Qualitative Analyse des Wissenstransfers bei der Paarprogrammierung[1]

Franz Zieris[2]

Abstract: Bei der Paarprogrammierung (PP) arbeiten zwei Softwareentwickler/innen an einem Computer eng zusammen an einer technischen Aufgabe. Praktiker erhoffen sich davon eine Reihe von Vorteilen, wie etwa schnelleren Fortschritt, höhere Qualität und den Austausch von Wissen. Während die bisherige Forschung oft auf unmittelbar messbare Effekte aus Laborsituationen fokussiert war, die auftretenden großen Streuungen aber nicht erklären konnte, richtet sich meine Forschung auf das Verstehen der zu Grunde liegenden Mechanismen. Ich habe Videoaufzeichnungen von 27 industriellen PP-Sitzungen qualitativ analysiert und eine *Grounded Theory* des Wissenstransfers bei der PP erarbeitet: Zentral in PP-Sitzungen ist *aufgaben-spezifisches Wissen über das Softwaresystem*. Paare gleichen zunächst ihr diesbezügliches Vorwissen an, bevor sie gemeinsam fehlendes Wissen aufbauen. Transfer von *Wissen über Softwareentwicklung allgemein* spielt hingegen eine viel kleinere Rolle und erfolgt erst, wenn das Paar seine Bedürfnisse nach System-Wissen geregelt hat. Paare, die ihr gemeinsames Verständnis pflegen, können kurze, aber sehr produktive *Fokusphasen* haben; ist es zu schwach, droht hingegen ein *Zusammenbruch* des Paarprozesses.

1 Einführung

Die Idee, sich als Softwareentwickler/in nicht allein, sondern zu zweit über ein Programmier-Problem zu beugen, ist vermutlich so alt wie das Handwerk selbst. In den 1990er Jahren wurde sie u.a. von Coplien als *Muster* beschrieben [Co98, S. 294] und erlangte durch Beck im Rahmen der agilen Entwicklungsmethode *eXtreme Programming* [Be99] unter dem Namen *Pair Programming* bzw. *Paarprogrammierung* (PP) große Bekanntheit.

Durch diesen von engster Zusammenarbeit und ständiger Kommunikation geprägten Entwicklungsstil erhoffen sich Praktiker in der Industrie eine Reihe von Vorteilen [BN08]:

- Zwei Entwickler/innen verfügen über mehr Wissen, können so mehr Ideen produzieren, an komplizierteren Aufgaben arbeiten als jede/r allein und weniger Defekte und bessere Entwürfe in kürzerer Zeit hervorbringen.
- Fehlendes Wissen für Defektsuche und Systemverstehen lässt sich zu zweit schneller und zuverlässiger aneignen und im Gedächtnis behalten.
- Die Entwickler/innen können für zukünftige Aufgaben voneinander oder gemeinsam Neues lernen, was das Risiko von Wissensinseln im Team senkt.

[1] Englischer Titel der Dissertation: “Qualitative Analysis of Knowledge Transfer in Pair Programming” [Zi20]

[2] Freie Universität Berlin, zieris@inf.fu-berlin.de

cc by sa

Obwohl die Paarprogrammierung seit den 1990er Jahren Gegenstand zahlreicher wisenschaftlicher Untersuchungen war, gibt es bislang keine klaren Erkenntnisse darüber, in dem welchem Grad sich diese Vorteile tatsächlich einstellen und welche Rahmenbedingungen dafür entscheidend sind. Wie ich in Abschnitt 2 erläutern werde, liegen die Gründe dafür in einem simplistischen und teils dogmatischen Verständnis der Paarprogrammierung, sowie einer Versteifung auf quantitative Untersuchungen unter Laborbedingungen. In Abschnitt 3 beschreibe ich, wie ich mit einem qualitativen Forschungsansatz in der industriellen Praxis erhobene Daten analysiert habe – vorrangig Videoaufzeichnungen von Paarprogrammierungssitzungen, ergänzt durch Feldbeobachtungen und Interviews. Ich gebe einen kurzen Einblick in einen Teil meiner Ergebnisse und wie ich diese mit Praktikern validiert habe (Abschnitt 4) bevor ich in Abschnitt 5 meine Arbeit zusammenfasse.

2 Überblick über die Erforschung der Paarprogrammierung

Wissenschaftliche Studien zur Paarprogrammierung fallen grob in zwei Kategorien. Auf der einen Seite sind Untersuchungen in kontrollierten Umgebungen, in denen oft zufällig zusammengesetzte Paare kleine Programmieraufgaben lösen, um mit Einzelarbeitenden quantitativ verglichen zu werden (Abschnitt 2.1). Auf der anderen Seite stehen qualitative Analysen natürlicher Situationen, in denen komplexe Aufgaben von selbstorganisierten und spontan gebildeten Paaren bearbeitet werden (Abschnitt 2.2).

Ich klammere in der folgenden Diskussion Studien zur Paarprogrammierung aus, die nur auf Fragebögen und Interviews basieren, da durch die Art der Datenerhebung dort Detailtiefe und Nähe zu realen Ereignissen nicht gewährleistet sind.

2.1 Quantitative Studien zur Paarprogrammierung: Ergebnisse und Probleme

Über die Jahre wurden kontrollierte Experimente durchgeführt, in denen die Arbeitsgeschwindigkeit und -qualtität von Paaren mit der von Alleinarbeitenden verglichen wurde. Hannay et al.'s Meta-Analyse über 18 solcher Experimente [Ha09] zeigt zwar im Mittel einen signifikaten positiven Effekt der Paarprogrammierung auf *Qualität* und *Bearbeitungsdauer in Zeitstunden*,[3] allerdings *keinen* statistisch signifikanten Effekt auf den *Aufwand in Personenstunden*.[4] Bedeutsamer ist aber noch die von Hannay et al. ermittelte *große Heterogenität* zwischen den Effekten der Primärstudien, die darauf hinweist, dass diese Studien womöglich wesentlich verschiedene Dinge gemessen haben und nicht vergleichbar sind. Die Schlussfolgerung von Hannay et al.: *Paarprogrammierung* sei nicht per se besser oder schlechter, sondern hänge von noch nicht verstandenen situativen Faktoren ab, wie etwa Aufgabenkomplexität, Entwicklererfahrung (im Programmieren *und* im Paarprogrammieren), Motivation und Teamklima.

[3] Kleine bzw. mittlere Effektgröße, 95%-Vertrauensbereich von Hedges $g = [0.07, 0.60]$ bzw. $g = [0.13, 0.94]$.
[4] 95%-Vertrauensbereich: $g = [-1.18, 0.13]$

Allerdings konnte selbst das großangelegte Experiment von Arisholm et al. [Ar07] trotz vieler Versuchspersonen (295 professionelle Softwareentwickler) keinen klaren moderierenden Einfluss von *Aufgabenschwere* (konkret: einfache vs. komplexe Architektur) und *Programmiererfahrung* auf die Bearbeitungsdauer und Korrektheit der Lösungen von Paaren im Vergleich zu Solos nachweisen. Ingesamt gehen die Probleme quantitativer PP-Studien jedoch noch über nicht verstandene Moderator-Variablen hinaus:

1. Es gibt zu viele potenzielle Einflussgrößen als dass sie realitischerweise in kontrollierten Experimenten durchgeprüft werden könnten. Das obige Experiment ist trotz immensen Aufwands an der Analyse nur *zweier* Faktoren gescheitert.
2. Die Experimentsituationen unterscheiden sich relevant von der industriellen Praxis, sodass etwaige Ergebnisse ohnehin nicht übertragbar wären: Die Versuchspersonen haben oft wenig Erfahrung mit dem Paar-Arbeitsmodus; werden an unbekannte Systeme an vorgegebene Spielzeugaufgaben[5] gesetzt, ohne entscheiden zu können, ob sie Paararbeit hier überhaupt für sinnvoll halten; und das mit ihnen zugewiesenen Partnern, anstatt dass sich Paare dynamisch aus dem Projektalltag ergeben.
3. Es wird stillschweigend angenommen, dass "Paarprogrammierung" etwas kanonisches ist, das Entwickler/innen einfach "tun" können. Wie aber geht Paarprogrammierung 'richtig'? Diskutiert man erst eine Reihe von Ideen und wählt dann sorgfältig eine aus? Oder folgt man der u.a. von Williams & Kessler [WK02] vorgeschlagenen Rollenteilung in aktiven "Driver" und kontrollierenden "Navigator"? Verfolgt man die Gedanken eines Partners so lange bis man in eine Sackgasse gerät und wechselt dann die Führung? Oder verfolgt ein Partner nur still das Geschehen, bis ihm ein Problem auffällt? Verschiedene Paare werden verschiedenen Prozessmustern folgen.

In der Konsequenz ist es nicht zielführend Solo- und Paarprogrammierung in künstlichen Umgebungen nur anhand summativer Ergebnis-Metriken quantitativ zu vergleichen. Für ein Verständis darüber, wie und wann Paarprogrammierung funktioniert, sind *qualitative* Forschungsmethoden nötig, die einerseits den *Prozess* in den Blick nehmen und andererseits die Praktik dort untersuchen, wo sie letztlich – übrigens ohnehin unabhängig von experimentellen Ergebnissen – eingesetzt wird: In echten Projekten in der Industrie.

2.2 Qualitative Studien zur Paarprogrammierung: Ergebnisse und Probleme

Einige qualitativ-quantative Studien haben industrielle (z.B. [BRB08]) oder industrie-nahe (z.B. [WH09]) PP-Sitzungen aufgezeichnet, Dialoge transkribiert, gelabelt und Zählstatistiken ausgewertet. Diese Studien zeigen, dass es eine ständige Kommunikation zwischen den Partnern gibt, und dass es bei der Art der Sprachbeiträge *keine* systematischen Unterschiede

[5] Das "komplexe" Experiment-System von Arisholm et al. [Ar07] umfasste z.B. nur 12 Klassen mit 287 Zeilen Code; die Änderungen dauerten im Mittel gerade einmal 60 Minuten.

zwischen vermeintlichen “Drivern” und “Navigatoren” gibt. Allerdings benutzen diese Studien ebenfalls summative Metriken, bei denen ganze PP-Sitzungen als einzelne Datenpunkte unter Vernachlässigung zeitlicher Bezüge betrachtet werden. Ohne Prozessdimension sind diese Forschungsansätze ebenfalls nicht geeignet, die Paarprogrammierung zu erklären.

Qualitative PP-Studien, die auf Theoriebildung statt Hypothesentests ausgerichtet sind, gehen weiter und charakterisieren PP-Prozesse auf einer konzeptionellen Ebene. Sie zeigen z.B. dass die Kommunikation guter Paare nach Mustern verläuft (z.B. implizite vs. explizite Erklärungen [Pl15] oder *Restarting*, *Planing*, *Action* [ZHR13]). Ein zentrales Problem der Ergebnisse vieler solcher Arbeiten ist allerdings, dass sie eher *beschreibende Taxonomien* von PP-Aspekten sind als dass sie auf eine *erklärende Theorie* der PP hinarbeiten, und so kaum weiterführende Forschung erlauben. Auch fehlt oft eine Praxisorientierung, die einen Nutzen für die Anwendung in der Industrie zumindest in Aussicht stellt.

3 Zielsetzung, Datengrundlage und Forschungsmethode

Das Ziel meiner qualitativen Forschung ist es, zu verstehen wie Wissenstransfer bei der Paarprogrammierung tatsächlich funktioniert. Es geht nicht darum zu klären, ob Paarprogrammierung “besser” ist, sondern Praktikern, die paarprogrammieren möchten, Hilfestellung zu geben, um Probleme zu vermeiden und Potentiale zu nutzen. Meine qualitative Forschung folgt der *Grounded Theory Methodology* (GTM) nach Strauss & Corbin [SC90] und baut auf die Vorarbeit von Salinger [Sa13] auf, der ein “Vokabular” zur Charakterisierung der Basisaktivitäten in einem PP-Prozess entwickelt hat.

Datengrundlage sind seit 2007 von Kollegen und mir in 13 Firmen gesammelte Aufzeichnungen von PP-Sitzungen,[6] in denen professionelle Softwareentwickler/innen selbstbestimmt an ihren alltäglichen Aufgaben arbeiten. Reflektierende Interviews, Gruppendiskussionen und Workshops zur Bewertung und Einordnung der Erkenntnisse ergänzen das Material. Ich habe der GTM-Methode des *Theoretischen Samplings* [SC90] folgend, iterativ insgesamt 27 Sitzungen aus 10 Firmen mit verschiedenen Technologiestacks und Anwendungsdomänen, sowie 29 Entwickler/inne/n verschiedener Erfahrungsstufen gewählt (siehe Tab. 1).

Meine Analysen habe ich nicht auf Transkripten, sondern direkt auf den Videos durchgeführt. Beobachtungsnotizen und reflektierende Gespräche mit den Entwickler/inne/n am Tag nach einer Aufzeichnung dienten v.a. zur Einordnung der Geschehnisse und der Vervollständigung von Kontextinformationen. In meiner Analyse habe ich die GTM-Praktiken des *offenen*, *axialen*, und *selektiven Kodierens* [SC90] angewendet, wobei ich mich schrittweise von der Ebene tausender einzelner Äußerungen, über hunderte Wissenstransfer-Episoden (einige Sekunden bis Minuten), zu Episoden-Clustern bis hin zur Gesamtsitzungsdynamik vorarbeitete, bis eine *theoretische Sättigung* [SC90] meiner Konzepte erreicht war. Im Folgenden gebe ich einen kurzen Einblick in einen Teil meiner Ergebnisse.

[6] Bestehend aus Bildschirmvideo, Webcam und Audio; Details sind ausführlich in einem technischen Bericht [ZP20] beschrieben.

Firma mit Anwendungsdomäne und Programmiersprachen		**Sitzungen**		**#Entwickler**
A:	Content-Management-System (Java, Objective-C, SQL)	1	02:22h	2
B:	Social Media (PHP, JavaScript, SQL, HTML, CSS)	3	06:30h	2
C:	Geoinformationssystem (Java)	5	07:49h	8
D:	Customer-Relationship-Management (Java, XML)	1	02:24h	2
E:	Logistik (C++, XML)	1	01:17h	2
J:	Rundfunk-Datenmanagement (Java)	2	02:22h	2
K:	Immobilienplattform (Java, SQL, CoffeeScript)	4	05:52h	3
M:	Datenanalyse in Energie & Logistik (SQL)	1	00:25h	2
O:	Online-Projektplanung (CoffeeScript)	4	05:11h	3
P:	Online-Autoteilehändler (PHP, SQL)	4	05:41h	3
	SUMME	27	39:53h	29

Tab. 1: Kontexte und Umfang der analysierten Sitzungsaufzeichnungen (Details in [ZP20]).

4 Überblick über die Ergebnisse

4.1 Paarhaftigkeit und Prozessflüssigkeit

Unter den analysierten Paaren gibt es große Unterschiede in Bezug auf die **Flüssigkeit** (*Fluency*) des Fortschrittes. Manche Paare haben **Fokusphasen** während derer es praktisch keine Sprechpausen gibt und die Partner gegenseitig ihre Gedanken, teilweise sogar Sätze und Programmcodezeilen ergänzen. Es ist unmöglich die Dynamik einer Fokusphase auf Papier greifbar zu machen. Als Näherung sind in Abb. 1 die Aktiväten und Sprachäußerungen einer Fokusphase im zeitlichen Verlauf dargestellt: Innerhalb von 60 Sekunden bespricht das Paar 11 (!) verschiedene Themen, spricht und editiert dabei parallel und nahezu ununterbrochen. Demgegenüber stehen Paare, deren Paarprozess einen **Zusammenbruch** erleidet, weil sie nicht mehr inhaltlich auf die Äußerungen ihres Partners Bezug nehmen, oder sogar in eine verlegene Starre verfallen (etwa bei A: *"Ah, der erwartet eine Zahl, kriegt aber ein Objekt!"* – B: *[bewegt stumm Mauscursor für 30 Sekunden ziellos über Bildschirm]*).

Maßgeblich für die Flüssigkeit eines Paarprogrammierprozesses ist die Leichtigkeit mit der die Partner gegenseitig die Intentionen ihrer Handlungen und Äußerungen verstehen können. Für diese **Paarhaftigkeit** (*Togetherness*) habe ich fünf Einflussfaktoren identifiziert:

1. Ein **gemeinsames Verständnis des Softwaresystems** erlaubt effiziente Kommunikation (z.B. durch kurze, aber verständliche Bezeichner – wie etwa *"Factory"* statt *'FeatureLayerAttributeTableCellRendererFactory'*); fehlt es, sind mehr Erklärungen nötig oder es kommt zu aufzuklärenden Missverständnissen.
2. Ein **gemeinsames Verständnis von Softwareentwicklung**, z.B. über gängige Architekturen, Entwurfsmuster und Lösungsansätze, erlaubt als implizites Wissen ebenfalls eine effiziente Kommunikation; andernfalls fällt es schwer Vorschläge des Partners zu bewerten (*"Ähm, okay? Mach mal weiter. Für mich ist das oberste Wissenskante!"*).

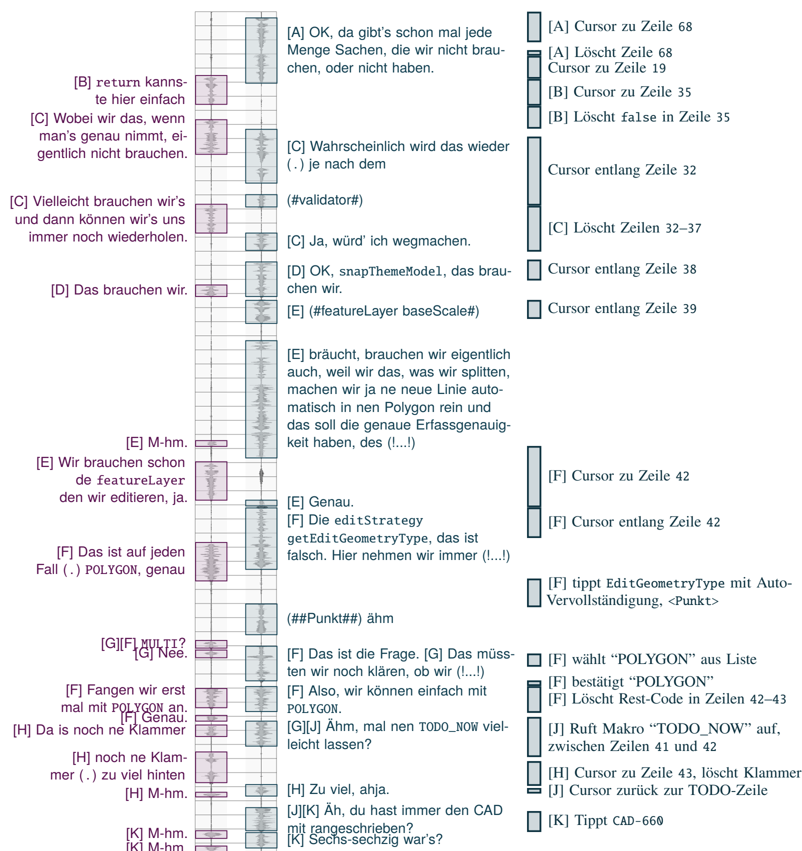

Abb. 1: Eine Fokusphase (60 Sekunden, Zeit von oben nach unten), in der das Paar elf Themen [A]–[K] bespricht (Spalten 1 und 2) und ein Partner direkt Code-Änderungen vornimmt (Spalte 3).

3. Ein **gemeinsamer Plan**, z.B. in der Sitzung getroffene strategische Entscheidungen, bietet den Hintergrund um taktische Vorschläge schnell zu verstehen und zu ergänzen (A: *"[öffnet Code] Okay, viel davon kann weg."* – B: *"Mach hier unten einfach 'return'."*); andernfalls kommt es zu Missverständnissen (A: *"Können wir das debuggen?"* – B: *"Haben wir doch gerade!"*).

4. **Gewahrsein der Arbeitsumgebung** ist z.B. bei verteilter Paarprogrammierung durch die räumliche Trennung reduziert, kann aber auch durch zu kleine Schrift ein Problem im lokalen Fall sein (*"Wo bist du gerade? In welcher Klasse?"*).

5. Eine **Sprachbarriere** kann durch Fremdsprachen bestehen (*"An 'offset' is a duration?"*) oder durch idiosynkratische Ausdrucksweisen (*"Wart mal kurz."* für *'Ich nehme mir jetzt die Tastatur und Maus.'*).

Gute Paare erkennen Defizite und kompensieren z.B. geringeres Gewahrsein (Faktor 4) durch Erläuterungen von Editieraktivitäten, oder nehmen sich die Zeit ein Idiom zu erklären, das den Partner irritiert hat (Faktor 2). Die Paarhaftigkeit ist keine statische Eigenschaft, sondern charaktersiert den momentanen Zusammenhalt und kann von den Entwickler/inne/n vernachlässigt (*“Mach mal. Ich sag, wenn ich wieder voll drin bin.”*) oder durch Reparaturen wiederhergestellt werden (*“Warum hattest du das gerade gemacht?”*).

Die bislang nicht erklärte Varianz der experimentellen Ergebnisse (siehe Abschnitt 2.1) ergibt sich womöglich durch unterschiedlich große Defizite der Paarhaftigkeit der Versuchspaare sowie durch unterschiedliche Kompetenzen mit diesen umzugehen (in Experimentsituation mindestens Faktoren 2 und 3). Klären lässt sich das im Nachhinein nicht. Es bleibt dabei, dass für Forscher Prozesseigenschaften wie *Flüssigkeit* und *Paarhaftigkeit* verborgen bleiben, solange sie Paarprogrammierungssitzungen nur auf summative Metriken wie verstrichene Zeit oder erzeugte Codezeilen reduzieren.

4.2 Wissensbedürfnisse, Wissensbedarfe und eine fundamentale Sitzungsdynamik

Im Kern befasste sich meine Forschung mit der Frage, wie Paarprogrammierer Wissen transferieren und sich gemeinsam neu aneignen. Eine zentrale Beobachtung ist hierbei, dass Wissenstransfer in **jeder Sitzung** stattfindet, sei es als erklärtes *Ziel* der Sitzung, in der ein Partner das Softwaresystem zum ersten Mal sieht, oder als *Nebeneffekt* nachdem das Paar festgestellt hat, dass eine/r eine zu schließende Lücke im System- oder Programmierverständnis hat (Reparatur des Zusammenhalts, siehe Abschnitt 4.1).

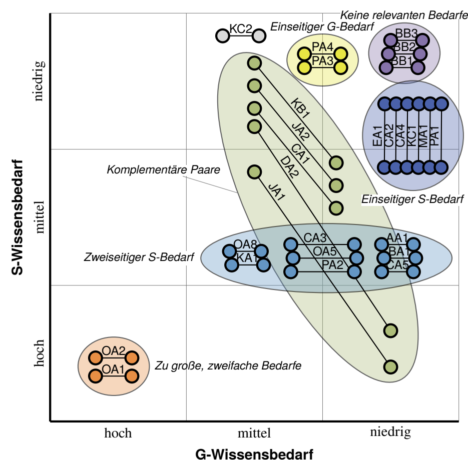

Abb. 2: Startkonstellationen der Wissensbedarfe in den analysierten Paarprogrammierungssitzungen

Ich unterscheide auf der einen Seite ein **Wissensbedürfnis** (*Knowledge Want*), das ein/e Entwickler/in in einer konkreten Situation verspürt und das eine Wissenstransfer-Episode motiviert, in der dann der Partner solange befragt, mit Erklärungen versorgt, oder in die Aneingung neuen Wissens durch Quellcode-Lesen o.ä. einbezogen wird, bis das Bedürfnis gestillt ist.

Der **Wissensbedarf** (*Knowledge Need*) ergibt sich dagegen aus inhaltlichen Anforderungen: Wie gut ist jedes Paarmitglied gewappnet, die Aufgabe erfolgreich zu bearbeiten? Der tatsächliche Wissensbedarf ergibt es sich erst im Laufe einer Sitzung; er kann sich als größer als gedacht herausstellen oder aber durch eine Eingrenzung der Aufgabe schrumpfen.

Basierend auf hunderten analysierten Episoden unterscheide ich zwei Arten von in industriellen Sitzungen relevanten Wissen: System-spezifisches **S-Wissen**, das Wissen über Anforderungen, Architektur und Entwurf, konkreten Technologieeinsatz, Quellcodeeigenschaften und über Defekte umfasst; sowie generisches **G-Wissen** über Softwareentwicklung an sich, das Entwurfsmuster, Programmiersprachen, Werkzeuge und Technologien abdeckt.

Paar-Sitzungen lassen sich nun danach charakterisieren, wie groß die Wissensbedarfe der Partner in den S- und G-Dimensionen *für ihre konkrete Aufgabe* sind, welchen Wissensbedürfnissen sie nachgehen und dadurch ihre Bedarfe erfüllen. Ingesamt habe ich sechs wiederkehrende Startkonstellationen identifiziert (Abb. 2), darunter z.B. *komplementäre Paare*, bei denen ein Partner über mehr aufgabenrelevantes G-Wissen verfügt (etwa bezogen auf Entwurfsmuster und Refactorings) und der andere über mehr S-Wissen (etwa Autorenwissen), oder den *einseitigen S-Bedarf*, der auftritt, wenn ein Partner schon mit der Aufgabe begonnen hat und bereits einen entsprechen S-Wissensvorsprung aufbauen konnte.

In der Zusammenschau der *aller* Sitzungsverläufe habe ich eine gemeinsame, **fundamentale Dynamik** entdeckt, die aus drei Phasen besteht. (1) Schließen der **Primären Wissenlücke**: Zunächst gleichen Paare einen etwaigen S-Wissensunterschied aus (*“Ich erzähl dir mal, was ich schon gemacht hab.”* oder *“Kennst du nicht, ne? Ich zeig dir das Ganze erstmal.”*). (2) Schließen der **Sekundären Wissenslücke**: Sie erkunden die Software durch Lesen oder mittels Debugger und erarbeiten sich so S-Wissen, das beiden noch fehlt. (3) Erst dann, wenn primäre und sekundäre Wissenslücke geschlossen sind, nutzen manche Paare die **Gelegenheit** zum Transfer von G-Wissen, die sich durch einen etwaigen Wissensunterschied bietet (*“Soll ich dir erklären wie OSGi-Classloading funtioniert?”*).

Es ist nun also eine empirische Beobachtung, dass in industrieller Paarprogrammierung der Transfer von system-unabhängigen G-Wissen erst erfolgt, wenn es keine offenen S-Wissensbedürfnisse mehr gibt, die Entwickler/innen also die für die Aufgabe relevanten Systemteile gut genug verstanden haben. Es zeigt sich eine klare Hackordnung der beiden Wissensarten: Die Aneignung von Systemverständnis war in fast allen Sitzungen eine Hauptsache. Unterschiede im Programmierwissen hingegen behinderten die Paare praktisch nicht, sondern blieben höchstens eine ungenutzte Gelegenheit. Wenn jedoch beiden Partnern zu viel G- und S-Wissen fehlt, wie es in zwei untersuchten Sitzungen der Fall war (Abb. 2, links unten), ist die Aufgabe ist schlicht zu schwer und das Paar macht nur wenig Fortschritt.

4.3 Anwendungsmöglichkeiten

Ich habe drei Praktiken zur Einbindung meiner Erkenntnisse in den Entwicklungsalltag eines Teams ausgearbeitet und in vier Firmen mit Workshops und Interviews evaluiert:

1. **Paare formen**: Teams können das G-S-Diagramm (Abb. 2) verwenden, um für konkrete Aufgaben strukturiert vorhandenes Vorwissen und günstige Paarkonstellationen zu besprechen (ob z.B. zwei Teammitglieder ein *komplementäres Paar* bilden).

2. **Sitzungsziel**: Müssen beide ihren kompletten S-Bedarf erfüllen, oder kann ein Teil der *primären Lücke* bleiben? Gibt es eine Gelegenheit zum Transfer von G-Wissen?
3. **Sitzungsreflektion**: Wurden die Wissensbedarfe erfüllt? Gibt es relevante S-/G-Lücken, die vorher nicht bekannt waren? Sollten diese im Team diskutiert werden?

Alle drei Ideen stießen bei den Praktikern auf breite Resonanz. Während Paare-formen und Sitzungsziel-festlegen wegen praktischer Beschränkungen wie Teamgröße, Entwicklerverfügbarkeit, und starrer Aufgaben wenig Wirkung entfalten konnte, waren die Sitzungsreflektionen ein Erfolg, da sie z.B. helfen, sich an erfolgreichen Wissensaustausch zu erinnern (*"Stimmt daran habe ich gar nicht gedacht, das war sogar richtig cool."*).

5 Zusammenfassung

In der Praxis dreht sich der Großteil einer jeden Paarprogrammierungssitzung um das Verstehen des Softwaresystems: Die mit Abstand meisten Wissenstransfer-Episoden haben System-Wissen zum Thema und Paare befassen sich *zuerst* mit ihren Bedürfnissen nach System-Wissen bevor Gelegenheit zum Austausch über allgemeinere Softwareentwicklungsthemen genutzt werden. Für das flüssige Voranschreiten innerhalb einer Sitzung ist es wichtig, dass das Paar sein gemeinsames Verständnis (u.a.) des Softwaresystems pflegt, da sonst ein Zusammenbruch des Paarprozesses droht.

Jahrelange Forschung unter Laborbedingungen hat Effekte der Paarprogrammierung auf Bearbeitungsdauer und Qualität nur in Tendenz und mit großer Streuung nachweisen, diese aber nicht erklären können. Nicht nur ist die Übertragbarkeit dieser Experimentalergebnissen ohnehin fragwürdig, wo doch der in der Praxis zentrale Rückgriff auf systemspezfisches Vorwissen und gemeinsames Verstehen eines komplexen Systems bei Spielzeugaufgaben komplett fehlt. Auch ist rückblickend nicht verwunderlich, dass quantitativ-orientierten Forschungsarbeiten Erklärungen fehlen, wenn sie nur summative Metriken im Blick hatten, nicht aber den eigentlichen Prozess, wodurch ihnen positive wie negative Ausprägungen entgangen sind, wie etwa Fokusphasen und Zusammenbrüche.

Literatur

[Ar07] Arisholm, E.; Gallis, H.; Dybå, T.; Sjøberg, D. I.: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. IEEE Transactions on Software Engineering 33/2, S. 65–86, 2007.

[Be99] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999, ISBN: 0201616416.

[BN08] Begel, A.; Nagappan, N.: Pair Programming: What's in it for Me? In: Proc. Second ACM-IEEE Intl. Symp. on Empirical Software Engineering and Measurement. ACM, S. 120–128, 2008.

[BRB08] Bryant, S.; Romero, P.; du Boulay, B.: Pair Programming and the Mysterious Role of the Navigator. Intl. J. of Human-Computer Studies 66/7, S. 519–529, 2008.

[Co98] Coplien, J.: A Generative Development-Process Pattern Language. In (Rising, L., Hrsg.): The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, S. 243–300, 1998, ISBN: 0-521-64818-1.

[Ha09] Hannay, J. E.; Dybå, T.; Arisholm, E.; Sjøberg, D. I.: The effectiveness of pair programming: A meta-analysis. Information and Software Technology 51/7, S. 1110–1122, 2009.

[Pl15] Plonka, L.; Sharp, H.; van der Linden, J.; Dittrich, Y.: Knowledge transfer in pair programming: An in-depth analysis. Intl. J. of Human-Computer Studies 73/, S. 66–78, 2015.

[Sa13] Salinger, S.: Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung, Diss., Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2013.

[SC90] Strauss, A.; Corbin, J.: Basics of Qualitative Research. Grounded Theory Procedure and Techniques. Sage Publications, 1990, ISBN: 978-0803932500.

[WH09] Walle, T.; Hannay, J. E.: Personality and the Nature of Collaboration in Pair Programming. In: Proc. 3rd Intl. Symp. on Empirical Software Engineering and Measurement. IEEE, S. 203–213, 2009.

[WK02] Williams, L.; Kessler, R. R.: Pair Programming Illuminated. Addison-Wesley Professional, 2002, ISBN: 978-0-201-74576-4.

[ZHR13] Zarb, M.; Hughes, J.; Richards, J.: Industry-Inspired Guidelines Improve Students' Pair Programming Communication. In: Proc. 18th ACM Conf. on Innovation and Technology in Computer Science Education. S. 135–140, 2013.

[Zi20] Zieris, F.: Qualitative Analysis of Knowledge Transfer in Pair Programming, Diss., Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2020.

[ZP20] Zieris, F.; Prechelt, L.: PP-ind: A Repository of Industrial Pair Programming Session Recordings, 2020, URL: https://arxiv.org/abs/2002.03121.

Franz Zieris wurde 1988 in Berlin geboren. Er studierte Informatik mit Psychologie im Nebenfach an der Freien Universität Berlin. Seit 2007 ist er freiberuflicher Softwareentwickler. Von 2012 bis 2020 war er Wissenschaftlicher Mitarbeiter der AG Software Engineering bei Prof. Dr. Lutz Prechelt, wo er mit qualitativen Forschungsmethoden industrienah zu Prozessthemen wie agiler Softwareentwicklung und Paarprogrammierung forschte und zu Themen der Softwaretechnik lehrte. Von 2012 bis 2018 leitete er das Open-Source-Projekt Saros (https://www.saros-project.org), das IDE-Plugins für verteilte Paarprogrammierung entwickelt.