejection* ab[49].

[49] Hier ist der Name des Konzepts *proposal_rejection* vielleicht nicht allzu glücklich gewählt. Allerdings könnte man auch den Vorschlag, etwas zu erklären, als *proposal* bezeichnen. Da der Sprecher aber die Äußerung seines Partner *unabhängig* vom Inhalt abwimmelt, ist das letztlich auch nicht entscheidend.

5.2.3 Phasentyp: Testen

Allgemeine Beschreibung

Das Ziel einer Testphase ist es, das tatsächliche Verhalten eines Programm(teil)s zu interpretieren, (falls vorhanden) mit einer persönlichen Erwartung zu vergleichen und so eine bestimmte Fragestellung zu beantworten. Ob es sich dabei um das schlichte Ausprobieren einer Anwendung (um sich z. B. einen Überblick zu verschaffen), um die Überprüfung einer Hypothese, oder um einen abschließenden Akzeptanztest zur Validierung einer Implementierung handelt, ist für die Untersuchung von Einflussnahmen irrelevant, da die tatsächlich ausgeführten Aktivitäten die gleichen sind.

Typische Aktivitäten

Je nach Komplexität des ausgeführten Testfalles, kann sich die direkte Interaktion mit dem Programm auf einzelne Klicks bzw. Tastaturanschläge beschränken, oder aber aus einem mehrstufigen Prozess bestehen.
Weiterhin muss die Testphase auch abgeschlossen werden, indem das Programmverhalten interpretiert und die ursprüngliche Fragestellung beantwortet wird (z. B. durch *propose_hypothesis*, *explain_finding*). Alternativ kann das Paar zu dem Schluss kommen, dass die Frage nicht beantwortet werden kann und geht dann zu einer anderen Phase über.

Typische Rollenverteilung

Hier wurde wiederum die klassische **Driver/Observer**-Verteilung beobachtet, wobei es je nach spezifischer Ausprägung der Phase Unterschiede im Engagement des Observers gibt: Bei einem abschließenden Akzeptanztest wird der Observer tendenziell aufmerksamer sein – v. a. wenn der Test beim vorigen Mal fehlgeschlagen ist – wenn der Driver nur spielerisch die GUI ausprobieren möchte, dann wird sich der Observer eher zurücklehnen – sowohl physisch als auch mental.
Wenn der Test an sich bereits ausgeführt ist, und lediglich die Ergebnisse (z. B. die Ausgabe) interpretiert werden müssen, dann löst sich diese Verteilung allerdings auf. Denn dann versuchen beide Entwickler gleichermaßen die Interpretation durchzuführen und es kommt häufiger zu einem *finding_conveyance*, mit dem der jeweilige Partner über die eigene Bewertung informiert wird.

Beobachtete Einflussnahmen

Es konnten Einflussnahmen in allen drei funktionalen Rollen beobachtet werden.

- **eröffnend:** Der Vorschlag, die Anwendung einfach mal auszuprobieren/zu starten, um zu „gucken was passiert“ (*work_proposal*, *P2* als Observer in Sitzung D07).
- **evolvierend:** Erweitern des manuellen Testfalls um weitere Schritte, weil sich die Fragestellung geändert hat.
- **schließend:** Interpretieren der Ausgabe eines automatischen Unit-Tests oder sonstige Ereignisse. In diesem Fall gibt es auch keine deutliche Rollenverteilung; es spricht der, der zuerst zu der entsprechenden Erkenntnis gelangt ist.

5.2.4 Phasentyp: Code verstehen

Allgemeine Beschreibung

Während der Arbeit an einem Softwareprojekt stoßen Entwickler irgendwann auf Stellen im Programmcode, die ihnen nicht geläufig sind und erst einmal verstanden werden müssen. Geschieht dies während einer Paarprogrammierungssitzung, gibt es zwei unterschiedliche Fälle: entweder einer der beiden ist im Vorteil, oder aber beiden Personen ist der Code unbekannt.
Wenn einer der beidem im Vorteil ist, dann kommt es zu einer Wissenstransfer-Phase (siehe Abschnitt 5.2.2) in der einer dem anderen sein Wissen vermittelt (wenn dieser ihn lässt). Wenn beide Entwickler mit dem Code nicht vertraut sind, dann hängt der weitere Verlauf davon ab, wie groß der Bereich ist, in dem navigiert werden muss, um das gewünschte Verständnis zu erlangen:

- Beschränkt sich die Unwissenheit auf eine einzelne Datei, dann wird diese für gewöhnlich zeilenweise von beiden Personen gelesen, wobei es abwechselnde *think aloud_activity*s (zu fremden und eigenen HCI-Aktivitäten), mit *read_sth*s, *explain_finding*s und *explain_standard of knowledge*s gibt. Man könnte auch von einem „Selbstgespräch zu zweit“ reden[50].

  Beim Verständnis des Programmcodes müssen die beiden Entwickler nicht zwingend gleich schnell arbeiten; es ergeben sich von alleine Unterschiede, je nach persönlicher Auffassungsgabe und Erfahrung. So kann es Einen geben, der die Suche vorantreibt, was er entweder an der Tastatur und Maus oder über Anweisungen machen kann. Fälle, in denen sich einer der Entwickler deutlich zurücknimmt oder gar physisch zurücklehnt, wurden nicht beobachtet.

- Ist nicht ganz klar, welche oder wie viele Dateien verstanden werden müssen (wie es häufig beim Debugging der Fall ist), gibt es eine große Informationsmenge, die zwangsläufig gefiltert werden muss, um sie zu verstehen. Dadurch ergeben sich häufig mehrere mögliche Vorgehensweisen, und damit potenziell auch verschiedene Ansichten bei den Entwicklern.

Typische Aktivitäten

(siehen oben unter „Allgemeine Beschreibung“)

Weitere Bemerkungen

Das Lesen von Programmcode auf dem Bildschirm durch beide Personen stellt sich ähnlich dar, wie das getrennte Lesen der Aufgabenstellung von einem Blatt Papier (in Sitzung St10B). Ein kleiner Unterschied besteht darin, dass beim Lesen auf dem Bildschirm meistens der Driver die Maus oder auch den Textcursor mitführt und dem Partner (gewollt oder ungewollt) so anzeigt, an welcher Stelle er gerade liest. Beim Lesen auf getrennten Papieren gibt es diesen Teil der „Awareness“ nicht. Dennoch konnte in beiden Szenarien beobachtet werden, wie die Entwickler bei entstandenen Unklarheiten sofort den Partner befragen.

[50] Im Anhang in Abschnitt B.3 sind zu diesem Begriff einige Gedanken zusammengefasst.

5.2.5 Phasentyp: Informationen einholen

Allgemeine Beschreibung

In mehreren Sitzungen gab es Gespräche mit weiteren Personen außerhalb des Paares, oftmals um Probleme zu klären, die das Paar selbst nicht bewältigen konnte. Da auch außerhalb der hier untersuchten Sitzungen Paarprogrammier selten völlig isoliert arbeiten, ergibt sich zumindest die *Möglichkeit* eines Informationsaustausches über die Paargrenze hinweg – auch während der laufenden Sitzung.

Eine andere Form der Informationsbeschaffung ist die Konsultation von Dokumentation. In einer Sitzung, die nur im Schnelldurchlauf angesehen, aber nicht analysiert wurde, haben die Entwickler einen wesentlichen Teil der Sitzungszeit mit dem Durchforsten und Lesen von Onlinedokumentationen verbracht, während sich die „Programmiertätigkeiten“ auf die wiederholte Anpassung einer einzelnen Konfigurationsdatei beschränkten.

5.2.6 Phasentyp: Diskutierend/Planend

Allgemeine Beschreibung

In einer Diskussions- oder Planungsphase werden Ideen hervorgebracht, erweitert, abgelehnt oder Gegenvorschläge gemacht. Diesen Phasen liegt ein kreativer Prozess zu Grunde, bei denen einer der beiden aktiver oder dominanter sein kann, aber nicht muss.

Typische Aktivitäten

In einer Diskussionsphase können theoretisch alle HHI-Konzepte verwendet werden (ausgenommen evtl. der *activity*-Konzepte). Das entscheidende Kriterium für diesen Phasentyp ist, dass währenddessen *kein* neuer Programmcode produziert wird.

5.2.7 Phasentyp: Umsetzung

Allgemeine Beschreibung

In einer Umsetzungsphase findet die eigentliche Paar*programmierung* statt. Hier werden die vorher gefassten Pläne umgesetzt – was nicht heißen muss, dass beide Entwickler diesen Plan kennen oder zumindest die gleiche Vorstellung davon haben.

Typische Rollenverteilungen

Hier findet sich die **Driver/Observer**-Konstellation. Der Driver setzt den Plan nach seinen Vorstellungen um, während der Observer die Aktivitäten des Drivers überwachen und korrigierend eingreifen *kann*.

5.2.8 Phasentyp: Kreative Tätigkeiten

Allgemeine Beschreibung

Während der Umsetzung eines Planes gibt es viele Punkte, bei denen Wahlfreiheit herrscht, z. B. wenn Methoden, Klassen oder Variablen benannt werden müssen oder wenn während einer organisatorischen Phase eine Commit-Message geschrieben werden muss. In diesen Phasen muss der Driver zu einem gewissen Teil kreativ tätig werden. Während solcher kreativen Phasen, die sich also in die anderen Phasen einbetten, spielen in vielen Fällen Konventionen eine Rolle, oder in spezielleren Fällen auch die Namen anderer Elemente (wie Klassen und Variablen).

An dieser Stelle gibt es einen Übergang zwischen den BS-Konzepttypen *design* und *knowledge*. Für die Verwendung eines *design*-Konzepts sind Wahlfreiheiten nötig[51]. Tatsächlich spielen aber sowohl Inhalte des Langzeitgedächtnisses (Namenskonventionen, Teile der Unternehmenskultur, etc.), als auch des Kurzzeitgedächtnisses (gerade eben gelesene Variablen- und Klassenamen), sowie Aufmerksamkeit und Konzentration eine wichtige Rolle.
Diese beiden Gedächtnistypen trennen sich wie folgt auf: die Konventionen sind im Allgemeinen bekannt bzw. antrainiert und werden in den kreativen Phasen vom Driver nur selten missachtet – sie befinden sich im LZG. Die o. g. Inhalte des KZG werden häufig vom Observer nachgereicht (*knowledge_conveyance* oder *finding_conveyance*), wenn dieser merkt, dass der Driver diese Informationen benötigt, oder sich vertippt hat[52].

Typische Rollenverteilung

(siehe oben unter „Allgemeine Beschreibung“)

Beobachtete Einflussnahmen

Zusammengefasst ist es während dieser Phasen tatsächlich so, dass der Observer die Randbedingungen überwacht und bei Bedarf *evolvierend* Einfluss nimmt.

[51] Statt *design* wurde in der BKM ursprünglich der Begriff *choice about degree of freedom* verwendet, um die Betonung auf *Gestaltungsmöglichkeiten* zu legen [29, Abschn. 7.4.1].
[52] Das ist im Übrigen ein Indiz für die *freien Ressourcen* des Observers, siehe Exkurs 3.

5.3 Abschließende Bemerkungen zu den Phasen

Im letzten Teil dieses Kapitels werden noch einige Feststellungen und Hypothesen zusammengefasst.

5.3.1 Äußerungen auf dem Backchannel und kurze Statusbeurteilungen

Auch wenn die Analyse der Phasentypen und der währenddessen auftretenden Beeinflussungstypen nicht so weit getrieben werden konnte, wie zwischenzeitlich erhofft, konnten zwei bestimmte Arten von Äußerungen während aller Phasen beobachtet werden. Es handelt sich hierbei um Äußerung auf dem *Backchannel* (siehe Exkurs 4) und um *kurze Statusbeurteilungen*.

Exkurs 4: Backchannel

Der Begriff „Back channel“ wurde vom Sprachwissenschaftler Victor Yngve geprägt und bezeichnet einen Kommunikationskanal, über den der Sprecher kurze Nachrichten vom Angesprochenen erhält [48]. Sprachlich stellen sich solche Äußerungen z. B. als „Hm“ oder „OK“ dar und sie zeigen dem Angesprochenen an, dass man ihm weiterhin zuhört und ihm inhaltlich folgt.

Derartige Äußerungen treten auch bei Paarprogrammierungssitzungen häufig auf. Sie zeigen dem Partner die eigene Aufmerksamkeit an und können so auch zum Weitermachen auffordern.

Die kurzen Beurteilungen zum Status oder dem Arbeitsfortschritt sind phonetisch von Äußerungen auf dem Backchannel nicht zu unterscheiden. Einige Personen äußern während einer Sitzung öfter ein kurzes „OK“, um dem Partner (oder auch nur sich selbst) anzuzeigen, dass sie mit dem Verlauf zufrieden sind (bzw. gerade zu eben dieser Einsicht gelangt sind) und so ebenfalls – wie ein Backchannel „OK“ – zum Weitermachen auffordern.

Beide Arten haben nur einen niedrigen *Miss of Expectations*-Wert, sodass sie nicht wirklich als Aktivitätsbeeinflussung gelten können. Dennoch scheinen sie der Trigger dafür zu sein, dass der Partner fortfährt und haben so zumindest einen kleinen Einfluss auf die Aktivität.

Es ist dabei allerdings nicht so, dass die Personen mit einem größeren Mitteilungsbedürfnis generell häufiger diese Äußerungen tätigen. Vielmehr scheint es so, als ob gerade für ruhigere Entwickler diese kurzen „OK“s eine einfache Möglichkeit darstellen, ihren Anteil an der Sitzung zu leisten, wenn sie sonst von einem dominanten Partner „überrannt“ werden.

5.3.2 PP-Phasen als Prozess

Strauss & Corbin beschreiben den Prozess als „eine Sequenz von Handlungen und Bewältigungsstrategien“. Sie postulieren ein über die Zeit relativ stabiles Ziel, das von den handelnden Personen erreicht werden will. Da sich im zeitlichen Verlauf die Bedingungen (der Kontext) verändern kann, müssen auch notwendigerweise die Handlungen zur Zielerreichung angepasst werden. Da die Handlungen der Personen nicht ohne Konsequenzen bleiben, beeinflussen die Personen selbst den Kontext, der sich so stetig verändert. [34, S. 118–126]

In der Paarprogrammierung wäre das relativ stabile Ziel schlicht die Zielsetzung der Sitzung, das u. a. durch die *organisatorischen* Phasen klar wird. Um die aktuelle Situation zu verstehen, können Phasen des *Testens/Ausprobierens*, *Code verstehens* und des *Wissenstransfers* nötig werden. Sind die Probleme verstanden, kann dann das Vorgehen *geplant* und *diskutiert* werden, im Bedarfsfall weitere *Informationen* eingeholt werden und schließlich die gefassten Pläne *umgesetzt* werden. Eingebettet in vieler dieser Abschnitte, können immer wieder *kreative Phasen* entstehen.
Jede dieser Phasen entsteht aus dem vorherigen Kontext heraus und innerhalb der Phasen wird der Kontext wiederum verändert. Die Aktivitätsbeeinflussungen spielen dabei mit ihren *eröffnenden*, *evolvierenden* und *schließenden* Eigenschaften eine wichtige Rolle.

Teil III
Hinweise zur Herleitung der Ergebnisse

Entsprechend dem GTM-Prozess werden in den nächsten beiden Kapiteln einige erwähnenswerte Ergebnisse der Überlegungen aus der Phase des offenen Kodierens (Abschnitt 6) und des axialen Kodierens (7) zusammengefasst.
Diese Zwischenergebnisse dienen v. a. zur Verdeutlichung, dass ein GTM-Forschungsprozess kein geradliniger Prozess, sondern eher einer mit mehreren Abfahrten und Umwegen ist. Die Tatsache, dass die nachfolgend genannten Resultate in der restlichen Arbeit zumindest nicht direkt sichtbar werden, zeugt genau davon.

6 Auszüge aus den Ergebnissen des offenen Kodierens

6.1 Erweiterung der *activity*-Konzepte

Betrachtet man die Äußerungen, die mit den *activity*-Konzepten der Basisschicht annotiert werden unter dem Blickwinkel der in Kapitel 3 vorgestellten Dimension *Negativity*, so erhält man folgende Übersicht:

Konzept	**Beschreibung**	***Negativity***
disagree_activity	Die HCI/HEI-Aktivität wird abgelehnt.	hoch
stop_activity	Ein Unterbrechen/Beenden wird der HCI/HEI-Aktivität vorgeschlagen.	*indifferent*
challenge_activity	Die HCI/HEI-Aktivität wird abgelehnt; ein Unterbrechen/Beenden wird vorgeschlagen; Ersatz wird vorgeschlagen.	*indifferent*
agree_activity	Der HCI/HEI-Aktivität wird zugestimmt.	gering
amend_activity	Der HCI/HEI-Aktivität wird zugestimmt; eine Erweiterung vorgeschlagen.	gering

Tabelle 6.1: BS-*activity*-Konzepte nach ihrer *Negativity*

Die Konzepte *stop_activity* und *challenge_activity* lassen sich auf der *Negativity*-Skala nicht eindeutig zuordnen, da es keine Aussage über die Beweggründe gibt: Der Sprecher kann der Meinung sein, dass sein Partner aufhören soll, weil die Aktivität generell nichts bringt oder weil er zurerst was anderes machen sollte.
Aus diesen Gründen wurden die *activity*-Konzepte erweitert und um ein neues Konzept ergänzt. Sie werden in Tabelle 6.2 aufgelistet.

Konzept ***Negativity***	**Beschreibung**
interrupt_activity sehr gering	Die Aktivitiät des Partners wird lediglich unterbrochen. (neu entwickelt)
obsolete_activity hoch	Der Sprecher drückt aus, dass die aktuelle Aktivität nicht (mehr) nötig, *obsolet* ist. (von *stop_activity*)
append_activity sehr gering	Der Sprecher stimmt der Aktivität zu und ergänzt sie um einen weiteren Vorschlag. (von *amend_activity*)
adjust_activity sehr gering	Der Sprecher stimmt der Aktivität zu und nimmt eine Konkretisierung vor, damit die Aktivität zielstrebiger ausgeführt werden kann. (von *amend_activity*)
prepend_activity mittel	Der Sprecher drückt aus, dass die aktuelle Aktivität nicht (vollständig) funktionieren kann, bzw. in dieser Form keinen Sinn macht. Er soll sie demnach unterbrechen, eine andere vorgeschlagene Aktivität einfügen und dann mit der ursprünglichen Aktivität fortfahren. (von *challenge_activity*)
replace_activity hoch	Der Sprecher drückt aus, dass die aktuelle Aktivität des Partners beendet werden soll und durch eine andere Aktivität ersetzt werden. (von *challenge_activity*)

Tabelle 6.2: Neue *activity*-Konzepte nach ihrer *Negativity*

6.2 Erste Klassifizierung von Beeinflussungen

Die erste Klassifizierung, die im Laufe der Untersuchung herausgearbeitet wurde, unterteilte die Beeinflussungen in drei Typen: Hilfestellung leisten (*Support*), auf Mängel hinweisen (*Error*) und Informationsaustausch *Information*.

Die Klasse ***Support*** war am detailliertesten ausgearbeitet und die betreffenden Phänomene wurden u. a. entlang der in Tabelle 6.3 aufgelisteten Dimensionen betrachtet.

Dimension	**Ausdehnung**
Basis	*knowledge* ⟷ *finding*
Bezugspunkt	Produkt ⟷ Entwicklungsumgebung ⟷ Prozess
Modus	initiativ ⟷ reaktiv

Tabelle 6.3: Dimensionen entlang derer die *Support*-Phänomene untersucht wurden. Die *Basis* der Phänomene können Erkenntnisse im Sinne von *findings*, oder aber Bestandswissen im Sinne von *knowledge* sein. Der *Bezugspunkt* einer solchen Äußerung kann das Produkt (in Form von Design-Vorschlägen), die Umgebung (bezogen auf die IDE oder andere Programme) sowie der Arbeitsprozess im größeren Maßstab sein. Schließlich unterschieden sich *Support*-Phänomene in ihrem *Modus*: entweder äußert sie der Sprecher von sich aus (= initiativ), oder geht auf eine (explizit oder implizit) gestellte Frage ein (= reaktiv).

Entlang der Dimension „Bezugspunkt“ wurden dann die drei Typen *produktorientierte*, *umgebungsorientierte* und *prozessorientierte Hilfestellung* unterschieden. Diese Typen sind wie in Tabelle 6.4 dargestellt in die aktuellen Beeinflussungstypen übergegangen.

früherer Typ	**jetzige Typen**
produktorientert	*work_proposal*
umgebungsorientiert	*guidance*, *finding_conveyance*
prozessorientiert	*knowledge_conveyance*, *finding_conveyance*

Tabelle 6.4: Übergang der früheren *Support*-Typen in die jetzigen Beeinflussungstypen

In der Klasse ***Error*** wurden sowohl Äußerungen zu Mängeln im Produkt (also Defekte) als auch im Prozess zusammengefasst. Der Umgang mit Defekten wird nun vom Konzept *defect_announcement* abgedeckt, während entdeckte Makel am Prozess heute v. a. durch das Konzept *work_proposal* erfasst werden.

Schließlich waren die heutigen *knowledge_request* und *knowledge_conveyance* in der Klasse ***Information*** vereint.

Als typische Observer-Tätigkeiten würden dabei wohl am ehesten die ersten beiden Klassen angesehen werden, während die letzte Klasse eher auf unkonventionelles Verhalten schließen ließe.

7 Überlegungen aus der Phase des axialen Kodierens

7.1 Einordnung von Interventionen

Zu Beginn meiner Untersuchung ging ich davon aus, dass die – häufig vom Observer – getätigten Beeinflussungen die Rolle des zentralen *Phänomens* im paradigmatischen Modell einnehmen.

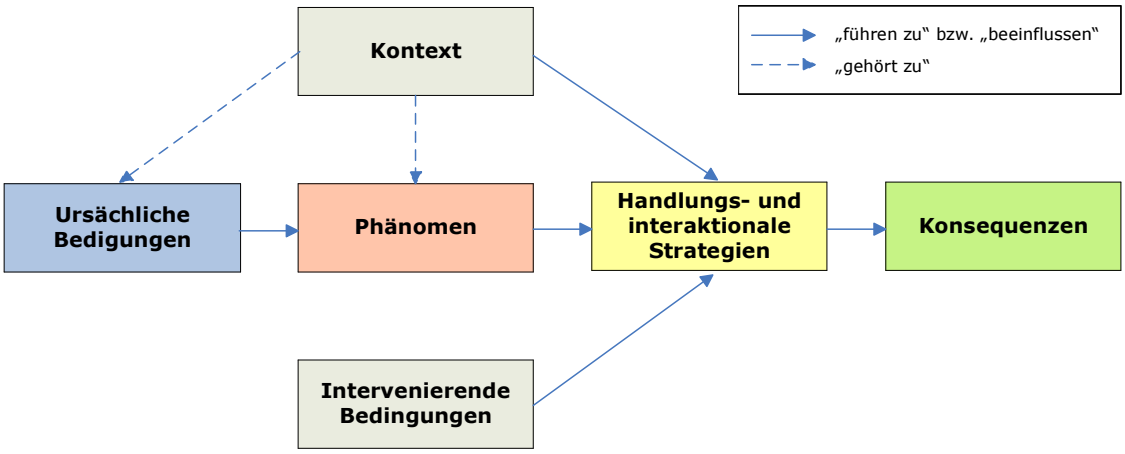

Abbildung 7.0: Darstellung des paradigmatischen Modells nach Strauss & Corbin [34]. (Wiederholung von Abbildung 2.2 von Seite 11.)

Strauss & Corbin beschreiben das Phänomen als „die zentrale Idee, das Ereignis, Geschehnis, auf das eine Reihe von Handlungen/Interaktionen gerichtet sind, um es zu bewältigen oder damit umzugehen." [34, S. 79]. Diese Definition brachte mich – nach einiger Zeit – zu der Erkenntnis, dass die Eingriffe des Observers gar nicht die eigentlichen *Phänomene* sind; vielmehr sind sie eine Möglichkeit für den Observer mit einem Phänomen umzugehen, stellen also eine *Handlungsstrategie* dar.

Probleme bei der Betrachtung von Interventionen als Handlungsstrategie

Aus dem oben bezogenen Blickwinkel ergibt sich automatisch die Frage, was denn stattdessen die Phänomene seien, auf die der Observer (unter gegebenen Umständen) mit einer Intervention oder einem Interrupt reagiert. Schon die damaligen Beobachtungen ließen darauf schließen, dass der Observer beim Driver ein Informationsdefizit oder -bedürfnis vermutet (siehe Abschnitt 4.2).

Aus dieser Schlussfolgerung ergab sich sofort das nächste Problem: ist das Phänomen das (womöglich) real existierende Informationsbedürfnis beim Driver, oder ist es nur die Wahrnehmung dieses Informationsbedürfnis beim Observer, also ein Abbild davon? Betrachtet man das *reale Informationsbedürfnis* als Phänomen, das vom Observer behandelt wird, dann ließe sich die Frage nach den ursächlichen Bedingungen anhand der Sitzungsaufzeichnungen nicht immer beantworten. Hypothetische Gründe könnten hier sein, dass der Driver neu in dieser Arbeitsumgebung ist, bisher an anderen Modulen gearbeitet hat, einen „schlechten Tag" hat, etc.
Die Ursachen für die *Wahrnehmung eines Informationsbedürfnisses* hingegen lassen sich viel einfacher in den Daten finden: sie zeigen sich z. B. in Form von vorsichtigen Vor-

schlägen[53], *explain_standard of knowledge*s mit einem niedrigen Level, vom Observer als falsch erkannten *explain_finding*s bzw. *explain_knowledge*s oder auch nur zögerlichen Mausbewegungen.
Aus diesen Gründen wurde zunächst die Wahrnehmung, also das mentale Abbild der partnerlichen Wissenslücke als Phänomen betrachtet.

Diese Modellierung hat einen markanten Haken. Die tatsächliche Existenz dieser Gedanken beim Observer lässt sich nicht direkt in den Daten belegen, da bisher keine Äußerung eines Observers beobachtet wurde, die eine Vermutung über die Wissenslücke beim Driver beinhaltet.
An dieser Stelle verlasse ich mich auf den „gesunden Menschenverstand“, indem ich davon ausgehe, dass der Observer über ein mentales Abbild der laufenden Sitzung und seinen Partner verfügt, in das er neue Informationen aus den Handlungen seines Partners integriert und auf Grundlage dessen er seine eigenen Aktivitäten koordiniert.
Es soll noch einmal betont werden, dass hier **nicht direkt Beobachtbares** als Phänomen (im Sinne des paradigmatischen Modells: „das, worum sich die Handlung dreht“ [34, S. 79]) betrachtet wird. Dies geschah aber – wie erläutert – aus gutem Grund.

7.2 Vorgehensweise beim axialen Kodieren

Nachdem die o. g. Gedankengänge einmal mühsam vollzogen wurden, konnte das folgende Vorgehensschema für jede untersuchte Episode angewendet werden:

1. Beschreiben, wie eigentlich gehandelt wurde. Worauf bezieht sich die Person, ist die Aktivitätsbeeinflussung die Absicht, oder nur die Konsequenz?

2. Zugrundeliegendes Phänomen „aufspüren“. Dafür Hypothesen aufstellen, überprüfen und verfeinern.

3. Ursachen für das Phänomen finden. Wiederum Hypothesen aufstellen, überprüfen und verfeinern.

Als zusätzlicher Punkt wurden dann noch Überlegungen über das „echte“ Phänomen angestellt, also versucht, Fragen wie *„Woher kommt der Defekt eigentlich?“* oder *„Warum weiß die Person das nicht?“* zu beantworten.

[53] Typ: Orientierung suchen, siehe [29, Abschn. 7.4.1]

Teil IV
Schluss

In Kapitel 8 werden Resultate dieser Arbeit bewertet und eingeordnet. Kapitel 9 enthält Vergleiche bisherigen Erkenntnissen aus der Literatur. Ein Ausblick in Kapitel 10 schließt den inhaltlichen Teil dieser Arbeit ab.

8 Bewertung und Einordnung

8.1 Was wurde in dieser Arbeit geleistet?

Die vorliegende Arbeit enthält eine ausführliche Diskussion von Beeinflussungen der Aktivität des Partners bei der Paarprogrammierung. Die Analyse wurde nach den Vorgaben der Grounded Theory Methodology durch die Phasen des offenen und des axialen Kodierens betrieben, wodurch sich verschiedene Formen von Informationsbedürfnissen als gemeinsames Element aller Beeinflussungen ergeben haben. Für die Vorbereitung auf das selektive Kodieren wurde die Entwicklung eines Phasenbegriffs und zugehöriger Phasentypen als notwendig herausgestellt und die entsprechenden Untersuchungen begonnen. Dabei wurden bereits erste Verbindungen zwischen den Beeinflussungen und den Phasen (in Form der drei funktionalen Rollen von Beeinflussungen) herausgearbeitet. Darüber hinaus wurden Indizien für die Existenz anderer Rollenverteilungen als das klassische Driver-Observer-Muster gesammelt.
Insgesamt wurde der Fokus der Arbeit dabei immer größer. Den Ausgangspunkt bildete die Betrachtung von *disagree_activity*-Phänomenen. Vergrößert wurde dies dann auf ein erweitertes Aktivitätenverständnis, als das in der Basisschicht verwendete, um so einen größeren Teil der Interaktionen bei der Paarprogrammierung zu erfassen. Schließlich erfolgte dann, wie oben beschrieben, die Weitung bis auf Phasenebene.

Neben den inhaltlichen Resultaten wurden auch methodische Erkenntnisse im Umgang mit dem paradigmatischen Modell gesammelt, für Anwendungsfälle, in denen keine physisch greifbaren Phänomene, sondern im weitesten Sinne „Bewusstseinszustände“ behandelt werden.

8.2 Verhältnis zur BS/BKM

Die Ergebnisse dieser Arbeit stellen einen Teil einer neuen Schicht oberhalb der Basisschicht dar. Dabei wurden bestehende Konzepte aus der BKM erweitert, sodass sie inhaltlich mehr fassen, andere wurden aufgeteilt, um ein genaueres „Hinsehen“ zu ermöglichen und schließlich wurde auch ein anderer Blickwinkel eingenommen, und völlig neue Konzepte eingeführt.

9 Vergleiche mit bisherigen Erkenntnissen aus der Literatur

Da es sich bei dieser Arbeit um eine qualitative Untersuchung handelt, ist es schwer bis fragwürdig, Vergleiche mit quantitativen Studien herzustellen. Aus diesem Grund werden hier Aussagen und Hinweise zum Vergleich gezogen, die ebenfalls qualitativer bzw. anekdotischer Natur sind.

Weiterführende Ideen, die u. a. Brücken zu anderen Forschungsgebieten schlagen, finden sich im nächsten Kapitel.

9.1 Verschiedene Perspektiven von Driver und Observer

Es ist eine weitverbreitete Ansicht, dass Driver und Observer verschiedene Sichten auf ihre Arbeit einnehmen. Der Observer arbeite sowohl auf einem niedrigen als auch auf einem hohem Niveau; der Driver dazwischen auf einem mittleren Niveau.
Typische Aufgaben auf einem niedrigen Niveau umfassen das Überprüfen der Programmsyntax und das Erinnern an Variablenbezeichner [16, Kap. 12]; Aufgaben auf einem höheren Niveau sind z. B. das Überwachen der Lesbarkeit des Codes, die Testabdeckung, das Einhalten von Codierungskonventionen, aber auch das befolgen einer konsistenten Strategie und das Vermeiden von Sackgassen [16, Kap. 12].

Tatsächlich beobachtet werden konnten v. a. Observer-Äußerungen, die auf ein *niedriges* Niveau schließen lassen: gefundene Defekte und ein besseres Gedächtnis für gerade gesehene Bezeichner sind oft die Gründe für Beeinflussungen.
Dass der Observer auch einen Blick für die größeren Zusammenhänge der Sitzung hat, dafür konnten keine schlagkräftigen Indizen entdeckt werden.

9.2 Abhängen des Partners und Rollenwechsel

Jeffries et al. [16] äußern sich zu den zusätzlichen Aufgaben des Drivers wie folgt:

> *The driver [. . .] is responsible for making sure that the partner can be completely engaged. The driver explains what she is doing, so the partner can understand it.*

In den untersuchten Sitzungen wurde der Driver, falls es nötig wurde, vom Observer darauf hingewiesen, dass er nicht mehr folgen kann (spezielles *knowledge_request*). Allerdings wäre das gar nicht nötig, wenn der Driver seine Pläne einfach immer kommentieren würde. Insofern kann diese Sichtweise unterstützt werden.

Weiterhin heißt es:

> *She also listens to what the partner is saying, and to what he is not saying. If she stops hearing encouraging noises from her partner, she needs to stop and see what's wrong.*

Diese Aussage korrespondiert mit der Beobachtung über die *kurzen Statusbeurteilungen*, die den Partner zum Fortfahren bewegen (siehe Abschnitt 5.3.1). Im Umkehrschluss sollte das Fehlen dieser „encouringing noises“ den Partner tatsächlich stutzig machen.

Dann geben die Autoren u. a. diesen Tipp:

> *The partner can say „Let me drive“ when he can't express an idea verbally [. . .]*

Dieses Verhalten wurde als *action_taking* konzeptualisiert, wobei die entsprechenden Vorkomnisse eher darauf hindeuteten, dass der Sprecher keine Lust hatte, seine Ideen zu verbalisieren, da das, was er dann tatsächlich gemacht hat, relativ leicht zu erklären gewesen wäre.

9.3 Eigene Hinweise

Ergänzend zum o. g. genannten Hinweis, dass der Driver dem Observer erklären soll, was er vorhat, kann ich nach der intensiven Untersuchung vieler Sitzungen noch hinzufügen: Es wäre sicherlich keine schlechte Idee, wenn sich beide Entwickler im Laufe der Sitzung Gedanken darüber machen, in welcher Phase sie sich gerade befinden, welche Rollen in dieser Phase *sinnvoll* sind, und was die entsprechenden Aufgaben sind, die die eigene Rolle und die des Partners erfüllen sollten.
Die Untersuchung der Phasenkonzepte wurde in dieser Arbeit lediglich begonnen, und es wurden auch nur einige und nicht alle Rollenverteilungen identifizert. Und selbst wenn die *auftretenden* Rollenverteilungen für die jeweiligen Phasen verstanden sind, müssten noch die *günstigsten* davon ermittelt werden. Dieser Hinweis ist also ausgesprochen vage.

Allerdings muss auch angemerkt werden, dass in den untersuchten Sitzungen keine katastrophalen Probleme aufgetreten sind, die das Paar durch Beherzigung irgendeines Ratschlages hätte vermeiden können.

10 Ausblick

10.1 Inspiration aus der Kleingruppenforschung

Ein Teilgebiet der Sozial- und Organisationspsychologie ist die *Kleingruppenforschung*.

Eine *Kleingruppe* ist eine Gruppe, in der sich alle Mitglieder gegenseitig kennen und die Möglichkeit für direkte Kommunikation gegeben ist [15, S. 385]. Folgende drei Kriterien werden gemeinhin für (Klein-)Gruppen angesetzt (nach [15]):

1. Es gibt ein **gemeinsames Ziel** und/oder gemeinsame Motive.

2. Es gibt so etwas wie ein **Gruppenbewusstsein** oder Zugehörigkeitsgefühl (die Mitglieder sprechen z. B. vom „wir“).

3. Die Mitglieder müssen über **längere Zeit** miteinander interagieren.

Eine *Dyade* ist dabei die „kleinste Kleingruppe“ und besteht aus zwei Personen[54].

Jeder Interaktionsschritt innerhalb einer Dyade wird dabei von inneren und äußeren Faktoren determiniert. Aus den relativen Anteilen innerer und äußerer Faktoren bei den beiden Mitgliedern haben Jones & Gerard [18] vier Typen von dyadischen Interaktionssequenzen abgeleitet (siehe Abbildung 10.1).
Die benannten Muster haben folgende Eigenschaften (Beschreibung an [15, S. 386f] angelehnt):

[54] Die Interaktion in größeren Kleingruppen lässt sich durch die dyadischen Interaktionen analysieren [15, S. 385]. Eine Kleingruppe mit den Personen $\{A, B, C\}$ kann man also durch die drei Dyaden $\{A, B\}$, $\{B, C\}$ und $\{A, C\}$ darstellen. Zwar wurden im Rahmen dieser Arbeit Paarprogrammierungssitzungen untersucht, sodass das Vorgehen bei Untersuchungen größerer Kleingruppen eigentlich keine Rolle spielen sollte, allerdings interagierte das Paar aus der Sitzunge Sitzung F02 zumindest vorübergehend mit einer dritten (und sogar vierten) Person. Für die Analyse dieser Prozesse könnte die Betrachtung aller Dyaden nötig werden.

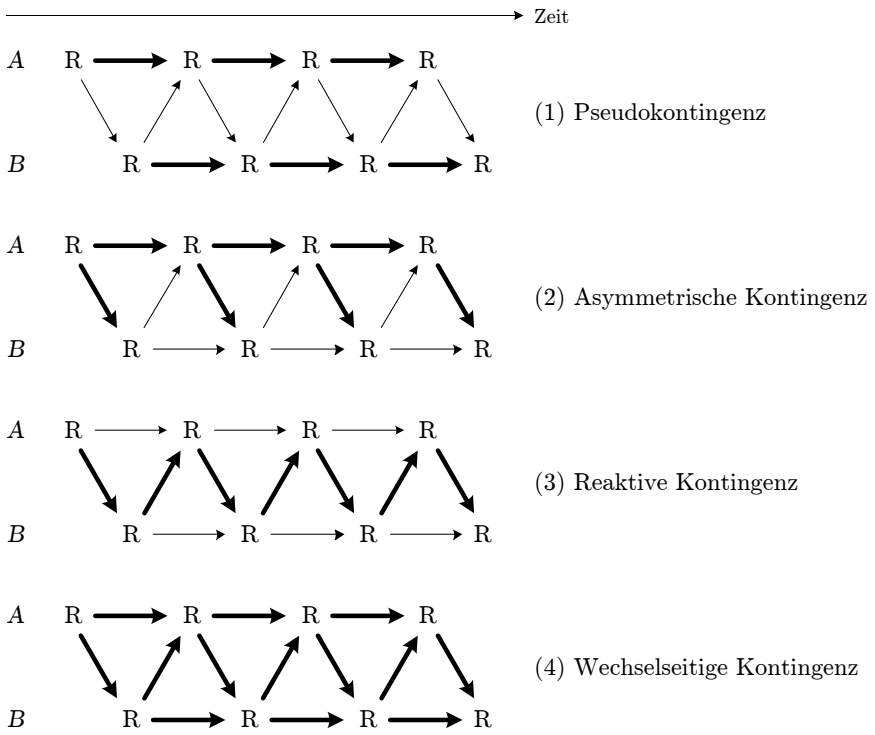

Abbildung 10.1: Dydische Interaktionssequenzen. Die Dicke der Pfeile zeigt die Stärke der Verhaltensdetermination an. Abbildung nach [15, S. 386].

(1) Werden die Personen A und B v. a. durch ihre inneren Faktoren (also z. B. ihre Pläne und Absichten) getrieben und nur wenig durch äußere Reize, ergibt sich ein Schema, das *Pseudokontingenz* genannt wird. Ein solches Interaktionsschema findet sich z. B. bei Schauspielern, die ihren Text auswendig sprechen können (innerer Faktor) und von außen lediglich „synchronisiert“ werden, also ihren Einsatz abpassen müssen.

(2) Ein anderes Muster ergibt sich, wenn eine der Personen sich selbst und die andere Person stark beeinflusst, wobei die andere Person weder einen fest eigenen Plan verfolgt, noch die erste Person merklich beeinflusst. Dieses Schema wird *Asymmetrische Kontingenz* genannt.

(3) Beim belanglosen Smalltalk lässt sich die *Reaktive Kontingenz* beobachten. Ausschlaggebend für das eigene Verhalten ist vor allem die voherige Äußerung des Partner. Ein (Gesprächs-)Ziel gibt es nicht wirklich.

(4) Die vollständige „Vernetzung“ ist die *Wechselseitige Kontingenz*. Hierbei werden von beiden Personen sowohl eigene Pläne verfolgt, als auch auf den jeweiligen Partner eingegangen. Die jeweiligen Ziele können, müssen aber nicht identisch oder kompatibel sein. Herkner nennt hier eine Verhandlung über die Kapitulation einer der Parteien als Beispiel für inkompatible Ziele.

Angewendet auf die Paarprogrammierung könnte man eine leicht veränderte Form der *asymmetrischen Kontingenz* – B müsste noch einen eigenen Plan verfolgen – als typische Driver/Observer-Situation betrachten. A als Driver verfolgt seinen Plan und wird von B nur wenig beeinflusst. B hingegen verfolgt die Schritte, die A geht, aktiv.

Bei den Untersuchungen der Sitzungen für diese Arbeit konnten auch die Verhaltensmuster (1) und (4) beobachtet werden. Zu einer Pseudokontingenz (1) kommt es vorübergehend, wenn beide Entwickler parallel den gleichen Quellcode lesen und verstehen, und dabei abwechselnd einzelne Bestandteile kommentieren. Das Verhalten wird hier hauptsächlich durch den eigenen Fortschritt beim Verstehen determiniert, aber auch durch die Äußerungen des Partner synchronisiert (siehe Selbstgespräche zu Zweit).
Eine wechselseitige Kontingenz (4) kann z. B. während einiger intensiver Diskussionen beobachtet werden. Beide Entwickler verfolgen ihre Ziele und werden vom Verhalten des Partners beeinflusst.

Eine ganzheitliche Betrachtung der Aktitivitätsbeeinflussungen und der zugehörigen dyadischen Interaktionssequenzen könnte ein ganz neues Licht auf die Paarprogrammierung werfen.

Anhänge

A Screenshots

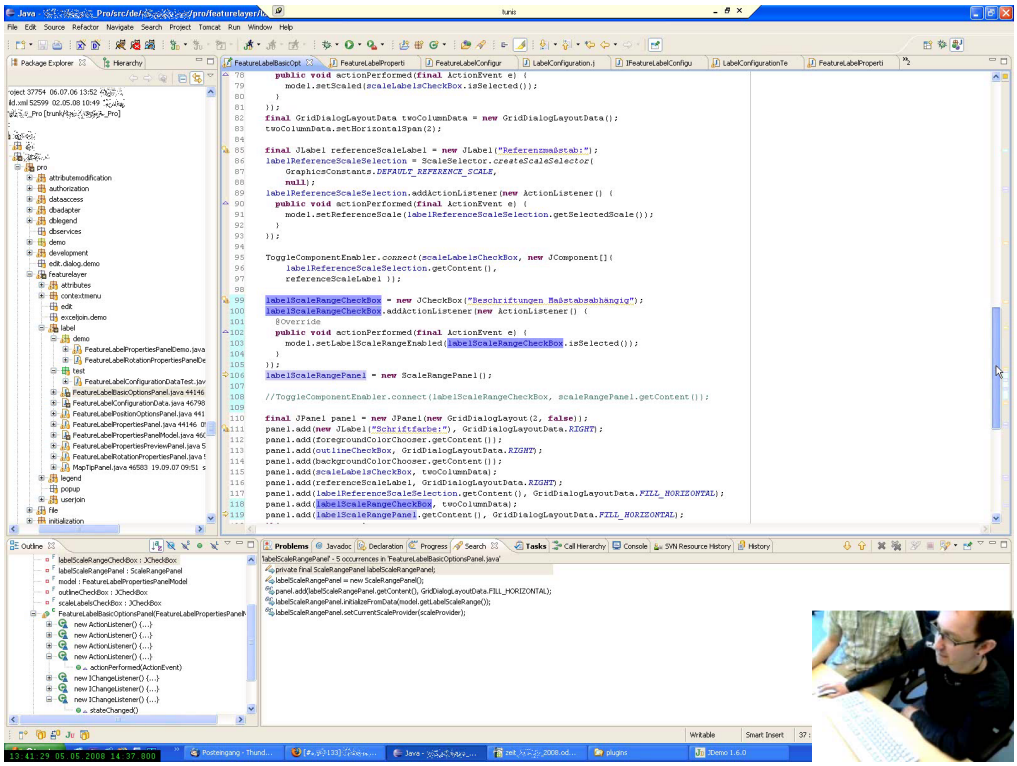

Abbildung A.1: Screenshot aus Sitzung D03. Die Sitzung läuft seit ca. 15 Minuten.

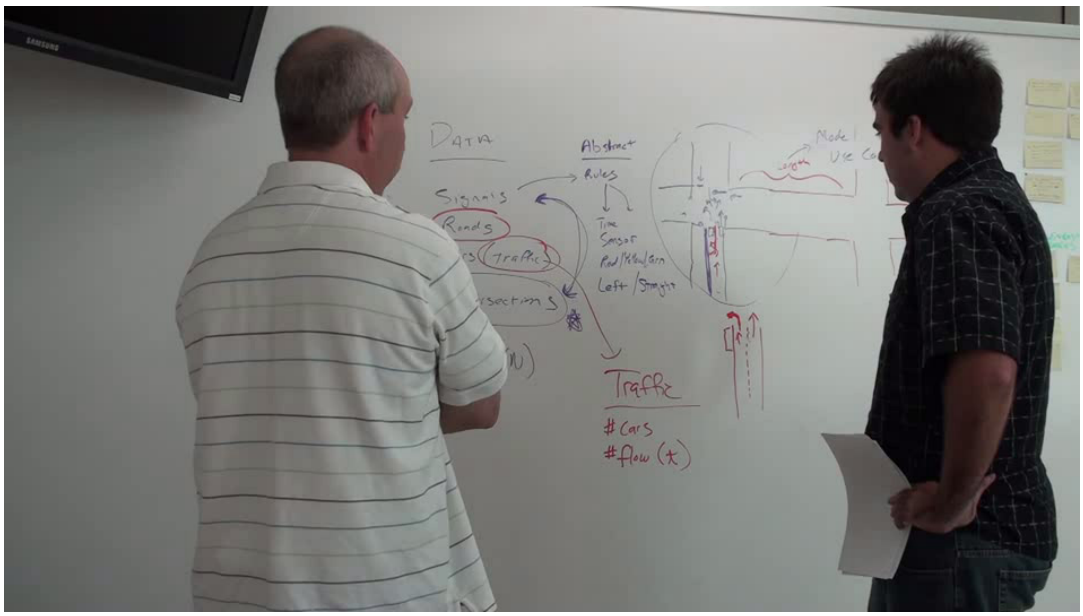

Abbildung A.2: Screenshot aus Sitzung W02. Die Sitzung läuft seit ca. 35 Minuten (siehe/see Acknowledgements).

B Nebenbei „abgefallene“ Erkenntnisse

B.1 Verhältnis zwischen *todo* und *design*

In einer untersuchten Sitzung wurde eine für die Entwickler intuitiv benutzte Verbindung von *todo*- und *design*-Konzepten beobachtet: Durch den Gebrauch von und die Diskussion über `TODO`- bzw. `TODO_NOW`-Kommentare(n)[55] – eindeutige *design*-Konzepte – wurden zukünftige Arbeitsschritte, deren Inhalt und Verantwortlichkeit diskutiert – klare Vertreter der *todo*-Klasse.

Die Gestaltung der (nicht funktionsträchtigen) Kommentare im Programmcode wurden also wie eine verbale Äußerung oder eine gezeichnete Skizze als Hilfsmittel der Kommuniktion über ein anderes Thema verwendet. Hier lässt sich eine Parallele zu einer Eigenschaft finden, die der Linguist Saussure der Sprache zuordnet: Er sagt, dass die Bedeutung von Worten „relational“ sei – die Bedeutung eines Wortes hänge von seinem Umfeld ab. Als bekanntes Beispiel wird hier der „8 Uhr 25-Zug von Genf nach Paris“ genannt. Was diesen Zug auszeichnet ist nicht seine Beschaffenheit und sein Material; es muss nicht mal ein *Zug* sein, da auch ein Bus als Schienenersatzverkehr eingesetzt werden könnte. Entscheidend ist nur, was er *nicht* ist bzw. wogegen er sich abgrenzt: z. B. ist er nicht der Zug der 7 Uhr 25 fährt. [4, S. 41f]

Eine solche Instrumentalisierung lässt sich eventuell auch in anderen Bereichen finden.

B.2 Wiederaufgreifen von *todo*: *remember_todo*

In Sitzung D03 schlägt der Observer vor, nun eine zuvor als *todo* aufgefasste Aufgabe anzugehen. Dies ist im Sinne der Basisschicht ein *propose_step*. Er hebt es zwar nicht explizit hervor, aber aus dem Kontext wird ersichtlich, dass er sich an die vorherige Absprache erinnert, sodass hier auch ein neues Konzept *remember_todo* verwendet werden könnte.

In der Regel werden bei der Diskussion über *todo*s (mittels *propose_todo* und *agree_todo*) keine Aussagen darüber getroffen, wann diese Aufgaben angegangen werden sollen – insbesondere nicht, ob sie noch während der Sitzung erledigt werden sollten. Falls sich solche *propose_todo*-Äußerungen doch beobachten lassen, könnte es interessant sein, in wie weit diese tatsächlich noch erledigt werden, oder ob sie im Laufe der Sitzung vergessen oder auch obsolet werden und wie die beiden Programmierer damit umgehen.

Unbestritten dürfte sein, dass es solche Vorkomnisse geben kann und dass sich diese negativ auf die Gesamtproduktivität auswirken können. Es wäre daher sicherlich hilfreich, wenn die beiden Programmierer festgestellte *todo*s sofort auf einem Zettel notieren oder digital direkt an die betreffende Klasse „heften“ würden, um so zu gegebener Zeit – z. B. vor dem Einchecken der Änderungen – daran erinnert zu werden, dass es noch offene Punkte gibt.

[55] In Eclipse werden diese „Task Tags“ in einem speziellen Bereich („Task View“) gesammelt. *http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fctskview.htm* (Abruf: 19.08.2011)

B.3 Selbstgespräche zu zweit

B.3.1 Forschungsstand zu Selbstgesprächen

Der Entwicklungspsychologe Adam Winsler [47] bezeichnet Selbstgespräche als „Werkzeuge des Denkens“. Der „Dialog mit sich selbst“ helfe Kindern bereits früh in ihrer Entwicklung ihre eigene Aufmerksamkeit zu fokussieren.

Der Sozialpsychologe Roy Baumeister, sowie auch der Persönlichkeitspsychologe Walter Mischel vermuteten, dass „Selbstgespräche eine wichtige Rolle bei der Umsetzung selbst gesetzter Ziele und bei der Handlungskontrolle einnehmen.“ [40][56]

Manfra & Winsler [20] ermittelten in einer Studie, dass etwa die Hälfte der Drei- bis Fünfjährigen sich darüber im Klaren ist, das sie bei einem Selbstgespräch mit sich selbst kommunizieren. Ab dem sechsten Lebensjahr werden die Selbstgespräche „nach und nach ins Unhörbare [verbannt, wahrscheinlich] aus anerzogener Rücksichtnahme.“ [40]

Allgemein werden Selbstgespräche aber seit einiger Zeit als Mittel zur Selbstregulation betrachtet, also jener bewussten und unbewussten psychischen Vorgänge, mit denen Menschen ihre Aufmerksamkeit, Gefühle und Impulse steuern [40].

B.3.2 Deutung von *think aloud_activity*s als Selbstgespräche

Erwachsene Menschen sprechen mit sich selbst meistens dann, wenn sie allein sind [10]. Das liegt vermutlich daran, dass Selbstgepräche bei Erwachsenen oft als Indiz für psychische Krankheiten betrachtet werden – lange Zeit auch aus Sicht der Forschung [40].

Bei Paarprogrammierungssitzungen kommunizieren die beiden Entwickler im Normalfall sehr viel – Kent Beck bezeichnet Paarprogrammierung sogar als *„Dialog zweier Leute, die versuchen gemeinsam zu programmieren“*, und als eine *„Konversation [. . .], die auf einen Computer fokussiert ist“* [6, Kap. 16]. Und da generell viel geredet wird, fiele es gar nicht negativ auf, wenn einige der Äußerungen nicht primär für den Partner bestimmt wären, sondern an die eigene Person adressiert sind. Die Hemmschwelle, mit sich selbst zu reden, ist im Kontext einer PP-Sitzung somit vermutlich wesentlich geringer als bei der Einzelarbeit.

Ein positiver Effekt dieser enthemmten Selbstgespräche ist natürlich, dass der Partner auf dem Laufenden gehalten wird, oder – auch wenn er den Gedankengängen nicht komplett nachvollziehen kann, weil er vielleicht selbst gerade konzentriert nachdenkt – zumindest Denkanstöße für katalysierte eigene Erkenntnisse erhält und das Paar so ingesamt synergetisch arbeiten kann.

C Transkriptionshinweise

Die Transkriptionen orientieren sich an dem von Salinger verwendeten Schema (ausführlich dazu [29, Anhang III]).

[56] Die ursprüngliche Quelle konnte leider nicht – auch nicht nach Rücksprache mit dem Verlag – ausfindig gemacht werden.

Notation	Beschreibung	Kommentar/Beispiel
(.) (..) (...)	Kurze (bis zu einer Sekunde), mittlere (mehr als eine und bis zu zwei Sekunden) bzw. längerer Sprechpause (mehr als zwei und bis zu drei Sekunden). Weitere Punkte entsprechend	Die verstrichene Zeit wurde i. d. R. geschätzt, nicht gemessen.
<* *K* *>	Ergänzungen/Kommentare *(K)* durch den Transkriptor.	
<**E**>	Ersetzung *(E)* durch den Transkriptor zum Zwecke der Anonymisierung.	<***Entwicklername***>
(∼)	Ein komplett unverständliches Wort.	
‿	Steht zwischen Worten oder Sätzen, falls der Sprecher diese in auffälliger Weise zusammen zieht bzw. schnell hintereinander ausspricht.	
(!...!)	Der Sprecher bricht mitten Satz ab, ohne unterbrochen worden zu sein.	
(!!...!!)	Der Sprecher bricht mitten Satz ab, weil er unterbrochen wird.	
(# A #)	Uneigentliches Sprechen, wie z. B. Vorlesen von Bildschirminhalten.	Außer dem Mitlesen gerade stattfindender Eingaben.
(## A ##)	Das Mitlesen gerade stattfindender Eingaben.	
[...]	Nicht wiedergegebener Teil einer Äußerung.	

Tabelle C.1: Transkriptionsregeln. Auszug aus [29].

D Acknowledgements

Die Aufzeichnung der Sitzung W02 wurde im Rahmen eines Workshops aus dem Jahre 2010 von der University of California angefertigt. Nähere Informationen zu diesem Workshop sind unter *http://www.ics.uci.edu/design-workshop* zu finden.

Results described in this paper are based upon videos and transcripts initially distributed for the 2010 international workshop „Studying Professional Software Design“, as partially supported by NSF grant CCF-0845840.

Literatur

[1] Arisholm, Erik ; Gallis, Hans ; Dybå, Tore ; I.K. Sjøberg, Dag: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. In: *IEEE Transactions on Software Engineering* 33 (2007), Februar, S. 65–86

[2] Baddeley, Alan: Working Memory. In: *Current Biology* 20 (2010), Nr. 4, S. R136–R140

[3] Baheti, Prashant ; Gehringer, Edward ; Stotts, David: Exploring the Efficacy of Distributed Pair Programming. In: Wells, Don (Hrsg.) ; Williams, Laurie (Hrsg.): *Extreme Programming and Agile Methods - XP/Agile Universe 2002* Bd. 2418. Springer Berlin / Heidelberg, 2002, S. 387–410

[4] Barry, Peter: *Beginning theory: An introduction to literary and cultural theory*. Manchester University Press, 1995

[5] Basili, Victor R. ; Perricone, Barry T.: Software errors and complexity: an empirical investigation. In: *Communications of the ACM* 27 (1984), January, S. 42–52

[6] Beck, Kent: *Extreme programming explained: embrace change*. Addison-Wesley, 2000 (The XP series)

[7] Beck, Kent: *Test-driven development: by example*. Addison-Wesley, 2003 (The Addison-Wesley signature series)

[8] Berti, Sebastian: Arbeitsgedächtnis: Vergangenheit, Gegenwart und Zukunft eines theoretischen Konstrukts. In: *Psychologische Rundschau* 61 (2010), Nr. 1, S. 3–9

[9] Birbaumer, Niels ; Schmidt, Robert F.: *Biologische Psychologie*. 6. Auflage. Springer, 2006

[10] Brinthaupt, Thomas M. ; Hein, Michael B. ; Kramer, Tracey E.: The Self-Talk Scale: Development, Factor Analysis, and Validation. In: *Journal of Personality Assessment* 91 (2009), Nr. 1, S. 82–92

[11] Brown, Ann L. ; Palincsar, Annemarie S.: Guided, Cooperative Learning and Individuals Knowledge Acquisition. In: Resnick, Lauren B. (Hrsg.) ; Glaser, Robert (Hrsg.): *Knowing, Learning and Instruction: Essays in Honor of Robert Glaser*. Routledge, 1989, Kapitel 13, S. 393–451

[12] Cockburn, Alistair ; Williams, Laurie: The costs and benefits of pair programming. In: Succi, Giancarlo (Hrsg.) ; Marchesi, Michele (Hrsg.): *Extreme programming examined*. Addison-Wesley Longman Publishing Co., Inc., 2001, S. 223–243

[13] Constantine, L. L.: *Constantine on Peopleware*. Yourdon Press, 1995 (Yourdon Press computing series)

[14] Glaser, Barney M. ; Strauss, Anselm L.: *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, 1967

[15] Herkner, Werner: *Lehrbuch Sozialpsychologie*. 5. Auflage. Verlag Hans Huber, 1996

[16] Jeffries, Ron E. ; Anderson, Ann ; Hendrickson, Chet: *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co., Inc., 2000

[17] Jekutsch, Sebastian: An Annotation Scheme to Support Analysis of Programming Activities. In: *Workshop on Ethnographies of Code, Infolab21, Lancaster University, UK, 30-31 March 2006*, 2006, S. 58–66

[18] Jones, Edward E. ; Gerard, Harold B.: *Foundations of social psychology*. Wiley, 1967

[19] MAGIX AG: *MAGIX Video Pro X3*. http://www.magix.com/de/video-pro-x, 2010

[20] Manfra, Louis ; Winsler, Adam: Preschool children's awareness of private speech. In: *International Journal of Behavioral Development* 30 (2006), Nr. 6, S. 537–549

[21] McBreen, Pete: *Questioning Extreme Programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002

[22] Muhr, Thomas: *User's Manual for ATLAS.ti 5.0*. http://www.atlasti.com (Abruf: 12.12.2011), Juni 2004

[23] Nawrocki, Jerzy ; Wojciechowski, Adam: *Experimental Evaluation of Pair Programming*. 2001

[24] Nosek, John T.: The case for collaborative programming. In: *Communications of the ACM* 41 (1998), March, S. 105–108

[25] Plonka, Laura: A comparison between Student and Professional Pair Programmers. In: *Proceedings of the 20th Annual Workshop of Psychology of Programming Interest Group (PPIG '08), Lancaster, September 10-12, 2008*, 2008

[26] Polcwiartek, Wojciech: *Charakterisierung des Designprozesses bei der Paarprogrammierung*, Freie Universität Berlin, Institut für Informatik, Diplomarbeit, Mai 2009

[27] Prechelt, Lutz: *Promotionsverfahren am Fachbereich Mathematik und Informatik*. Version: August 2005. https://www.mi.fu-berlin.de/w/Mi/PromotionsVerfahren?rev=30, Abruf: 08.01.2012

[28] Prechelt, Lutz: Errors and Defects. In: *Course „Spezielle Themen der Softwaretechnik"*, 2009

[29] Salinger, Stephan: *Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung (Arbeitstitel)*. 2012 Dissertation

[30] Salinger, Stephan ; Plonka, Laura ; Pre, Lutz: A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming. In: *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group*, 2007, S. 144–157

[31] Salinger, Stephan ; Plonka, Laura ; Prechelt, Lutz: A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming. In: *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* 4 (2008), Mai, S. 9–25

[32] Salinger, Stephan ; Prechelt, Lutz: What happens during Pair Programming. In: *Proceedings of the 20th Annual Workshop of Psychology of Programming Interest Group (PPIG '08), Lancaster, September 10-12, 2008.*, 2008

[33] Schönpflug, Wolfgang: *Einführung in die Psychologie*. Psychologie Verlagsunion, 2006

[34] Strauss, Anselm ; Corbin, Juliet: *Grounded Theory: Grundlagen Qualitativer Sozialforschung*. Psychologie Verlags Union, 1996

[35] Tang, Antony ; Aleti, Aldeida ; Burge, Janet ; van Vliet, Hans: What makes software design effective? In: *Design Studies* 31 (2010), S. 614–640

[36] TechSmith Corporation: *Camtasia Studio*. http://www.techsmith.com/camtasia.html,

[37] Vanhanen, Jari: *Effects of Pair Programming at the Development Team Level: An Experiment*, Helsinki University of Technology, Licentiate Thesis, 2005

[38] Walle, Thorbjorn ; Hannay, Jo E.: Personality and the nature of collaboration in pair programming. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, 2009 (ESEM '09), S. 203–213

[39] Webb, Noreen M. ; Lewis, Scott: The Social Context of Learning Computer Programming. In: Mayer, Richard E. (Hrsg.): *Training and Learning Computer Programming: Multiple Research Perspectives*. Mallory International, 1988, Kapitel 8, S. 179–206

[40] Wilhelm, Klaus: Schweigen ist Silber, Reden ist Gold. In: *Gehirn & Geist* 12 (2011), Dezember, S. 14–18

[41] Williams, Laurie A.: *The collaborative software process*, The University of Utah, Dissertation, 2000

[42] Williams, Laurie A. ; Kessler, Robert R.: All I really need to know about pair programming I learned in kindergarten. In: *Communications of the ACM* 43 (2000), May, S. 108–114

[43] Williams, Laurie A. ; Kessler, Robert R.: The Effects of „Pair-Pressure" and „Pair-Learning" on Software Engineering Education. In: *Proceedings of the 13th Conference on Software Engineering Education & Training*, IEEE Computer Society, 2000 (CSEET '00), S. 59–68

[44] Williams, Laurie A. ; Kessler, Robert R.: *Pair programming illuminated*. Addison-Wesley, 2003

[45] Williams, Laurie A. ; Kessler, Robert R. ; Cunningham, Ward ; Jeffries, Ron: Strengthening the Case for Pair Programming. In: *IEEE Software* 17 (2000), S. 19–25

[46] Wilson, Judith D. ; Hoskin, Nathan ; Nosek, John T.: The benefits of collaboration for student programmers. In: *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, ACM, 1993 (SIGCSE '93), S. 160–164

[47] Winsler, Adam ; Manfra, Louis ; Diaz, Rafael M.: „Should I let them talk?“: Private speech and task performance among preschool children with and without behavior problems. In: *Early Childhood Research Quarterly* 22 (2007), S. 215–231

[48] Yngve, Victor H.: On getting a word in edgewise. In: *Papers from the 6th regional meeting of the Chicago Linguistic Society* (1970), April, S. 567–578

[49] Zieris, Franz: *Das „a“*, Freie Universität Berlin, Institut für Informatik, Bachelorarbeit, August 2009