Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Untersuchung von Fokus-Phasen in Paar-Programmierungs-Sitzungen

Thomas Harms

Matrikelnummer: 4914008

t.harms@mad-co.de

Betreuer: Franz Zieris

Eingereicht bei: Prof. Dr. Lutz Prechelt

Zweitgutachter: Prof. Dr. Müller-Birn

Berlin, den 19. Oktober 2017

Zusammenfassung

Die Erwartungen an Paar-Programmierungs-Sitzungen sind sowohl von Forschern, als auch anwendenden, handelnen Programmierern gleichermassen: erhöhte Geschwindigkeit im Arbeitsablauf, bessere Design-Ergebnisse, höhere Erfolgswahrscheinlichkeit bei der Lösung komplexer Probleme und geringere Anzahl von Defekten der Software während der Laufzeit im Vergleich zu Solo-Programmierung. In meinen Untersuchungen habe ich festgestellt, dass die Erwartungen an einen erhöhten Arbeitsfluss und sehr schnelle Fortschritte bis hin zu Fokus-Phasen erfüllt werden können. Mit dieser Arbeit soll ein Einblick vermittelt werden, wie Fokus-Phasen erkannt werden und welche Faktoren dazu beitragen, dass selbige eintreten können oder behindert werden. Auf der Basis der „Grounded Theory Methodology“ (GTM) habe ich voraufgezeichnete Paar-Programmierungs-Sitzungen professioneller Programmierer in Unternehmen analysiert.

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.

19. Oktober 2017
Thomas Harms

Inhaltsverzeichnis

1 Einführung

Paar-Programmierung (PP) ist eine Arbeitstechnik in der Softwareentwicklung, in der zwei Programmierer gemeinsam Softwareentwicklungsprobleme lösen. Unter anderem zählen dazu: Programmieren von Code, Erarbeiten von Designs, Testen von Software etc.

Die am weitesten verbreitete Definition von Paar-Programmierung ist die folgende, aus dem Englischen übersetzte:

In Paar-Programmierungs-Sitzungen produzieren zwei Programmierer gemeinsam ein Artifakt (Design, Algorithmus, Code). Die beiden Programmierer sind wie ein vereinter, intelligenter Organismus mit einem Verstand arbeitend, verantwortlich für jeden Aspekt des Artifaktes. [Der Schreibende] arbeitet [...] an der Tastatur und schreibt den Code. Der andere Partner observiert fortwährend und aktiv des Schreibendens Arbeit, erkennt Fehler, denkt über Alternativen nach, schlägt Ressourcen nach und strategische Ausrichtungen vor. Vgl. [8]

Auf zwei Aspekte dieser Definition möchte ich folgend eingehen:

1. Beide Programmierer arbeiten tatsächlich mit größerer Wahrscheinlichkeit zusammen an einem Thema und entwickeln Ideen gemeinsam. Forscher haben bereits bestätigt (Vgl.[7]), dass die Aufgabenteilung in einen schreibenden und observierenden Teil des Paares selten geschieht.
2. Die Vorstellung zweier Programmierer, als ein „vereinter, intelligenter Organismus mit einem Verstand arbeitend“ ist in dem mir zur Verfügung stehendem Material nicht zu beobachten. Tatsächlich sind große Unterschiede zu beobachten, in wie weit Paare in der Lage sind, zusammenzuarbeiten.

1.1 Problembeschreibung

Jenseits dieser beiden beschriebenen Aspekte kann es in Ausnahmen vorkommen, dass phasenweise tatsächlich ein Zustand erreicht wird, in der die beiden Entwickler „mental synchronisiert“ wirken. Prechelt und Zieris beschreiben eine der wesentlichen Faktoren in PP im Aufbauen und Aufrechterhalten von **Togetherness**. Vgl. [4, S. 1, I] Falls dies erfolgreich ist, kann ein mental synchronisierter Zustand der beiden Entwickler („mit einem Verstand arbeitend“) beobachtet werden Vgl. [4, S. 1, I], welcher zu schnellen Fortschritten innerhalb des Arbeitsprozesses führen kann.
Sie vervollständigen und ergänzen gegenseitig ihre Sätze oder müssen teilweise gar nicht oder nur phrasenweise kommunizieren und verstehen einander trotzdem. Die Kommunikationsinhalte sind dabei trotzdem komplex und nicht trivial. Die Anzahl der getroffenen Entscheidungen während des Arbeitsprozesses ist hoch in erstaunlich kurzer Zeit. Forscher berichteten bereits im Jahr 2007 von diesem Paarverhalten in ihren Analysen von PP-Sitzungen Vgl.[6]. Die Entwickler tauchen in eine Art „Tunnel fokussierten Abarbeitens“ ein und erreichen somit eine hohe Produktivität.

Es stellen sich in diesem Zusammenhang mehrere Fragen:

1. Warum treten so große Unterschiede bzgl. der Togetherness der Entwickler auf?
2. Welche Faktoren tragen dazu bei, dass die Togetherness von Paaren hoch ist?

Die Relevanz der Beantwortung liegt in der Erfüllung der Erwartungen an PP: erhöhte Geschwindigkeit im Arbeitsablauf, bessere Design-Ergebnisse, höhere Erfolgswahrscheinlichkeit bei

der Lösung komplexer Probleme und geringere Anzahl der Defekte im Code im Vergleich zu Solo-Programmierung.

Unternehmen investieren durch den Einsatz des zusätzlichen Entwicklers bei der Lösung eines Problems mittels PP zusätzliche Personenstunden. Entsprechend ist es verständlich, dass die Hoffnung auf die Erfüllung der Erwartungen an PP besteht. Insbesondere ist es wünschenswert, durch PP eine erhöhte Produktivität, bessere Design-Entscheidungen und eine geringere Anzahl der Defekte während der Laufzeit zu erlangen.

PP-Sitzungen werden darüber hinaus oft benutzt, um Wissenstransfers zwischen den Entwicklern zu befördern. Bspw. in Trainingssituationen kann PP genutzt werden, um einem Praktikanten oder Junior Wissen im Vergleich zum Selbststudium effizienter zu vermitteln. Der Wissensgeber profitiert aufgrund der Notwendigkeit der konkreten Ausformulierung der Inhalte im Wissenstranfer.

Ein weiterer Grund für den Einsatz von PP kann ebenso das vorherige Scheitern eines einzelnen Entwicklers bei der Lösung eines komplexen Problemes sein. Diese Situationen habe ich nicht selbst untersucht, sondern wurden mir in Gesprächen mit Forschern berichtet.

Mit meinen Analysen möchte ich zeigen, dass in PP eine erhöhte Togetherness der Partner entscheidend zum Erfolg hinsichtlich der Lösung der Aufgabe in der PP-Sitzung beiträgt. Des weiteren ist es mein Ziel Faktoren aufzuzeigen und zu klären, welche die Togetherness von PP-Paaren beeinflussen und welche Auswirkungen dies auf das Arbeitsergebnis und den Ablauf einer PP-Sitzung haben kann.

1.2 **Forschungsgrundlagen**

In ihren Analysen untersuchten die Autoren in [9] PP-Sitzungen auf experimenteller Basis. In der Veröffentlichung werden u.a. Erkenntnisse hinsichtlich des Aufwandes, der Dauer und der Qualtität der Lösung im Vergleich zu Solo-Programmierung präsentiert. [9, S. 1115] Die Autoren annoncieren interessante Schlüsse, wie bestimmte Faktoren (Komplexität der Aufgabe, Erfahrung der Programmierer, etc.) die Effektivität der genannten Parameter beeinflussen können. Die Konzentration auf den Aufwand ist grundlegend für diese Arbeit. In den Schlussfolgerungen [9, S. 1120, Abschnitt 5] weisen die Forscher darauf hin, dass weitergehende Untersuchungen in der Praxis jenseits von Labor-Umgebungen nötig sind, um konkrete Effekte von PP verstehen zu können. Die Ergebnisse der experimentellen Umgebung sind unklar. Somit dind Untersuchungen auf der Basis qualitativer Analysen norwendig. Diese sind Bestandteil meiner Arbeit und die Erkenntnisse werden in folgenden Kapiteln erläutert.

Die formulierten Grundlagen für die Untersuchungen in [4, S.1] richten den Fokus (ähnlich dieser Arbeit) auf die Togetherness von Paaren. Diese Arbeit baut darauf auf. Jedoch trennt sich der thematische Fokus stark durch unterschiedliche Ausrichtung des Themas. In dieser Arbeit soll es lediglich peripher um das Schwierigkeits-Niveau der Aufgabenstellung und ineffizeinte PP-Sitzungen bis hin zu deren Zusammenbruch gehen. Meine Analyse von Fokus-Phasen richtet sich gezielt auf effiziente Prozesse in PP-Sitzungen und deren Faktoren.

1.3 Forschungsmethode

Mit dem Ziel PP-Sitzungen, die Dynamik der agierenden Programmierer und insbesondere die Auswirkungen auf die Geschwindigkeit hinsichtlich der erzielten Fortschritte zu untersuchen, habe ich mittels qualitativer Analysen vorab aufgezeichnete PP-Sitzungen analysiert. Die Paare bestehen aus professionellen Programmierern deutscher Software-Unternehmen. Die Analysen habe ich auf der Basis der „Grounded Theory Methodology“ (GTM) durchgeführt und mich methodisch an [1] und [2] und im Prozess des Open Coding an [3] orientiert. Die Schritte der Datenanalyse wurden mittels der Software für qualitative Datenanalyse „ATLAS.ti“ durchgeführt.

Das untersuchte Material besteht aus Video- und Audioinhalten. Die Videoinhalte bestehen aus der Bildschirmansicht der PP-Sitzung und einer Webcam mit Blick auf die beiden agierenden Programmierer. Ein Mikrofon nimmt während des Arbeitsprozesses die die gesprochenen Worte des Paares auf. Die Sitzungen dauern zwischen eineinhalb und dreieinhalb Stunden. Aus einer Vorauswahl von elf Sitzungen habe ich mich für eine nähere Untersuchung von zwei Sitzungen entschieden: CA5 und AA1.

Zu Beginn habe ich mir beide Sitzungen vollständig angesehen, um einen Eindruck des Ablaufs und der einzelnen Abschnitte zu bekommen. Anschließend habe ich die Sitzungen seperat nach interessanten Phänomenen für das gestellte Problem untersucht. Das Augenmerk war das Finden von Konzepten, welche die erzielten Fortschritte bei der Bewältigung der Aufgabe beeinflussen. Den umfangreiche Kodierungsprozess (Open Coding) auf der Grundlage von [3] habe ich mehrmals wiederholt. Die für mein Dafürhalten interessanten Phänome habe ich auf der Basis der getätigten Äußerungen Satz für Satz mit Hinblick auf die Illuktionen analysiert. Anschließend habe ich die gefundenen Konzepte im Prozess des Axial Codings in Beziehung zueinander gestellt. Mit dem Ziel Meta-Konzepte zu finden, welche relevant erscheinen und sich für eine Präsentation in dieser Arbeit eignen, habe ich mich auf selbige konzentriert und den Prozess des Open Codings wiederholt. Die Erkenntnisse präsentiere ich in folgenden Kapiteln.

1.4 Sitzungsbeschreibung CA5

Zwei Entwickler C3 und C4 implementieren eine neue Programmfunktionalität für ein grafisches Geo-Informationssystem. Die Aufgabe besteht im Teilen bereits vorhandender Geometrien (Polygone, Linien, Punkte) auf der Basis zu wählender Geometrien. Anfangs wird der Stand des Software-Systems diskutiert, um einen Einstiegspunkt zu finden. Anschließend wird die Strategie festgelegt durch Unterteilung der Gesamtaufgabe in die Teilaufgaben: Erstellung eines Menüpunktes „Teilen“, Erstellen des Teil-Polygons und Umsetzen der Teilungsfunktionalität. Die Teilaufgaben beinhalten seperate Tests der Implementationsschritte. Die Sitzungsdauer beträgt 94 Minuten.

1.5 Sitzungsbeschreibung AA1

Die Entwickler A1 und A2 sind während einer PP-Sitzung vor die Aufgabe gestellt, Anpassungen an ein CMS (Content Management System) vorzunehmen. Die Aufgabe besteht in der Anpassung der Darstellung von Listen aus Linkzielen in Abhängigkeit ihres Status (nicht erreichbar, gelöscht, aktiv, etc.) Die Entwickler benötigen einen ausführlichem Recherche- und Findungsprozess und stoßen auf weitere Probleme im Code, welche sie versuchen zu beseitigen. Währenddessen werden Testszenarien durchgeführt. Die Sitzungsdauer beträgt 153 Minuten.

1.6 **Basis-Schicht**

Die Analysen der Sitzungen werden auf der Basis der Kommunikation der Paare durchgeführt. Ich habe jede Äußerung in Abschnitten interessanter Phänomene analysiert. Der Fokus bei der Analyse der Kommunkation liegt in den Illuktionen: „Was ist die Absicht der getätigten Äußerung?“ Die Grundlage für diese Analysen bereiten die Arbeiten von Lutz Prechelt und Stefan Salinger, insbesondere [2], [3]. In [3] wird die „Basis-Schicht“ mit der Absicht erklärt, sie in der Praxis einsetzen zu können.

Der Mittelpunkt der Untersuchungen ist die Abischt der getätigten Äußerungen der Partner. Diese beziehen sich auf das Produkt oder den Arbeitsprozess. Darüber hinaus existieren universale Konzepte als Kommunikationsabsicht.

Produktorientierte Äußerungen beziehen sich auf ein Design oder eine Anforderung an den Code. Gekoppelt mit Verben ist es sehr handlich möglich, Äußerungen mit der Absicht Code-Artifakte (design) oder Anforderungen an den Code (requirement) zu beschreiben, einzuordnen. Die Einordnung bzgl. des Basis-Schicht-Objektes „design“ erfolgt folgendermaßen:

- Wird bspw. ein Vorschlag für eine Änderung im Code durch einen der beiden Partner geäußert, handelt es sich um ein Konzept der Basis-Schicht: „propose_design“.
- Wird diesem Vorschlag anschließend zugestimmt, handelt es sich um: „agree_design“.
- Ein Vorschlag, welcher einen vorangegangenen Vorschlag aufgreift und ihn ergänzt oder erweitert, ohne ihn zu negieren, wird als „amend_design“ konzeptionalisiert.
- Wird eine Äußerung getätigt, welche einen vorigen Vorschlag nicht unterstützt und statt dessen eine Alternative bietet, so handelt es sich um: „challenge_design“.
- Eine, einen vorangegangen Vorschlag ablehnende Äußerung, ist ein „disagree_design.
- Ein Basis-Schicht-Konzept „ask_design“ verlangt nach einer Idee bzgl. der Struktur oder des Inhaltes des Codes.

Prozessorientierte Äußerungen beziehen sich auf den Arbeitsablauf. Das Basis-Schicht-Objekt „step“ konzeptionalisiert den nächsten folgenden, taktischen Arbeitschritt im Diskurs der Entwickler. Dies kann wie am o.g. Beispiel „design“ beschrieben, eine Frage, ein Vorschlag, eine Erweiterung u.s.w. sein. Weitere prozessorientierte Objekte sind Äußerungen bzgl. einer Lösungsstrategie („strategy“), des Fertigungszustandes eines Arbeitsschrittes („completion“) oder der Strategie („state“), als auch die Verschiebung eines Arbeitsschrittes auf einen späteren Zeitpunkt („todo“). In Kombination mit Teilen der bereits genannten Verben entsteht somit die Möglichkeit, Äußerungen über den Ablauf der Sitzung einzuordnen. Nicht jedes Verb kann mit jedem Objekt kombiniert werden.

Für ein besseres Verständnis der Basis-Schicht Konzepte für zwischenmenschliche Kommunikation (HHI) empfehle ich einen Blick auf die Tabellen in [4, S. 50-51]. Die Basis-Schicht Konzepte sind in folgenden Kapiteln relevant, um die Interatkion der PP-Paare zu verstehen und einordnen zu können.

1.7 **Format der Darstellung der Gesprächsverläufe**

Die Gesprächsverläufe in den folgenden, zu betrachtenden Beispielen folgen einem bestimmtem, von mir gewählten Format:

Entscheidungsschritt X. <Base_Layer_Konzept> Entwicklerbezeichner: „Äußerung“ <Kontextbeschreibung>

- Der Entscheidungsschritt ist eine Nummer, welche sich auf einen vorab beschriebenen Entscheidungsprozess bezieht.
- Das Basis-Schicht-Konzept stellt das von mir gewählte Konzept dar. (Vgl. [3, Kapitel 2 ff.]
- Der Entwicklerbezeichner ist die gewählte Referenz des Entwicklers, welcher folgend etwas verbal oder nonverbal ausdrückt.
- Die „Äußerung“ sind gesprochene Worte des Entwicklers
- Die in {} eingefasste Kontextbeschreibung beschreibt wichtige Details, die zu einem besserem Verständnis der Situation beitragen sollen.

Die genaue Analyse der Kommunikation, insbesondere die Absicht der Äußerungen, sind abgesehen von Beispiel im Anhang zu finden. Es ist für ein besseres Verständnis hilfreich, diese zu lesen. In Auszügen werde ich u.U. einzelne Entscheidungsprozesse in Beispielen exemplarisch in den Beispielen erläutern.

Es ist wichtig zu verstehen, dass oft ohne eine Reaktion auf einen Vorschlag des Partners Zustimmung signalisiert wird. Dies ist oft nicht offensichtlich verständlich. Wird durch einen Entwickler ein Vorschlag geäußert und umgesetzt und der andere Partner ist sichtlich konzentriert bei der Aufgabe und antwortet nicht, stimmt er dem Vorschlag nonverbal zu. Der Partner hätte in diesem Moment die Möglichkeit eine Alternative zu bieten, den Vorschlag zu hinterfragen oder viele weitere Optionen zu reagieren. Bleibt dies aus kann davon ausgegangen werden, dass ein Zustimmung erfolgt.

2 Analyse von Fokus-Phasen

Im folgendem Kapitel möchte ich Fokus-Phasen und analysierte Bedingungen näher beschreiben. Was sind Fokus-Phasen und welche Faktoren können selbige beeinlussen?

In Kapitel 1.2. habe ich erwähnt, dass ein wesentlicher Faktor innerhalb einer PP-Sitzung das Aufbauen und Aufrechterhalten von Togetherness ist, damit die Entwickler tatsächlich „mit einem Verstand“ ein Problem bearbeiten können. Togetherness ist ein zentrales Konzept bei der Analyse von PP. Arbeiten die beiden Entwickler nicht vereint an einer Aufgabe und sind somit gezwungen lange Diskussionen darüber zu führen, ob oder wie ein Teilproblem gelöst wird, werden entsprechend weniger Fortschritte in einer bestimmten Zeit erzielt werden können. Ein Mehraufwand an Wissenstransfer oder Diskussionen entsteht, um Entscheidungen treffen zu können.

Notwendige oder hilfreiche „PP-Fertigkeiten“ von Programmierern tragen sicher dazu bei, um den Aufwand beim Prozess des Erreichen und Erhaltens von Togetherness zu reduzieren. Diese lassen sich allerdings nur sehr schwer klar definieren oder umreissen. Vgl. [4, S.1] Bestimmte Fertigkeiten der Partner tragen dazu bei, dass ein Paar vereint und effizient arbeitet und andere Paare dazu weniger in der Lage sind Vgl. Kapitel 3.5. Diese Unterschiede sind in Sitzungen zu beobachten. Jedoch ist die Menge der Faktoren zu gross und zu komplex, um sie einfach aufzuzählen und über ihre Ursachen und Bedeutungen zu reden.

Togetherness von Paaren ist ein schwer zu fassendes oder zu beschreibendes Konzept. Der Begriff ist nicht trivial zu definieren. Es obliegt des Beobachters Wahrnehmung, Erfahrung und Bewertung eine Einschätzung geben zu können, inwiefern Paare synchronisiert und vereint arbeiten oder Unverständnis, Missverständnisse oder Seperation selbigen beeinflussen. Menschliche Interaktionen und Motivation des Handeln sind zu komplex, um sie trivial oder skalierbar zu definieren.

Eine Auswirkung von Togetherness kann man in einem leicht zu beobachtendem Konzpet festhalten: dem **Arbeitsfluss**. Die Flüssigkeit des Arbeitsprozesses und die Geschwindigkeit, in der Fortschritte während einer PP-Sitzung auftreten, lässt sich leicht wahrnehmen. Mit Hilfe des Arbeitsflusses werde ich folgend Togetherness von PP-Paaren charakterisieren.

2.1 Arbeitsfluss

Mit dem Begriff Arbeitsfluss wird beschrieben, wie schnell und flüssig Fortschritte im Arbeitsprozess erzielt werden.

Es werden drei Kategorien des Arbeitsflusses unterschieden (Vgl [4, S. 1, I.B.]):

- PP_{normal} beschreibt einen normalen Arbeitsfluss. Die Geschwindigkeit kann sich zwischenzeitlich ändern, wobei insgesamt Fortschritte erzielt werden.
- PP_{fast} beschriebt einen ungewöhnlich und extrem fließenden Arbeitsfluss mit sehr schnellem Fortschritt.
- $PP_{breakdown}$ zeichnet sich durch einen zähen Arbeitsprozess und/oder sehr uneffiziente Arbeitsweise aus.

Mittels qualitativer Datenanalyse der untersuchten PP-Sitzungen soll verdeutlicht werden, wie Togetherness der Partner dazu beiträgt, einen flüssigen Arbeitsfluss zu erlangen. In den folgenden Abschnitten werde ich den Zusammenhang des Arbeitsflusses und PP-Fertigkeiten beschreiben.

2.2 Beispiel für PP_{fast} als Teil einer Fokus-Phase

In diesem Abschnitt möchte an einem konkretem, analysiertem Beispiel PP_{fast} besser beschreiben, um einen Eindruck zu vermittlen, was man sich konkret darunter vorstellen kann. Das gewählte Beispiel soll zeigen, wie ein schneller Arbeitsfluss vorstellbar ist und inwiefern sich PP_{fast} beobachten und einordnen lässt.

Eingehenden Abschnitten von Orientierungs-Phasen über den Stand des Software-Systems, Besprechung und Einigung über das strategische Vorgehen zur Lösung der gestellten Aufgabe in PP-Sitzungen folgend, kann ein Paar in einen Zustand fokussiertem Abarbeitens eintreten. Die Partner verschaffen sich vorab Klarheit über das Problem und entwickeln eine Lösungsstrategie, welche von beiden detailliert verstanden wird. Die Betrachtung der Arbeit der beiden Entwickler in diesen Abschnitten der PP-Sitzung macht tatsächtatsächlich den Eindruck, als würden sie „mit einem Verstand“ das Problem lösen. Sie ernten somit die Früchte der vorab aufgebauten Togetherness durch eines vollumfänglichen Bewusstseins darüber, was getan werden muss, welche Voraussetzungen dafür erfüllt sein müssen und wie taktisch vorgegangenen wird. Dieser Prozess erfolgt, wie in 2.1. beschrieben, in PP_{fast}.

Der Abschnitt der PP-Sitzung fokussiertem Arbeitens in PP_{fast} wird folgend als „Fokus-Phase“ bezeichnet.

Für ein besseres Verständnis von PP_{fast} möchte ich diesen Zustand des Arbeitsfluss im folgenden *Beispiel 1* näher betrachten. Das Beispiel ist ein Ausschnitt einer Fokus-Phase der Sitzung CA5.

Beispiel 1, Sitzung CA5, Teil I, 20:32 – 21:39:

Situation:
Die Aufgabe der PP-Sitzung besteht in der Implementierung einer Klasse, welche ein Polygon erstellt, mittels dessen Geometrie ein vorhandenes Polygon geteilt werden soll. Nachdem sich die Entwickler über die Natur der Aufgabe und den Stand der Entwicklung des Software-Systems verständigt haben, einigen sie sich über die Vererbungshierarchie der zu erstellenden Klasse und entschließen sich, bereits bestehenden Code mittels „Kopieren und Einfügen“ zu verwenden und anzupassen. Die Teilaufgabe besteht in der Implementierung einer Klasse, welche die Erstellung des Teilungs-Polygons realisieren soll.

Verlauf:
Innerhalb von einer Minute werden folgende Entscheidungen (teilweise ist der Entscheidungsweg beinhaltet) durch die beiden Entwickler getroffen:

1. Der Rückgabewert „false“ in der Return Anweisung der bearbeiteten Methode soll gelöscht werden.
2. Exception-Anweisungen sollen vorerst nicht gelöscht werden. C4 hinterfragt die Existenzberechtigung dieser Exception.
3. Die Programm-Bibliothek „GeometryFactory“ kann benutzt werden.
4. Die Funktionalität für den „EditorLog“ ist nicht vorhandenen. Es folgt ein kurzer Wissenstransfer mit schnellem Konsens, dass die Methode „abstrakt“ war und im Anschluss durch einbinden der bereits vorhandenen Funktionalität implementiert und an die zu bearbeitende Methode übergeben wird.
5. Der vorhandene Code für den „EditDialogFactory“ wird benötigt und bleibt unverändert.
6. Der Code für die „Display“-Funktionalität wird unverändert übernommen.
7. Der Code für die „DisplayToolbar“ wird unverändert übernommen.
8. Der Code für die „configureEditData“ Funktionalität wird gelöscht. Nach kurzem Wissentransfer und Evaluierung mittels Nachschlagen in der Programm-Biobliothek einigen sich die Entwickler.
9. Der Code für die „EditToolbar“ wird benötigt und unverändert übernommen.
10. Eine Code-Zeile für den Code des „DisplayTool“ wird benötigt und unverändert übernommen.
11. Die Einbindung der „DisplayToolbar“-Funktionalität wird benötigt und unverändert übernommen.
12. Mit abschließendem Einverständnis der Vollständigkeit stimmen beide Entwickler zu, den vorab in Schritt 4 verschobenen taktischen Schritt wieder aufzunehmen.

Diese zwölf Entscheidungen erfolgen in einer extrem hohem Geschwindigkeit innerhalb von 67 Sekunden. Der Arbeitsfluss ist erstaunlich flüssig und der Kommunikationsverlauf effizient und

kompakt. Das Verständnis beider Paartner füreinander ist sehr hoch und somit kann ohne viele Worte zu benutzen, eine schnelle Abarbeitung der Änderung des Codes und somit eine Vielzahl von Entscheidungen in kurzer Zeit getroffen werden.

Der Gesprächsverlauf der Entscheidungsprozesse 5-11 ist ein Auszug von insgesamt 30 Sekunden:

5. <propose_design> C3: {Cursor via Keyboard navigiert zu beschriebener Stelle im Code, markiert die Zeile kurz}
   „Ähm... EditDialogFactory. Okay, ist auch klar. Bleibt wie es ist, würde ich sagen."

   <agree_design> C4: {stimmt zu, keine Reaktion, Blick auf Bildschirm}

6. <propose_design> C3: {Cursor via Keyboard navigiert zu beschriebener Stelle im Code, markiert die Zeile kurz}
   „display,..."

   <agree_design> C4: {keine Reaktion, Blick auf Bildschirm}

7. <propose_design> C3: {Cursor via Keyboard navigiert zu beschriebener Stelle im Code, markiert die Zeile kurz}
   ... „displayToolbar brauchen wir halt."

   <agree_design> C4: {stimmt zu, keine Reaktion, Blick auf Bildschirm}

8. <propose_step> C3: {Cursor navigiert via Tastatur zu beschriebener Stelle im Code, markiert die Zeile kurz}
   „configureEditData"

   <propose_hypothesis> C4: „Da macht er im Menü nichts, soweit ich weiss..."
   {Kopf schüttelnd, Kinn auf rechte Hand gestützt, Mund leicht verdeckt}
   „ ...bei der newAction."

   <agree_hypothesis> C3: „Mhmm. Denk ich auch."

   <explain_finding> C4: „Kann ich mal?"
   {rechte Hand geht zur Maus, zeitglich nimmt C3 rechte Hand von seiner Maus weg, C4 übernimmt fließend Eingabekontrolle, C4 wechselt Tab in Eclipse entsprechende Klasse enthaltend, navigiert zu der Methode und markiert sie kurz}
   „Da. Nichts."

   <challenge_finding> C3: {rechte Hand geht zur Maus, C3 übernimmt Eingabekontrolle, beide haben Eingabekontrolle, C4 beobachtend inaktiv, C3 linke Hand deutet auf den Bildschirm}
   „Moment!"
   <agree_finding> {C3 wechselt Tab in Eclipse entsprechende Klasse enthaltend, navigiert zu der Methode und markiert sie kurz}
   „FeatureNewAction. Hier. Guck mal da... Ach ne. Ne."

   <amend_finding> C4: „Der andere ist der Performer."
   {rechte Hand bewegt sich unter das Kinn weg von der Maus}

   <agree_finding> C3: „Das ist der Performer. Ok." {beide Hände bewegen sich zum Keyboard, C3 wechselt den Tab zurück, die zu bearbeitende Klasse enthaltend}

<propose_design> C4: „Kann weg.“

<agree_design> C3: „Okay, dann den weg.“
{löscht entsprechend den Code}

9. <propose_design> C3: {navigiert Cursor linear durch den Code und kommentiert entsprechende Funktionalität}
„EditTool already attached...“

<agree_design> C4: {zeitgleich zu C3}
“EditToolbar brauchen wir.“

10. <propose_design> C3: {navigiert Cursor über die folgenden beiden Code-Zeilen, kurze, sehr schnelle Sprünge mit Markierungen von Methoden}
„DisplayTool, ...“

<agree_design> C4: {stimmt zu im Sinne, dass kein Löschen erfolgt; keine Reaktion}

11. <propose_design> C3: „...DisplayToolbar“

<agree_design> C4: {Kopf schüttelnd im Kontext "nicht löschen", den schnellen Markierungssprüngen im Code durch Eingabe von C3 folgend}
„...brauchen wir, brauchen wir, brauchen wir...“

<agree_design> C3: {zeitgleich zu C4}
„Wunderbar.“

Auch ohne eine nähere Betrachtung der Kodierung fällt auf, dass ein extrem schneller Arbeitsfluss vorhanden ist. Der produktive Prozess ist sehr effizient. Beiden Entwicklern ist bewusst, wie der wiederverwertete Code funktioniert, was die Anforderungen an den zu erstellenden Code sind und wie der duplizierte Code zu verändern ist. Es sind, neben einem kurzem Abschnitt des Wissenstransfers, keine Diskussionen darüber notwendig, was getan werden soll. Auf dieser Basis entsteht eine schnelle Abfolge von Entscheidungen, ob der wiederverwertete Code übernommen, angepasst oder gelöscht werden muss.

Mit Hilfe der Betrachtung der Basis-Schicht-Konzepte lässt sich der Zustand PP_{fast} abbilden. Abgesehen von Entscheidungsprozess 8, in dem ein kurzer Wissenstransfer stattfindet, ist in allen weiteren Entscheidungen ein Schema im Kommunikationsverlauf zu beobachten:

1. Entwickler C3 referenziert kurz durch Markierung des Artifaktes oder eine verbale Bennenung auf eine Codezeile. Hiermit wird deutlich gemacht, welcher der nächste taktische, zu bearbeitende Arbeitsschritt ist. Dies erfolgt zum überwiegenden Teil durch eine kurze Markierung der Stelle im Code mittels der Maus oder Tastatur.
2. Zeitgleich äußert C3 einen Vorschlag, wie mit dem markiertem Code verfahren wird. Diese Äußerungen sind mit dem Basis-Schicht-Konzept <propose_design> kodiert.
3. Direkt anschließend stimmt Paartner C4 dem Vorschlag von C3 zu. Auch wenn C4 keine Zustimmung verbal äußert, kommt das Basis-Schicht-Konzept <agree_design> zur Anwendung. C4 hätte im entsprechenden Moment die Möglichkeit Alternativen zu bieten, Ablehnung zum erbrachten Vorschlag zu signalisieren oder den Vorschlag des Partners mit

einem eigenen Vorschlag zu erweitern. Da diese Äußerungen ausbleiben und C4 konzentriert die Änderungen durch C3 beobachtet, wird eine nonverbale Zustimmung signalisiert.

Die Entscheidungen 5, 6, 7, 9, 10 und 11 folgen dem beschriebenem Schema. Innerhalb von Sekunden findet die Referenzierung der folgenden wichtigen Codezeile, der Unterbreitung eines Implementierungsvorschlages und der sofortig folgenden Zustimmung durch den Paartner statt. Dieser Vorgang wird anschließend direkt wiederholt und somit werden durch einen flüssigen Verlauf im Arbeitsprozess schnelle Fortschritte erzielt.

Beide Entwickler sind sich darüber einig, was bearbeitet wird, wie die Bearbeitung geschehen soll und welches Ergebnis erwartet wird. Es entstehen zu allen drei Aspekten keine Diskussionen aufgrund von Wissenslücken oder Unverständnis mindestens einer der beiden Entwickler, noch aufgrund von Meinungsverschiedenheiten, Sichtweisen oder alternativen Vorschlägen. Diese schnelle Abfolge (<propose_design> folgend <agree_design>) ist charakteristisch für den Zustand PP_{fast}.
Die beschriebene Fokus-Phase entsteht jedoch keineswegs von allein. Die von mir analysierten Voraussetzungen für Fokus-Phasen werden in Kapitel 3 beschrieben.

2.3 Charakterisierung von Fokus-Phasen

Ein überdurchschnittlich flüssiger Prozessverlauf in PP mit schnellen Fortschritten, welcher PP_{fast} und folgend Fokus-Phasen charakterisiert, können durch eine schnelle Abfolge bestimmter Äußerungen beobachtet werden. Einerseits in Hinblick auf produktorientierte und andererseits auf prozessorientierte Kommunikation. Die Abfolgen sind nicht immer gleich, folgen jedoch einem bestimmtem Schema.

Zwei Konzepte treten wiederholt während des „Open Coding“ Prozesses in meinen Analysen auf und sind charakterisierend für Fokus-Phasen. Während eines längeren Abschnittes in PP_{fast} in einer Fokus-Phase können sowohl Implementationsschritte, als auch strategische oder taktische Überlegungen geschehen. Im ersten Fall handelt es sich um produktorientiere Codes und im zweiten um prozessorientierte Codes. Beide können die Kriterien von PP_{fast} erfüllen und somit Bestandteil einer Fokus-Phase sein.

Ein Schema, wiederholt auftretender, produktorientierter Phänomene habe ich im Code „propose.design.fluency“ zusammengefasst. Bestimmte prozessorienterte Phänomene habe ich mit dem Code „propose.step.fluency“ versehen. Beide Konzepte treten zusammen und einzeln auf. Es können also u.U. flüssige Entscheidungsverläufe in Hinsicht auf Code-Artifakte und gleichzeitig schnelle Abflolgen von Vorgaben taktischer Arbeitsschritte beobachtet werden. Im Folgenden möchte ich beide näher erläutern, um ein besseres Verständnis von Fokus-Phasen zu geben.

2.3.1 **Analyse des Arbeitsflusses in produktorientierter Kommunikation**

Beispiel 1 ist eine Beschreibung des Konzeptes flüssiger Design-Entscheidungen. Mit *Beispiel 2* möchte ich im Folgenden dieses Konzept näher beleuchten und für jede Entscheidung erklären.

Beispiel 2, Sitzung CA5, Teil I, 19:08 – 19:52:

Situation:
Die Teilaufgabe ist die ähnlich angelegt wie in *Beispiel 1*. Die Arbeitsschritte beinhalten die Implementierung einer Klasse, welche ein Polygon erstellt, welches ein vorhandenes Polygon teilt. Auch hier wird die Strategie verfolgt, implementierten Code zu kopieren und einzufügen, um ihn folgend anzupassen. Beide Entwickler haben sich auf den zu kopierenden Code einer vorhandenen Klasse geeinigt und die Vererbungsstrategie für die zu erstellende Klasse festgelegt. Folgend erfolgt eine erste Betrachtung und der erste Prozess der Anpassung des Codes an die Bedürfnisse der Funktionalität der zu erstellenden Klasse.

Verlauf:
Innerhalb von 44 Sekunden werden folgende Entscheidungen durch die beiden Entwickler getroffen:

1. Return Rückgabewert der Methode wird gelöscht.
2. Return Rückgabewert einer Funktion wird gelöscht.
3. Mehrere Codezeilen einer Validierungs-Subfunktion werden gelöscht.
4. Ein Todo wird vereinbart, die in Schritt 3 gelöschte Funktion evtl. später wieder zu implementieren.
5. Das snapThemeModel wird benötigt und unverändert belassen.
6. Skalierungsfunktionalität für den FeatureLayer wird benötigt unverändert belassen.
7. Die Funktionalität, die EditGeometry mittels einer Methode getEditStrategy zu berechnen, ist falsch.
8. Die EditGeometry ist immer ein Polygon und wird durch den GeometryType ermittelt.

Kommunikationsverlauf:

1. <propose_design> C3: „Okay, da gibts schon mal jede Menge Sachen die wir nicht brauchen oder nicht haben.“
   {C3 löscht return Zeile der kopierten Methode während des „Browsens“ durch den Code}

   <agree_design> C4: {kurzes Nicken mit dem Kopf}

2. <propose_design> C4: „Return... Da kannst du ja anfangen.“
   {C4 zeigt auf return Zeile}

   <agree_design> C3: {C3 löscht Rückgabewert entsprechend C4´s Vorschlag}

3. <propose_design> C4: „Wobei wir das validator nicht brauchen.“
   {C4 meint die Message-Funktionalität den „validator“, gemeint ist die komplette Funktion zu löschen und nicht nur den Rückgabewert}

   <amend_design> C3: „Wahrscheinlich wird das wieder... je nachdem.. validator...“

4. <propose_todo> C4: „Vielleicht brauchen wir es, aber das kannst du es immer noch wieder holen."

   <agree_design> C3: „Ja, würde ich wegmachen."
   {löscht Code entsprechend Vorschlag}

5. <propose_design> C3: "Okay, SnapThemeModel... Das brauchen wir."

   <agree_design> C4: „Das brauchen wir."

6. <propose_design> C3: "FeatureLayerBaseScale... Das brauchen wir eigentlich auch, weil wir das spliten.. „
   {rechte Hand zeigt auf eine Stelle auf dem Screen}
   „...machen wir ja eine neue Linie..."
   {Hände illustrieren den verbal beschriebenen geometrischen Vorgang, Blick wandert mehrmals vom Screen zum Partner und zurück}
   „...automatisch ins Polygon rein und das sollte genaue Passgenauigkeit haben."

   <agree_design> C4: {Blick zum Partner, Blick abwendend und Kopf nickend, Hände spielen in Haaren, zustimmend}
   „Mh-hmm, wir brauchen schon den FeatureLayer den wir editieren wollen."
   {Körperhaltung geht nach vorn, Blick auf den Screen gerichtet}

   <agree_design> C3: „Genau."

7. <propose_design> C3: {bewegt Maus-Cursor zu entsprechenden Methoden, markiert EditGeometryType}
   „getEditStrategy, EditGeometryType: Das ist falsch!"

8. <amend_design> C4: {löst Körperhaltung auf, Hände illustrieren Worte}
   „Das ist auf jeden Fall... Polygon, genau."

   <agree_design> C3: „Hier nehmen wir immer..."
   {zeitgleich zu C4, editiert den Code entsprechend des formulierten Konsens}

In *Beispiel 2* und *Beispiel 1* handelt es sich um produktorientierte Kommunikation. Die Äußerungen beziehen sich inhaltlich auf Code-Artefakte. Das Basis-Schicht-Objekt für die unterbreiteten Vorschläge ist „design". Vgl. [3, S. 59 ff., Kapitel 4, 5]

Bei der Betrachtung der Kommunikation in *Beispiel 2* wird das vorab in Kapitel 2.2. beschriebene Schema erfüllt, in dem ein Vorschlag unterbreitet und folgend direkt durch den Partner bestätigt wird.

Analyse der Entscheidungsprozesse (Schritt 1 ist hier exemplarisch beihnaltet, alle weiteren Schritte sind im Anhang zu finden):

1.
In Entscheidung 1 sagt C3 zu Beginn des *Beispiels 2*: „Okay, da gibts schon mal jede Menge Sachen die wir nicht brauchen oder nicht haben." C3 erklärt hiermit implizit, wie die folgenden, kommenden, taktischen Schritte durchgeführt werden sollen. C4 versteht diesen Vorschlag und nimmt ihn an. Dies zeigt sich insbesondere durch ihr Verhalten in Kombination mit dem Ausbleiben einer Antwort.

Nichts desto trotz habe ich die Illuktion der Aussage aus zwei Gründen mit „propose_design“ kodiert. Erstens wiederholt C3 hiermit nur, was vorher bereits als Strategie festgelegt wurde. Zweitens löscht C3 gleichzeitig die Return Anweisung der bearbeiteten Methode. Der engere Fokus liegt in der produktorientierten Handlung. C4 bestätigt den Vorschlag durch kurzes, gutierendes Nicken mit dem Kopf.

Schlussfolgerungen:

Die Synchronisation der beiden Partner funktioniert in *Beispiel 2* gut. Die vorhergehende Verabredung darüber, wie mit Code-Artifakten umzugehen ist, erübrigt jede weitere Diskussion darüber. Das Paar wird somit in die Lage versetzt, gemeinsam Ideen zu äußern, ohne dass es Mißverständnisse geben könnte, was gemeint ist. Die Togetherness der beiden Entwickler ist zu diesem Zeitpunkt sehr hoch.

Darüber hinaus ist Sprache für den konkreten Arbeitsverlauf nicht vorrangig entscheidend. Sie dient hauptsächlich dem Zweck die Togetherness aufrecht zu erhalten. Von einer kurzen Unsicherheit, welche zu einer schnellen Design-Entscheidung in Punkt 6 führt, abgesehen, arbeiten die beiden Partner mit einem Verstand lediglich „rezeptartig“ ab.
Im einfachsten Fall folgt auf ein propose_design direkt ein agree_design, wie in den Entscheidungen 1, 2, 5 und 6 sehr gut zu beobachten ist. Dieser Umstand befördert einen überdurchschnittlich fließenden Arbeitsprozess mit schnellen Fortschritten. Die Abwesenheit der Notwendigkeit von Erklärungen der zu vollziehenden Arbeitsschritte zeigt ein hohes Maß an Synchronisation der beiden Partner. Es wird nicht diskutiert, was oder wie die nächste taktische Schritte sind.

In *Beispiel 2* ist in Entscheidung 7 und 8 zu beobachten, dass ein Partner mit einer Idee antwortet, welche den vorab unterbreiteten Vorschlag erweitert (amend_design). Es entsteht einem Vorschlag folgend keine Diskussion oder Uneinigkeit. Darüber hinaus besteht zwischen C3 und C4 keine Notwendigkeit eines Wissenstranfers. Falls ein Wissenstransfer stattfindet, so ist er kurz, effizient und wird sehr schnell bestätigt.
Im Falle eines erweiterten Design-Vorschlages ist es wichtig zwischen zwei Arten zu unterscheiden: dem vorab (Kapitel 1.5.) erwähnten amend_design und einem challenge_design.

- Ersterer negiert die Idee des eingangs unterbreiteten Vorschlages (propose_design) nicht. Die Idee wird aufgegriffen und erweitert. Wie erwähnt ist dies in *Beispiel 2* in Punkt 7 und 8 zu beobachten.
- Im Falle eines Vorschlages, der die vorab unterbreitete Idee bzgl. eines Code-Artifaktes nicht unterstützt und statt dessen eine Alternative bietet, handelt es sich um ein challenge_design.

In *Beispiel 2* sind bestimmte Äußerungen nicht zu beobachten. Äußerungen kodiert mit Basis-Schicht-Konzept <challenge_design> oder <disagree_design> werden u.U. die Phase von PP_{fast} beenden. Es kann vorkommen, dass aufgrund der Uneinigkeit der Entwickler anschließend eine Diskussion mit beinhaltetendem Wissentransfer stattfindet. Der Arbeitsfluss und die Togetherness und Synchronisation der Partner im laufenden Arbeitsprozess wird hierdurch deutlich reduziert. Falls einem challenge_design nach kurzer Zeit ein agree_design (oder decide_design) des Partners folgt, signalisiert dies eine schnelle Einigung über das Problem und eine Entscheidung. Die Voraussetzung ist eine hohe Effizienz im Wissenstransfer und schneller Konsens. Somit kann der hohe Arbeitsfluss aufrecht erhalten werden und der Zustand PP_{fast} wird nicht verlassen.

Zusammenfassung:
Durch die in *Beispiel 2* illustrierte, schnelle Abfolge von Vorschlägen und direkt folgenden Zustimmungen wird ein hoher Arbeitsfluss charakterisiert. Es können somit viele Entscheidungen in

sehr kurzen Zeitabschnitten getroffen werden. Diese schnelle Abfolge ist charakterisierend für PP_{fast}. Mit Hilfe dessen können Fokus-Phasen erkannt werden. Die Abwesenheit von langen Abschnitten des Wissenstransfers und Diskussionen trägt entscheidend zu PP_{fast} bei.

2.3.2 **Analyse des Arbeitsflusses in prozessorientierter Kommunikation**

Nachdem ein Implementierungsvorgang von verschiedenen Code-Artifakten abgeschlossen ist, kann sich dem direkt ein weiterer anschließen. Dies setzt eine sehr gute, vorab verabredete Konzeption der PP-Sitzung voraus.
In Sitzung CA5 ist zu beobachten, dass nach einem abgeschlossenem Abschnitt von Code-Änderungen, Implementierungen oder Tests der Implementierung eine Phase stattfindet, in der die Strategie für folgende Arbeitsschritte und das taktische Vorgehen besprochen werden. Eine Fokus-Phase muss durch diesen Übergang nicht unterbrochen werden.
Die Aufgabe einer PP-Sitzung kann ein Entwurf einer Software sein, ohne dass ein Code-Artifakt implementiert wird. Diesen Fall habe ich nicht analysiert, jedoch wurde mir in Gesprächen mit Forschern davon berichtet.

Im vorangegangenen Abschnitt 2.3.1. habe ich Merkmale produktorientierter Kommunikation besprochen. Ähnlich zu selbigen existieren Charakteristika in einer PP-Sitzung oder Teilen davon in Bezug auf prozessorientierte Kommunikation.

Insbesondere gehören dazu Äußerungen mit Bezug auf Vgl. [3, S. 73 ff, Kapitel 6, 7, 8, 9, 10]:

- Strategien: wie wird das Problem oder ein Teilproblem im Ganzen gelöst
- Taktiken: wie gestaltet sich der nächste folgende Schritt, was ist zu tun
- Status: in weit wurde der Arbeitsplan für die Sitzung erfüllt
- Vollständigkeit: in weit fern ist die besprochene Taktik vollständig abgearbeitet
- Todo: Verschiebung eines Arbeitsschrittes auf einen späteren Zeitpunkt

Im Basis-Schicht-Model existieren weitere universelle, Konzept-Objekte, welche ich in Teilen hier mit einbeziehen möchte. Sie unterliegen in der Betrachtung des Arbeitsflusses den gleichen Kriterien. In den analysierten Sitzungen vermischen sich Basis-Schicht-Objekte prozessorientierter Kommunikation oft mit gewissen universellen Konzept-Objekten. Hierzu gehören insbesondere Vgl. [3, S 109 ff, Kapitel 11, 12, 13, 14, 15, 16]:

- Wissen: ein Entwickler gibt seinem Partner Informationen über deklaratives Wissen Vgl. [3, S. 151 ff, Kapitel 16]
- Wissenslücken: ein Entwickler weist seinen Partner auf eine Wissenslücke hin Vgl. [3, S. 147 ff, Kapitel 15]
- Hypothesen: eine Vermutung wird ausgeprochen Vgl. [3, S. 131 ff, Kapitel 13]
- Erkenntnis: Äußerung auf eine Beobachtung oder eine Erkenntnis bezogen Vgl. [3, S. 113 ff, Kapitel 12]

Darüber hinaus existieren weitere universelle Konzept-Objekte, die speziell und für die folgenden Beispiele nicht streng relevant sind und daher hier vernachlässigt werden können.

In Phasen der Richtungsfindung und Besprechung über das weitere Vorgehen findet oft ein Recherche-Prozess statt. Die ideale Situation, dass beide Partner ein vollständig abgeschlossenen Überblick über das zur Lösung notwendige Wissen besitzen, ist unwahrscheinlich, wenn auch möglich. Ich habe es im Material nicht beobachten können. Dem zur Folge ist in Phasen, in denen die Prozeduren für das weitere Vorgehen besprochen werden, ein Wissenstransfer und Evaluierungen über den Wissenstand beinhaltet. Insbesondere gilt dies für den Fall: wenn Code bearbeitet, getestet oder auf diesem aufgebaut werden soll, den die Entwickler nicht selbst geschrieben haben. Im Folgenden möchte ich illustrieren, wie in diesen Situationen Fokus-Phasen charakterisiert werden können.

Beispiel 3, Sitzung CA5, Video Teil II, 0:00 – 00:37:

Situation:
Die Erstellung der Geometrie für die Erstellung eines Polygons zum Teilen eines bestehenden Polygons wurde implementiert. Im Anschluss daran erfolgte eine Test-Phase. Das Programm wurde gestartet und die Erstellungsfunktionalität und einige Ausnahmefälle wurden durch die Entwickler getestet. Der Arbeitsfluss ging während der Test-phase deutlich zurück. Entsprechend folgen nun Überlegungen, wie fortgefahren werden soll. Die Vereintheit der Entwickler für weitere, folgende Implementierungsvorgänge muss wieder aufgebaut werden. Die Entwickler entscheiden sich wiederholt für eine Lösungstrategie, in der bestehender Code kopiert, eingefügt und den Bedürfnissen angepasst werden soll.

Verlauf:
Innerhalb von 37 Sekunden werden folgende Entscheidungen durch die beiden Entwickler getroffen:

1. Auf der Basis der Wiederverwendung des bestehendehenden Codes der getEditStrategy wird die umzusetzende Funktionalität realisiert.
2. Der zu kopierende Code befindet sich in der Funktion applyGeometryChange.
3. Die zu übergebende Geometrie muss ermittelt werden.
4. Verabredung über die strategische Einordnung der folgenden Schritte, wegschneiden der bestehenden Geometrie.

Kommunikationsverlauf:

1. <propose_step> C4: „Okay. Jetzt gibt es irgendwo diese getEditStrategy..."
   {fügt Code entsprechend der Äußerung ein und betrachtet Tooltip anschließend für Vorschläge}

   <agree_step> C3: {keine Reaktion mit Blick auf Screen}

2. <propose_step> C4: „...und da gibt es einen con..."
   {tippt und löscht Eingabe wieder}
   „...applyGeometryChange..."

   <agree_step> C3: <kurzes Nicken mit dem Kopf>

3. <propose_strategy> C4: „...und die Geometry müssen wir ermitteln."

<agree_strategy> C3: “Mh-hmm.”

4. <propose_strategy> C3: „Was wir jetzt machen... also was wir jetzt machen könnten, wäre aus dem bestehendem Objekt den Teil wegschneiden, der dann in ein neues kommt.“

   <agree_strategy> C4: „Ja genau. Das ist die Aufgabe.“

   <explain_state> C3: „Na es ist ein Teil der Aufgabe.“

   <agree_state> C4: „Das ist jetzt für diesen Task die Aufgabe.“

   <agree_strategy> C3: „Okay. Also wegschneiden, gut dann...“

Beispiel 3 beinhaltet prozessorientierte Kommunikation. Der Gesprächsinhalt bezieht sich im Gegensatz zu *Beispiel 1* und *Beispiel 2* nicht auf konkrete Code-Artifakte. Die Absicht dieses Gespräches ist die Richtungsfindung, wie im weiteren vorgegangen werden soll. Währenddessen findet ein Wissenstransfer statt. Während der Strategiebesprechung für den nächsten Abschnitt der PP-Sitzung wird paralell in der Bibliothek recherchiert, wie die Umsetzung der Lösung für die nächste Teilaufgabe möglich ist. Die Ideenfindung findet auf der Basis der Betrachtung von bestehendem Code statt.

Zusammenfassung:

Ähnlich wie die im Abschnitt 2.3.1. beschriebenen Merkmale für einen extrem flüssigen Verlauf des Arbeitsprozesses mit schnellen Forschritten, existieren diese paralell für prozessorientierte Kommunkation. In *Beispiel 3* fällt ebenso wie in den Beschreibungen von *Beispiel 2* auf, dass die Konzept-Verben „propose“ und „agree“ aufeinander folgen. Diese schnelle Abfolge von verbalisierten Vorschlägen und direkt folgenden Zustimmungen charakterisiert PP_{fast} ebensfalls in Abschnitten einer PP-Sitzung, in denen vorangig prozessorientierte Kommunikation stattfindet. Es wird sehr schnell eine Lösungstrategie gefunden, verbalisiert, vom Partner verstanden und anschließend eine Entscheidung getroffen. Es findet in diesem Beispiel kein Wissenstransfer aufgrund von Wissenlücken eines der beiden Entwickler statt.
Ein sehr hohes Maß an Wissen über bereits implementierten Code, effizientem Umgang mit der Entwicklungsumgebung und Verständnis der Anforderungen an die gestellte Aufgabe voraussetzend ist es den Entwicklern möglich, innerhalb von 37 Sekunden eine vollständige Erarbeitung der Strategie für den kommenden Implementierungsvorgang festzulegen. Diese Voraussetzungen ermöglichen den Zustand des Arbeitsflusses PP_{fast} und werden in Kapitel 3 konkreter analysiert.

Keiner der beiden Entwickler äußert während des Gespräches eine eigene Wissenslücke oder die des Partners. Dies würde die Folge von propose und agree unterbrechen und den Arbeitsfluss reduzieren. An Stelle eines „agree“ würde ein universelles Basis-Schicht-Konzept rücken, bspw. „explain_gap_in_knowledge“, „explain_knowledge“ oder „propose_hypothesis“, etc. Anschließend müssten Wissenlücken ausgeräumt oder Alternativen besprochen werden. Es würde eine Phase des Wissenstransfers eingerückt werden müssen, bis der vorab erbrachte Vorschlag mit einem „agree” bestätigt, mit „disagree“ abgelehnt oder durch das Konzept-Verb „decide“ eine Entscheidung getroffen wird. Wichtig ist: All dies braucht Zeit, welche den Arbeitsfluss reduziert.

Es ist wichtig zu erwähnen, dass Wissenstransfer per se nicht zwingend einen Abbruch einer Fokus-Phasen bedeutet. Allerdings muss vorausgesetzt sein, dass der Wissenstransfer im Falle von Wissenslücken seitens eines Entwicklers oder Uneinigkeiten zwischen beiden Entwicklern effizient und mit schnell folgendem Konsens gelöst werden. Lange Diskussionen oder Erklärungsversuche

reduzieren den Arbeitsfluss deutlich und würden somit eine Fokus-Phase beenden oder verhindern.

2.4 **Rezeptartiges Abarbeiten**

Ich habe bereits mehrfach gezeigt, dass in der PP-Sitzung CA5 sowohl Entwickler C3 als auch C4 einen sehr hohen Wissensstand bzgl. der Programmbibliothek und den enthaltenen, implementierten Klassen und deren Funktion besitzen. Des weiteren verstehen sie vollumfänglich, was die an sie gestellte Aufgabe ist. Auf dieser Basis sind sie in der Lage eine Lösungsstrategie zu verfolgen, bereits bestehenden Programm-Code wiederzuverwenden und anzupassen.

Da beiden Entwicklern bewußt ist wie sie den Code verändern müssen, habe ich in meinen Analysen eine Art und Weise des Abarbeitens beobachtet, welche einen hohen Arbeitsfluss ermöglicht. Wie in 2.2.1 dargestellt, „sprinten“ C3 und C4 durch die Codezeilen und benennen lediglich, ob das momentan annoncierte Code-Artifakt benötigt wird, ob es gelöscht oder verändert werden soll. Falls es verändert wird, ist beiden Entwicklern klar wie. Das Ausbleiben von produktorientierter Kommunikation in aufeinander folgenden taktischen Arbeitsschritten über das „Wie etwas geschehen soll“ und die blosse Benennung „Was betrachtet wird“, bringt einen gesteigerten Arbeitsfluss hervor. Dieses Konzept nenne ich **rezeptartiges Abarbeiten**. In Ausschnitten ist dieser Effekt bereits in *Beispiel 1* und *Beispiel 2* dargestellt. Obwohl Beispiel 1 diesen Effekt sehr gut beschreibt, möchte ich an dieser Stelle ein weiteres Beispiel bemühen:

Beispiel 4, Sitzung CA5, Teil I, 25:05 – 25:24:

Situation:
Nach kurzer Phase der Recherche und des Wissenstransfers beenden C3 und C4 die Aufgabe der Anpassung des kopierten und eingefügten Codes. Die Funktionalität zur Erstellung eines Polygons, mit dem ein vorhandenes Polygon geteilt werden soll, wird umgesetzt. Im Folgenden geschieht die Anpassung der Display-Funktionalität.

Verlauf:
Innerhalb von 19 Sekunden werden folgende Entscheidungen durch die beiden Entwickler getroffen:

1. Identifikation der zu verändernden Stellen im Programm-Code, in denen Display-Toolbar oder Display-Funktionen auftreten.
2. DisplayToolbar Funktion im ensure wird verändert und über die Klasse displayContext mit der Methode getDisplayToolbar eingebunden.
3. Weitere Vorkommen von DisplayToolbar werden wie vorab verabredet über die Klasse displayContext mit der Methode getDisplayToolbar eingebunden.
4. Display Funktionalität wird über über die Klasse displayContext mit der Methode getDisplay eingebunden.

Kommunikationsverlauf:

1. <propose_step> C3: “Display, Display, DisplayToolbar...”
   {C3 navigiert durch Code, jeweilige Stellen markierend}

   <agree_step> C4: „Nö.“
   {gemeint im Sinne: „Kein weiteres vorkommen von Displayfunktionen“}

2. <propose_design> C3: „Das ist dann der DisplayContext.“
   {markiert Stelle im ensure und ändert Code von DisplayToolbar zu DisplayContext.getDisplayToolbar}
   “... getDisplayToolbar... bla, blub...“

   <agree_design> C4: < langsames nicken>

3. <propose_design> C3: „Toolbar..“
   {navigiert zu weiterer Stelle im Code, ändert Code mittels Copy Paste}

   <agree_design> C4: {keine Reaktion, Code-Änderungen beobachtend}

   <propose_step> C3: “noch irgendwo Toolbar?”

   <agree_step> C4: {suchend nach weiteren Vorkommen im Code, keine verbale Reaktion}

   <propose_design> C3: „...hier ist Toolbar.“
   {navigiert zu weiterer Stelle im Code, markiert und ändert Code mittels Copy Paste}

   <agree_design> C4: „Mh-hmm.”

4. <propose_design> C3: {C3 markiert Stelle "Display" im Code}
   „Das ist so ähnlich. Das ist wahrscheinlich einfach getdisplay.“
   {ändert Code}

   <agree_design> C4: {keine Reaktion, Code-Änderungen beobachtend}

Wie in 2.2.1 beschrieben ist in *Beispiel 4* ein sehr hoher Arbeitsfluss durch schnelle, effiziente, produktorientierte Äußerungen zu beobachten. Dies wird aufgrund der schnellen Folge von Vorschlägen (propose) und direkt folgender Zustimmung durch den Partner (agree) deutlich.

Ebenso wie in *Beispiel 1* findet keine bis kaum Kommunikation darüber statt, wie der weitere Arbeitsverlauf aussehen soll. C3 erwähnt lediglich kurz, welches Code-Artifakt als nächstes zu bearbeiten ist. Ist beiden Entwicklern klar, wie die nächsten Arbeitsschritte geschehen sollen, wird lediglich nur noch rezeptartig abgearbeitet. Das Programm-Code geändert, beibehalten oder gelöscht wird, ist in *Beispiel 4* unausgeprochen Konsen. Wie der Code geändert wird, wurde vorab besprochen und beschreibt das Rezept. In Arbeitsschritten, in denen eine Codeänderung erforderlich ist, wird die Funktionalität der Entwicklungsumgebung effizient eingesetzt. In Zusammenhang mit dem Wissen über die Funktionalität und Inhalt des Codes der Programm-Bibliothek ist es C3 möglich, sofort eine passende Klasse und enthaltende Methode zu finden. Selbst wenn es eine Erwähnung über das weitere Verfahren eines Code-Artifaktes geben sollte (vgl. *Beispiel 1*), ist durch diese drei verabredeten Möglichkeiten das Rezept abgebildet.
Das Ausbleiben von Kommunikation bzgl. der nächsten Arbeitsschritte über das „Wie“ etwas geschehen soll und die blosse Benennung oder Aufzeigen durch die Input-Mittel (Maus, Tastatur) davon, was das nächste zu betrachtende Code-Artifakt ist, bezeichne ich als „rezeptartiges Abarbeiten“.

Diese Arbeitsweise trägt zu einem sehr hohem Grad zu einem erhöhtem Arbetisfluss und PP_{fast} bei. Die Vorbedingungen werden, wie eingangs erwähnt, durch die grenzwertig vollständige Kenntnis des bestehenden Programm-Codes, inklusive der Klassen- und Vererbungshierarchie und dem Entschluss diesen wiederzuverwenden, befördert. Allerdings ist dies keine notwendige Vorausetzung für rezeptartiges Abarbeiten. Es ist ebenso möglich, dass dieser Effekt nach hinreichend genauer Konzeption und hohem Verständnis des Design-Entwurfs ebenso während einer Implementations-Phase auftreten könnte, ohne vorhandenen Code wiederzuverwenden. Wichtig ist die Einigung und das Verständnis beider Entwickler auf ein Rezept, welches entsprechend algorithmisch abgearbeitet wird. In den vorab erbrachten Beispielen wird in einer Schleife ein Vorgang mit unterschiedlichen Code-Artifakten wiederholt. Ein Rezept kann ebenfalls aufeinander folgende Schritte beinhalten, welche sich nicht wiederholen, sondern jedesmal unterschiedlich sind. Für jeden dieser Schritte muss den Entwicklern klar sein, wie mit den jeweiligen Code-Artifakten umgegangen werden soll, um diese Voraussetzung für PP_{fast} ermöglichen zu können.

In *Beispiel 4* ist dieses Konzept insbesondere durch die Vermischung von produktorientierter Kommunkation (propose_design -> agree_design) und prozessorientierter Kommunikation (propose_step -> agree_step) zu erkennen. In den Äußerungen prozessorientierter Kommunikation ist genau diese Abwesenheit von inhaltlichen Bezügen auf konkrete Vorschläge hinsichtlich der Änderung von Code-Artifakten zu erkennen. Es wird lediglich benannt, welche Code-Zeile(n) als nächstes zu betrachten sind. Es besteht keine Notwendigkeit darüber zu reden, wie etwas geschehen muss.

Dies wird klarer bei der Analyse von Entscheidungsprozess 3. C3 benennt mit „Toolbar“ eine entsprechende Stelle im Code und ändert ihn mittels Kopieren und Einfügen. Es wird verbal kein konkreter Vorschlag geäußert, jedoch signalisiert C3 durch die gleichzeitige Änderung des Codes C4 einen Design-Vorschlag (propose_design). Es findet darüber weder Erklärung (warum dies so geschehen muss), noch Wissenstransfer statt. Entsprechend antwortet C4 nonverbal zustimmend (agree_design).
Um weitere Vorkommen der „Toolbar“ anzupassen, gibt C3 mit der Frage an C4 den nächsten folgenden Arbeitschritt vor: „Noch irgendwo Toolbar?“ (propose_step). Hier wird keine Referenz auf konkrete Code-Artifakte zum Ausdruck gebracht. C3 möchte lediglich mitteilen, dass die Suche nach entsprechenden Stellen im Code der nächste Schritt ist. C4 nimmt dies an (agree_step). Ohne nennenswerte Zeitverzögerung wird eine Stelle im Code gefunden und C3 kommentiert dies: „...hier ist Toolbar.“ Wiederum ist keine Kommunikation zu beobachten, wie mit diesem gefundenem Code-Artifakt umgegangen werden soll. Statt dessen erfolgt eine Änderung des selbigen durch C3 (propose_design). C4 stimmt der Änderung mit dem Kommentar: „Mh-hmm“ zu (agree_design).

Die Folgen von Vorschlag (propose) und direkter Zustimmung (agree) beschreiben den Zustand PP_{fast}, welcher durch das beschriebene Konzept „rezeptartiges Abarbeiten“ ermöglicht wird.

3 Vorbedingungen von Fokus-Phasen

Der in Kapitel 2 beschriebene schnelle Arbeitsfluss PP_{fast} ist kein Zustand, der plötzlich unvorhergesehen passiert. Gegenteilig sind bestimmte Vorbedingungen notwendig, welche die Togetherness und Synchronisation eines Paares befördern. Im dritten Kapitel möchte ich einen Überblick über die beobachteten Voraussetzungen geben, ohne einen Anspruch auf Vollständigkeit

zu erheben. Es ist durchaus denkbar und wahrscheinlich, dass bei der Analyse weiterer PP-Sitzungen Vorbedingungen analysiert werden, welche in dem mir zur Verfügung stehenden Material nicht vorkommen. Dies kann bspw. mit der Art der Aufgabe und dem Entwicklungsprozess (Software-Entwurf, Test von Software, Implementation, Debugging, etc.) zu tun haben, welche an das Paar der PP-Sitzung gestellt wird.

3.1 Vollständiges, detailliertes Verständnis des Problems

Die im vorangegangenen Abschnitt erläuterte Vorausetzung (rezeptartiges Abarbeiten) ermöglicht PP_{fast}. Dieses Konzept kann nur dann umgesetzt werden, wenn beiden Partner detailliert klar ist, was die Anforderungen an ihre Aufgabe ist. Sie müssen das Problem vollumfänglich verstehen. Sind Teile unklar, kann kein „Rezept“ für ein schnelles, effizientes Abarbeiten gefunden werden. Anstelle dessen tritt an dem Punkt im Arbeitsprozess, an dem es Unklarheiten gibt, ein Abschnitt der Recherche und eine Diskussion darüber, wie weiterverfahren werden soll. Implizit ist der nächst folgende Arbeitsschritt unklar. Der Zustand PP_{fast} wird unterbrochen, da vorerst keine Fortschritte erzielt werden. Die in Kapitel 2 beschriebene, Fokus-Phasen charakterisierende, schnelle Abfolge von Vorschlägen und sich dem direkt anschließenden Zustimmungen wird hierdurch unterbrochen.
Meist muss externes Wissen eines Supervisors herangezogen werden, welcher im Zweifel nicht sofort zur Verfügung steht.

In Sitzung AA1 besteht eingangs das Problem, dass den Entwicklern nicht vollständig bewusst ist, wie die Änderungen stilistisch umzusetzten sind. Über einen Zeitraum von 6 Minuten und 50 Sekunden diskutieren sie, ob bestimmte Objekte durchgestrichen, unterstrichen, ohne oder mit anderer Formatierung dargestellt werden sollen, um verschiedene Zustände von Linkzielen auszudrücken. Dieser Vorgang geschieht von 3:40 bis 10:30. A2 äußert einen Vorschlag für einen Workaround, die CSS Klassen später anzupassen und vorerst durch einen Identifier „inactive“ in einem „span-tag“ zu signalisieren, welche Objekte betroffen sind. Vorab wird versucht, einen Supervisor telefonisch zu erreichen, welcher nicht reagiert.
Nachdem 1 Stunde und 8 Minuten der PP-Sitzung vergangen sind, erscheint der Supervisor und das Problem kann konkret besprochen werden. Die Entwickler unterbrechen ihren derzeitigen Arbeitsprozess. Es folgen sieben Minuten Erklärung der Unklarheit für den Supervisor und dem Annoncieren einer detaillierten Anforderung an die Entwickler. Nachdem die Entwickler verstanden haben, wie genau die stilistische Umsetzung der verschiedener Linkziele aussehen soll, nehmen sie den vorab unterbrochenen Vorgang wieder auf. Die Togetherness des Paares muss kurz wieder aufgebaut werden, indem sich beide kurz erinnern, wo sie standen und was sie machen wollten. Der Arbeitsfluss wird durch die Ablenkung und Unterbrechung des Arbeitsprozesses reduziert. Sowohl in der eingehenden Phase der Strategiefindung, als auch in der Phase des Verschaffens von Klarheit durch die Expertise und Anweisungen des Supervisors, werden kaum Fortschritte erzielt.

Diese Voraussetzung für eine Fokus-Phase ist statisch einzuordnen. Sie muss vor einer PP-Sitzung geklärt werden. Während einer Sitzung ist es schwer, diesen Nachteil auszugleichen. Eine hinreichend genaue Vorbesprechung der PP-Sitzung oder mittels Dokumentationen verfügbar gemachtes Wissen ermöglicht eine detailierte Klarheit des Problems.
Je nachdem, wie gravierend die Wissenslücken des Paares hinsichtlich der an sie gestellten Aufgabe sind, können sich Paare mit Work-Arounds oder „Todo´s“ Abhilfe schaffen. Die Aufgabe wird in dem

Fall nur teilweise erfüllt, jedoch können weiterhin Fortschritte erzielt und PP_{fast} weiterhin aufrecht erhalten werden. Die Fokus-Phase wird somit nicht beendet. Zu erwarten ist ein temporärer Übergang zu PP_{normal}. Für den Fall, dass vorerst keine Idee gefunden wird, ist $PP_{breakdown}$ zu erwarten und die Togetherness des Paares muss erneut aufgebaut werden.

3.2 Bewußte Erarbeitung einer Lösungsstrategie

Das Erarbeiten und vollständige Verständnis einer Lösungsstrategie ist eine wichtige Voraussetzung für einen überdurchschnittlich flüssigen Arbeitsprozess. Rezeptartiges Abarbeiten als Voraussetzung für PP_{fast} setzt, wie in Kapitel 2.4. beschrieben, ein vollständiges Verständnis voraus, wie die Bearbeitung geschehen soll. Die Abwesenheit über Kommunikation bzgl. des „Wie Code-Artifakte bearbeitet werden“ charakterisiert das Rezept. Die Entwickler müssen die Lösungsstrategie verinnerlicht haben, um über die konkrete Vorgehensweise in einzelnen taktischen Schritten (das Rezept, Vgl. 2.4.) nicht mehr nachdenken oder diskutieren zu müssen. Dies würde den Arbeitsfluss in Richtung PP_{normal} reduzieren.

In der Strategiefindung sind Unterschiede zu beobachten, welche rezeptartiges Abarbeiten ermöglichen oder wenigstens beeinflussen. Auf der einen Seite kann das Paar bewußt eine Lösung für das an sie gestellte Problem entwickeln. Den Entwicklern ist, wie in Abschnitt 3.1. beschrieben, vollständig und detailliert bewußt, was die Anforderungen sind. Im Gegensatz dazu kann selbstverständlich auch keine Strategie gefunden werden, was folgend vernachlässigt wird, da eine Fokus-Phase somit defacto ausgeschlossen ist. Interessant ist eine weitere Möglichkeit, in der die Entwickler während der Phase einer Strategiefindung eine erzwungene Lösung wählen müssen. Dies geschieht in meinen Analysen deswegen, weil Abhängigkeiten unklar sind und die Anforderung an die gestellte Aufgabe nicht im Detail verstanden werden.

Die Entwickler in Sitzung CA5 erarbeiten zu Beginn eines jeden strategischen Abschnittes der Sitzung eine Lösungsstrategie für den kommenden Arbeitsabschnitt. Ihnen ist detailliert bewußt, wie die Funktionalität ihres Moduls auszusehen hat. Sie sind in somit in der Lage, auf bestehendem Code aufbauend, Teilabschnitte für die Lösung zu definieren und detailliert die jeweiligen Ziele festzulegen, welche in dem Arbeitsmodul erfüllt werden sollen. Die Strategiebesprechung wird anhand von konkreten bestehenden Code-Artifakten evaluiert und kann somit anschließend abgearbeitet werden. Beinhalteter Wissenstransfer findet schnell und effizient statt. Darüber hinaus sogar Vorschläge des Partners ergänzend. Teilweise wirkt dies tatsächlich wie ein Paar, welches mit einem Verstand denkt. Ideen des Partners werden ernst genommen, verfolgt und gemeinsam evaluiert.

Ein interessantes Phänomen wird in Diskussionen darüber beschrieben, in wie fern die Aufgabe oder Teilaufgabe erfüllt und welche Aufgabenbereiche in einer weiteren PP-Sitzung umzusetzen sind. Jedes Mal erfolgt ein schneller Konsens bzgl. des Umfangs des an sie gestellten Problems. Beide Partner schaffen es, sich schnell auf konkrete strategische Ziele zu einigen, welche die Erarbeitung einer Lösungsstrategie ermöglicht. In der Nachbesprechung erfolgt eine kurze Diskussion über dieses Phänomen, welches ich im folgenden darstellen möchte:

Beispiel 5, Sitzung CA5, Teil I, 37:49 – 38:57:

Situation:
Nachdem die Aufgabe der PP-Sitzung erfüllt wurde, nehmen sich die Entwickler ca. 10 Minuten Zeit, über ihre Erfahrungen während der Sitzung auszutauschen und Feedback zu geben. Sie besprechen Punkte hinsichtlich der Kommunkation und des Arbeitsablaufes.

Kommunikationsverlauf:

C4: "Ne, ich hatte mehr ein Problem insgesamt mit dem Tempo. Also für mich, hab ich da immer noch was hinterlassen."

C3: "Mh-hmm"

C4: "Also auch an anderen Stellen. Du hinterlässt verhältnismäßig viele Todo-Now´s und ich schließe Dinge lieber rund ab und bin da Safe. Und dann gehe ich erst zum nächsten Punkt über."

C3: "Mh-hmm. Das ist interessant, weil ich.. Also wenn ich das gemacht hätte.. also wenn wir das nicht zu zweit gemacht hätten, hätte ich mehr Sachen gleich gemacht bevor ich weiter gegangen wär. Aber ich hätte Sachen gemacht, wo du jetzt gesagt hast: Ne, das ist ne andere Karte."

C4: "Ja. Das glaub ich gern."

C3: "Ich hätte zum Beispiel gar nicht mit dem Polygon etwas gemacht sondern erst mal geschaut, dass die Action erst mal enabled ist und so weiter. So das ich weiß, bis zu dem Punkt ist alles erledigt."

C4: "Ja gut. Da weiß ich aber das da noch Karten sind."

C3: "Mh-hmm"

C4: "....die das dann erledigen. Also das geht nicht weg. Aber so innerhalb eines Arbeitspunktes sind die Sachen.. In diesem Arbeitspunkt sind die Sachen auf der Karte wo drauf steht.. dass ich die clean hab bevor ich zum nächsten wander. Das mach ich so."

Auswertung:

Während der Phasen in Sitzung CA5 hinsichtlich der Besprechung der Strategie des folgenden Abschnittes kommt es tatsächlich mehrmals vor, dass C4 Vorschläge von C3 für ein strategisches Ziel mit dem Argument ablehnt, dass dies nicht Bestandteil der Sitzung ist. C4 hat offensichtlich ein sehr gutes Verständnis über die Meta-Ebene des Entwicklungsprozesses der Software, jenseits dieser konkreten Sitzung. C4 ist bewußt, welche weiteren Teilaufgaben definiert wurden. Somit ist C4 in der Lage, bewusst strategische Ziele zu definieren. C4 tut dies auf der Basis des detaillierten Verständnis für das an das Paar gestellte Problem abgrenzend zu anderen Aufgaben, welche in weiteren Sitzungen erfüllt werden sollen. C3 und C4 entwickeln bewußt eine Lösungsstrategie für ihre Aufgabe und konkrete, detailliert ausformulierte strategische Ziele für Teilprobleme. C3 ermöglicht dies, in dem C4 gefolgt wird und eine Anpassung des Verhaltens erfolgt. Dies drückt C3 im obigen Beispiel aus.

Im Gegensatz zu Sitzung CA5 sind die Entwickler in Sitzung AA1 gezwungen, eine Lösungsstrategie oder wenigstens Teile davon zu wählen, die nicht bewußt erarbeitet werden kann. Sie sind kontextgebunden (wie in Abschnitt 3.1. beschrieben) dazu gezwungen, einen Work-Around zu wählen. Ihnen ist nicht detailliert klar, was die Anforderung an sie sind und wie das Software-Produkt aussehen soll. A2 macht einen Vorschlag für einen Work-Around, welcher schlüssig ist. In den folgenden strategischen Überlegungen und somit im weiteren Arbeitsprozess muss diesen Abhängigkeiten gefolgt werden. Handelt es sich bspw. um Unklarheiten bzgl. eines Teilproblems, auf dem im weiteren Softwareentwicklungsprozess aufgebaut wird, so behindert diese erzwungene Erarbeitung der Lösungsstrategie für die Aufgabe PP_{fast}. Rezeptartiges Abarbeiten wird hierdurch erschwert.

3.3 spezifisches Wissen

Spezifisches Wissen konzeptionalisiert das Verständnis von relevanten Aspekten des Software Systems, insbesondere die folgenden oder Teile davon beinhaltend: seine Anforderungen, seine Architecktur, das Design, Design-Abhängingkeiten, bestehenden Code (ebenso Tests, Skripte, etc.), bekannte Fehler u.w. Vgl. [5] Spezifisches Wissen ist voraussetzend für das Lösen des gestellten Problems und kann während einer PP-Sitzung durch Recherche oder Wissenstransfer erlangt werden. Vgl. [5]

Spezifisches Wissen kann in drei Kategorien eingeteilt werden: Vgl. [5]

1. **Hoch**:
   Der Entwickler hat ein umfangreiches und mehr oder weniger vollständiges Verständnis für das System und seine Abhängigkeiten, als auch für die relevanten Aspekte des zum Lösen der gestellten Aufgabe. Hierdurch wird ermöglicht, neue Erkenntnisse herauszufinden und einzuordnen und dem Partner den existierenden Status quo des Systems zu erklären.
2. **Mittel:**
   Der Entwickler hat ein gutes Verständnis für das System im allgemeinen, jedoch zu wenig, um alle konkreten Anforderungen an das gestellte Problem lösen zu können oder ist nicht auf dem neuesten Stand, kürzliche Änderungen betreffend. Konkret treten während der PP-Sitzung durch den Entwickler Fragen oder Äußerungen auf, konkrete Aspekte bzgl. Vorschlägen erst evaluieren oder nachschlagen zu müssen. Die Wissenslücken müssen erst gefüllt werden. Für den Fall, dass sich ein Entwickler diesen Wissenslücken nicht bewußt ist, kann dies zu wiederholten Ablehnungen seitens des Partners bzgl. erbrachter Vorschläge führen.
3. **Niedrig:**
   Der Entwickler versteht kaum etwas von den relevanten Aspekten des Software-Systems. Es wird ausgedrückt, dass Wissen das System betreffend nicht vorhanden ist und es existieren keine bis sehr wenige Äußerungen hinsichtlich der Aspekte und Abhängigkeiten, die noch nicht zur Sprache kamen.

Die Operationalisierung spezifischem Wissens eines Entwicklers erfolgt „top-down“. Es wird vorab angenommen, dass ein Entwickler über hohes spezifisches Wissen verfügt. Jede Äußerung, welche eine Wissenslücke ausdrückt oder vom Partner signalisiert wird, führt zu einer Herabstufung. Nicht

jede Wissenslücke führt direkt von Hoch zu Mittel. Die Einstufung ist im Gesamtkontext der Sitzung vorzunehmen. Vgl. [5]

Spezifisches Wissen als Voraussetzung für Fokus-Phasen

In Kapitel 2 und den einzelnen Abschnitten habe ich mehrfach erwähnt, dass die schnelle Folge von Vorschlägen und folgenden Zustimmungen Fokus-Phasen charakterisiert. Zwei Voraussetzungen, welche rezeptartiges Abarbeiten als Voraussetzung für PP_{fast} ermöglichen, wurden in den vorangegangenen Abschnitten erläutert (Kapitel 3.1 und 3.2.).

In den Abschnitten von Kapitel 2 ist darüber hinaus mehrfach zur Sprache gekommen, dass eine weitere Voraussetzung für die in den Beispielen dargestellen flüssigen Arbeitsprozesse und das hohe Maß an Synchronisation der beiden Paartner, das Verständnis für den bestehenden Code und die Programmbibliothek sind. Dies möchte ich im folgenden konkreter beschreiben.

Spezifisches Wissen konzeptionalisiert alle Aspekte des Software-Systems, welche zur Lösung des Problems relevant sind. U.a. gehört dazu insbesondere bereits umgesetztes Design, der bestehende Code, Abhängigkeiten des Systems etc. Ein sich in meinen Analysen wiederholender und wichtiger Aspekt ist die Frage: „Warum gelingt es den Entwicklern in Sitzung CA5 flüssiger zu arbeiten und in dem Paar in Sitzung AA1 weniger?“ Die Antwort liegt in der Bedeutung spezifischem Wissens für Fokus-Phasen. Um diesen Zusammenhang zu erklären, werde ich folgend beide Sitzungen hinsichtlich Lücken spezifischem Wissens einordnen.

Einordnung Sitzung CA5:
Beide Entwickler haben ein sehr gutes Verständnis für das Software-System, dessen Abhängigkeiten und sind auf dem neuesten Stand. Sie verstehen die Programmbibliothek vollständig und sind in der Lage, auf bestehendem Code aufzubauen. Insbesondere in Phasen der Strategiefindung für kommende Arbeitsschritte zeigt sich dies deutlich (siehe *Beispiel 4*). Beide Entwickler wissen ohne langen Recherche-Prozess genau, wo sie in der Programmbibliothek nachsehen müssen. Sie ergänzen sich darüber hinaus durch weiterführende Strategievorschläge. Die Aufmerksamkeit beider Entwickler ist in den Findungs-Phasen hoch, wenn sie der observierende Teil sind.
C4 zeigt keine wesentlichen Wissenslücken bzgl. des Software-Systems und drückt keine aus. Darüber hinaus hat C4, wie in *Beipiel 5* gezeigt, ein umfängliches Verständnis für weitere Entwicklungsprozesse und ist somit in der Lage, das Problem von anderen Aufgaben abzugrenzen.
C3 zeigt an zwei Stellen der PP-Sitzung durch Vorschläge kleine Wissenslücken, welche für den Arbeitsablauf nicht relevant sind. C3 hat ebenfalls einen umfänglichen Überblick über das Software-System, ist in der Lage Alternativ-Vorschläge und Verbesserungen zu annoncieren.
Beide Entwickler werden mit Hoch eingestuft. Die Abwesenheit von für den Arbeitsablauf notwendigen Phasen des Wissenstransfers, Aussagen über eigene Wissenslücken oder die des Partners, untermauern dies.

Um die analysierte Aufmerksamkeit des beobachtenden Partners auf der Basis spzeifischem Wissens und die Bedeutung zu verdeutlichen, folgend ein kurzes Beispiel:

Beispiel 6, Sitzung CA5, Teil I, 25:05 – 25:24:

Situation:
Die Phase der Strategiefindung für das weitere Vorgehen ist abgeschlossen. Die Funktionalität für das

Erstellen eines Polygons, auf der Basis dessen Geometrie ein bestehendes Polygon geteilt werden soll, ist abgeschlossen. C3 und C4 haben die Aufgabe der Anpassung des kopierten und eingefügten Codes. Im Folgenden geschieht die Anpassung des Selection-Models für die Teilungsfunktionalität. C4 beginnt mit dem ersten taktischen Arbeitsschritt.

Verlauf:
Innerhalb von 20 Sekunden werden folgende Enstscheidungsprozesse besprochen:

1. C4 entscheidet mit der Anpassung des SelectionModels zu beginnen.
2. C3 bittet C4 die Entscheidung 1 zu revidieren und die Eingabe zu beenden.
3. C3 erklärt einen neuen Alternativ-Vorschlag.
4. C3 übernimmt die Eingabemittel und erklärt seinen Strategievorschlag, der eine effizientere Lösung verspricht.

Kommunikationsverlauf:

1. <ask_step> C3: “Was machst du?”

   <propose_step> C4: „SelectionModel?“
   {tippt während dessen bereits SelectionModel Klasse und Methode für Auswahl}

2. <stop_activity> C3: „Ne. Warte. Das ist...“
   <challenge_step> C3: „ich glaub das brauchen wir nicht mal.“

   <explain_knowledge> C4: „Wieso? Du musst doch augenblicklich...“
   {wird unterbrochen}

3. <propose_strategy> C3: „Das wäre der Standardweg. Aber ich glaube es gibt eine Abkürzung hier, weil wir ja das sowieso an anderen Stellen schon... Also ich...“

4. C3: {schaut auf die Tastatur, Hände gestikulieren den Wunsch einer Eingabe, beginnt mit Auswahl von Klassen in der Programm-Bibliothek}

Hohes spezifisches Wissen beider Entwickler trägt zu einem hohem Mass zur Togetherness bei und ermöglicht Fokus-Phasen.

3.4 Gradliniges Arbeiten

Die Aufgabe einer PP-Sitzung in produktiven Software-Entwicklungsprozessen kann mehrere strategische Abschnitte beinhalten. Die Gesamtaufgabe wird vorab in Arbeitspakete unterteilt. Für einen erfolgreichen Abschluss einer Teilaufgabe und schnelle Fortschritte während der Lösung selbiger ist es hilfreich, sich auf genau diesen Lösungsprozess zu konzentrieren und gradlinig zu arbeiten. Dies können die Entwickler aktiv beeinflussen. In einer Teilaufgabe Fokus-Phasen befördernde Faktoren sind die folgenden drei:

1. Einsatz und aktive Pfege von Versionierungssystemen
2. Verschieben interessanter, für den Moment nicht essentieller Teil- oder Nebenprobleme
3. Wiederaufnehmen vorab verschobener Arbeitsschritte an einem späteren Zeitpunkt

3.4.1 **Bedeutung von Versionierungssystemen**

In fast allen beobachteten Projekten nutzen Software-Unternehmen Versionierungssysteme für Projekte. In diesem Abschnitt möchte ich kurz auf die Relevanz für die Entwickler und den Einfluss auf Fokus-Phasen eingehen.

Der aktive Umgang mit Versionierungssystemen, völlig gleich welches es ist (SVN, Git, CVC, etc.) bringt neben offensichtlich Vorteilen ebenso Faktoren mit sich, welche Fokus-Phasen befördern können.

Entwickler werden u.a. durch den Einsatz der Versionierungssoftware in die Lage versetzt, die an sie gestellten Aufgabe zu verstehen und von anderen Teilproblemen abzugrenzen. Die Bedeutung dessen habe ich in Kapitel 3.1. beschrieben. Der Fokus auf den Kern des zu lösenden Problems ist wichtig, um effizient schnelle Lösungsstrategien zu entwickeln und sich auf das wesentliche zu konzentrieren. Darüber hinaus erhöht der aktive und bewußte Umgang mit den einzutragenden und bereits formulierten Inhalten des Versionierungssystems die togetherness. Das Paar muss hinreichend genau formulieren, was getan werden muss oder gelöst wurde. Somit muss ein Einigungsprozess stattgefunden haben und beide Entwickler müssen sich verständigt haben, was genau das Zeil des Arbeitsschrittes oder des Gesamtproblems ist, an dem sie arbeiten. Wird ein Arbeitsabschnitt beendet, muss vor einem Eintrag zwangsläufig beiderseitig signalisiert worden sein, dass die Umsetzung vollständig und abgeschlossen ist. Sowohl in weiteren beobachteten Sitzung, als auch der analysierten Sitzung AA1 ist zu erkennen, dass dieser Prozess oftmals nicht trivial ist. Ein Entwickler ist bspw. der Meinung, dass vor einem Eintrag (commit) in das Versionierungssystem noch ein Test implementiert werden muss, während der andere Entwickler diesen für nicht nötig empfindet. Durch die Diskussion und eine folgende Entscheidung verständigen sich die Entwickler auf eine Lösung und wissen anschließend, an welchem Punkt des Arbeitsprozesses sie stehen und welcher der nächste Arbeitsschritt ist.

Darüber hinaus müssen Inhalte konkret formuliert werden, damit auch Dritte diese verstehen. Die Entscheidung für eine konkrete Formulierung ist ebenso hilfreich, um togetherness aufzubauen. Details sind hierbei entscheidend: bspw. die Formulierung der Umsetzung einer Fuktionalität im Konjunktiv oder Imperativ.

3.4.2 **Verschieben von Nebenproblemen**

Während einer Fokus-Phase in einem Prozess rezeptartigem Abarbeitens kann sich der nächste Arbeitsschritt auf ein Code-Artifakt beziehen, bei dem sich die Entwickler nicht sicher sind, wie mit selbigem verfahren werden muss. Hier geschieht eine kritische Entscheidung, welche die Fokus-Phase leicht beenden kann. Entscheiden sich die Entwickler, diesen Gedanken weiterzuverfolgen und zu recherchieren, was mit dem entsprechendem Code-Artifakt geschehen soll, wird der Prozess des rezeptartigen Abarbeitens unterbrochen und an Stelle dessen erfolgt ein Recherche-Vorgang oder eine Diskussion. Der Arbeitsfluss wird reduziert. Die Entwickler in Sitzung CA5 werden mehrmals vor die Entscheidung gestellt, wie weiterverfahren werden soll und einigen sich, diesen Arbeitsschritt zu verschieben und später wieder aufzunehmen. Die Fokus-Phase wird fortgesetzt.

Der verschobene Arbeitsschritt muss nebenläufig erfüllbar sein. Handelt es sich um eine Code-Artifakt, auf dem weitere aufbauen, kann er nicht verschoben werden. In dem Fall kann eine Lücke

im spezifischem Wissens zu beobachten sein, da diese Abhängigkeit hätte klar sein müssen. Eine schnelle Lösung wird togetherness nicht signifikant reduzieren und das Paar müsste diese nicht erneut aufbauen, so dass anschließende schnelle Fortschritte möglich sind.

Während des Entwicklungsprozesses ergeben sich darüber hinaus oft interessante, aber für die Lösung des Problems unwichtige Teilprobleme. Dies kann bspw. Diskussionen über Ausnahmen oder Extremfälle beinhalten. Innhalb einer Fokus-Phase ist es hilfreich, diese zu verschieben und die folgenden Arbeitsschritte abzuarbeiten, um den Arbeitsfluss nicht zu unterbrechen.

3.4.3 **Wiederaufnehmen verschobener Arbeitsschritte**

Wurde, wie im vorangegangenem Abschnitt beschrieben, ein atomarer, taktischer Arbeitsschritt verschoben, muss dieser zu gegebener Zeit fortgesetzt werden. Dieser Zusammenhang mag trivial erscheinen. In der Praxis ist dies nicht immer der Fall und somit entstehen Fehler in der Software, welche anschließend nur mit Mühe zu finden sind. Insbesondere in längeren strategischen Abschnitten ist leicht, einen verschobenen Arbeitsschritt zu vergessen. Ein Todo-Kommentar im Code oder die Aufmerksamkeit mindestens eines Entwicklers kann helfen, dies nicht zu vergessen.

Das Timing ist ebenfalls entscheidend. Verschobene Schritte sollten erledigt werden, bevor zum nächsten strategischem Abschnitt übergegangen wird. Wird dies vergessen, kann eine folgende, spätere Fokus-Phase heirdurch negativ beeinflusst werden und der Arbeitsfluss wird reduziert. Die Entwickler müssen sich erst wieder erinnern, wo genau das Problem war, bevor mit einer Lösungsfindung begonnen werden kann.

3.5 **Generisches Wissen**

Der Begriff generisches Wissen beinhaltet generisches, system-unabhängiges Wissen wie z.B.: Programmiersprachen, Frameworks. Entwicklungsmethoden, Test- und Debugging-Methoden u.a., welche für das zu lösende Problem wichtig sind. Darüber hinaus beinhaltet generisches Wissen Soft-Skills, insbesondere im Zusammenhang mit PP die Fertigkeit, welche Prechelt und Zieris PP-Skill nennen. Hierunter werden Fertigkeiten der Entwickler zusammengefaßt, welche Kollaboration jenseits technischer Fertigkeiten befördern. Vgl. [5]

Im Vergleich zu spezifischem Wissen ist generisches Wissen nicht zwingend erforderlich, um die Aufgabe lösen zu können. Jedoch werden hierdurch wichtige Arbeitsschritte erleichtert, wie bspw.: Schließen von Lücken im spezifischem Wissen, Erarbeitung eleganter Lösungen, flüssigere Implementierung und Debugging einer Lösung. Vgl. [5]

Ähnlich wie im Abschnitt 3.3. beschriebene Einteilung spezifischen Wissens, wird generischen Wissen ebenfalls in Hoch, Mittel und Niedrig unterschieden. Die Operationalisierung ist ähnlich und bezieht sich auf die eingangs genannten Eigenschaften. Vlg. [5]Ich möchte diese nicht näher erläutern, da diese nur teilweise relevant für Fokus-Phasen ist.

Generisches Wissen ist sehr hilfreich um Fokus-Phasen zu erlangen und diese aufrecht zu erhalten. Im folgenden möchte ich nur einige Aspekte nennen.

In einer PP-Sitzung wird es vorkommen, dass eine Lösung für einen atomaren, taktischen oder auch strategischen Schritt unklar ist oder eine gewisse Unsicherheit über die Validität der Lösung besteht. Ein Recherche Prozess muss erfolgen. Der effiziente Umgang mit der Entwicklungsumgebung und dessen Hilfsmitteln, der verwendeten Programmiersprache, dem Framework etc. trägt entscheidend dazu bei, diese Lücken im spezifischem Wissen schnell zu schließen. Die Entwickler werden durch hohes generisches Wissen schneller in die Lage versetzt, eine Entscheidung zu treffen und zum nächsten Schritt überzugehen. Dies erhöht den Arbeitsfluss. Im Gegensatz dazu werden lange Recherche-Prozesse den Arbeitsfluss deutlich reduzieren.

PP-Skill umfasst eine Vielzahl von Fertigkeiten eines Entwicklers und ist nur sehr schwer vollständig zu beschreiben oder zu definieren. Der Einfluss auf den Arbeitsfluss ist jedoch deutlich. Bspw. befördert die Fähigkeit der Entwickler, kollaborativ zu arbeiten, mit Konfliktlösungen besser umzugehen und allgemeine Bereitschaft Kompromisse zu suchen, die Geschwindigkeit in der Fortschritte erreicht werden können. Insbesondere in Phasen der Strategiefindung für einen Arbeitsabschnitt oder Vorlieben bzgl. Programmierstile und die anschließende Auflösung des Konfliktes machen diesen Zusammenhang deutlich. Vgl. Kapitel 4.1.

In Debugging-Prozessen ist der effiziente Umgang mit den Werkzeugen, Methoden und Erfahrung essentiell für einen schnellen, erfolgreichen Arbeitsprozess. Im Gegensatz dazu wird der Arbeitsfluss durch die Abwesenheit generischen Wissens deutlich reduziert.

Entwickler mit niedrigem und mittlerem generischem Wissen wird es schwerer fallen, effizient Informationen bereitzustellen, um einen Entscheidung im Entwicklungsprozess zu erlangen. Der Arbeitsfluss wird reduziert, PP_{fast} ist ausgeschlossen und der Arbeitsprozess allgemein kann stockend und ungradlinig werden, da die Entwickler die Orientierung verlieren können.

4 Fokus-Phasen negierende Faktoren und Lösungsstrategien

Die in Kapitel 3 beschriebenen Vorbedingungen für Fokus-Phasen können nicht immer ermöglicht werden. Es existieren eine Vielzahl von Problemen, die der Realisierung selbiger entgegenstehen. Diese Faktoren können dem Erreichen eines hohen Arbeitsflusses im Wege stehen oder Fokus-Phasen beenden. Im folgenden Kapitel möchte ich wesentliche, in dem mir zur Verfügung stehendem Material analyisierte Faktoren näher beleuchten und Lösungsstrategien anhand geben.

4.1 Lösungsstrategien in Konfliktsituationen

Während einer PP-Sitzung kann es vorkommen, dass bzgl. konkreter Implementierungen, Methoden, Teilen einer Programmiersprache, Teilen der Bibliothek oder Algorithmen aufgrund von Vorlieben oder Vorurteilen keine Entscheidung gefunden werden kann. Der Arbeitsprozess wird deutlich reduziert. Was der Grund für den Konflikt ist, spielt im folgenden keine grosse Rolle. Anhand eines Beispiels möchte ich das Problem kurz verdeutlichen.

Beispiel 7, Sitzung AA5, 23:20 – 24:20

Situation:
C4 und C3 befinden sich in einer Fokus-Phase während eines Prozesses zezeptartigem Abarbeitens.

Die Teilungsfunktionalität eines vorab erstelltem Polygons, auf Basis dessen Geometrie eines bereits bestehendes Polygon geteilt werden soll, wird umgesetzt. Die Strategie ist beiden Entwicklern klar, der Arbeitsfluss hoch. Aufgrund der folgenden Uneinigkeit bricht der Arbeitsfluss zu $PP_{breakdown}$ ab.

Verlauf:
Innerhalb von 60 Sekunden sind folgende Entscheidungen relevant:

1. C3 schlägt die Implementierung von ensure-Funktionen vor.
2. C4 lehnt den Vorschlag kategorisch ab.
3. C4 macht einen Gegenvorschlag.
4. C3 schlägt vor, die Funktionen wie in 1 vorgeschlagen vorläufig zu implementieren.
5. C4 nimmt den Vorschlag mit der Bedingung an, dass sie später wieder entfernt werden.
6. C3 schlägt eine externe, spätere Diskussion vor, um eine permanente Lösung zu finden.

Kommunikationsverlauf:

1.
<propose_design> C3: „Jetzt weißt du was bei mir kommt."
{fügt eine ensure-Funktion in den Code ein}

2.
<disagree_design> C4: {Schultern verkrampfen, Mimik: Ekel wird ausgedrückt, rechte Hand bewegt sich an die Nase}
„Nein! Nein!"

<mumble_sth> C3: {Blick richtet sich zur Partnerin, keine Eingabe mehr, sehr leise, vorsichtige Aussprache}
„Ich..."
{folgend weiterhin inaktiv mit Blick auf Partnerin, verhaltenes Lächeln}

<disagree_design> C4: {Hände nehmen eine Gebetsposition ein, Blick senkt sich, Kopf fällt auf noch immer gefalltete Hände}
„Bitte, Bitte nicht!"

3.
<challenge_design> C4: {Körperhaltung wird offen, Schulter gehen zurück, Kopf hebt sich, Hände heben Gebetsposition auf, Gesicht wird klar mit fröhlichem Ausdruck}
„Lass uns Test schreiben."
{Gesichtsausdruck erstarrt mit zusammengepressten Lippen über 1.8 Sekunden mit Blick auf Screen, C4 Blick wendet sich zum Partner, Hände fassen einander vor dem Körper weg vom Partner gerichtet, spricht lachend}
„...dann willst du diese ensures da nicht haben."
{kurzes Lachen mit tiefem Luftholen, Blick richtet sich hinter den Partner ins Off, Körper bewegt sich zurück, länger lachend, Gesicht bewegt sich zu den gefalteten Hände und lacht in sie hinein, Kopf weg vom Partner gerichtet}

4.
<propose_design> C3: „Tun sie dir weh wenn ich sie jetzt hinmach?“

<disagree_design> C4: “JA! Total!”
{Hände sind wieder in Gebetsposition, Köpf und Hände bewegen sich gebetsartig, flehend vor und zurück}

5.
<propose_todo> C4: aber gut...
{Haltung löst sich spontat auf, Hände breiten sich vor dem Körper aus und gehen in eine gewohnte Haltung zurück, rechte Hand Mund verdeckend}
„wenn es dir nicht weh tut, wenn ich sie später beim Testen wieder weg mache...“

<agree_todo> C3: {kurzes Lachen, immer noch ohne Eingabe, Blick auf Screen, Gesicht entspannt, Kopf nickend}
„Wenn du sie beim Testen...äh... wenn sie dir beim Testen im Weg sind...“
<decide_design> C3: {setzt jetzt die Eingabe der ensure-Funktionalität fort}
„...aber so lang sie nicht im Weg sind.“

<disagree_design> C4: {räuspert sich mehrmals, Gesichtfarbe leicht röter als zuvor}
„Ohhh, sie sind im Weg.“
{atmet schwer, nimmt einen Stift in beide Hände und führt ihn vor die Nase, verharrt in der Position 2 Sekunden und nimmt wieder gewohnte Sitzhaltung ein, rechte Hand stützt das Kinn den Stift greifend, Blick auf Screen gerichtet}

<decide_design> C3: {setzt Eingabe fort und beendet}

<disagree_design> C4: {fokussiert den Screen währenddessen mehrere Sekunden, schüttelt mehrmals den Kopf, lächelnd}
„Da werden wir uns nie einig.“
{wendet den Kopf von C3 schnell ab}
„Neee.“

6.
<propose_todo> C3: {fokussiert den Screen, markiert entsprechenden Code}
„Na wenn eine Lösung da ist, die das ganze besser macht, zum Beispiel über die Exceptions?
{Blick wendet sich zu C4}

Für den Fall, dass zwei Entwickler vor einer Entscheidung stehen, zur der es aus Gründen die nichts mit der Funktionalität oder systemischen Abhängigkeiten, sondern im Gegensatz dazu mit Vorlieben oder Vorurteilen bzgl. des weiteren Vorgehens zu tun haben, sind folgende Punkte hiflreich:

1. Bewusster Umgang mit dem Konflikt, bewusstes Hineingehen in den Konflikt
2. Lösung, Kompromiss oder Einigung für eine kurzfristigen Lösungsstrategie
3. Erörterung von Vorschlägen und Einigung auf eine mittelfristige Lösungsstrategie
4. Anbieten einer langfristigen Lösungsstrategie oder eine Perspektive

Diese vier Punkte erleichtern die Lösung des Konfliktes im *Beispiel 7* und vesetzen das Paar in die Lage, an den vorangegangenen Arbeitsprozess anzuschließen. Generisches und spezifisches Wissen ist hilfreich hinsichtlich der Angebote für Lösungen. C3 und C4 ist es anschließend somit möglich, einen schnellen Arbeitsfluss wieder herzustellen und die vorab beendete Fokus-Phase fortzusetzen. Togehterness muss nicht durch einen langen Prozess wieder aufgebaut werden.

4.2 Lücken in spezifischem Wissen

In Abschnitt 3.3. habe ich erläutert, dass hohes spezifisches Wissen voraussetzend für Fokus-Phasen ist. Werden während des Entwicklungsprozesses Lücken im spezifischem Wissen deutlich, wird anstelle der nächst folgenden Entscheidung ein Recherche-Prozess treten. Hierdurch reduziert sich der Arbeitsfluss deutlich hin zu PP_{normal} oder $PP_{breakdown}$. Diesen Zusammenhang möchte ich im folgenden verdeutlichen und Lösungen anhand geben.

Einordnung Sitzung AA1:

Die Entwickler A1 und A2 haben einen generelles Verständnis des Software-Systems. In vielen Situationen fehlt beiden spezifisches Wissen, um eine Lösungsstrategie zu erarbeiten. Weite Teile des Beginns der PP-Sitzung (ca. 30 Minuten) müssen investiert werden, um die Lücken hinsichtlich des zur Lösung der Aufgabe notwendigen, relevanten Wissens zu schließen. Erst dann erlangen beide einen Überblick und sind in der Lage, über eine Lösungen für konkrete Probleme nachdenken zu können. Jede weitere, neue Aufgabe stellt die Entwickler vor dieselbe Herausforderung, sich spezifisches Wissen aneignen zu müssen. Der Vorgang wiederholt sich durch die Sitzung hindurch. Durch diesen mühsamen Prozess werden sie in die Lage versetzt, das an sie gestellte Problem zu lösen.
A1 hat einen guten Überblick über das Software-System. Zusammenhänge, Funktionalität des Codes und dessen Abhängigkeiten sind A1 bewusst. In Details ergeben sich oft artikulierte Wissenslücken. Nach eigener Einschätzung stufe ich A1 bzgl. des spezifischem Wissens „Mittel“ ein, wenn auch Richtung „Hoch“ tendierend.
A2 drückt wiederholt und mehrfach spezifische Wissenslücken aus. Mittels Phasen des Wissenstransfers durch A1 werden die Lücken geschlossen. Es besteht im allgemeinen ein Verständnis für das Software-System. Die Einstufung muss deutlich unter A1 erfolgen, jedoch im „Mittel“ festzulegen.

Mit einem kurzem Beispiel möchte ich kurz, ein sich häufig in AA1 wiederholendes Szenario illustrieren:

Beispiel 8, Sitzung AA1, 29:37 – 29:57

Situation:
Beide Entwickler sind seit Beginn der Sitzung in einer Phase, bestehenden Programm-Code zu verstehen, um eine Lösung für das gestellte Problem zu entwickeln. Die Aufgabe besteht in der Änderung des Anzeigens bestimmter Klassen von Link-Zielen, je nachdem ob das Ziel gültig, offline oder veraltet ist. Die Entwickler versuchen herauszufinden, wo im Code die Anzeigefunktionalität umgesetzt worden ist, um sie anzupassen und weiterentwickeln zu können.

Verlauf:
Innerhalb von 20 Sekunden werden folgende Enstscheidungsprozesse besprochen:

1. A2 stellt fest, dass die Listing Page keine zur Lösung der Aufgabe relevanten Code enthält.
2. A1 stellt die in Schritt 1 artikulierte Feststellung in Frage und gibt einen Anhaltspunkt wo A2 weitersuchen soll.
3. A2 erklärt mit Verweis auf den Code, das der in Schritt 2 vorgeschlagene Teil irrelevant ist.
4. A2 möchte Wissen, was das in Schritt 2 annoncierte Modul ist.
5. A1 gibt Unwissenheit zum Ausdruck und bittet A2, zu recherchieren.

Kommunikationsverlauf:

1. <explain_finding> A2: “Die Listing Page rendert nix.”

2. <challenge_finding> A1: „Wieso? Die ist doch erreichbar durch die Component-Page. Die muss doch irgendwas rendern.“

3. <amend_finding> A2: Ja. Die rendert den Main Part.

   <agree_finding> A1: Jo.

4. <ask_knowlege> A2: Und was ist Component?

5. <explain_gap_in_knowledge> A1: Ja weiß ich auch nicht.
   <propose_hypothesis> A1: Das muss ja irgendwo definiert sein.

In Sitzung AA1 existieren Unterschiede zwischen den Entwicklern im Hinblick auf spezifisches Wissen. Durch Wissenstransfer können diese mit Aufwand ausgeglichen werden. Der wesentlich Unterschied zu Sitzung CA5 besteht in der Einordnung des spezifischen Wissens. Beide Entwickler drücken wiederholt Lücken des spezifischen Wissens aus. Diese müssen geschlossen werden, bevor das Paar in der Lage ist, mit dem Arbeitsprozess zu beginnen. Die Entwickler können keine Fokus-Phase erreichen, so lange ihnen nicht detailliert bewußt ist, wie sie das an sie gestellte Problem lösen können. Sie werden eine Strategie und deren taktische Schritte erst dann definieren können, wenn ein hinreichendes, die wichtigen Aspekte der an sie gestellten Aufgabe und der Lösung umfassendes Level an spezifischem Wissens erreicht ist.

Im Gegensatz dazu existieren in Sitzung CA5 keine relevanten, die Sitzung dominierenden, spezifischen Wissenslücken. Die Entwickler sind somit befähigt, direkt in die Aufgabe einzusteigen und mit der Arbeit, einer Phase der Strategiefindung zu beginnen. Die Gemeinsamkeit, dass es keinen relevanten Unterschied in der Kategorisierung spezifischen Wissens zwischen den beiden Entwicklern gibt, erübrigt Phasen des Wissenstransfers mit dem Zweck, selbige bei einem der Entwickler schließen zu müssen. Dieser Fakt befördert die Vereintheit des Paares und stellt die Voraussetzung der Synchronisation des Paares. Eine Fokus-Phase ist in Zusammenhang mit einer effizienten und validen Lösungsstrategie für das Problem und deren Teile ermöglicht.

Ein hoher Grad an sezifischem Wissen ist unablässig für eine Fokus-Phase. Ein Paar mit signifikanten Lücken kann PP_{fast} nicht erreichen. Nur nachdem die Lücken bewußt ausgedrückt und geschlossen

werden, kann eine Steigerung im Arbeitsfluss geschehen. Spezifisches Wissen im Gegensatz zu generischem Wissen kurzfristig angelegt und schnell erlernbar. Es ist durch Wissenstransfer möglich, diese Lücken innerhalb einer Sitzung zu schließen. Je nach Verhältnis der Lücken im spezifischen Wissen ergeben sich offensichtliche Lösungsstrategien.

1. Sind die Lücken im spezifischen Wissen einseitg und nicht zu gravierend in Richtung Ahnungslosigkeit eines der beiden Entwickler Vgl. [5, S. 3 f, Kapitel 4.1.], muss der Arbeitsprozess gestoppt und durch eine Phase des Wissenstransfers ausgeglichen werden. Somit wird Togetherness erhöht und anschließend durch die Fortsetzung des produktiven Prozesses eine Fokus-Phase ermöglicht.

2. Sind die Wissenslücken beiderseitig mittel einzustufen, muss der Arbeitsprozess gestoppt werden und in Zusammenhang mit hohem generischem Wissen kann schnell eine Schließung der Lücken erfolgen. Es erfolgt ein Wissenstransfer bei asynchronen Lücken oder eine gemeinsamer Recherche-Prozess. Anschließend kann der produktive Prozess weitergeführt werden und u.U. eine Fokus-Phase erfolgen.

3. Sind die Lücken einseitig zu gross, um sie kurzfristig zu schließen, kann die Aufgabe der PP-Sitzung dahingehend geändert werden, dass sie dem Wissentransfer dient. Andererseits kann der Entwickler ohne grosse Lücken im spezifischem Wissen die Aufgabe allein lösen, was im wahrscheinlichen Fall nicht Sinn der Sache ist. In einer nächsten anschließenden Sitzung kann der produktive Prozess fortgesetzt werden und sowohl Togetherness und der Arbeitsfluss in dieser wird deutlich erhöht sein.

4. Für den Fall, dass beide Entwickler Lücken spezifischem Wissens aufzeigen, welche zu gross sind um sie kurzfristig in einer PP-Sitzung zu schließen, kann sich das Paar entschließen die Sitzung abzubrechen und sich das fehlende, spezifische Wissen im Selbststudium, gemeinsam oder durch Dritte anzueignen. Anschließend kann ein produktiver Prozess erfolgen, welcher einen erhöhten Arbeitsfluss hervorbringen wird.

Entscheidend im Umgang mit Lücken im spezifischem Wissen ist, mit selbigen bewußt umzugehen. Sie müssen erkannt und beseitigt werden, um einen hohen Arbeitsfluss zu ermöglichen. Sind sich beide Entwickler oder einer der Entwickler nicht im Klaren, dass Lücken im spezifischem Wissen vorhanden sind, werden sie das an sie gestellte Problem der PP-Sitzung entweder nur mit grossem Zeitaufwand, sehr stockendem Arbeitsfluss zwischen alternierenden Phasen in PP_{normal} und $PP_{breakdown}$ oder gar nicht lösen können.

4.3 Transition strategischer Abschnitte

In Abschnitt 2.4. habe ich rezeptartiges Arbeiten als Voraussetzung für Fokus-Phasen erläutert. Das Rezept wird vorab in der Phase der Strategiefindnung definiert. Für jeden strategischen Abschnitt existiert dieses Rezept und muss beiden Entwicklern detailliert klar sein. Innerhalb einer Fokus-Phase muss zwingend ein folgender Arbeitsschritt klar sein und eine Entscheidung schnell gefunden werden. Mit dem Abschluss eines strategischen Abschnittes ist dies meist nicht der Fall. Somit wird die Fokus-Phase beendet.

Wird ein strategischer Abschnitt abgeschlossen, folgt in den analysierten Sitzungen eine Test-Phase, ein Eintrag in das Versionierungssystems und anschließend die strategische Festlegung kommender Abschnitte. Die Togehterness wird durch die Strategiebesprechung anschließend wieder aufgebaut. Beiden Entwicklern ist anschließend wieder klar, wie die kommenden taktischen Schritte aussehen werden und ein neues Rezept wird definiert. Dies ermöglicht eine weitere Fokus-Phase.

Darüber hinaus ist es möglich die komplette Sitzung vorab zu besprechen. Jeder Teilabschnitt muss entsprechend definiert werden, was sehr hohes spezifisches Wissen voraussetzt. Entsprechend ist es somit möglich, nach einem abschließendem letztem Arbeitsschrittes im direkten Anschluss den ersten, dem nächsten strategischen Abschnitt zugehörigem Schritt, folgen zu lassen. Sind die in Kapitel 2 erläuterten Voraussetzungen für Fokus-Phasen gegeben, wird eine vorangegangene nicht unterbrochen. Dabei ist es unabhängig davon, ob anschließend getestet, implementiert oder andere Softwareentwicklungs-Prozesse durchgeführt werden.

5. Zusammenfassung

Diese Arbeit auf der Basis qualitativer Daten-Analysen soll einen Überblick über die Charakterisierung von Fokus-Phasen in PP geben. Das Ziel ist ein Verständnis zu geben, was Fokus-Phasen sind, welche Vorbedingungen für selbige existieren und welche Faktoren PP_{fast} beeinflussen. Darüber hinaus habe ich für einige Faktoren Lösungsstrategien erläutert. Diese sollen helfen Togehterness aufzubauen und aufrecht zu erhalten, den Arbeitsfluss zu beschleunigen und folgend die Produktivität zu verbessern.

Die wesentlichen, beschriebenen Beobachtungen und resultierenden Erkenntnisse dieser Arbeit sind:

- Aufbau und Erhalt von Togetherness ist wesentlicher Bestandteil einer PP-Sitzung.
- Togehterness ist Voraussetzung für einen hohen Arbeitsfluss (PP_{fast}).
- Fokus-Phasen werden charakterisiert durch Folgen einer Vielzahl von Entscheidungen in kurzen Zeitabständen. Entscheidungsprozesses beinhalten Vorschläge mit schnell folgender Zustimmungen des Partners auf der Basis von Konsens im Entscheidungsprozess.
- Rezeptartiges Abarbeiten ist eine Voraussetzung von PP_{fast}. Kurze, schnelle Referenzierung auf den nächst folgenden Arbeitsschritt, das nächste zu betrachtende Code-Artifakt oder Entwurfsobjekt ohne Diskussion darüber, wie damit umzugehen ist, ist voraussetzend für PP_{fast}.
- Der Konsens über ein Rezept für folgende, atomare Arbeitsschritte erhöht Togetherness und ermöglicht rezeptartiges Abarbeiten.
- Hohes spezifisches Wissen ist eine notwendige Voraussetzung für Fokus-Phasen.
- Detaillierte Klarheit des Problems der zu lösenden Aufgabe ist voraussetzend für PP_{fast}.
- Die Geschwindigkeit der Erarbeitung und die Qualität der Lösungsstrategie setzen ein umfängliches, detailliertes Verständnis der zu lösenden Aufgabe voraus.
- Gradliniges Arbeiten ermöglicht PP_{fast} und hilft Togetherness aufrechtzuerhalten. Darüber wird die Bewältigung der Aufgabe erleichtert und kann Komplexität und Schwierigkeitsgrad verringern.
- Lücken im spezifischen Wissen reduzieren Togetherness und folgend den Arbeitsfluss. Sie können Fokus-Phasen abbrechen oder verhindern.

- Hohes generisches Wissen hilft, Lücken im spezifischem Wissen zu schließen, um anschließend PP_{fast} zu ermöglichen.
- Konfliktlösungsstrategien helfen Togetherness aufrechtzuerhalten, insbesondere durch Konsens einer kurz-, mittel- und langfristigen Lösungsstrategie der Konfliktursache

6. Perspektiven weiterer Arbeiten

Während der qualitativen Analysen bin ich, über die hier erläuterten Konzepte hinaus, auf weitere, speziellere aufmerksam geworden, welche nicht Bestandteil dieser Arbeit werden konnten. Ich möchte diesen Abschnit nutzen, um diese für weitergehende Studien zu annoncieren:

1. Rollenwechsel und Togetherness

Die Unterscheidung in Schreibenden (Driver) und Observierenden (Observer) als grundlegend festgelegte Binome der Paardynamik ist in der Praxis so nicht zu beobachten. Vielleicht gibt es diese strenge Teilung der Entwickler in diese Rollen, jedoch konnte ich selbige in dem mir zur Verfügung stehendem Material nicht bestätigen. Es existieren aber Rollen innerhalb der Paardynamik. Ein interessanter Aspekt ist der Wechsel der beiden Rollen für strategischen Abschnitte.

In der Praxis ändert sich das Verhalten oft, insbesondere innerhalb von Fokus-Phasen. Der schreibende Partner gibt den Ton an und bestimmt dominant die Richtung. Der observierende Partner stimmt zu oder bringt Vorschläge. Mit einem Wechsel ändert sich die Rollenverteilung ebenso. Dies läßt den Schluß zu, dass sich dieses Phänomen nicht aufgrund von Persönlichkeitsstrukturen selbst implementiert. Eher wird bei konzentriertem, rezeptartigem Abarbeiten unter der Voraussetzung etablierter Togetherness einem Partner gefolgt. Dies geschieht so lange keine Einwände oder Fragen aufkommen. In den analysierten Situationen in Fokus-Phasen ist dies der schreibende Partner, was nicht so sein muss.

2. Rollenwechsel und PP-Skill

Der Rollenwechsel, wie im ersten Punkt beschrieben, befördert u.U. PP-Skill und die Synchronisation des Paares. Somit wird zukünftig Togetherness leichter erreicht und einfacher aufrechterhalten. Hierdurch wird die Produktivität erhöht und Aufwand reduziert. Das Setup der Arbeitsplätze (Anzahl der Eingabe- und Ausgabegeräte etc.) spielt keine entscheidende Rolle dabei.

3. Einflüsse von eingespielten PP-Paaren auf Togetherness

Eingespielte Paare werden in Konfliktsituationen gelassener reagieren und effizienter eine Lösung finden können. Sie kennen ihre Stärke und Schwächen und die des Anderen. Darüber hinaus wissen sie mit Eigenheiten des Partners umzugehen, was das Konfliktpotential verringern kann. Auf der anderen Seite könnten Probleme durch aufgestaute, ungelöste Situationen auftreten. Durch diese Phänomene steht ein Einfluss auf die Togetherness und den Arbeitsfluss zu vermuten.

4. Reduzierung des Schwierigkeitsgrades der Aufgabe

Der Einfluss eines erhöhten Schwierigkeitsgrades der Aufgabe wird Togetherness erschweren. Ich vermute bei hohem PP-Skill und spezifischem Wissens andererseits das Potential, dieses Dilemma

auflösen zu können. Bspw. könnte die Komplexität der Aufgabe durch Unterteilung in strategische Abschnitte unterschiedlicher Schwierigkeitsgrade reduziert werden. Das Verschieben der schwerer zu lösenden Teile wird die Möglichkeit für Fokus-Phasen schaffen. Die erreichte Togetherness könnte u.U. in folgende, schwere Teilaufgaben überführt werden. Darüber hinaus würden beim Lösen der leichten Aufgabenteile Lerneffekte durch Wissenstransfer, falls dies nötig sein sollte, spezifisches Wissen erhöhen.

5. Sonderfälle spezifischem Wissens

Ein Sonderfall im Zusammenhang mit spezifischem Wissen ist für mein Dafürhalten interessant. Es existieren Aufgabenstellungen in PP-Sitzungen, in denen per se kein spezifisches Wissen vorhanden sein kann. Dies ist genau dann der Fall, wenn es kein System gibt, auf welchem aufgebaut wird: bspw. ein Software-Entwurf oder der Beginn einer Implementierung ohne bereits bestehenden Code etc. In diesen Fällen sollte spezifisches Wissen als Voraussetzung für PP_{fast} untersucht werden.

Literaturverzeichnis

1. **A. Strauss, J. Corbin.** *Basis of Qualitative Research, Grounded Theory Procedures and Techniques.* s.l. : SAGE Publications, Inc, 1990. ISBN 0-8039-3251-0.

2. **Salinger, S.** *Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung.* 2013. Dissertation am FB Mathematik und Informatik, FU Berlin

3. **L. Prechelt, S. Salinger.** *Understanding Pair Programming, The Base Layer.* s.l. : Books on Demand, 2013. ISBN 978-3-7322-8193-0.

4. **L. Prechelt, F. Zieris.** *Pair Programming Feasibility Critical Depends on Task Difficulty 2017 (in Vorbereitung)*.

5. **L. Prechelt, F. Zieris**. *When Pair Programming is most helpful: Explaining session dynamics from knowledge gaps.* ICSE'18, 2018, Gothenburg, Sweden, 2017. S. 3. Kapitel 3.1.

6. **J. Chong, T. Hurlbutt.** *The social dynamics of pair programming.* 29th Int'l. Conf. on Software Engineering. 2007. S. 4. chapter 5.1.2..

7. **L. Plonka, J. Segal, H.Sharp, J. van der Linden.** *Collaboration in Pair Programming: Driving and Switching. Center of Research in Computing, The Open University, Milton Keynes, UK,* s.l. : Springer, *2011*

8. **R. R. Kessler, L. Williams, W. Cunningham, R. Jeffries.** *"Strengthening the case for pair programming".* no. 4. s.l. : IEEE Sofware4, 2000. pp. 19-25. Vol. 17.

9. **J. E. Hannay, T. Dybå, E. Arisholm, D. I. K. Sjøberg.** *The effectiveness of pair programming: A meta analysis.* 7, July 2009, Information and Software Technology, Bd. 51, S. 1110-1122.

A Anhang

Analyse der Äußerungen in *Beispiel 2*:

1.
In Entscheidung 1 sagt C3 zu Beginn des *Beispiels 2*: „Okay, da gibts schon mal jede Menge Sachen die wir nicht brauchen oder nicht haben.“ C3 erklärt hiermit implizit, wie die folgenden, kommenden, taktischen Schritte durchgeführt werden sollen. C4 versteht diesen Vorschlag und nimmt ihn an. Dies zeigt sich insbesondere durch ihr Verhalten in Kombination mit dem Ausbleiben einer Antwort. Nichts desto trotz habe ich die Illuktion der Aussage aus zwei Gründen mit „propose_design“ kodiert. Erstens wiederholt C3 hiermit nur, was vorher bereits als Strategie festgelegt wurde. Zweitens löscht C3 gleichzeitig die Return Anweisung der bearbeiteten Methode. Der engere Fokus liegt in der produktorientierten Handlung. C4 bestätigt den Vorschlag durch kurzes, gutierendes Nicken mit dem Kopf.

2.
C4 macht aufbauend den Design-Vorschlag, die Return-Anweisung einer Subfunktion innerhalb der Methode zu ändern und dort zu beginnen. C4 ist bewusst, dass C3 bereits in Entscheidung 1 den Rückgabewert der Methode gelöscht hat. C4 sagt wörtlich: „Return... Da kannst du ja anfangen“. Geichzeitig zeigt sie kurz illustrierend auf den Bildschirm. C3 reagiert prompt und löscht den Rückgabewert. Auch hier handelt es sich wörtlich um einen Vorschlag, einen nächsten Arbeitsschritt aufzuzeigen. Da C4 einen Vorschlag bzgl. eines Code-Artifaktes macht und C3 dies durch die Löschung des Rückgabewertes so versteht, habe ich die Aussage mit „propose_design“ kodiert. Aufgrund der direkten Anpassung des Codes durch C3 erfolgt, stimmt C3 dem Vorschlag direkt zu.

3., 4.
Bei näherer Betrachtung macht C4 anschließend den Design-Vorschlag, nicht nur den Rückgabewert der Subfunktion anzupassen, sondern sie komplett zu löschen: „Wobei wir das validator nicht brauchen.“ C3 gibt gleichzeitig Skepsis darüber zum Ausdruck, ob die Validierungsfunktion tatsächlich nicht gebraucht wird: „Wahrscheinlich wird das wieder... je nachdem.. validator...“. Im strengen Sinne ist kein konkreter Vorschlag enthalten. C4 versteht die Bedenken C3´s trotzdem und antwortet: „Vielleicht brauchen wir es, aber das kannst du es immer noch wiederholen.“ Es wird mit dieser Aussage ein „todo“ vorgeschlagen, um kurzfristig eine Lösung für das weitere Verfahren dieses Code-Artifaktes zu erhalten. C3 stimmt daraufhin dem Vorschlag, die Subfunktion später wieder hinzuzufügen, zu und löscht die Anweisungen: „Ja, würde ich wegmachen.“ Die Annoncierung eines kurzfristigen Lösungsvorschlages (jetzt zu Löschen) und einer mittelfristigen Perspektive (wieder einfügen, falls die Funktion benötigt wird) ermöglicht eine schnelle Entscheidung.

5.
Ohne Verzögerung geht C3 zum nächsten Schritt über. Der Vorschlag: „Okay, SnapThemeModel... Das brauchen wir.“ gibt C4 den nächsten taktischen Arbeitsschritt vor, welche Code-Zeile zu betrachten ist. C3 schaut während der Äußerung nicht zu C4. C4 bestätigt den Vorschlag zwar kopfnickend folgend: „Das brauchen wir“, allerdings wird mit dem Vorschlag lediglich Information weitergegeben, ohne eine Evaluierung des Vorschlages durch den Partner zu erwarten. Dieser Fakt in Kombination mit der folgenden Zustimmung signalisiert ein hohes Maß an Synchronisation des Paares und ein „mit einem Verstand“ arbeitenden Zustand. Die Entscheidung 5, den Code der Methode „SnapThemeModel“ beizubehalten, wird somit konsolidiert.

6.
Der Vorschlag für die Design-Entscheidung 6 beinhaltet eine Aufforderung zur Evaluierung durch den Partner. C3 blickt während des Vorschlages zu C4 und ergänzt seinen Vorschlag begründend mit zusätzlichen Informationen: „FeatureLayerBaseScale... Das brauchen wir eigentlich auch, weil wir das

spliten (...) machen wir eine neue Linie (...) ins Polygon rein und das sollte Passgenauigkeit haben.“ C4 wartet den Vorschlag von C3 ab und bestätigt ihn kurz: „Mh-hmm, wir brauchen schon den FeatureLayer den wir editieren wollen.“ C3 nimmt die Bestätigung an und erwidert zustimmend: „Genau“. In dieser Entscheidung wird trotz offensichtlich erforderlicher, im Vorschlag enthaltenden, zusätzlichen Kontexten schnell eine Entscheidung getroffen. PP_{fast} bleibt bestehen.

7., 8.

Zeitgleich markiert C3, den nächsten taktischen Arbeitsschritt anzeigend, die nächste zu betrachtende Code-Zeile und macht den Vorschlag: „getEditStrategy, EditGeometryType: Das ist falsch!“ C3 drückt damit aus, dass hier eine Änderung geschehen muss. Wie diese Änderung auszusehen hat ist sagt er zwar nicht, jedoch wissen C4 und C3 was die Anforderungen an ihren Code sein müssen. C4 nimmt dies zum Anlass und antwortet: „Das ist auf jeden Fall... Polygon, genau.“ C3 benötigt die enthaltende Information der Antwort nicht und gibt zeitgleich zu C4´s vorhergehender Aussage zu verstehen: „Hier nehmen wir immer...“ In Zusammenhang mit den schnellen Änderungen im Code ohne jede Verzögerung, dienen die sprachlichen Äußerungen zu diesem Zeitpunkt nur dem Zweck der Beibehaltung der Synchronisation. Beide sind sich einig über die Art und Weise der Änderung des Codes. Die Vorschläge werden direkt umgesetzt ohne eine Antwort abzuwarten. Es erfolgt kein Gegenvorschlag oder eine Erweiterung des Vorschlages des Partners.

Analyse der Äußerungen in *Beispiel 3*:

1.
C4 beginnt den Dialog mit dem Vorschlag, als nächsten taktischen Schritt eine Betrachtung der Methode „getEditStrategy“ vorzunehmen. C4 drückt damit mehrere Dinge auf verschiedenen Ebenen gleichzeitig aus, welche in Schritt 1 zusammengefasst sind. Die einzelnen Implikationen sind folgende:

1. Der nächste taktische Schritt ist die Betrachtung der getEditStrategy.
2. C4 legt damit fest, dass zur Lösung der nächsten Teilaufgabe eine Strategie verfolgt wird, bestehenden Code wiederzuverwenden und darauf aufzubauen.
3. C4 hat bereits eine Idee, welcher Code in der Programmbibliothek dafür benutzt werden kann.

C3 versteht den Ansatz und hat nicht einzuwenden oder hinzuzufügen.
C4 nutzt die Funktionalität der Entwicklungsumgebung „Eclipse“ effizient und tippt „getEditStrategy“ ein, um Vorschläge vor Methoden zu erhalten, die bereits ähnliche Funktionalitäten umsetzen, welche die Anforderungen der nächsten, an die Entwickler gestellten Teilaufgabe erfüllen.

2.
C4 hat wiederum bereits eine Idee und äußert in Schritt zwei: „...und da gibt es einen con....“. C4 betrachtet das aufgehende Kontext-Menü kurz. In weniger als zwei Sekunden entscheidet C4, dass es sich um eine andere Methode handelt, die gesucht wird und löscht die Eingabe „con“ wieder. C4 entscheidet sich für die Betrachtung aller Methoden, Vorschläge durch die Entwicklungsumgebung erwartend und äußert während des browsens durch die Methodenliste: „...applyGeometryChange...“. Der Mauszeiger wird auf dem gleichnamigen Methodennamen plaziert, um weitere Informationen durch die Entwicklungsumgebung zu erhalten. Nach einer Sekunde entschließt sich C4 für die Auswahl der Methode und bestätigt sie durch Eingabe. Folgend betrachten beide Entwickler die zu übergebenen Parameter für die Methode.

3.
Öhne Verzögerung markiert C4 den Übergabepararmeter „getFeatureLayerSelection()“. Durch die Entwicklungsumgebung bereitgestellte Information wird der Kontext des markierten Parameters angezeigt. C4 sagt im direkten Anschluss: „...und die Geometry müssen wir ermitteln.“ Mit diesem Satz äußert C4 abschließend einen Vorschlag für die Strategie des folgenden Arbeitsabschnittes.

Darüber hinaus werden zeitgleich mehrere Dinge auf verschiedenen Ebenen zum Ausdruck gebracht:

1. Die Methode applyGeometryChange() erfüllt die gewünschten Voraussetzungen und kann verwendet werden, um die nächste Teilaufgabe umzusetzen.
2. Die Übergabeparameter „featureLayerSelection“ und „featureProxy“ müssen nicht näher betrachtet werden, da C4 bewusst ist, wie sie eingebunden werden können.
3. Der dritte Übergabeparameter des Typs „Geometry“ muss noch ermittelt werden.
4. C4 drückt durch Punkt 3 aus, dass es möglich ist, dies zu tun. Implizit wird das dadurch ausgedrückt, dass C4 den Vorschlag gutierend äußert und lediglich Information sendet, ohne den Wunsch bzgl. Evaluierung durch ihren Partner.

Ein überdurchschnittlich hohes Maß an Synchronisation der beiden Partner voraussetzend, versteht C3 die vier Punkte und bestätigt dies gutierend anschließend kurz durch nicken mit dem Kopf und verbal: “Mh-hmm.”

4.
C3 wiederholt anschließend den konkreten Vorschlag der durch C4 erläuterten Lösungsstrategie auf abstrakter Ebene: „Was wir jetzt machen... also was wir jetzt machen könnten, wäre aus dem

bestehendem Objekt den Teil wegschneiden, der dann in ein neues kommt.“ Direkt anschließend bestätigt C4 den Strategievorschlag: „Ja genau. Das ist die Aufgabe.“
C3 ordnet anschließend die gewählte Strategie in den Gesamtkontext der Sitzung ein und sagt: „Na es ist ein Teil der Aufgabe.“ C3 weist darauf hin, dass die Entwickler mit dem Abschluss der besprochenen Teilaufgabe nicht fertig sein werden. C4 bestätigt dies: „Das ist jetzt für diesen Task die Aufgabe.“ Durch diese beiderseitig einvernehmliche Entscheidung sind beide Entwickler für den weiteren Arbeitsablauf des kommenden, produktorientierten Teiles der Sitzung eingeordnet und synchronisiert. C3 bestätigt noch einmal sein Verständnis für die Anforderungen und bzgl. des durch C4 beschriebenen Lösungsansatzes: „Okay. Also wegschneiden, gut dann...“.

Analyse der Äußerungen in *Beispiel 6*:

1.

C3 beobachtet die Eingaben von C4 aufmerksam zu Beginn eines beginnenden strategischen Schrittes. C4 fängt ohne Vorbesprechung einen nächsten taktischen Schritt an. C3 fragt nach der Lösungsidee von C4. C4 referenziert kurz auf das nächste, zu bearbeitende Code-Artifakt.

2.

C3 signalisiert C4 die Eingabe zu stoppen und äußert den Wunsch nach Kommunikation. C3 gibt zu verstehen, dass es eine weitere Lösung gibt, welche schneller implementiert werden ohne zu diesem Zeitpunkt genau zu erklären, welche das ist.

3.

C3 erklärt, dass der von C4 gewählte Weg die nahe liegende Vorgehensweise wäre. Augrund hohem spezifischem Wissens ist C3 in der Lage, die Validität des Lösungsvorschlages von C4 zu evaluieren. C3 ordnet bereits zu Beginn der Implementierung durch C4 die Lösungsidee ein und signalisiert C4 eine schnellere Lösungsidee, welche C3 gern illustrieren würde.

4.

C3 bietet durch kurze, konkrete Referenzen auf Stellen in der Bibliothek eine Verbesserung, welche C3 ohne sehr gute, detaillierte Kenntnis des Software-Systems, insbesondere des bestehenden Code und dessen Abhängigkeiten, als auch den Anforderungen an die für die Lösung benötigten Mittel, nicht kennen würde. Ein hohes spezifisches Wissen wird hiermit ausgedrückt. Auch wenn es in der Operationalisierung spezifischem Wissens um das Ausdrücken von Lücken geht, spielt das in diesem Beispiel gezeigte hohe Maß im Gesamtkontext eine Rolle und unterstützt die Einordnung „Hoch“.

Analyse der Äußerungen in *Beispiel 7*:

1.
Wichtig zu erkennen ist, dass C3 mit der Art und Weise des Vorschlages, sehr bewußt in den Konflikt geht. Gemeint ist nicht der Wunsch nach Eskalation einer Situation. Vielmehr geht es um das Bewußtsein, dass im folgenden Vorschlag die ensure-Funktion zu benutzen, Streit-Potential vorhanden ist. Trotzdem äußert C3 den Vorschlag und wartet auf die Reaktion von C4.

2.
Wie erwartet lehnt C4 den Vorschlag vehement und deutlich ab. Die analysierte Körpersprache soll über die gesprochenen Worte hinaus einen Einblick geben, wie deutlich dies passiert. C4 reagiert sehr emotional, wenn auch damit spielend. C4 steigert sich während der kommenden Diskussion in die hier noch spontane emotionale Reaktion, folgned bewusst hinein.

3.
C4 macht einen alternativen, den eingangs von C3 geäußerten Design-Vorschlag nicht beinhaltenden Gegenvorschlag. Beide Entwickler sind zu diesem Zeitpunkt nicht mehr zusammen. Togetherness ist nicht existent, falls kein Kompromiss oder einhellige Lösung gefunden werden kann.

4.
C3 bietet eine kurzfristige Lösung, welche keine generelle Lösung darstellen soll. Die Absicht ist nicht sich durchzusetzen, sondern einen Kompromiss anzubieten. C4 lehnt dies zwar nicht kategorisch ab, bringt allerdings zum Ausdruck, dass C4 nicht glücklich über den Vorschlag ist.

5.
C4 nimmt den Vorschlag für die kurzfristige Lösung unter der Bedingung an, dass der Vorschlag weder eine mittelfristige oder langfristige Lösung sein soll. Die Option die ensures während der späteren Test-Phase zu entfernen ist essentiell für die Zustimmung.

6.
C3 macht einen langfristigen Lösungsvorschlag, in einer späteren Diskussion Vorschläge zu erörtern, wie die Funktionalität besser umgesetzt werden kann. C3 signalisiert Kompromissbereitschaft. Die Bedingung ist ein sinnvoller Vorschlag, bspw. durch Einbinden eines Packages. welches dies anstelle der ensures umsetzt.

Danksagung

Hiermit möchte ich mich gern bei Franz Zieris bedanken. Ohne die zahlreichen Stunden der Diskussionen wäre diese Arbeit in der Form nicht möglich gewesen. Vielen lieben Dank für deine Zeit und die Geduld mit mir.

Vielen Dank ebenfalls an Dr. Holger Harms für die Hinweise und Tips bei den abschliessenden Korrekturen.